Arbitrary Precision Arithmetic – SIMD Style *

S. Balakrishnan and S. K. Nandy
Supercomputer Education and Research Center
Indian Institute of Science
Bangalore 560 012
India.
e-mail: {sbalki,nandy}serc.iisc.ernet.in

Abstract

Current day general purpose processors have been enhanced with what is called "media instruction set" to achieve performance gains in applications that are media processing intensive. The instruction set that have been added exploit the fact that media applications have small native datatypes and have widths much less than that supported by commercial processors and the plethora of data-parallelism in such applications. Current processors enhanced with the "media instruction set" support arithmetic on sub-datatypes of only 8-bit, 16-bit, 32-bit and 64-bit precision. In this paper we motivate the need for arbitrary precision packed arithmetic wherein the width of the sub-datatypes are programmable by the user and propose an implementation for arithmetic on such packed datatypes. The proposed scheme has marginal hardware overhead over conventional implementations of arithmetic on processors incorporating a multimedia extended instruction set.

1 Introduction

The current trend of incorporating special instructions for multimedia applications in general purpose processors has been motivated by the fact that most media applications have several common characteristics:

- Small native data types.
- Repeated compute intensive operations on these data types.
- High data-parallelism.

These properties along with the fact that technology is sufficiently mature to support 64-bit internal bus widths and ALUs, indicate that a possible way of enhancing the performance of general purpose processors is by exploiting SIMD parallelism. Thus, for example, the Intel MMX [1], the VIS instruction set of SPARC [2] and the MAX-2 instruction set of PA-RISC 2.0 [3] have instructions that can operate on eight 8-bit, four 16-bit, or two 32-bit partitioned components of 64-bit operands in parallel. This has resulted in a performance gain of 50% – 100% over processors that do not support SIMD style arithmetic.

One interesting observation regarding the choice of width of the partitioned operands is that they have been chosen more from the point of view of convenience of implementation than from the size of data types used in actual implementation of applications. For example, while most pixel oriented data is 8-bits in length it is a fact that some applications like medical imaging use 12-bit pixels. Similarly IDCT values in MPEG are 12-bits [4]. Media processors also need a special 9-bit data type for performing DCT for compression and decompression. Several such instances where there is a need of native data types that are different from the conventional "power-of-two" boundaries supported by architectures implementing the Intel MMX or the Ultra Sparc's VIS instructions have been tabulated in [5]. The table has been reproduced below for convenience (Table 1). It is also interesting to note that MICROUNITY's media processor architecture [6] supports arithmetic operations on $2 \times 64, 4 \times 32, 8 \times 16, 16 \times 8, 32 \times 4, 64 \times 2$ *or* 128×1 partitions of 128-bit registers.

The speedups achieved with media applications programmed with instructions that manipulate data with sizes equal to that of the native data types of media applications have been the emphasis of current media processing technologies and the advantages of it cannot be gainsaid. Thus, if the instruction set has a provision for manipulating 12-bit packed data types in a 64-bit register, five pixel values can be ma-

*This research was supported in part by the Department of Electronics, Government of India, under sponsored project DE-NMC/SP-054, and the TUD-IISc. collaboration project between Technical University, Delft, The Netherlands and The Indian Institute of Science.

Media Task	Natural data Type
MPEG	8-bit/9-bit fixed point
Fax/Modems	16-bit fixed point
3D Graphics	24-bit/32-bit floating point
Audio	16-bit/20-bit fixed point

Table 1: Native data Types of Media Applications

nipulated simultaneously compared to four that can be manipulated in a conventional MMX type technology enhanced processor. Another argument for using operand widths equal to that of a native data type of an applications is that there is power efficiency in terms of the total work done. Thus for example an application programmed using an architecture that uses 12-bit data types packed in a 64-bit register can be algorithmically designed to be 25% more power efficient than an architecture that defines only 16-bit packed arithmetic operations.

Having motivated the need for architectures with instruction sets that manipulate data types that correspond closely to that of the native data types manipulated by the application, in this paper we propose a fast algorithm based on the carry lookahead scheme for SIMD fashion addition of user defined segments of two registers.

The rest of the paper is organized as follows. In section 2 we establish the theoretical basis of an arbitrary precision carry-lookahead scheme. Though most of the section reproduces the results given in [10] the last part of the section applies these results to establish the correctness of our scheme for addition. In section 3 we discuss the carry-lookahead scheme and propose the idea of user defined precision for addition. We also propose a scheme for arithmetic on user defined, application specific packed data types. In section 3 we compare our idea with those proposed in literature. In section 4 we summarize the results.

2 Addition as a Digit-Set Conversion Problem

We herein describe the basic notations that are essential in understanding the digit set conversion problem for a fixed radix and its application to addition. The notation used is the same as that introduced by [10].

A radix β polynomial is an expression of the form:

$$P = \sum_{i=\ell}^{m} d_i[\beta]^i,$$

where the digits $d_i \in \mathbb{Z}$, belonging to a digit set D such that D is finite and $0 \in D$. The integer radix β is such that $\mid \beta \mid \geq 2$. The set of radix β polynomials over a digit set D is represented by $\mathcal{P}[\beta, D]$:

$$\mathcal{P}[\beta, D] = \left\{ P = \sum_{i=\ell}^{m} d_i[\beta]^i \;\middle|\; d_i \in D \right\}.$$

The set of integer radix polynomials are represented by:

$$\mathcal{P}_I[\beta, D] = \left\{ P = \sum_{i=\ell}^{m} d_i[\beta]^i \;\middle|\; P \in \mathcal{P}[\beta, \mathcal{D}], \prime \leq \ell \right\}.$$

The set of polynomials representing a given integer value i is given by:

$$\nu_{\beta,D}(i) = \{P \in \mathcal{P}_I[\beta, D] \mid \|P\| = i\}$$

where $\|P\|$ is the real value of the polynomial P given by

$$\|P\| = \sum_{i=\ell}^{m} d_i \beta^i.$$

A digit set D is :

$$\begin{array}{rl} \text{complete for radix } \beta & \text{iff } \forall i \in \mathbb{Z} : \mid \nu_{\beta,D}(i) \mid \geq 1, \\ \text{redundant for radix } \beta & \text{iff } \exists i \in \mathbb{Z} : \mid \nu_{\beta,D}(i) \mid > 1, \\ \text{non-redundant for radix } \beta & \text{iff } \forall i \in \mathbb{Z} : \mid \nu_{\beta,D}(i) \mid \leq 1, \end{array}$$

where the cardinality of a set S is given by $\mid S \mid$.

Let D be a complete digit set and E be a non-redundant and complete digit set for radix β, $D \neq E$. Given a $P \in \mathcal{P}[\beta, D], P = \sum_{\ell}^{m} d_i[\beta]^i, d_i \in D$, it has been shown in [10] that we can find a $Q \in \mathcal{P}[\beta, E]$ such that $\|P\| = \|Q\|$. For this conversion, any digit $d \in D$ is rewritten uniquely as $d = c\beta + e$, where $e \in E$ and $c \in C$, a *carry* set. An incoming carry has to be, in general, added before conversion. Thus one can define the conversion as a mapping

$$\alpha : C \times D \rightarrow C \times E$$

where the set C is such that for all $(c', d) \in C \times D$ there exists a $(c, e) \in C \times E$ such that

$$c' + d = c\beta + e.$$

In [10] the addition of two binary numbers has been posed as a digit set conversion problem for a fixed radix. Here the conversion is from a redundant digit set of base 2 with the digit set {0, 1, 2} to a non-redundant digit set of the same base and the digit set comprising of the digits {0, 1}. Here the two addends are interpreted using the *"carry-save encoding"*:

$$\begin{array}{lcl} 0 & \sim & 00 \\ 1 & \sim & 01 \text{ or } 10 \\ 2 & \sim & 11 \end{array}$$

The pairs of digits on the right denote the two addend digits at any bit position.

The conversion mapping is then given by the following table:

		D		
	α	0	1	2
C	0	00	01	10
	1	01	10	11

where each entry is of the form ce representing a pair $(c, e) \in C \times E$. One can split the table shown above into two parts describing two sets of functions. One, the *carry-transfer* set of functions γ_d that describe the mapping of an incoming carry c' into an outgoing carry c when a digit d is being converted and the other defining the digit-mapping set of functions ξ_d that maps an incoming carry to a digit $e \in E$ when a digit $d \in D$ is being converted. The split tables giving $\{\gamma_d\}$ and $\{\xi_d\}$are show below.

		γ_0	γ_1	γ_2
C	0	0	0	1
	1	0	1	1

		ξ_0	ξ_1	ξ_2
C	0	0	1	0
	1	1	0	1

The following points are worth noting: γ_0 kills an incoming carry, γ_1 propagates an incoming carry and γ_2 generates a carry irrespective of the incoming carry.

If we have two carry-transfer functions:

$$\gamma' = \left\{ \begin{array}{l} a' \\ b' \end{array} \right\} \text{ and } \gamma'' = \left\{ \begin{array}{l} a'' \\ b'' \end{array} \right\}$$

the composition of these two functions after simplification is given by:

$$\gamma = \gamma' \circ \gamma'' = \left\{ \begin{array}{l} a' + b'a'' \\ a' + b'b'' \end{array} \right\}$$

since functional composition is associative, one can use the *parallel prefix computation* techniques to compute the composition and hence compute the c_is at any bit position as $c_i = \gamma_{d_i d_{i-1} \cdots d_l}(0)$. This can then be used to compute the final sum digits (using the ξ_ds) as $s_i = c_i \oplus x_i \oplus y_i$. The composition function describes precisely the nature of computation at each cell of a *Conditional Sum Adder*. The composition function described is different from that described by [11] where the composition is carried out according to the rules given in equations 7 and 8 of section 3. This is because it is common practice to associate the function $\left\{ \begin{array}{l} 1 \\ 0 \end{array} \right\}$ with a carry generate, while it has been defined here as $\left\{ \begin{array}{l} 1 \\ 1 \end{array} \right\}$. However, the time complexity of the circuit used to implement the logic shown here is identical to that given by the equations.

We now investigate on what segmentation means in terms of the digit set conversion problem.

The problem can still be posed as a digit set conversion problem with the conversion being from a redundant digit set of radix 2 and digit set {0, 1, 2, 3, 4, 5} to a non-redundant digit set of the same base and the digit set comprising of the digits {0, 1}. Here the two addends are interpreted using an extension of the *"carry save encoding"*:

$$\begin{array}{lcl} 0 & \sim & 00 \\ 1 & \sim & 01 \text{ or } 10 \\ 2 & \sim & 11 \\ 3 & \sim & 0 \,|\, 0 \,| \\ 4 & \sim & 0 \,|\, 1 \,| \text{ or } 1 \,|\, 0 \,| \\ 5 & \sim & 1 \,|\, 1 \,| \end{array}$$

The '|' sign is used to denote a *"boundary"* and the pairs of digits, the addend digits at any bit position.

The conversion mapping table is given below:

		D					
	α	0	1	2	3	4	5
C	0	00	01	10	00	01	10
	1	01	10	11	00	01	10

from which we can derive the $\{\gamma_d\}$ and $\{\xi_d\}$ sets:

		γ_0	γ_1	γ_2	γ_3	γ_4	γ_5
C	0	0	0	1	0	0	1
	1	0	1	1	0	0	1

		ξ_0	ξ_1	ξ_2	ξ_3	ξ_4	ξ_5
C	0	0	1	0	0	1	0
	1	1	0	1	0	1	0

We note that the γ_3 and the γ_4 functions act exactly like γ_0 and the γ_5 function is identical to the γ_2 function. This observation hints that no extra hardware over that of a normal carry lookahead scheme, might be needed to realize the composition functions in the parallel prefix tree.

3 Arbitrary Boundary Packed Addition

In this section we revisit the carry-lookahead (CLA) scheme and provide a method to perform arbitrary boundary packed CLA.

3.1 Carry-Lookahead Addition

Let $A_{n-1}A_{n-2}\ldots A_0$ and $B_{n-1}B_{n-2}\ldots B_0$ be two n-bit binary numbers with a sum bits $S_{n-1}S_{n-2}\ldots S_0$ and carry bits $C_nC_{n-1}\ldots C_0$. Here the index 0 refers to the least significant bit of the numbers.

The sum bits S_is are computed as follows in the carry-lookahead scheme:

$$k_i = \overline{A_i}.\overline{B_i} \ \ (carry\ kill) \tag{1}$$

$$p_i = A_i \oplus B_i \ \ (carry\ propagate) \tag{2}$$

$$g_i = A_i.B_i \ \ (carry\ generate) \tag{3}$$

$$C_i = g_i + p_i.C_{i-1},\ C_0 = 0. \tag{4}$$

$$S_i = A_i \oplus B_i \oplus C_i,\ \ i = 1, \cdots, n. \tag{5}$$

While, ideally the whole carry-lookahead hardware to compute the C_is can be constructed, it is impractical to do so because of limitations in fan-in, fan-out, irregularity in structure and the inordinately long wires that might be necessary. To ameliorate the problems mentioned above one can implement the carry-lookahead scheme in a tree like structure. Such an implementation gives a regular structure and is amenable to efficient implementation [7]. The carry-lookahead tree can be implemented by defining block-kill, block-propagate and block-generate expressions for a block of the addends as follows:

$$K_{l,n} = K_{l,m} + P_{l,m}.K_{m-1,n} \ \ (block\ carry\ kill) \tag{6}$$

$$P_{l,n} = P_{l,m}.P_{m-1,n} \ \ (block\ carry\ propagate) \tag{7}$$

$$G_{l,n} = G_{l,m} + P_{l,m}.G_{m-1,n} \ \ (block\ carry\ generate) \tag{8}$$

$$C_m = G_{m-1,n} + P_{m-1,n}.C_n \tag{9}$$

where $l \geq m > n, G_{l,l} = g_l$ and $P_{l,l} = p_l$. The block kills, propagates and generates can be computed in any order given the initial g_is, p_is and k_is because the operations defined by equations 6, 7 and 8 are associative. It is to be noted that the symbol k_i has been included here just for the sake of completion of the truth table values and are implicitly coded in the logic when g_i and p_i are zero simultaneously.

3.2 Arbitrary Boundary Packed Carry-Lookahead Addition

In this subsection we present a scheme for performing arbitrary precision addition on user defined segments of two n-bit numbers. This scheme is based on the parallel segmented prefix computation algorithm described in [8].

To cast the packed addition problem into a parallel segmented prefix computation problem one can extend the input carry set of k_i, p_i and g_i by adding k_i |, p_i | and g_i |. The | to the right of a symbol identifies the boundaries of a packed sub-datatype. While an elaborate scheme can be worked out in which the extended carry set is used and an associative operator defined on them, it is however unnecessary owing to the fact the carry-transfer functions that are defined for a segmented addition scheme posed as a digit set conversion problem indicates that k_i | and p_i | behave exactly like k_i and g_i | behaves exactly like g_i. Also, once the carries are generated it is only a question of interpreting the results using the digit-mapping functions. The associativity of the carry-transfer functions defined for a segmented addition scheme can now be exploited to use the carry-lookahead tree to perform the addition in parallel.

In the following we assume that a mask register specifying the boundaries of a segment is loaded prior to the addition. This means that a separate instruction loads the mask register corresponding to the segment boundaries. This is assumed to be a one time operation whenever one wants to change the precision of the operands. In view of the above mentioned facts the equations for k_i, g_i, $K_{l,n}$, $P_{l,n}$ and $G_{l,n}$ can be used without any modification. The equation for p_i however has to be modified and is given by:

$$p_i = \overline{M_i}.(A_i \oplus B_i) \tag{10}$$

where $M_{n-1}M_{n-2}\ldots M_0$ is the mask register. M_i is '1' if the boundary of the sub-datatype to which the bit belongs is to the immediate right. Here it is assumed that the rightmost bit is the least significant bit. The digit mapping function of a conventional carry-lookahead adder given in equation 5 also has to be modified for a arbitrary precision packed data type adder and is given by:

$$S_i = A_i \oplus B_i \oplus \overline{M_i}.C_i \tag{11}$$

An arbitrary precision addition scheme can now be built around existing carry lookahead adder circuit since the scheme does not affect the logic of the circuit that actually computes the C_is.

The scheme discussed so far ignores block carries generated by a sub-datatype and is not considered a critical problem in non-media applications. However, the block carry generated may become necessary to implement saturating arithmetic wherein the packed sub-datatype is saturated to the maximum integer value with the precision specified by the mask register. An interesting observation is that the block carries generated by each sub-datatype of the packed operand appears as the carry at the least significant bit of the block immediately to the left (The least significant bit is assumed to be at the right of the packed

operand). Thus the set of block carries can be generated in a register (carry register) by performing $M_i.C_i$ and shifting the result right by one. The equation to generate the block carry at the most significant subdatatype of a n-bit register has to be however computed independently as:

$$C_n = (g_{n-1} + p_{n-1}.C_{n-1}) \quad (12)$$

Segmented addition schemes can be easily extended to multiply two arbitrary-boundary packed operands. In particular, if we consider the carry save adder (CSA) based multiplier of Santoro [9], segmented multiplication can be performed by casting the addition of partial products as segmented additions. While we we will not delve into the details of this scheme in this paper, it is sufficient to mention that in the multiplier reported in [9], partial products are added along the columns using a tree of $4 \rightarrow 2$ compressors to derive the SUM and CARRY words. These are later added to form the result of the multiplication. In the context of the discussion so far, the mask register can be used to generate the partial products appropriately. The SUM and CARRY results obtained using the normal $4 \rightarrow 2$ trees can then be added using the adder proposed earlier.

4 Comparison with Similar Adders

Brent *et* al. [7] give a pipelined scheme for implementing addition of large n-bit numbers as a sum of n/w w-bit numbers with the results of a lesser significant w-bit addition combined with the results of the next higher w-bit addition in a pipelined fashion. One can inhibit carries generated by every w-bit partition to implement a packed addition of n/w numbers in an n-bit register. A problem with such an approach with current technologies is that the latch delays can far outweigh the gate delays of such a pipelined adder. Even if such a scheme were feasible, the value of w is defined at the time of implementation and hence cannot be programmed by an application programmer. The segmented carry-lookahead scheme outlined above aims at giving the programmer the choice of programming his application with operand precision equal to that demanded by the application.

5 Conclusion

In this paper we have motivated the need for supporting arithmetic on the native datatypes of an application and also the need to let the programmer/application decide the precision of operands in arithmetic. We have also presented an addition scheme with very little hardware overhead over conventional carry-lookahead adders.

Acknowledgments

Thanks are due to Prof. V. Visvanathan, whose suggestions helped focus the presentation of the paper.

References

[1] Alex Peleg, Sam Wilkie and Uri Weiser, *"Intel MMX for Multimedia PCs"*, Communications of the ACM, Vol. 40, No. 1, January 1997, pp 25-38.

[2] Marc Trembley, J. Michael O'Connor, V. Narayanan, Liang He, *"VIS Speeds New Media Processing"*, IEEE Micro, August 1996, pp 10-20.

[3] Ruby B. Lee, *"Subword Parallelism with MAX-2"*, IEEE Micro 16, 4, August 1996, pp 51-59.

[4] Ruby B. Lee, *"Accelerating Multimedia with Enhanced Microprocessors"*, IEEE Micro 1995.

[5] Sunil Nanda, *"Media Processors"*, Proceedings of the 10^{th} International Conference on VLSI Design.

[6] Craig Hansen, *"MicroUnity's MediaProcessor Architecture"*, IEEE Micro, Auguest 1996, pp 34-41.

[7] Richard P. Brent and H. T. Kung, *"A Regular Layout for Parallel Adders"*, IEEE Transactions on Computers, Vol. C-31, No. 3, March 1982.

[8] F. Thomson Leighton, *"Introduction to Parallel Algorithms and Architectures"*, Morgan Kaufmann Publishers, San Mateo California, 1992.

[9] Mark Ronald Santoro, *"Design and Clocking VLSI Multipliers"*, Technical Report No. CSL-TR-89-397, Stanford University, Computer Systems Laboratory, October 1989.

[10] Peter Kornerup, *"Digit-Set Conversions: Generalizations and Applications"*, IEEE Transactions on Computers, Vol 43, No. 5, May 1994.

[11] R. E. Ladner & M. J. Fischer, *"Parallel Prefix Computation"*, Journal of the ACM, Vol 27, No. 4, October 1980, pp 831-838.