# Polynomial evaluation on multimedia processors *

Julio Villalba Gerardo Bandera Mario A. Gonzalez Javier Hormigo
Emilio L. Zapata
*Computer Architecture Dept., University of Malaga*
*{julio, bandera, mario, hormigo, ezapata}@ac.uma.es*

## Abstract

***In this paper we deal with the polynomial evaluation based on new processor architectures for multimedia applications. We introduce some algorithms to take advantage of the new attributes on multimedia processors, such as VLIW and SIMD architectures. Algorithms to support the polynomial evaluation based only in addition/shift operations and other different algorithms with MAC instructions are analyzed and tailored to subword parallelism units of the new processors. Both potential instruction–level and machine–level parallelism are fully exploited through concurrent use of all functional units.***

## 1: Introduction

Polynomial evaluation is the basis on the computation of a wide variety of mathematical functions. Digital Signal Processing uses this kind of functions in an intensive way. The inherent parallelism of the audio and video applications encourages processor architecture modifications to satisfy the increasing demand of multimedia applications. As a consequence, new techniques as subword parallelism have been introduced on general purpose processors in the last few years. They are based on the Single Instruction Multiple Data architecture (SIMD), and produces an extension of the processor basic set of instructions commonly called *Multimedia Extensions.* The current trend of the processor design is to promote multimedia capabilities by adding new multimedia instructions (i.e, extension to floating point arithmetic), increasing the complexity of the elementary multimedia operations (i.e, to make flexible the data movement), etc... The SIMD processor architecture has also reached to most of the Digital Signal Processors (DSP) like the TMS320C64 family of Texas Instrument[9].

On the other hand, architectures like Very Long Instruction Word (VLIW) or Explicit Parallel Instruction Computing (EPIC) have emerged to exploit the instruction–level parallelism. The last generation of Texas Instrument digital signal processors (the TMS320C6xxx family) [8] and the Intel Itanium Processor [3] (which includes multimedia extension) are some good examples of a high performance processors based on VLIW and EPIC architecture respectively. The combination of SIMD and VLIW architectures is found in the TMS320C64x family of Texas Instruments. To obtain an optimal performance from these architectures is necessary to use a qualified compiler able to extract parallel instructions from the source code.

Our paper presents a way to take advantage of these modern architectures to perform the evaluation of polynomials. Most of the current processors have both one or more Multiply-and-Add (MAC) units supporting DSP algorithms, and one or more units with addition/shift/logical capabilities. These units can work in parallel if the source code has enough parallelism and the compiler is able to extract it. The main goal of this paper is to achieve that most of the units were simultaneously working with the polynomial evaluations.

* This work was supported in part by the Ministry of Education and Culture (CICYT) of Spain under project TIC2000-1658.

The paper has been structured as follows: Section 2 includes some notes about state-of-the-art of the current processors with multimedia capability; Section 3 describes our algorithms to perform polynomial evaluation based on subword parallelism units; and finally, we evaluate our approaches and present the conclusions of this work.

## 2: Multimedia processors

In this section we give some notes about the current multimedia processors. We study the evolution and remark some relevant issues for both the algorithms and architectures proposed in the next section.

The first general–purpose microprocessor to include specific support for graphics was the Intel i860[4] in 1989. It provides six instructions along with their pipelined versions. After that, Motorola introduces on the 88110[14] (1991) a set of nine graphics instructions handling various data types tailored to graphics and imaging. It works with 4, 8, 16 and 32 fixed point values. The Hewlett-Packard's PA-7100LC[7] (1993) introduced five instructions that operated on two 16–bit components in parallel. The extension MAX-2 (Multimedia Acceleration eXtension) is implemented on HP PA-RISC 2.0[11]. The Visual Instruction Set (VIS) is the multimedia extension described by Sun Microsystems in 1994 and implemented in the UltraSparc I in 1995[12]. In 1996 Intel introduced the MMX technology (Matrix Math eXtension)[17] on the Pentium processors. The MVI (Motion Video Instructions) is the multimedia extension of the DEC Alpha processors[1]. For the Motorola Power PC, the new instruction set is called AltiVec[15]. 3DNow![10] is the extension of the AMD to support multimedia applications, and it is compatible with MMX. The basic idea behind the multimedia extension is performing parallel computation of subwords by following a SIMD architecture.

The evolution of the multimedia extension of the processor leads to increase the number of new multimedia instructions (including support for floating–point arithmetic) and to make a more flexible data movement within the functional units that support subword parallelism (from this point on, let us call *subword parallelism units* to this kind of functional units). An example of that is the evolution of the MMX extension of Intel to SSE (*S*treaming *S*IMD *E*xtension), and then to SSE2 [5]. These last two extensions include 70 and 144 new instructions respectively. Most of them are introduced to support floating point arithmetic. Other instructions like *PSHUFW* allow movements among subwords. AltiVec has a very powerful instruction to perform subword permutations: *VPERM*. Another trend that we can observe in bibliography is the increasing of the length of registers supporting the SIMD operations: from 32 bits in the first generation processors (i.e. extension MAX-1 of HP) to 128 bits for the last generations (i.e, AltiVec of Motorola and SSE of Intel). Some authors call *vector* to these large registers, and we follow this convention in the rest of the paper.

On the other hand, both EPIC and VLIW architectures have emerged to exploit the program instruction–level of parallelism. For example, the TMS320C6000 processor family of Texas Instruments are based on VLIW architecture with eight functional units, including two multipliers and six arithmetic logic units [8]. The CPU can execute up to eight instructions per cycle and all the instructions can operate conditionally. The newest member of this family, the 'C64x [9], supports 16-bits subword parallelism in all the functional units, and 8-bits subword parallelism in six units, including both multipliers. In the Intel architecture IA-64, instructions come in groups of three, called *bundles* [2]. Each 128 bits bundle contains three 40–bit fixed-format instructions and an 8–bit template. This scheme, plus the existence of many registers, allows the compiler to both isolate blocks of instructions, and to inform to the CPU which of them can be executed in parallel 1. The predication technique allows the instructions to operate conditionally. Furthermore, Intel adds the aforementioned SSE2 extension to their last generation processor (Itanium).

## 3: Polynomial evaluation on SIMD & VLIW architectures

In this section we describe different algorithms to perform the complete evaluation of a polynomial based on MAC units or addition/shift units. The first algorithm allows to implement the polynomial evaluation using only *addition/shift* units and the second one is only based on *MAC* units. In both cases, the units must support subword parallelism (SIMD architecture).

### 3.1: Polynomial evaluation based on addition/shift units

In [13] a specific architecture to support polynomial evaluation based on shift and add operations is presented. The authors propose a parallel algorithm for fixed–point signed–digit arithmetic based on the Taylor's series expansion of the polynomial function. In this section we adapt the algorithm to the architecture of fixed–point subword parallelism units and non–redundant arithmetic. Let $x$ be the point where the polynomial is evaluated ($-1 \leq x < 1$). Assume that $\sigma_i$ is binary digit ($\sigma_i = 0, 1$) of $x$ at the weight $2^{-i}$, that is:

$$x = (-1)^{\sigma_0} + \sum_{i=1}^{m} \sigma_i 2^{-i} \tag{1}$$

(finite precision of $m$ fractional bits is considered). Assume that $F$ is a degree-$n$ polynomial:

$$F(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \tag{2}$$

and let us denote $F_k^{(i)}$ the value of the $i^{th}$ derivative of $F$ at point $x_k$, where $x_k = (-1)^{\sigma_0} + \sum_{r=1}^{k} \sigma_r 2^{-r}$. From Taylor's expansion of the derivatives of $F$ we deduce:

$$F_{k+1}^{(i)} = \sum_{j=0}^{n-i} \sigma_{k+1} 2^{-j(k+1)} \frac{F_k^{(i+j)}}{j!} \tag{3}$$

To reduce the number of variable shifters involved in the previous expression we define $P_k^{(i)} = 2^{-ik} \frac{F_k^{(i)}}{i!}$ as referred to [13]. Therefore:

$$P_{k+1}^{(i)} = \sum_{j=0}^{n-i} \sigma_{k+1} 2^{-i-j} \frac{(i+j)!}{i!j!} P_k^{(i+j)} \tag{4}$$

Notice that $P_k^{(0)} = F_k^{(0)}$ and the polynomial to be evaluated ($F(x)$) corresponds to the value of $F_m^{(0)}$. Expression (4) defines several recurrences that allow to obtain $P_{k+1}^{(0)}$ from the values of $P_k^{(0)}, P_k^{(1)}, \ldots, P_k^{(n)}$, $P_{k+1}^{(1)}$ from the values of $P_k^{(1)}, P_k^{(2)}, \ldots, P_k^{(n)}$, etc. Furthermore, the computation of $P_{k+1}^{(0)}, P_{k+1}^{(1)} \ldots$ can be performed in parallel. We transform these expressions in such a way that they are carried out following a SIMD programming model.

The current vector length used in subword parallelism units is up to 128 bits. Without any loss of generality we describe the processing of a degree-4 polynomial based on 128–bit units. The recurrence based on expression (4) for this system is

$$\begin{array}{lcccccccccc}
P_{k+1}^{(0)} & = & P_k^{(0)} & + & \sigma_{k+1}\frac{1}{2}P_k^{(1)} & + & \sigma_{k+1}\frac{1}{4}P_k^{(2)} & + & \sigma_{k+1}\frac{1}{8}P_k^{(3)} & + & \sigma_{k+1}\frac{1}{16}P_k^{(4)} \\
P_{k+1}^{(1)} & = & & & \frac{1}{2}P_k^{(1)} & + & \sigma_{k+1}\frac{1}{2}P_k^{(2)} & + & \sigma_{k+1}\frac{3}{8}P_k^{(3)} & + & \sigma_{k+1}\frac{1}{4}P_k^{(4)} \\
P_{k+1}^{(2)} & = & & & & & \frac{1}{4}P_k^{(2)} & + & \sigma_{k+1}\frac{3}{8}P_k^{(3)} & + & \sigma_{k+1}\frac{3}{8}P_k^{(4)} \\
P_{k+1}^{(3)} & = & & & & & & & \sigma_{k+1}\frac{1}{8}P_k^{(3)} & + & \sigma_{k+1}\frac{1}{4}P_k^{(4)} \\
P_{k+1}^{(4)} & = & & & & & & & & & \frac{1}{16}P_k^{(4)}
\end{array} \tag{5}$$

All these equations can be performed in parallel. Moreover, they are simultaneously carried out based on functional units with subword parallelism capability. At first, for $m$-bit precision, $m$ iterations are required. Let us study the $\sigma_{k+1} = 0$ and $\sigma_{k+1} = 1$ values separately:

- $\underline{\sigma_{k+1}} = 0$. In this case, the equations (5) become:

$$\begin{aligned} P^{(0)}_{k+1} &= P^{(0)}_k \\ P^{(1)}_{k+1} &= \tfrac{1}{2}P^{(1)}_k \\ P^{(2)}_{k+1} &= \tfrac{1}{4}P^{(2)}_k \\ P^{(3)}_{k+1} &= \tfrac{1}{8}P^{(3)}_k \\ P^{(4)}_{k+1} &= \tfrac{1}{16}P^{(4)}_k \end{aligned} \tag{6}$$

  The only operations involved in these equations are shifts. It is possible to perform these operations in one cycle if we use a subword parallelism unit which is able to manage a different shifting amount on each subword.
  If two or more consecutive digits "0" are found in the value of $x$, it is possible to save iterations by only performing one parallel shift instruction. The number of consecutive digit "0" must be taken into account to select the suitable shift on each subword for a unique parallel shift instruction (i.e. for two consecutive digit "0" ($\sigma_{k+1} = \sigma_{k+2} = 0$) the equations (6) become $P^{(0)}_{k+2} = P^{(0)}_k, P^{(1)}_{k+2} = \frac{1}{4}P^{(1)}_k, P^{(2)}_{k+2} = \frac{1}{8}P^{(2)}_k, P^{(3)}_{k+2} = \frac{1}{16}P^{(3)}_k, P^{(4)}_{k+2} = \frac{1}{32}P^{(4)}_k$ which is performed by only one instruction).

- $\underline{\sigma_{k+1} = 1}$. Now equations (5) involve shift, addition and multiplication operations. To avoid the multiplication by 3 of some terms of the second and third equations we decompose this factor in a sum of two terms which only involve shifts. Hence, we can rewrite these equations as follows:

$$\begin{aligned} P^{(0)}_{k+1} &= P^{(0)}_k + \tfrac{1}{2}P^{(1)}_k + \tfrac{1}{4}P^{(2)}_k + \tfrac{1}{8}P^{(3)}_k + \tfrac{1}{16}P^{(4)}_k \\ P^{(1)}_{k+1} &= \tfrac{1}{2}P^{(1)}_k + \tfrac{1}{2}P^{(2)}_k + \tfrac{1}{8}P^{(3)}_k + \tfrac{1}{4}P^{(3)}_k + \tfrac{1}{4}P^{(4)}_k \\ P^{(2)}_{k+1} &= \tfrac{1}{4}P^{(2)}_k + \tfrac{1}{8}P^{(3)}_k + \tfrac{1}{4}P^{(3)}_k + \tfrac{1}{8}P^{(4)}_k + \tfrac{1}{4}P^{(4)}_k \\ P^{(3)}_{k+1} &= \tfrac{1}{8}P^{(3)}_k + \tfrac{1}{4}P^{(4)}_k \\ P^{(4)}_{k+1} &= + \tfrac{1}{16}P^{(4)}_k \end{aligned} \tag{7}$$

  Now, only addition and shift operations are required. Notice that the maximum number of additions per equation is not increased from equations (5) to (7), which means that worst case time is not modified.

To carry out equations (5) on a 128–bit subword parallelism unit we consider subword of 16 bits. Therefore, 8 subwords are contained on each vector. Figure 1 shows the data flow and the operations required. At the beginning of each iteration *vector A* supports the values of $P^{(0)}_k, P^{(1)}_k, P^{(2)}_k, P^{(3)}_k, P^{(4)}_k$ distributed as shown. The first operation to be performed is a variable shift of every subword (step 1). If a sequence of consecutive digit "0" is found before the digit $\sigma_k = 1$, the number of bits to be shifted must be increased by 1 for every 0 of the sequence. In this way, the step 1 of figure 1 also includes the successive computations of equations (6) corresponding to each preceding digit "0".

Steps 2 to 4 on figure 1 require some parallel additions and permutations of the data contained on each subword. The different datapaths have been selected in such a way that the different terms of equations (7) are calculated and kept inside the same 128–bit *vector A*. Therefore, after three additions we have computed equations (7) and one iteration will be completed.

Since the computation of consecutive digit "0" are included in the first step of the computation of the next digit "1", the total number of iterations coincides with the number of digit "1" of $x$. Consequently, if we consider the same number of "1's" and "0's" on $x$, then the mean number of iterations is reduced by a half.

The operation involved on each step of the proposed algorithm can not be carried out in a single parallel instruction on current processors, in spite of keeping the SIMD programming schedule. The operation implied in the step 1 requires that the number of bits to be shifted in every subword was different. The addition and data movement of steps 2 to 4 require several parallel instructions since data movement plus addition is not directly supported. Nevertheless, the data movement

$$P_{k+1}^{(0)} = \underset{(1)}{P_k^{(0)}} + \underset{(2)}{\frac{1}{2} P_k^{(1)}} + \underset{(3)}{\frac{1}{4} P_k^{(2)}} + \underset{(4)}{\frac{1}{8} P_k^{(3)}} + \underset{(5)}{\frac{1}{16} P_k^{(4)}}$$

$$P_{k+1}^{(1)} = \underset{(6)}{\frac{1}{2} P_k^{(1)}} + \underset{(7)}{\frac{1}{2} P_k^{(2)}} + \underset{(8)}{\frac{1}{8} P_k^{(3)}} + \underset{(9)}{\frac{1}{4} P_k^{(3)}} + \underset{(10)}{\frac{1}{4} P_k^{(4)}}$$

$$P_{k+1}^{(2)} = \underset{(11)}{\frac{1}{4} P_k^{(2)}} + \underset{(8)}{\frac{1}{8} P_k^{(3)}} + \underset{(9)}{\frac{1}{4} P_k^{(3)}} + \underset{(14)}{\frac{1}{4} P_k} + \underset{(15)}{\frac{1}{8} P_k^{(4)}}$$

$$P_{k+1}^{(3)} = \underset{(16)}{\frac{1}{8} P_k^{(3)}} + \underset{(17)}{\frac{1}{4} P_k^{(4)}}$$

$$P_{k+1}^{(4)} = \underset{(18)}{\frac{1}{16} P_k^{(4)}}$$

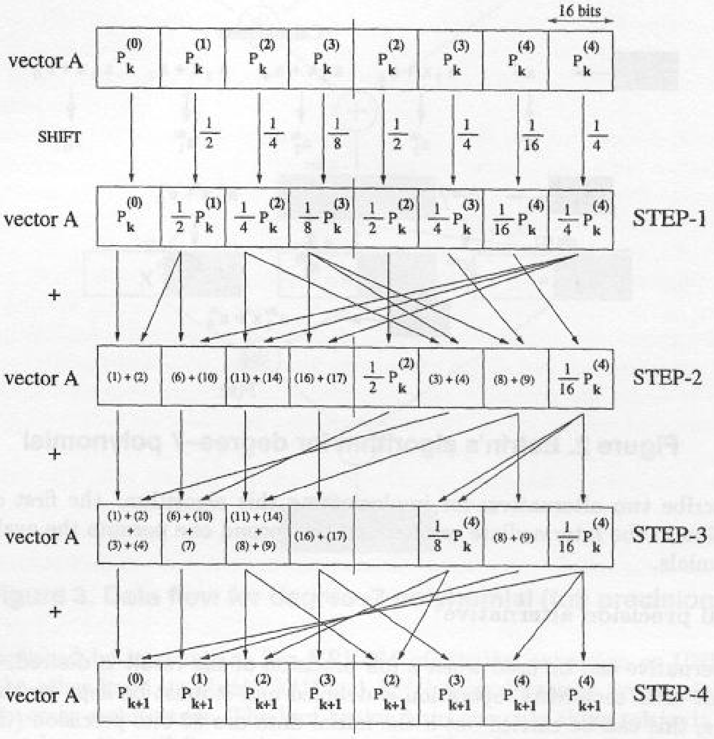

**Figure 1. Data flow for degree–4 polynomial**

flexibility of some of the current multimedia extension like AltiVec (i.e. *VPERM* instruction) makes feasible that the next generation of multimedia extension of the general and digital signal processors can support the aforementioned operations in an unique parallel instruction (one step, one parallel instruction).

If we want to make a complete parallel control over the algorithm, an additional unit must be able to perform the detection of the next "1" of $x$ jointly with the execution of the algorithm. In this way, the number of bits to be shifted on the next iteration is calculated and it is ready to be used at the beginning of the following iteration. This can be easily accomplished by both the VLIW or EPIC architectures, which allow the execution of several instructions in parallel. On the other hand, the predication of instructions of the current processors generation help to reduce, or even to eliminate, the overhead involved in the iterative nature of the proposed algorithm.

### 3.2: Polynomial evaluation based on MAC units

A second well-known approach to evaluate polynomials is the Estrin's algorithm[16]. It allows to compute a polynomial based on parallel multiplications and accumulations. This approach has been used to make hardwired polynomial evaluation, as referred to [6]. In this subsection we analyze the application of this algorithm to processors containing subword parallelism MAC units.

Without any loss of generality we describe the Estrin's algorithm to compute a degree–7 polynomial ($F(x) = a_0 + a_1x + ... + a_7x^7$) and present two implementation alternatives based on a 128 bits subword parallelism MAC unit. Figure 2 shows the three steps of the Estrin's algorithm for this case. As we can observe, for the first step four parallel MAC operations plus an extra multiplication are needed to compute $x^2$. On the step–2, only two MAC operations and an extra multiplication are involved. And finally, only a single MAC operation is required for the last step of the algorithm.

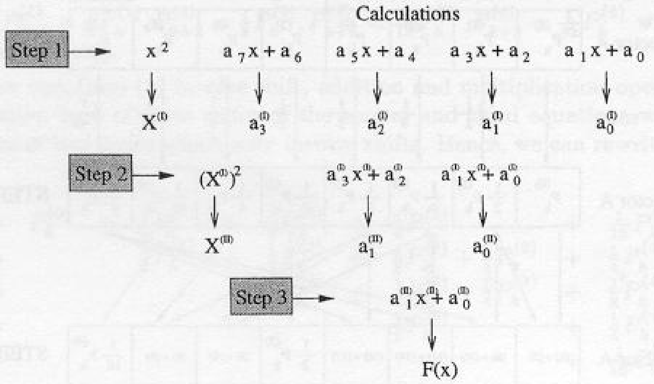

**Figure 2. Estrin's algorithm for degree–7 polynomial**

Now we describe two alternatives for implementing this algorithm: the first one allows us to keep full precision in the intermediate results, and the second one permits the evaluation of higher degrees polynomials.

#### 3.2.1: The full precision alternative

The first alternative can be used when a full precision of the result is desired. In this case, the size of the result after each MAC operation is doubled and it must be kept for the next step. For 128-bits vectors, this can be carried out if the initial data are 16 bits precision (that is, variable $x$ and coefficients $a_i$) and it is embedded into a 32 bits subword. This is emphasized in Figure 3 by shading the right half of the subword. Also, this figure shows the data flow for this alternative. We use three 128–bits vectors to carry out every step of the Estrin's algorithm. The first step of figure 2 is implemented with one parallel-MAC instruction (four 32 bits subwords). After this operation, the four 32 bits results are moved and expanded onto two new 128 bits vectors, as shown in figure 3. The second step of the algorithm works on 64 bits subword and the last step works over the full vector length. On the other hand, the computation of $X^{(I)} = x^2$ can be performed in a separate unit and copied to a 128 bit vector before the starting of the second step. A similar solution is required for the $X^{(II)}$ calculation.

To carry out the full precision alternative several instructions are required. The parallel-MAC instruction of AltiVec extension support three operand as input, just like required by the proposed design. The movement and expansion of the intermediate results (i.e. from 32 bits to 64 bits) can

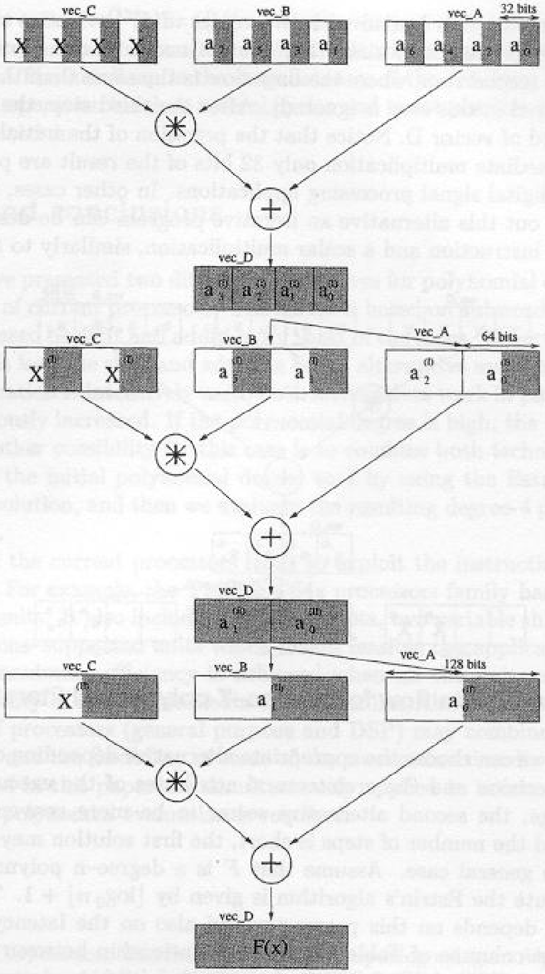

**Figure 3. Data flow for degree–7 polynomial (full precision).**

be easily accomplished by instructions like VPERM of AltiVec extension or UNPACK of MMX extension. On the other hand, to obtain $X^{(I)}$ and $X^{(II)}$ several options are possible depending on the complete processor architecture. The best option is to compute these values in a scalar unit in parallel with the subword parallelism unit, in such a way that the required values were ready before the next step. For a degree–7 polynomial this implies that scalar units will support operands with a 32–bits precision, and produce 64–bits results. In an Explicit Parallelism or VLIW architecture this can be performed in parallel under user control. In a superscalar processor, it is likely but not certain. In the worst case, the computation of $X^{(I)}$ and $X^{(II)}$ can be performed sequentially, but it supposes an important overhead.

### 3.2.2: High polynomial degree

The second alternative allows the use of a higher polynomial degree, but the full precision of intermediate results are not maintained. Figure 4 show the data flow for the first step of the Estrin's algorithm. The coefficients $a_0, a_1, ..., a_7$ and variable $x$ are distributed on three vectors as shown in figure 4. After the first parallel-MAC instruction, the four results are properly moved to vectors A and B, while vector C is loaded with two copies of $X^{(I)}$. Notice that the two most left subwords of vector A and B are empty since they are not needed for the next iteration. Now, we are ready to perform the second step, where the data flow is the same than the first step (the result of the subword not involved in this step is ignored). After the third step, the final result is located at the most-right subword of vector D. Notice that the precision of the initial data can be up to 32 bits, but after the intermediate multiplication only 32 bits of the result are produced. It is usually enough for most of the digital signal processing applications. In other cases, some guard bits must be considered. To carry out this alternative an iterative program can be designed. Each iteration requires a parallel-MAC instruction and a scalar multiplication, similarly to the first alternative.

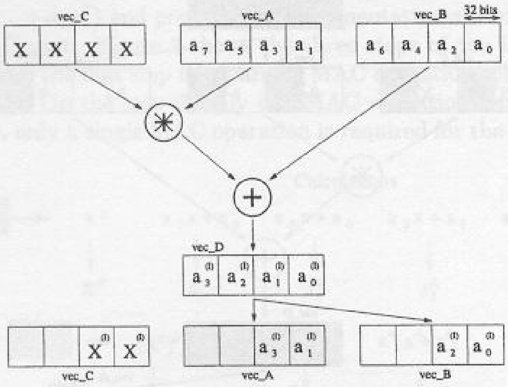

**Figure 4. Data flow for degree–7 polynomial (iterative)**

For the general case, we can choose the appropriate alternative depending on both the polynomial degree, the required precision and the architectural attributes of the executing processor. If the number of steps is large, the second alternative seems to be more convenient. However, if full precision is required and the number of steps is short, the first solution may be a good choice.

Let us return to the general case. Assume that $F$ is a degree–n polynomial. The number of steps required to compute the Estrin's algorithm is given by $\lfloor \log_2 n \rfloor + 1$. The total computation time for the algorithm depends on this parameter and also on the latency of the parallel MAC operation. The first two columns of Table 1 show the relationship between the polynomial degree and the number of steps of the algorithm. We show the results from degree–2 to 64 polynomials. The number of subwords required for the first step of the algorithm fixes the vector length of the system for the required precision for the data (subword length). This number is given by $\lfloor \frac{n}{2} \rfloor + 1$. Table 1 also shows the number of subwords needed for the first step and the vector length to keep subword precision of 16 and 32 bits. The parameters of the degree–7 polynomial used in the example of figure 4 has been remarked.

| Polynomial degree | Num. steps | Num. subwords (first step) | Vector length subword | |
|---|---|---|---|---|
| | | | 16 bits | 32 bits |
| 2,3 | 2 | 2 | 32 | 64 |
| 4-7 | 3 | 3-4 | 64 | 128 |
| 8-15 | 4 | 5-8 | 128 | 256 |
| 16-31 | 5 | 9-16 | 256 | 512 |
| 32-64 | 6 | 17-32 | 512 | 1024 |

Table 1. Relationship between polynomial degree and precision

Other solution is given by the implementation of the Horner's scheme [16]. For example, for degree–7 polynomial we compute $F(x) = (((((a_7x+a_6)x+a_5)x+a_4)x+a_3)x+a_2)a_1+x_0$. This can be accomplished by using a subword parallelism MAC unit where each subword supports a different polynomial evaluation. The handicap is the large latency of the resulting system. This is due to the fact that the MAC operation involves several cycles and is not possible to take advantage of the potential pipeline architecture.

## 4: Evaluation and conclusions

In this paper we have presented two different alternatives for polynomial evaluation based on the multimedia extensions of current processors. The solution based on subword parallelism MAC units is faster than the one based on shift and addition for most of the cases. Nevertheless, if the number of digit "1" of variable $x$ is low, the shift and addition based alternative may be better. In applications where polynomial evaluation is intensively used, both alternatives work in parallel, and the efficiency of the system is notoriously increased. If the polynomial degree is high, the MAC alternative seems more appropriate. Another possibility for this case is to combine both techniques in a pipeline way. To do this, we reduce the initial polynomial degree to 4 by using the Estrin's algorithm and the proposed MAC based solution, and then we evaluate the resulting degree-4 polynomial by using our addition/shift solution.

The development of the current processors leads to exploit the instruction–level and data–level (subword) parallelism. For example, the TMS320C64x processors family has a subword parallelism in his eight functional units. It also includes two MAC units, two variable shift–supported units and two arithmetic operations–supported units which makes feasible the application of both techniques simultaneously. The maximum efficiency is achieved when all the units are working in parallel. Nevertheless, the flexibility of other multimedia extensions like AltiVec is not achieved by this DSP. The next generation of processors (general purpose and DSP) may combine the parallel capability of the VLIW processor and the flexibility of some of the current multimedia extension. In this case, the algorithms proposed in this paper are simultaneously applied to maximize the efficiency of the system when intensive polynomial evaluation is required.

## References

[1] Digital Equipment Corporation. The future of Visual Computing using an Alpha Microprocessor: New Motion Video Instructions. *Mariland, Massachussets, EC-R4GDA*, 1997.

[2] Intel Corporation. *IA-64 Application Developer's Architecture Guide.*

[3] Intel Corporation. *Intel(R) Itanium(TM) Processor Hardware Developer's Manual.*

[4] Intel Corporation. i860 64-bit microprocessor. *Data Sheet,Santa Clara*, 1989.

[5] Intel Corporation. Intel pentium 4 processor optimization reference manual. 2000.

[6] J. Duprat and J. M. Muller. Hardwired polynomial evaluation. *Journal of Parallel and Distributed Computing*, Special Issue on Parallelism in Computer Arithmetic(5), 1988.

[7] P. Knelbel et al. Hp pa 7100lc: A low-cost superscalar pa-risc processor. *Proc. Compcon, IEEE CS Press*, pages 441–447, 1993.

[8] Texas Instrument Incorporated. *TMS320C6000 CPU and Instruction Set Reference Guide".* Tarrant Dallas Printing, Inc., 1997.

[9] Texas Instrument Incorporated. *TMS320C64x Technical Overview.* Tarrant Dallas Printing, Inc., 2000.

[10] Howard Kalish and Jerry Isaac. The AMD K6-3D Processor: Revolucionary Multimedia Performance. 19.

[11] Rubby B. Lee. Subword Paralelism with MAX-2. *IEEE Micro*, 16:51–59, August 1996.

[12] V. Narayanan M. Trembaly, J.M. O'Connor and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, pages 10–20, August 1996.

[13] X. Merrheim, J. M. Muller, and H. J. Yeh. Fast evaluation of polynomials and inverses of polynomials. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 186–192, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[14] Motorola. Mc88110 second generation risc microprocessor user's manual. *Austin, Texas*, 1991.

[15] Motorola. AltiVec Technology programing Enviroments Manual. 1998.

[16] J.M. Muller. *Elementary Functions, Algorithms and Implementation.* Birkhauser, Boston, 1997.

[17] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, August 1996.