

More on Arbitrary Boundary Packed Arithmetic

P S Karthikeyan
Indian Institute of Technology, Madras
Department Of Aerospace Engineering
IIT Madras, India 600036
ae95015@aero.iitm.ernet.in

P S Ranganathan
Sri Venkateswara College of Engineering
Department of Computer Science and Engineering
SVCE, Pennalur, India 602105
psranga@mail.utexas.edu

Abstract

Recent microprocessors have been enhanced with media instruction sets for accelerating media algorithms. They exploit the fact that media algorithms have small data types, and widths much less than that of the processor. Current media instruction sets support only 8-, 16- and 32-bit sub-datatypes. This scheme is inefficient in several applications where bit lengths of 9, 12 and so on are used. We need user programmable sub-datatype bit lengths. [1] discusses arbitrary boundary packed addition.

Many media algorithms are based on multiply-accumulate algorithms. For full acceleration we also need arbitrary boundary packed multiplication. We present such a scheme based on Wallace tree multiplication. We also expand on [1] and provide a detailed treatment of the intermediate carries of sub-datatypes which were lost in the previous work. These carries could be used for saturation arithmetic and flow control.

1. Introduction

We find that general purpose processors are incorporating special instructions to manipulate multimedia data. MMX in x86, VIS in SPARC and MAX-2 of PA-RISC are examples of this trend. This has been motivated by the fact that small increases in hardware results in appreciable increase in speed in operations like media algorithms, because of factors like:

- Small data types
- Same operations repeated many times
- High data parallelism

These instruction sets contain special integer instructions that operate on packed units of integers. A 64-bit integer could be interpreted as having eight packed 8-bit integers.

Adding two such 64-bit numbers (with minor changes to the algorithm) will result in adding the eight 8-bit integers. Similar instructions are provided for multiply-accumulate and other operations.

An interesting observation has been made by Balakrishnan and Nandy in [1] that the size of the sub-datatypes allowed has been chosen for convenience of implementation. These architectures only allow 64-bit data to be subdivided into 8-, 16- and 32-bit parts. There are several instances where other data sizes are required. These are summarized from [1].

MPEG uses 8- and 9-bit fixed point data. Fax and modem needs 16-bit data. 3D graphics needs 24 or 32 bits. Audio needs 16- and 20-bit data. Applications like medical imaging use 12-bit data and the IDCT value of MPEG is also 12-bits. There are several such cases where there is a mismatch between the actual and available sub-datatype sizes.

The speedup achievable using existing architectures is not optimum. For example, processing 9-bit values requires the 16-bit subtypes. This leads to slower processing. Efficiency can be improved by allowing all sub-datatype widths.

2. Previous Work

In [1], a scheme for adding arbitrarily packed sub-datatypes has been described. It is based on the carry-lookahead algorithm. We provide a brief synopsis.

In a carry-lookahead scheme, all the carries are generated in parallel. Two terms called *carry propagate* and *carry generate* are defined. For binary numbers $A_{n-1}A_{n-2}\dots A_0$, and $B_{n-1}B_{n-2}\dots B_0$, they are defined as:

$$p_i = A_i \oplus B_i \text{ (carry propagate)} \quad (1)$$

$$g_i = A_i \cdot B_i \text{ (carry generate)} \quad (2)$$

Consider adding two 8-bit numbers with two packed 4-bit sub-datatypes.

```
0001 1100
1101 1100
```

A carry will be generated in bit 3 (assuming the LSB is bit 0). Since the two packed words are separate numbers, the carry from the lower nibble must not be propagated to the higher nibble. This can be achieved by logically and'ing with a 0 if the bit position is 3.

More generally, we have a term with which the carry propagate should be and'ed and another term with which the sum should be exclusive-or'ed.

We can think of a mask register M of the width of the original datatype whose bits contain 1 if the boundary of the sub-datatype to which that bit belongs is to the immediate right. This leads to the modified expressions for the carry propagate and sum.

$$p_i = \overline{M}_i \cdot (A_i \oplus B_i) \quad (3)$$

$$S_i = A_i \oplus B_i \oplus \overline{M}_i \cdot C_i \quad (4)$$

$$C_i = G_i \oplus P_i \cdot C_{i-1}, C_{-1} = 0 \quad (5)$$

In summary, we have a mask register that is used to mark the boundaries of the sub-datatypes. The sub-datatypes need not all be of the same width. Modified expressions are used for carry propagate and sum.

3. Motivation

The idea of arbitrary boundary packed arithmetic is useful only if all operations are available in that form. Till now, only addition as dealt with in [1] has been done.

Many media algorithms are based on the multiply-add operation. Hence for full acceleration, arbitrary boundary packed multiply-accumulate must also be implemented. Hence we present a method of arbitrary boundary packed multiplication based on the Wallace tree algorithm described in [2].

In the previous work, block carries generated by a sub-datatype were ignored. There is no record of any carries generated during the addition of sub-datatypes. Considering the importance of saturation arithmetic especially while handling numbers representing sound samples and color intensities, we have dealt with generation and usage of sub-datatype carries in detail.

4. Arbitrary Boundary Packed Addition

We will first describe a carry lookahead addition scheme and then describe how this can be extended to arbitrary boundary packed addition.

4.1. Carry Lookahead Addition

Let the two binary numbers be $A_0, A_1, A_2, A_3 \dots A_{n-1}$ and $B_0, B_1, B_2, \dots B_{n-1}$. The sum bits are $S_0 \dots S_{n-1}$ and carry bits are $C_{-1}, C_0 \dots C_n$.

The sum bits in a CLA are calculated using the following scheme:

$$\text{Propagate } P_i = A_i \oplus B_i \quad (6)$$

$$\text{Generate } G_i = A_i \cdot B_i \quad (7)$$

$$\text{Carry } C_i = G_i + P_i \cdot C_{i-1}, C_{-1} = 0 \quad (8)$$

$$\text{Sum } S_i = A_i \oplus B_i \oplus C_{i-1} \quad (9)$$

Ideally, the whole carry lookahead hardware can be constructed for any bit length digits, it is impractical because of limitations of fan-in, fan-out, irregularity in structure and length of wires that may be required. To overcome the problem, group carry adders are used, which are essentially small carry lookahead adders stringed together. Every group of smaller CLA will generate a group generate and a group propagate similar to the propagate and generate of the normal CLA, and a final group carry is generated for each group which is used as group carry in of next group.

4.2. Arbitrary Packed CLA

In this section we will be dealing with arbitrarily boundary packed addition of two N-bit numbers using CLA. The scheme in this section can be extended to group carry addition and for the sake of simplicity and brevity we will discuss the implementation for the CLA.

To define the boundary packing we will define a mask register M which is loaded with its value once every time the packing is changed. The mask register will be an N-bit register on an N-bit machine.

For a 32-bit machine, with bit packing of 4×8 , the mask will contain:

```
0000 0001 0000 0001 0000 0001 0000 0001
```

For a 32-bit machine with bit packing of 9, 4, 4, 3, 12, the mask will contain:

```
0000 0000 0001 001 0001 0001 0000 00001
```

that is, the LSB of the sub-datatype will contain the value 1 and the rest will be zero. In the following examples we will be using the mask given in the second example.

We also have an extra N-bit carry register C' which contains the carries generated by the previous addition, bubbled back to be used as the C_{-1} for the next addition.

The equations in the CLA will now change to:

$$\text{Propagate } P_i = A_i \oplus B_i \quad (10)$$

$$\text{Generate } G_i = A_i \cdot B_i \quad (11)$$

$$\text{Carry } C_i = G_i + P_i \cdot C_{i-1}, C_{-1} = 0 \quad (12)$$

$$\text{Sum } S_i = A_i \oplus B_i \oplus (C_{i-1} \cdot \overline{M_i} + C'_i \cdot M_i) \quad (13)$$

$$\text{Carry out } C' = \text{previous add's carries} \quad (14)$$

4.3. Sub-datatype Carry Handling

C' is defined such that it takes care of the carry that was generated by the previous addition. Suppose in the first addition the first datatype generated an extra carry out of the 9th bit, then the carry status of the C will be:

Mask 1 0 0000 0001

Carry x 1 xxxx xxxx

In the next addition this 1 should be used as the carry-in of the first sub-datatype. To do this we need to bubble through this carry to the LSB of the sub-datatype, and in this case zeroth location. These bubbled carries are saved in C' , and are updated after addition, to be used for the next addition or for extra hardware to implement saturation arithmetic or for controlling program flow by means of test-and-jump instructions.

C' is then defined as

$$C'_i = C_i \cdot M_{i+1} + (C'_{i+1} \cdot \overline{M_{i+1}}) \quad (15)$$

C' initially set to zero. The values being set from left to right.

If the definition is implemented using this recursive definition it will introduce a clock delay of N for calculating the product terms, for an N-bit adder, which is unacceptable. We therefore unroll the recursion and redefine as:

$$C'_i = C_i \cdot M_{i+1} + \sum_{j=N}^{i+1} (C_j \cdot M_{j+1} \cdot \prod_{k=j}^{i+1} \overline{M_k}) \quad (16)$$

This can be further simplified and speeded up.

We can see that for each evaluation of C' , $\prod_{k=j}^{i+1} \overline{M_k}$ will introduce propagation delay of the AND gates. For C'_0 this will be a maximum of N. These products then have to be summed up, and will further introduce a delay of N, if addition is done term by term. To reduce this considerably, we can precompute all the product terms and save them once when the mask register's value is set, since the product is purely a function of M and independent of C.

The total number of product terms will be $N \times (N-1)/2$. These can be saved in N-bit pseudo-registers. We call these pseudo-registers because they are not really registers requiring complex wiring, but just memory locations which are

wired to one single place on the circuit. We will require $(N-1)/2$ such registers. For $N = 32$, we will require 16 pseudo-registers.

It can be seen that from the definition of Equation 15 that of the two terms, always only one is zero, and the other is one. What this implies is that in the C'_{i-1} term we do not need to carry the excess baggage of both the terms and consider

$$C'_{i-1} = C_{i-1} \cdot M_i + (C'_i \cdot \overline{M_i}) \quad (17)$$

$$= C_{i-1} \cdot M_i + C_i \cdot M_{i+1} \cdot \overline{M_i} + C'_{i+1} \cdot \overline{M_{i+1}} \cdot \overline{M_i} \quad (18)$$

Instead we can rewrite as

$$C'_{i-1} = \begin{cases} C_{i-1} \cdot M_i + (C_i \cdot M_{i+1} \cdot \overline{M_i}) & \text{if } \overline{M_{i+1}} = 0 \\ C_{i-1} \cdot M_i + (C'_{i+1} \cdot \overline{M_{i+1}} \cdot \overline{M_i}) & \text{if } \overline{M_{i+1}} \neq 0 \end{cases} \quad (19)$$

Using Equation 19, the number of terms for every C'_i will remain constantly 2. This is being further investigated to reduce complexity and increase speed.

5. Arbitrary Boundary Packed Multiplication

Booth multiplication [3] which gives 50% speedup over sequential multiplication and Wallace tree algorithm [2, 4] are popular methods for implementing multiplication hardware.

We describe an arbitrary boundary packed algorithm for a high speed multiplication based on Wallace trees [5, 6, pages 183–203].

5.1. Wallace Tree Multiplication

The essential idea behind using Wallace tree multipliers is to split the multiplication into smaller operands; multiply these operands separately and in parallel and then add the partial products.

The smaller operands are typically 4-bit operands and their multiplication is done using purely combinatorial logic. For a 8-bit computer using 4 bit operands:

A is rewritten as $A_h A_l$ where A_h and A_l are higher and lower order operands respectively. $B = B_h B_l$

$$\begin{aligned} P &= A \times B \\ &= (A_h \cdot A_l) \times (B_h \cdot B_l) \\ &= A_h \times B_h + A_h \times B_l + A_l \times B_h + A_l \times B_l \\ &= P_{hh} + P_{hl} + P_{lh} + P_{ll} \end{aligned}$$

The partial products are added with correct bit weights.

5.2. Arbitrary Boundary Packed Multiplication

We will now deal with how we can use the above discussed multiplier for our problem.

Consider a hypothetical 9-bit machine. This machine will be used to explain the operation of the modified Wallace Tree multiplier. It has two 4-bit combinatorial multiply operands. So we will have a Mask of 01 001 001. Figures 1 and 2 pertain to this computer.

When we multiply, we require only some of the partial products and the rest are to be killed. This is shown in figure 1. The dotted lines represent the products that are to be ignored and the thick lines represent the products that should be retained. This can be done by using a 2D array of mask bits; we call this the *mask array* and refer to it as \mathcal{M} .

Figure 2 shows the bits that are to be used. They are boxed within thick lines.

This can be implemented by making some changes to the combinatorial multiplier circuit. The original multiplier circuit can be found in [6, page 197]. The modified circuit is shown in figure 3

To kill the bit products that are not required, we replace the 2 input AND gates with 3 input AND gates and the third input of a gate whose two other inputs are A_i and B_j is the mask array element \mathcal{M}_{ij} .

Mask array initialisation of \mathcal{M} is done once when the mask register is loaded with its value such that:

$$\mathcal{M}_{ij} = \begin{cases} 1 & \text{if the bit product } A_i B_j \text{ is valid} \\ 0 & \text{otherwise} \end{cases}$$

The pseudo-code below explains the algorithm for setting the \mathcal{M}_{ij} values. This can be implemented using a sequential circuit with a time delay of N clocks for a N-bit computer.

```
fillarray( $\mathcal{M}$ , 0);
/* initialise mask array to 0 */

for (i = 0; i < N; i++) {
    subdatalimit = 0;
    for (j = i; j < N; j++) {
        if (M[j] == 1) {
            /* reached next mask bit
            ⇒ one subdatatype done */
            subdatalimit = j-1;
            break;
        }
    }
}
if (subdatalimit == 0)
    subdatalimit = N-1;
for (k = i; k <= subdatalimit;
    k++) {
    for (l = i; l <= subdatalimit;
```

```
        l++) {
             $\mathcal{M}[k][l] = 1;$ 
        }
    }
    i = subdatalimit;
}
```

This array can be initialized using a sequential circuit.

The Mask array will require $N \times N$ bits which is N pseudo registers. This is the same level of complexity as addition.

6. Applications

The DFT, DCT, convolution and many such DSP operations are essentially multiply-add which is the motivation behind such an instruction in the MMX instruction set.

The architecture can be useful for the general purpose graphics and multimedia techniques used in alpha blending, sound mixing. YUV822, YUV422 are common data formats used by frame grabbers. For such data, arbitrary boundary packed multiplication-add will be efficient.

7. Conclusion

We have motivated the need for comprehensive arbitrary boundary packed schemes to support different native data types for future architectures. We have presented a scheme for such multiplication and addition with small overheads compared to conventional adders and multipliers.

References

- [1] S. Balakrishnan and S. K. Nandy, "Arbitrary precision arithmetic – SIMD style," in *Proceedings Eleventh International Conference on VLSI Design*, pp. 128–132, 1998.
- [2] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electron. Comp.*, vol. EC-13, pp. 14–17, Feb 1964.
- [3] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. and Appl. Math.*, vol. IV, pp. 236–240, 1951.
- [4] J. Fadavi-Ardekani, " $M \times N$ Booth encoded multiplier generator using optimized Wallace trees," *IEEE Trans. on VLSI*, vol. 1, pp. 120–125, Jun 1993.
- [5] J. Pihl and E. J. Aan, "A multiplier and squarer generator for high performance DSP applications," tech. rep., Norwegian University of Science and Technology, 1996.

[6] J. J. F. Cavanagh, *Digital Computer Arithmetic Design and Implementation*. McGraw-Hill Book Company, 1985.

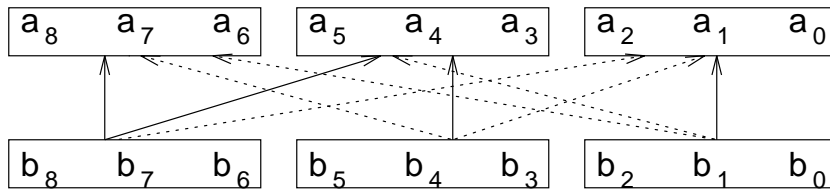


Figure 1. Bitwise multiplication blocks

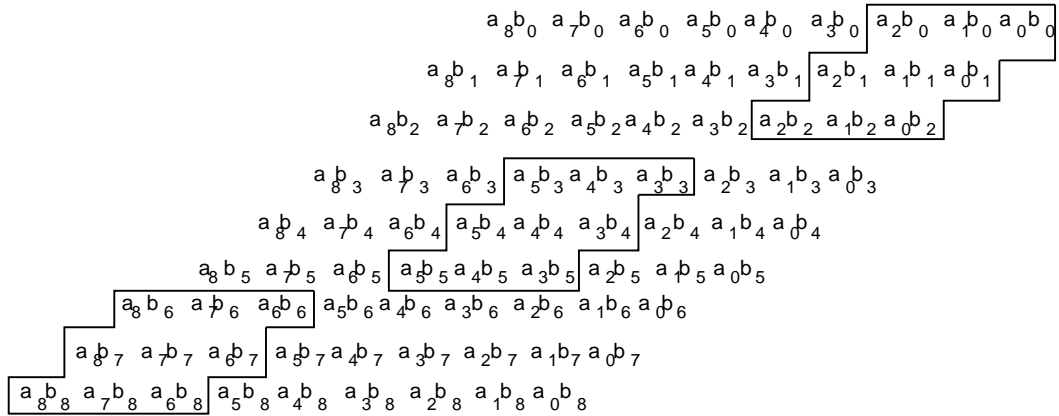


Figure 2. Bitwise multiplication digits

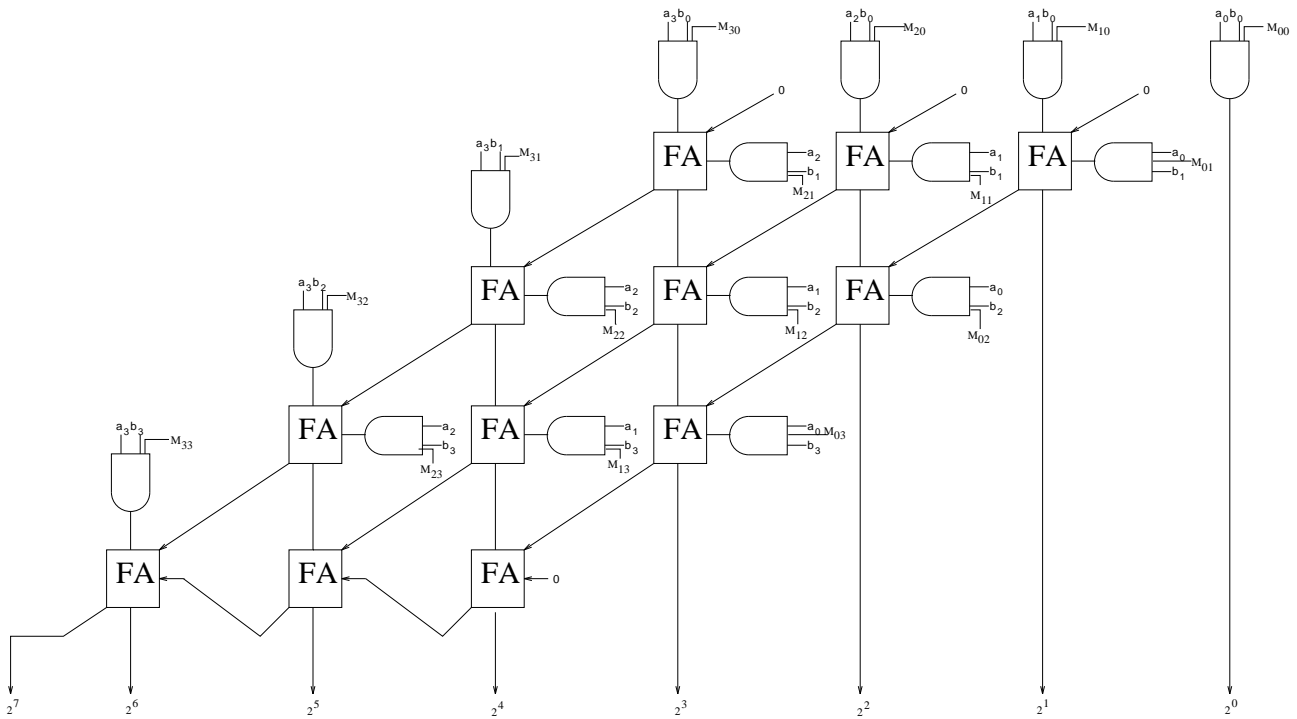


Figure 3. A 4×4 combinatorial multiplier modified for arbitrary bit packing.