

# The Sum-Absolute-Difference Motion Estimation Accelerator

S. Vassiliadis, E.A. Hakkennes, J.S.S.M. Wong, G.G. Pechanek  
Electrical Engineering Dept. BOPS Inc.  
Delft University of Technology 6340 Quadrangle Drive  
Mekelweg 4, 2628 CD Delft Suite 210  
The Netherlands Chapel Hill, NC 27514  
E.Hakkennes@ET.TUdelft.NL

## Abstract

*In this paper we investigate the Sum Absolute Difference (SAD) operation, an operation frequently used by a number of algorithms for digital motion estimation. For such operation, we propose a single vector instruction that can be performed (in hardware) on an entire block of data in parallel. We investigate possible implementations for such an instruction. Assuming a machine cycle comparable to the cycle of a two cycle multiply, we show that for a block of 16x1 or 16x16, the SAD operation can be performed in 3 or 4 machine cycles respectively. The proposed implementation operates as follows: first we determine in parallel which of the operands is the smallest in a pair of operands. Second we compute the absolute value of the difference of each pairs by subtracting the smallest value from the largest and finally we compute the accumulation. The operations associated with the second and the third step are performed in parallel resulting in a multiply (accumulate) type of operation. Our approach covers also the Mean Absolute Difference (MAD) operation at the exclusion of a shifting (division) operation.*

## 1. Introduction

In block-based motion estimation [10, 3], that is motion estimation performed on a set of pixels, every frame is divided into blocks of equal size and for each block in the *current* frame a search is performed in the *reference* frame to find the block resembling the current block the most. Because a search performed over the whole reference frame for each block in the current frame is computational intensive and movements in video sequences are usually small, the search is limited to a search area. After finding the best match for the current block, the motion vector (i.e. the displacement relative to the current block) is stored together with the differences between the two blocks. In determining

which block in the searching area of the reference frame is the best match with the current frame, a best match method is employed. The best match is usually established with the use of the *mean absolute difference* (MAD) and the *sum of absolute differences* (SAD).

In this paper our primary concern is to propose a hardware solution to the SAD and the MAD operations. That is our primary concern is to propose instructions that have “convenient” hardware implementations, where “convenient” in the context of our discussion mainly means parallel hardware vector related implementations. We note here that if the division (shifting) operation is excluded from the MAD then both operations can be viewed as equivalent. In essence, discussing the SAD operation will also cover the MAD with an additional shift (divide) of the final result, thus MAD is no longer considered in the discussion to follow.

Given that the SAD operation is usually considered for 16x16 pixels (pels) blocks [10] and because the search area could involve a high number of blocks, performing the SAD operation could be time-consuming if traditional methods are used for its computation<sup>1</sup>. In this paper we propose a new instruction that is capable of producing the direct SAD operation. Furthermore we also show that the proposed instruction is scalable, depending on the constraints of the technology considered for the design. This is shown by considering a 16x1 sub-block element and an entire 16x16 element and showing that the implementation will require 3 machine cycles<sup>2</sup> for a 16x1 sub-block and 4 cycles for a 16x16 block. The 16x16 block performance is achieved by using hardware proportional in size to a 16x1 sub-block unit, that is we achieve a 4 cycle 16x16 block SAD using

<sup>1</sup>Traditional here means that performing SAD requires a number of subtractions with proper complementation to produce the absolute value which are followed by an accumulation to perform the final operation.

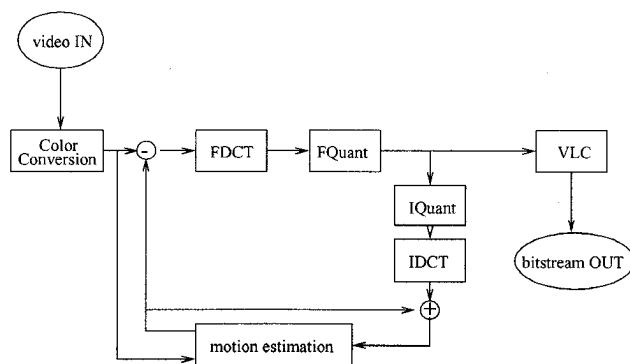
<sup>2</sup>A cycle here is considered to be comparable to the cycle of a high-speed, 2-cycle, 32x32 bit multiplier [11, 17, 18]. Other implementations including array systolic implementations are also possible.

approximately 16 times the area of the 16x1 SAD.

The rest of this paper is organized as follows. Section 2 gives some background information about motion estimation and how it fits in the MPEG standard, followed by a discussion of the SAD operation. Section 3 describes the basic operation of our proposed Sum-Absolute-Difference unit, and Section 4 gives a sample implementation of the proposed unit. Section 5 concludes this paper with some remarks and future research directions.

## 2. Background

In MPEG [10, 4], video-sequences are compressed by exploiting both spatial and temporal redundancies. Spatial redundancies can be seen as small differences between local pels. In many encoding schemes the spatial redundancies are exploited using DCT [1, 7, 15] or predictive coding [12]. Temporal redundancies can be seen as small differences between two temporally close video frames. These kind of redundancies can also be exploited using predictive coding, but more compression can be reached by using it together with motion compensation [6]. As an example, a diagram of the MPEG encoding process is given below. Because the MPEG standard does not specify the encoding process, this diagram is only one possible implementation for the MPEG encoding process. In the diagram, FDCT denotes the Forward Discrete Cosine Transformation, FQuant denotes the Forward Quantization and VLC denotes Variable Length Coding. The IQuant and IDCT are the inverse operations needed to reproduce the picture as it is available at the decoder. The motion estimation block uses these decoded images as reference instead of original images, because the decoder only has access to decoded images. Adding “differences from original images” to decoded images which are already slightly different from the original images, would introduce unnecessary errors in the decoded images.



**Figure 1. Diagram of a MPEG encoder implementation.**

In the MPEG coding, there are two kinds of blocks: the 16x16 (pels) macro-block and the 8x8 (pels) basic block. The basic block is used when the DCT is performed and the macro-block is used for motion estimation. The encoding of a video stream is done in several steps. Each of the steps depicted in Figure 1 are explained below. For simplicity, many details regarding the MPEG standard are left out.

**Color conversion** In this step the input color-space is transformed into the YCbCr color-space. Furthermore, the chrominances are subsampled by a factor of two in both the horizontal and vertical direction. Thus, a 16x16 block from the video signal results in four 8x8 luminance blocks, one 8x8 Cb block, and one 8x8 Cr block. These 8x8 blocks are used by the DCT. The 16x16 luminance block is used by the motion estimation.

**Motion Estimation** In this step, for each block of 16x16 luminance pels in the current frame, a motion vector is computed. This motion vector contains the relative position of the block most closely resembling the current block in the reference frame (either in the past or future). To exploit redundancies between difference values, the difference values are also put through the DCT process.

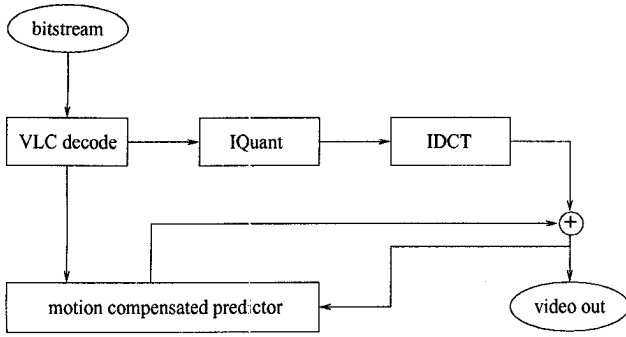
**Discrete Cosine Transformation** In this step a DCT is performed on each 8x8 blocks which can be either blocks from the frame or difference blocks.

**Quantization** In this step the DCT coefficients computed in the DCT process are quantized. This step is the main contribution to the lossiness of the MPEG coding standard. However, the information lost in this step is thought to be (almost) not perceivable due to the use of the DCT. Depending on constraints on the bitstream rate, the quantization can be adjusted to meet these constraints which can result in lower quality video if the quantization is too coarse.

**Variable Length Coding** In this step the results of the quantization process are serialized into a bitstream using run-length coding and variable length coding (in this case Huffman coding).

The decoding process is depicted in Figure 2 which is basically the lower part of the encoding process.

First, the incoming bitstream is decoded using variable length decoding. Second, the results are fed into an inverse quantization step and an inverse DCT step. If the blocks were not motion-compensated difference blocks, they can be directly fed to the output of the decoder. If motion vectors are decoded, then they are used to fetch data from reference frames, to which the decoded difference values are



**Figure 2. Diagram of a MPEG decoder implementation.**

added. Worth noting is that the MPEG standard is not symmetric, meaning that the computational requirements for the encoder and the decoder are different. For example, the encoder has to put lots of effort in calculating motion vectors, while the decoder just uses motion vectors to fetch the right block from memory.

This paper focuses on ways to speed up the motion estimation part of MPEG encoding, also denoted as *motion vector search*. There are several algorithms to compute which block in the reference frame most closely resembles the current block. At one end of the spectrum is the exhaustive search [12] which is time-consuming, but produces the best possible result, and at the other end there are several heuristic-based algorithms, which are much faster at the cost of a possibly less optimal result. Examples of these heuristic-based search algorithms are the Three Step Search [8] and the Two Dimensional Logarithmic search [5, 14]. The assumption made by these algorithms is that the global minimum can be reached by following the steepest descent. If the results are less optimal, this will yield an increase in the number of difference values and larger difference values. This in turn results in larger bandwidth requirements or quality degradation if bandwidth constraints apply, because coarser quantization must be applied.

Irrelevant of the search algorithm to determine the best resemblance, a metric is used which indicates the “closeness” between compared blocks. The two most common metrics found in different search algorithms are the *mean square error* (MSE) and the *mean absolute difference* (MAD). The MSE is performed by the following (assuming blocks of 16x16 pixels):

$$MSE(x, y, r, s) = \frac{1}{256} \sum_{i=0}^{15} \sum_{j=0}^{15} (A_{(x+i, y+j)} - B_{((x+r)+i, (y+s)+j)})^2 \quad (1)$$

The MAD is performed by

$$MAD(x, y, r, s) = \frac{1}{256} \sum_{i=0}^{15} \sum_{j=0}^{15} |A_{(x+i, y+j)} - B_{((x+r)+i, (y+s)+j)}| \quad (2)$$

MAD can also be rewritten as

$$MAD(x, y, r, s) = \frac{SAD(x, y, r, s)}{256} \quad (3)$$

where SAD is the summation of the absolute differences, that is:

$$SAD(x, y, r, s) = \sum_{i=0}^{15} \sum_{j=0}^{15} |A_{(x+i, y+j)} - B_{((x+r)+i, (y+s)+j)}| \quad (4)$$

In these equations (x,y) is the position of the current block and (r,s) denotes the motion vector, i.e. the displacement of the current block (A) relative to the block in the reference frame (B). The *x* and *y* in Equations 1 though 4 are multiples of 16<sup>3</sup> for MPEG1 and the values of *r* and *s* are determined by the algorithm.

Given that the MAD metric, due to its computational simplicity, is used more often, we will not consider the MSE in our discussion. In the section to follow, we introduce a novell approach for the computation of the SAD which leads to the computation of the MAD with a trivial extension.

### 3. Computing the Sum Absolute Difference

In this section we proceed by investigating the SAD operation and propose some possible parallel implementations leading to an instruction proposal for SAD. The general algorithm computing the Sum Absolute Difference of two blocks is depicted in Equation 4. A direct approach in computation the SAD consists of the following steps:

- Compute  $(A_i - B_i)$  for all 16x16 pixels in the two blocks A and B.
- Determine which  $A_i - B_i$  are less than zero and produce in that case  $B_i - A_i$  as the absolute value, else produce  $A_i - B_i$ .
- Perform the accumulate operation to all 16x16 absolute values.

<sup>3</sup>We note that we assume in the remaining of the presentation 16x16 pixel MPEG1 blocks. This assumption is not restrictive as our proposal supports arbitrary block-sizes.

In order to speed up the computation, we perform a multiplicity of operations in a single operation. In the case of the computation of the SAD we want to eliminate the absolute-difference operations. Generally, it is not possible to eliminate these operations, because of the inability to take an absolute operation out of a summation.

$$\sum |A_i - B_i| \neq |\sum (A_i - B_i)| \quad (5)$$

Our solution to this problem is as follows. By determining the smallest of both operands and subtracting it from a constant, it becomes possible to eliminate the absolute operations. This subtraction is a trivial operation, if the constant is chosen correctly.

To achieve our goal, we first briefly describe an unit capable of computing the SAD of 16x1 pels in parallel, where each pel(pixel) is represented in 8 bits (in unsigned binary notation).

**Determine the smallest of two operands** This is done by inverting one of the operands, and computing the carry-out which would arise from the addition of both operands.

**Invert the smallest operand and pass both operands to an adder tree.** The smallest operand is inverted, which means that its value changes to  $2^8 - 1 - X = 255 - X$ . Both the inverted smallest and the largest values are passed to the adder-tree, which corrects for this constant ( $2^8 - 1 = 255$ ).

The above two steps can be carried out in parallel for 16 pels. The result is 32 8-bit values, on which the following steps are applied.

**Addition of a correction term** The correction term is added to account for the  $2^n - 1$ 's introduced by the inverting of the smallest value. If the number of pels on which the unit is operating is a power of 2, the correction term is equal to that number, as the sum of the  $2^n$  adds up to one "simple eliminatable bit". If the number of pels the unit operates on is not a power of two, we also have to account for the additional  $2^n$  per pel.

**Reduce the 32 rows to 2** The resulting 32 rows passed to the adder tree and the correction-term is 33 rows, are reduced to 2 rows by using a counter scheme, see for example [19, 2, 16].

**Reduce the 2 rows to 1 (accumulation)** In this final step, a full summation of the two remaining rows is performed. The total sum of all constants, which has to be discarded, is the carry\_out of this addition.

A more thorough explanation of each step follows below, which is using  $n$  for the number of bits to represent the luminance pels and  $m$  instead of 16 for the number of pels on which the unit operates. Note that if  $m$  is a power of 2, we have a special case which may simplify some computations.

**Step 1, Determining the smallest:** Both operands  $A$  and  $B$  are positive numbers in binary representation in  $n$  bits and range from 0 to  $(2^n - 1)$ . The result of  $|A - B|$  is also in unsigned binary representation, and has also the same range. To avoid the absolute operation, we can substitute  $|A - B|$  with  $A - B$  or  $B - A$ , depending whether  $A$  or  $B$  is the smallest. To determine which one is the smallest, we have to check whether the following inequality is true or false:

$$B > A \quad (6)$$

$$B - A > 0 \quad (7)$$

Generally it is not possible to subtract two positive numbers without the possibility of producing a negative result which can not be represented as an unsigned number. If we subtract  $A$  from its maximum value,  $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ , the result is always positive or zero. The result of the subtraction  $(2^n - 1 - A)$  is  $\bar{A}$ , the binary bit by bit inversion of  $A$ . This can be concluded from the following equation:

$$A + \bar{A} = \sum_{i=0}^{n-1} 2^i = 2^n - 1 \quad (8)$$

$$\bar{A} = 2^n - 1 - A \quad (9)$$

We still have to check the following inequality:  $B > A$ . We rewrite this inequality to:

$$B > A \quad (10)$$

$$-A + B > 0 \quad (11)$$

$$2^n - 1 - A + B > 2^n - 1 \quad (12)$$

$$\bar{A} + B > 2^n - 1 \quad (13)$$

$$\bar{A} + B \geq 2^n \quad (14)$$

The last step is possible because we are dealing with natural, non-fractional, numbers.

The maximum value of  $\bar{A} + B$  is  $2 * (2^n - 1) = 2^{n+1} - 2$ . This is a  $n + 1$  bit number. The most-significant bit, with weight  $2^n$ , is computed as the carry-out of the  $n$  bit addition. Thus checking whether  $\bar{A} + B \geq 2^n$  means checking whether the addition of the bit inverted  $A$  and the operand  $B$  produces a carry\_out.

**Step 2, Inverting the smallest value:** To compute  $|A - B|$  in a single step (which will improve the computation of the SAD) we can compute separately  $A - B$  and  $B - A$  and

determine which of the two has a negative result. Consequently, we could choose (multiplex) between the two results choosing the “positive” value. There are two drawbacks with this approach. One relates to the hardware, the other to delay. To perform the entire operation in parallel we must consider two adders per single operation and pay, in addition to the adder delay, the multiplexer delay. The two problems can be alleviated by doing the following. Instead of adding the two input values, we convert the smallest input value to  $2^n - 1 - X = \overline{X}$ , (that is the one’s complement of  $X$ ). In the remaining discussion, these two values form two rows and they are denoted as a couple.

There are two cases arising from the previous step:

**No carry was generated** This implies  $B \not> A$ . In this case we should invert  $B$  to  $\overline{B}$ . As stated previously, the value of  $\overline{B}$  is equal to the positive number  $2^n - 1 - B$ . This number is again in unsigned binary representation. The value  $A$  should be propagated unmodified. Their sum equals  $2^n - 1 - B + A = 2^n - 1 + |A - B|$ .

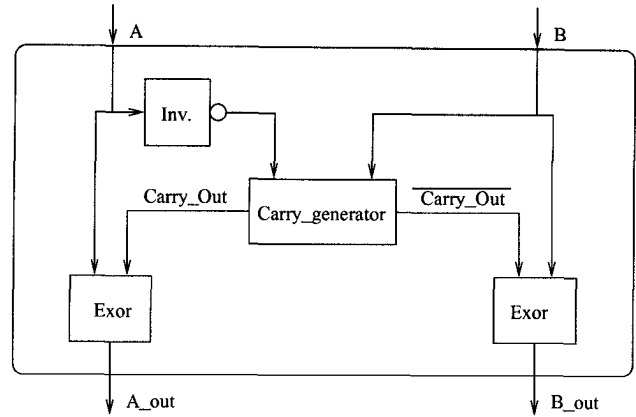
**A carry was generated** This implies  $B > A$ . In this case we should invert  $A$  to  $\overline{A}$  and propagate  $B$  unmodified. Their sum equals  $2^n - 1 - A + B = 2^n - 1 + |A - B|$ .

Thus in both cases, the  $n + 1$  bit sum of the two values is equal to  $2^n - 1 + |A - B|$ , which is the desired value  $|A - B|$  plus a constant of  $2^n - 1$ . In the next step, this constant will be eliminated. It should be noted here, as also indicated in step1, that the inversion of the operands  $A$  or  $B$  is not known a priori. To determine which operand ( $A$  or  $B$ ) to invert, it is enough to compute the carry out of the operation  $\overline{A} + B$  (see step1 for further elaboration).

Step 2 takes one level of multiplexers and can be performed in the same first cycle. This step produces two  $n$ -bit numbers, which are added in step 4. Figure 3 gives a graphical representation of the first two steps. We note here that steps 1 and 2 substitute two adders and multiplexing logic of the output of the adders with carry-out-detection logic and multiplexing of operands improving both the hardware and the delay requirements.

**Step 3, Adding a correction term:** In order to parallelize the computation of the SAD, the two previous steps are performed on  $m$  couples (that is  $A, B$  operands) in parallel. The  $2m$  rows, the result of step 1 and 2, are positioned into a matrix which is then reduced (summed) using some well known counter-scheme and discussed in steps 4 and 5.

Each of the  $m$  couples has a sum equal to  $2^n - 1 + |A_i - B_i|$ , assuming  $A_i$  and  $B_i$  of length  $n$ . Thus the sum of all



**Figure 3. Graphical representation of the first two steps in computing the Sum Absolute Difference (SAD).**

couples has the value

$$Sum = m * (2^n - 1) + \sum_{i=0}^{m-1} |A_i - B_i| \quad (15)$$

which can be rewritten as:

$$Sum = m * 2^n - m + \sum_{i=0}^{m-1} |A_i - B_i| \quad (16)$$

Because  $|A_i - B_i|$  is always less than  $2^n$ , the sum  $\sum_{i=0}^{m-1} |A_i - B_i|$  will always be less than  $m * 2^n$ . The desired sum is therefore always representable in  $n + \lceil \log_2(m) \rceil$  bits.

For this discussion, we define  $q = \lceil \log_2(m) \rceil$ .

In order to eliminate the constant  $m * 2^n - m$  from the sum without subtraction, we have to be able to split the result in two parts. The first part consists of the lower  $(q + n)$  bits, and the higher part consists of the most-significant bit, with value  $2^{q+n}$ . We now have to make sure that the sum of the constants equals this most-significant bit, by adding an extra constant. The value of this extra constant is computed at design-time with the following formula:

$$Extra\_Constant = 2^{q+n} - m * 2^n + m \quad (17)$$

Note that in the case that  $m$  is a power of 2, it simply takes the value  $m$ . (fill in  $2^q$  for  $m$  in the above equation)

After adding this extra\_constant, the total sum will be:

$$\begin{aligned}
 Total\_Sum &= Extra\_Constant + Sum \\
 Total\_Sum &= 2^{q+n} - m * 2^n + m + \\
 &\quad + m * 2^n - m + \sum_{i=0}^{m-1} |A_i - B_i| \\
 Total\_Sum &= 2^{q+n} + \sum_{i=0}^{m-1} |A_i - B_i| \quad (18)
 \end{aligned}$$

Given that  $2^{q+n}$  is not required to represent the result, it can be discarded producing the needed Final\_Sum as:

$$\begin{aligned}
 Final\_Sum &= Total\_Sum - 2^{q+n} \\
 Final\_Sum &= 2^{q+n} + \sum_{i=0}^{m-1} |A_i - B_i| - 2^{q+n} \\
 Final\_Sum &= \sum_{i=0}^{m-1} |A_i - B_i| \quad (19)
 \end{aligned}$$

**Step 4, Matrix reduction:** In step 4, we reduce the matrix of  $2m + 1$  rows of  $n$  bits to 2 rows.

This matrix reduction can be done in several ways. We could use for example Lim counters [9], 6-2 counters [16, 13], or a tree of Carry-Save-Adders (CSA)[19, 2]. The Carry-Save-Adder-Tree approach is used in the example in Section 4 and shows that for  $m = 16$  and  $n = 8$ , 260 CSA's in 8 levels can reduce the 33 rows to 2.

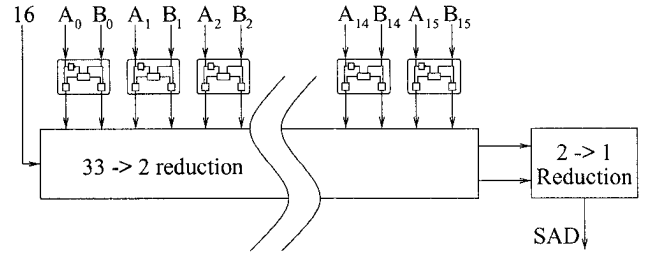
**Step 5, Final reduction:** The last step is the final reduction of the matrix. This is done using a fast carry-lookahead scheme.

Figure 4 shows a graphical representation of a  $16 \times 1$  unit, that is a unit operation on 16 couples of elements producing a single output value. The top half shows 16 times steps 1 and 2 in parallel, and steps 4 and 5 are depicted in the bottom half. Step 3 is represented by the addition term at the left (16).

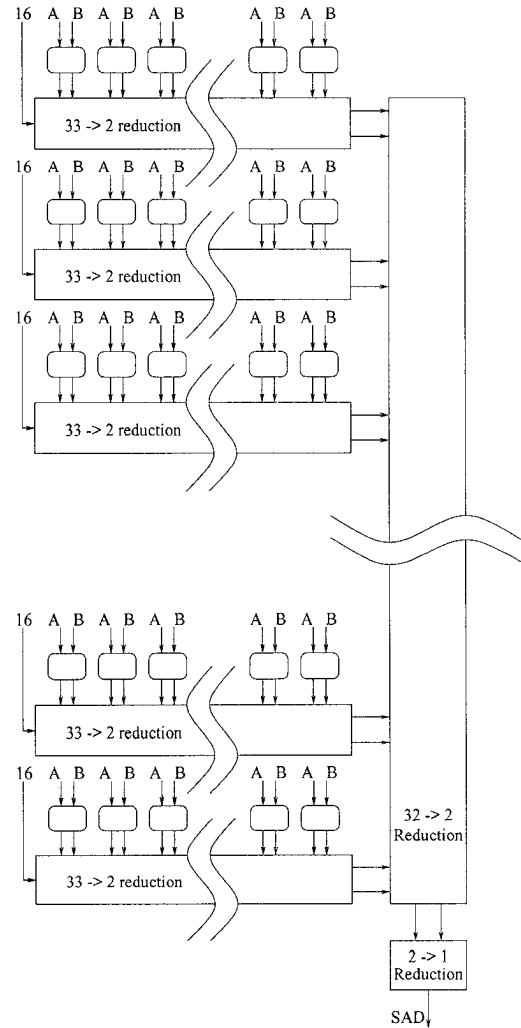
The concept can be expanded to an array capable of computing the SAD of  $16 \times 16$  pel blocks. In this case, the 2 rows going into the 2-to-1 reduction should go into another 32-to-2 reduction unit, together with the 30 rows of the 15 other units. The result of this 32-to-2 reduction is then reduced by a 2-to-1 final adder. This saves both the execution time and the area of 15 2-to-1 reduction units. For a block diagram of this extension see Figure 5

#### 4. A Sample Hardware Implementation

As an example, we describe the implementation of an unit which computes the SAD of two  $16 \times 1$  blocks, which



**Figure 4. Graphical representation of the SAD computation of a  $16 \times 1$  block by using 16 of the blocks from Figure 3 in parallel and a multiplier-like tree reduction.**



**Figure 5. A  $16 \times 16$  pel SAD computation unit. Note that each block is assumed to take one cycle, thereby making the total number of cycles for this unit equal to 4.**

can be either a row or a column of a 16x16 macroblock. We assume 8 bit values which is common in MPEG.

**Step 1, Determining the smallest:** We need 16 parallel blocks to perform step 1. In each of these blocks, we first need to compute the  $C_{out}$  of the n bit addition  $\overline{A} + B$ . This is formed by:

$$C_{out} = G_0^7 + P_0^7 * C_{in} \quad (20)$$

In Equation 20, P stands for Propagate and G for Generate. Because  $C_{in}$  is always zero in this case, we can ignore  $P_0^7$ . The remaining term to compute is  $G_0^7$ . This can be computed in 4 stages using 2x2 And-Or-Invert as the most complex gate as follows:

Stage 1

$$G_0^0 = \overline{a_0} * b_0 \quad (21)$$

....

$$G_7^7 = \overline{a_7} * b_7 \quad (22)$$

$$P_1^1 = \overline{a_1} + b_1 \quad (23)$$

....

$$P_7^7 = \overline{a_7} + b_7 \quad (24)$$

Stage 2

$$G_6^7 = G_7^7 + P_7^7 * G_6^6 \quad (25)$$

$$G_4^5 = G_5^5 + P_5^5 * G_4^4 \quad (26)$$

$$G_2^3 = G_3^3 + P_3^3 * G_2^2 \quad (27)$$

$$G_0^1 = G_1^1 + P_1^1 * G_0^0 \quad (28)$$

$$P_6^7 = P_7^7 * P_6^6 \quad (29)$$

$$P_4^5 = P_5^5 * P_4^4 \quad (30)$$

$$P_2^3 = P_3^3 * P_2^2 \quad (31)$$

Stage 3

$$G_4^7 = G_6^7 + P_6^7 * G_4^5 \quad (32)$$

$$G_0^3 = G_2^3 + P_2^3 * G_0^1 \quad (33)$$

$$P_4^7 = P_6^7 * P_4^5 \quad (34)$$

Stage 4

$$G_0^7 = G_4^7 + P_4^7 * G_0^3 \quad (35)$$

It might be convenient to compute  $\overline{G_0^7}$  in the same stage.

$$\overline{G_0^7} = \overline{G_4^7 + P_4^7 * G_0^3} \quad (36)$$

$$\overline{G_0^7} = \overline{G_4^7 * P_4^7 * G_0^3} \quad (37)$$

Depending on the chosen technology, it might be possible to skip the first stage, and to merge it with the second stage.

**Step 2, Inverting the smallest:** In the second step, we have to invert the smallest of A and B. Again, this needs to be done for 16 input-pairs. In terms of hardware, this operation is merged with the operation in step 1 and performed as follows.

Stage 5

$$\forall i \in \{0..7\},$$

$$a_{i,out} = G_0^7 * \overline{a_i} + \overline{G_0^7} * a_i \quad (38)$$

$$b_{i,out} = G_0^7 * b_i + \overline{G_0^7} * \overline{b_i} \quad (39)$$

These 5 stages of step 1 and 2 can be executed in the first cycle.

**Step 3, Placing the correction term:** In step 3 we place a correction-term to the terms to be added. Therefore, the number of rows to add up in step 4 becomes  $16 * 2 + 1 = 33$ . The correction term has a predetermined value, computed at design time with Equation 17. This step does not take any execution time.

**Step 4, Reducing the matrix:** In step 4, we perform the matrix reduction. For a 33-to-2 reduction, a total of 260 Carry Save Adders in 8 levels suffices.

**Step 5, Final addition:** Step 5 is the final 2-to-1 addition. This is done using a carry-lookahead scheme.

The last two steps each take one cycle, making the total number of cycles needed equal to 3.

## 5. Conclusions and future work

In this paper, we proposed a hardware unit capable of computing the SAD instruction. In particular, we considered two example implementations assuming 16x1 and a 16x16 pel blocks. The proposed sample implementation schemes compute the SAD in 3 or 4 cycles respectively.

We are able to perform the SAD in a small amount of cycles, because of the following two reasons:

- we have substituted complex operations (i.e subtract and absolute operation) with two simple operations (determining and inverting the smallest).
- we have substituted the subtractions and the accumulation operation by one multi-operand addition.

This speed advantage is especially beneficial for data-dependent algorithms, such as the three-step search algorithm. These algorithms need the SAD of the blocks in their first step to compute the addresses of the blocks in the second step.

There are however two problems which must be addressed in future research.

**Bandwidth** The bandwidth required to feed our unit is rather high. This means that some sort of on-chip cache is needed to store the reference frames.

**Data-alignment** The current-frames are always positioned on 16 byte boundaries, while most reference frames will not be aligned. This means that special hardware is needed to align the data.

## References

- [1] N. Ahmed, T. Natarajan, and K. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, pages 90–93, Jan 1974.
- [2] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34:349–356, May 1965.
- [3] B. Furht, J. Greenberg, and R. Westwater. *Motion Estimation Algorithms for Video Compression*. Kluwer Academic Publishers, 1997.
- [4] D. L. Gall. Mpeg: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.
- [5] J. R. Jain and A. K. Jain. Displacement measurement and its applications in interframe image coding. *IEEE Transactions on Communications*, COM-29(12):1799–1808, December 1981.
- [6] S. Kappagantula and K. Rao. Motion compensated predictive coding. In *Proc. Int. Tech. Symp. SPIE*, San Diego, CA, August 1983.
- [7] H. Kitjima. A symmetric cosine transform. *IEEE Transactions on Computers*, c-29(4):317–323, April 1980.
- [8] T. Koga, K. Linuma, A. Hirano, Y. Iijima, and T. Ishiguro. Motion-compensated interframe coding for video conferencing. In *NTC 81 Proc.*, pages G5.3.1–5, New Orleans, LA, December 1981.
- [9] R. Lim. High-speed multiplication and multiple summand addition. In *Proc. IEEE 4th Symp. Com. Arithmetic*, pages 149–153, October 1978.
- [10] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. LeGall. *MPEG Video Compression Standard*. Digital Multimedia Standard Series. Chapman and Hall, 1996.
- [11] R. Montoye, E. Hokenek, and S. Runyon. Design of the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research and Development*, 34(1):59–70, January 1990.
- [12] A. N. Netravali and B. G. Haskell. *Digital Pictures; Representation, Compression, and Standards*. Plenum Press, 1994.
- [13] P. Song and G. De Michelli. Circuits and architecture trade-offs for high-speed multiplication. *IEEE Journal of Solid-State Circuits*, SC-26(9):1184–1198, 1991.
- [14] R. Srinivasan and K. Rao. Predictive coding based on motion estimation. *IEEE Transactions on Communications*, COM-33:1011–1014, September 1985.
- [15] B. Tseng and W. Miller. On computing the discrete cosine transform. *IEEE Transactions on Computers*, c-27(10):966–968, Oct 1978.
- [16] S. Vassiliadis, J. Hoekstra, and H.-T. Chiu. Array multiplication scheme using (p,2) counters and pre-addition. *Electronics Letters*, 31(8):619–620, April 1995.
- [17] S. Vassiliadis, E. M. Schwarz, and D. J. Hanrahan. A general proof of overlapped multiple-bit scanning multiplications. *IEEE Transactions on Computers*, 38(2):172–183, February 1989.
- [18] S. Vassiliadis, E. M. Schwarz, and B. M. Sung. Hard-wired multipliers with encoded partial products. *IEEE Transactions on Computers*, 40(11):1181–1197, November 1991.
- [19] C. Wallace. A suggestion for parallel multipliers. *IEEE Trans. Electron. Comput.*, EC-13:14–17, 1964.