Synthesis of delay functions in DSP compilers

A. Delaruelle, O. McArdle, J. van Meerbergen, C. Niessen
Philips Research Labs, Eindhoven, The Netherlands

Abstract
An efficient method for the synthesis of delay lines is reported. The solution is based on a well-defined way of organising the data in memory which allows an efficient synthesis of delayed arrays, and has been implemented in a DSP compiler. Examples demonstrate that the efficiency of the automatic implementation is comparable to manual solutions.

1 Introduction
Delay lines are essential in many DSP applications. High level specification languages like Silage [1] therefore have special operators ($\oplus$) to represent the $z^{-1}$ delay operation. The mapping of these operators is a step during the design process that is very error-prone and requires a lot of book-keeping. Accordingly the automatic synthesis of these operations is an important step for a DSP compiler.

When storing delay lines in memory the complete delay line must be correctly updated for the next sampling period. Two possibilities exist. The first solution moves the data in the memory, which requires a lot of extra operations. A second approach will therefore be followed, where circular buffers using pointers can be used in such a way that the data always remains at the same physical position in the memory. In this case the necessary hardware support for modulo address operations and masking must be present. Both techniques are well known and are supported by most general purpose DSP processors. However most of the current DSP compilers for ASIC chips pay little attention to this problem, despite the fact that this can be more important than the data path design.

Circular buffers are one-dimensional data structures in memory. Using them permits one-dimensional delay lines, which occur for example in digital filters by delaying scalar values, to be mapped directly. The problem becomes much more complex however when different elements of an array must be given different delay values (delays on arrays). This basically adds a new dimension to the problem because considering these as individual delay lines may become very inefficient.

The paper is organised as follows. First an example will be given to show the relevance of the problem. Next the mapping of delayed arrays using circular buffers with pointer addressing will be discussed. The method can be divided in two steps:

1. First the organisation of the memory will be discussed and the position of the elements will be determined. The main goal in this step is to minimise the area by minimising the memory size. In section 3 a new memory organisation (called element store) will be introduced whose results are close to those achieved manually.

2. In a second step addresses will be generated in such a way that the number of calculations is minimal. This will be discussed in section 4.

Finally conclusions will be drawn.

2 An example
As an example, let us consider the deinterleaving function as used in the Compact Disc error correction algorithm. This part of the CD system is shown in figure 1.
The data is read from disc and grouped into blocks (frames). An input array (In) consists of 32 words. This is offered to a first Reed-Solomon decoder (C1) which performs a number of arithmetic operations (not shown in detail). The resulting array (x) is input for the deinterleaving, which performs some shuffling on the data whereby each data component of a block is given a different delay, the goal being to increase the burst error correction capability. The output of the deinterleaving (y) is offered to a second Reed-Solomon decoder (C2) which generates the output (Out). Making an abstraction of the internal details of the C1 and C2 decoders, the problem can be written in Silage as follows:

```
func main (In : num<8,0>) Out : num =
begin
x[i] = C1(In(i));
(1:0..27):

y[i] = x[i] @ (4*i); Out[i] = C2(y[i]);
end;
```

Silage is a functional language where we can make use of arrays, but where nothing is said about their implementation. This is elegant from a specification point of view, but as a consequence the synthesis tools are responsible for the correct addressing.

From figure 1 it can be seen that the minimal amount of memory is equal to 12 Kbit. This minimum size has been obtained in previous CD generation chips using a manual design style.

As a first solution we could try to implement every delay individually - we then end up with 28 delay lines. If we use circular buffer techniques we need 28 different pointers, and 28 different modulo registers, which leads to an extra area cost. Before accessing any of these 28 delay lines we have to update its pointer. This generates a number of extra addressing instructions before we can even start accessing the delay line. This design is therefore far from what can be obtained manually, and in conclusion we can state that treating delay lines individually like this would not be acceptable for the user, in terms of both area and performance. In what follows an alternative and more efficient solution will be discussed - the basic idea being to consider the delay construct as one entity instead of a collection of individual delay lines.

The problem that we will discuss in this paper can therefore be formulated as follows. We have to find an efficient implementation for delays on arrays, where every element of the array can have a different delay. In Silage terms we are looking for an efficient mapping of an expression of the following type:

\[ x[P_1(i) \oplus P_2(i)] \quad (1a) \]

where \( P_1(i) \) and \( P_2(i) \) represent polynomial functions of a variable \( i \), which is the index of the loop in which the expression occurs.

### 3 Memory organisation using one circular buffer: from array store towards element store

As discussed in the introduction, the first step is the definition of a memory map, i.e. a well-defined organisation of all the variables in memory. Two possibilities using circular buffers for storing delay lines in memory will be compared. The first one is called array store, which considers a vector \( x \) as one entity and stores complete delayed versions of the array together in memory. For every possible value of the delay expression a complete array is held. Figure 2 shows how such an array is stored in memory. The advantage of this method is that the structure can be controlled with 1 pointer, which is updated on every repetition of the algorithm with a step equal to the number of elements in the array. The disadvantage of this method is that a complete version of the array must be kept in memory until the last read of any individual element, which can lead to a considerable overhead and unacceptable solutions. In the case of the CD example this would lead to a doubling of the RAM size compared with manual solutions (24K instead of 12K). Since this memory already consumes 50% of the chip area the solution is totally unacceptable.

A new memory map has therefore been designed, called element store. In element store all delayed versions of each array element are stored together - figure 3 shows the same delayed array as in figure 2, but now stored with element store. In this way the disadvantage of array store, when the number of delayed versions that must be stored can be different for every element, can be avoided. The whole structure can still be controlled with 1 pointer. This will be illustrated using an example.

Assume that we have to map the expression

\[ x[i] @ i \quad (1b) \]

where \( i \) takes the values from 1 to 4. This is a special case of equation 1a. Figure 4 shows the situation in the memory after every execution of the algorithm i.e. after every sampling.
Figure 3: Element Store

Memory positions

<table>
<thead>
<tr>
<th>m1</th>
<th>m2</th>
<th>m3</th>
<th>m4</th>
<th>m5</th>
<th>m6</th>
<th>m7</th>
<th>m8</th>
<th>m9</th>
<th>m10</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p=2</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p=3</td>
<td>x1</td>
<td>x2</td>
<td>x1</td>
<td>x3</td>
<td>x4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p=4</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td></td>
</tr>
<tr>
<td>p=5</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td></td>
</tr>
<tr>
<td>p=6</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: period. We will show that this problem can be mapped using a memory containing only 10 locations. The memory positions are represented by m1, m2, m3 ... m10, the sampling periods by p. We will assume modulo-addressing.

For a sampling period p we have 4 input data, represented as x1, x2, x3, x4. The subscripts indicate the element number of array x, the superscripts the sampling period. During period 1, x1, x2, x3, x4 are written at the positions indicated in the figure. During sampling period 2, x1 is read, and x2, x3, x4 are written. We can see that the spacings between write positions are exactly the same as in period 1. Assuming the read operation occurs before the write, we may write x2 at the same position as x1. During sampling period 3, x1, x2 are read and x3, x4 are written. Again x1 may be written in place with x3, and the same holds for x2 and x4. Continuing this exercise, we can see that all four new input data may always be written at positions that have been read just before, giving a minimum memory size. The shift of the write-positions in every sampling period combined with the above-mentioned modulo-addressing performs the rotation of the delay line in memory. The system works with only 1 pointer - relative to this pointer the addresses are the same in every cycle. This already indicates simple addressing. If we check vertical columns we can see that x1 resides in memory for 1, x2 for 2, x3 for 3 and x4 for 4 sampling periods respectively. This is consistent with equation 1b.

Figure 5: Memory map for two consecutive points in time

This method will now be generalised and formalised. Figure 5 shows what happens in general between two consecutive points in time, the first column showing the situation at time (t-1). We now go to the next time point and decrement the pointer (pz) by 1 (modulo the size of the delay line). Because we step to the next point in time, all delays are incremented. The second column shows the same situation, except that all positions where a read occurs have been marked as empty. New data is coming in, and the method allows us to write at exactly the same positions which have just been read. The memory usage is therefore optimal and as efficient as manual design, unlike array store.

Figure 6 presents a memory map in a different form. We assume the same example (equation 1b) except that the index now runs from 1 to n. Shown are the number of delayed versions for every element of the array that we have to keep in memory. The slanting line marks the maximum delay for each element and the numbers indicate the memory positions (e.g. x[4][4] is stored at position 10). 10 positions are sufficient when the index runs from 1 to 4. The maximum delay per element is calculated at compile time, and enough space is allocated to store only this and nothing more, leading to the memory usage of figure 6. This example would result in the minimum 'triangle' of memory being allocated, a saving of 50% over array-store.

Because we need it in the next section a function Q(k) will now be defined which indicates the number of delays for array
element \( k \) stored in the memory. This function is defined as:

\[ Q(k) = P_2(j) \text{ where } j \text{ satisfies } P_1(j) = k \]  

(2a)

and \( P_1, P_2 \) are as introduced in equation 1a. For example 1b this function reduces to:

\[ Q(k) = k \text{ for } k = 1, 2, 3, \ldots n \]  

(2b)

4 Generation of addresses

Once the memory map has been fixed, we can start the address generation. In the PIRAMID [2] and CATHEDRAL II [3] compilers an Address Calculation Unit (ACU) is defined for this purpose. The instruction set includes dyadic operations (addition and subtraction) and monadic operations (increment, decrement, pass). All operations can be executed using modulo arithmetic. Input data are available in two input register files (see figure 7).

We now have to generate a number of basic operations in such a way that the correct precedences are calculated. We also have to take account of the correct precedences. The graph constituting the input for the scheduling step is constructed in the following way, in the order given:

1. On processor-initialisation the pointer of the delay line (represented as \( p_u \)) must be loaded with a start value.
2. Because several modulo operations can be merged on the same ACU the correct modulo value must be loaded in its modulo register.
3. Now we can update the pointer. In pseudo code this can be represented as: \( p_v = p_v \oplus l - 1 \mod \text{size of delay line} \). This performs the required rotation by 1 every sampling period.
4. We next have to calculate the address needed for the first iteration of the loop.
5. Within the loop we have to calculate the address for the next iteration. This will be done incrementally, i.e. using the address of the current iteration. This part is the most critical one because it is repeated for every iteration.

From the memory map an expression for the address can be obtained, because the position in the memory can be calculated by accumulating all delays \( Q(k) \) for all elements 1, 2, 3, \ldots \( (P_1(i) - 1) \). This leads to:

\[ k \times [P_1(i)] \oplus P_2(i) = \text{pos}(i) = p_v + \sum_{k=1}^{i-1} Q(k) + P_2(i) \]  

(3a)

Now two cases can be distinguished:

(a) \( P_1(i) \) is of the form \( i + a \) where \( a \) is a constant

Since \( k = P_1(i) = i + a \) we obtain \( i = k - a \) and \( Q(k) = P_2(k - a) \). It can thus be seen that \( Q(k) \) is a polynomial. As a consequence a set of difference equations can be used to calculate the position of every element to be read from the memory. The number of difference equations depends on the order of \( P_2(i) \), but the solution can become very simple.

Let us consider again the example that corresponds to equation 1b. Equation 3a now becomes:

\[ k \times [P_1(i)] \oplus P_2(i) = \text{pos}(i) = p_v + \sum_{k=1}^{i-1} k + i = p_v + (i^2 + i)/2 \]  

(3b)

This leads to the following difference equations, which can easily be executed on an ACU.

Initialisation:

\[ k \times [i+1] \oplus (i+1) = k \times [i] \oplus i + D(i) \]

\[ D(i+1) = D(i) + 1 \]

For \( i = 1, \ldots n \)

\[ k \times [i+1] \oplus (i+1) = k \times [i] \oplus i + D(i) \]

This shows that for practical examples such as the deinterleaving used in the Compact Disc application this method can lead to very simple but nevertheless efficient implementations. Figure 7a shows the corresponding architecture for the address calculation. An adder/subtractor with modulo arithmetic, and 2 register files containing 6 registers are sufficient.

(b) \( P_1(i) \) is not of the form \( i + a \)

In this case all the elements of \( \sum_{k=1}^{P_1(i)-1} Q(k) \) are known at compile time, as are all elements of \( P_2(i) \). These we can combine into one ROM table:

\[ Q(i) = \sum_{k=1}^{P_2(i)-1} Q(k) + P_2(i) \]

This data can then be read from the ROM, after which the calculation of equation 3a on the ACU is straightforward.

For \( i = 1, \ldots n \)

\[ Q(i) = \text{read(address_rom, } k \times Q(i) \text{)} \]

\[ \&c[P_1(i)] \oplus P_2(i) = Q(i) + p_v \]

\[ Q(i+1) = Q(i) + 1 \]

The architecture for this method is shown in figure 7b. It is basically the same as in figure 7a, with the addition of the address Rom. Where the polynomial \( P_2 \) is more complex than that used in 1b, the address calculation can in here be simpler than required in case (a), and also requires a smaller number of registers (4).
References


Figure 7: 2 possible architectures for address computation

5 Conclusions

Delay functions are essential in many DSP applications. Their efficient synthesis is a key element of a whole synthesis system. In this paper we have reported general, powerful synthesis techniques which start from a well-defined optimal memory organisation. The ideas have been incorporated in a tool which has been used for the design of real life VLSI systems such as a Compact Disc circuit. The results in terms of area and timing are comparable with manual designs.

6 Acknowledgements

This work is part of the CD90 project and supported by the EEC under Esprit project 2260 (Sprite). The authors are grateful to their colleagues at Philips Research Laboratories, and would like to thank Imec, Racal and Siemens as partners in the Sprite project. Thanks are also due to the reviewers, whose comments were invaluable in the preparation of this paper.