Journal of VLSI Signal Processing 28, 221–234, 2001

Multimedia Execution Hardware Accelerator*

EDWIN HAKKENNES AND STAMATIS VASSILIADIS
Department of Electrical Engineering, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands

Received October 1999; Revised April 2000

Abstract. In this paper we show that some expressions frequently used in multimedia applications can be formulated as a general add-multiply-add operation. We further show a hardwired implementation of the Add-Multiply-Add instruction which is no more complex than the multiplier implementation. Furthermore we show that two frequently motion estimation operations, the Sum and Mean of Absolute Differences, can be implemented in hardware requiring also approximately the same cycle time as the multiplication. We also show that our approach can be extended easily to provide the computation of the Sum and Mean of Absolute Difference of a 16×16 pixel block in no more than four machine cycles. Additionally we propose a codec hardwired mechanism for the Paeth predictor used in the Portable Network Standard (PNG) that requires at most two general purpose ALU cycles. We extend the paeth unit to include the median, maximum and minimum operations on three inputs with no additional cycle time and we also extend the Add-Multiply-Add unit to include the mean of three numbers. Finally we propose a multimedia hardware accelerator to accommodate all the proposed operations. The proposed unit is an extension of the multiply pipeline with ALU extensions with no extra stages added. The unit operates on 32 instructions in total.

Keywords: multimedia, hardware accelerators, multimedia architectures, subword parallelism, multimedia instruction set, vector architectures, multimedia processors, SIMD processors, compound instructions

1. Introduction

In order to improve the processing of multimedia applications, three types of processors have been investigated, namely:

Specialized multimedia standard processors: In this class of processors a specific standard such as MPEG2 is assumed, and for such a standard a processor is designed that uniquely performs this standards requirements. There exist several processors available that assume this approach. Examples of such processors can be found in [1, 2] and [3]

Specialized augmented multimedia processors: For this kind of processors, programmability is assumed and no restriction to a standard is imposed. That is the processing follows the usual general purpose paradigm of programmability and instructions set definition, with the caveat that all processor architectural requirements are imposed by multimedia processing needs, with additions to the architecture to permit stand alone processing. Example architectures for such a class include the Philips Trimedia architecture and processors [4] and the Texas Instruments Multimedia Video Processor (MVP) [5]

General purpose processors: This class of processors constitutes the third and final family of processors. In this scenario the general purpose machine is extended with coprocessing capabilities to improve the performance of multimedia formats. This approach follows the traditional extension oriented processing. That is in order to improve a certain application domain a general purpose processor architecture is

*This paper is dedicated to my mother, Lida Hakkennes, for the love and support she has given me over the years.

extended with new architectural features that allow the design of coprocessors specialized for the considered application. Examples for such extensions include the floating point and vector extension of general purpose computing. Regarding multimedia examples of this type of extensions include the Intel MMX [6] and the ALPHA MVI [7] extensions.

All three classes of processors provide improvements in multimedia processing and there is discussion as to which of the approaches should be followed. In this paper we propose mechanisms that provide some improvements to all possible types of processors by proposing new execution units. In particular we propose new instructions that:

- Can be added to existing units, e.g. a multiplier unit, with small additional cost and that
- Can be implemented by a separate "ALU like unit" with cycle times comparable to general purpose ALU/multiply cycles.

These additions we introduce are meant to be the initiation of new unit design and the proposed research direction is to solidify this multimedia unit(s) by adding instructions in the future with similar hardwired requirements.

The organization of the discussion is as follows: We first present the proposed unit designs. Consequently we discuss the architecture and a possible high level design of the multimedia hardware accelerator we propose. This paper is concluded with final remarks.

2. Add-Multiply-Add

It is widely accepted that true data dependencies [8] constitute one of the major obstacles for the improvement of speed in the computer based computational paradigm. In the recent past it has been shown that some important classes of true data dependencies for the general purpose computational paradigm can be resolved resulting in a substantial gain of performance [9, 10]. Regarding multimedia, the application analysis presented by Onion et al. [11] for applications such as 2D convolution, filters, FFT, DCT, histogram flattening, edge detection, etc., shows that more elaborate data dependency collapsing hardware is required to resolve most of the true data dependencies. More in particular the investigation in [11] has revealed that the following expressions (assuming the compiler can expose them) appear frequently in the benchmarks they have considered: Add-Add, Add-Multiply, Multiply-Add, Add-Multiply-Add, and Multiply-Add-Add.

In this section we consider fixed point, two's complement number representations and combine four of the above five exposed expressions in a single instruction. Our investigation strongly suggests that the expression we consider, in addition to the covering of the expressions or operations revealed in [11], can potentially be implemented using a parallel hardware organization and potentially be executed within two machine cycles. The previous conjecture is put in place by showing that the partial product matrix associated with the expression requires no more than $n+2$ rows, n being the number of bits of the input-values, which will most likely require no more cycle time than a fixed point multiplier in most implementations. That is we perform the following: Given that the two's complement notation $-X$ is almost equal to $\bar{X}$ (to be precise: $-X = \bar{X} + 1$),we rewrite the $(A \pm B) * C$ so that we compute it with $A * C + B * \bar{C}$ (or $A * C + \bar{B} * \bar{C}$), in which C chooses between passing A or $B(\bar{B})$. As an example, we will compute $(A - B) * C$. We can do so by rewriting it to: $(A - B) * C = A * C - B * C = A * C + B * (-C)$. In this last equation, we use the two's complement rule in that we rewrite $-C$ as $\bar{C} + 1$. The hot one is not added directly in this case. This makes $(A - B) * C = A * C + B * (\bar{C} + 1) = A * C + B * \bar{C} + B$.

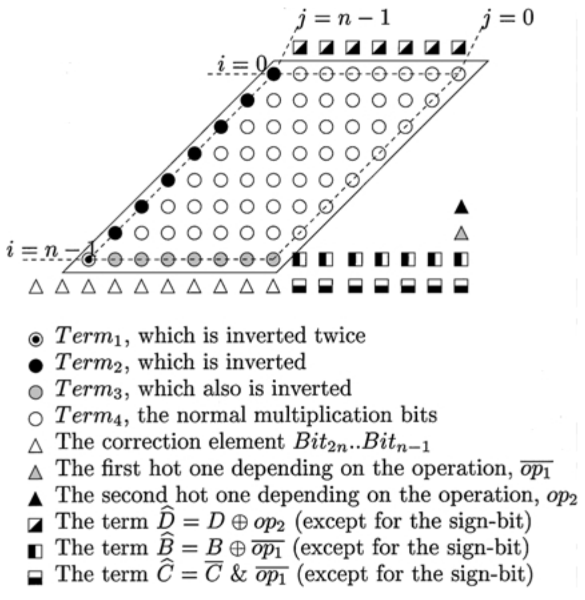
This means that for each bit of C, we have to add either A (if the bit of C is one) or B (if it is zero) to the partial product matrix, in stead of both or neither of them. C is used as a set of select signals to choose between them. Normally, if a certain bit of C is one, we would have to feed A and $-B$ to the partial product matrix on that position. If that bit would have been zero, we would have to add nothing. The maximum number of partial product rows would be equal to two times the number of bits of C. In our scheme, each row is always filled with either A or B, and the number of rows equals the number of bits of C and one extra row for B. The other three instances of $(A \pm B) * C \pm D$ can be rewritten in a similar way. This yields the expressions of Table 1. In the Table op_1 and op_2 (the operations) are the control signals to the hardware to indicate what needs to be computed. Equal to zero implies addition and equal to one implies subtraction.

The four instances can be combined into a single expression as follows:

$$\begin{aligned}&(A\ oper_1 B) * C\ oper_2 D\\ &\quad = A * C + \hat{B} * \bar{C} + \hat{B} + \hat{C} + \hat{D} + \overline{op_1} + op_2\end{aligned}$$

Table 1. The possible operations.

op_1	op_2	Expression	Expression used for the Inversion Selection Technique
0	0	$(A+B)*C+D$	$A*C+\bar{B}*\bar{C}+\bar{B}+\bar{C}+D+1$
0	1	$(A+B)*C-D$	$A*C+\bar{B}*\bar{C}+\bar{B}+\bar{C}+\bar{D}+2$
1	0	$(A-B)*C+D$	$A*C+B*\bar{C}+B+D$
1	1	$(A-B)*C-D$	$A*C+B*\bar{C}+B+\bar{D}+1$



j = n − 1
j = 0
i = 0
i = n − 1
⊚ Term₁, which is inverted twice
● Term₂, which is inverted
● Term₃, which also is inverted
○ Term₄, the normal multiplication bits
△ The correction element Bit₂ₙ..Bitₙ₋₁
△ The first hot one depending on the operation, op₁ (overlined)
▲ The second hot one depending on the operation, op₂
◪ The term D̂ = D ⊕ op₂ (except for the sign-bit)
◧ The term B̂ = B ⊕ op₁ (overlined) (except for the sign-bit)
⬓ The term Ĉ = C̄ & op₁ (overlined) (except for the sign-bit)

Figure 1. Graphical representation of the Add-Multiply-Add.

in which $\hat{B} = B \oplus \overline{op_1}$, $\hat{C} = \bar{C}\ \&\ \overline{op_1}$ and $\hat{D} = D \oplus op_2$. The graphical organization of this structure is described in Fig. 1.

In this section, we have introduced a technique which uses the fact that the partial products generated by $c_i * A$ and those generated by $\bar{c_i} * B$ can be added by simply OR-ing them together. This results in $n+2$ partial product lines, that can be summed up in an arbitrary counter structure. In the next section, we will introduce the Sum of Absolute Differences unit, which jointly uses the partial product reduction tree.

3. Sum and Mean Absolute Difference

In this section we proceed by investigating the Sum of Absolute Differences (SAD) operation. The SAD operation is used as a metric in determining the closeness of two blocks of successive frames of a video sequence [12]. In general, the SAD is computed in software using the following expression.

$$SAD(x, y, r, s) \\ \times \sum_{i=0}^{i=15} \sum_{j=0}^{j=15} |A_{(x+i,y+j)} - B_{((x+r)+1,(y+j)+s)}|$$

In this equation, (x, y) are the coordinates of the upper-left pixel of the original block and (r, s) is the motion vector of which the SAD will be computed.

A direct approach for the computation of the SAD consists of the following steps:

- Compute $(A_i - B_i)$ for all 16×16 pixels in the two blocks A and B.
- Determine which $A_i - B_i$ are less that 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 16×16 absolute values.

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. Our solution to this problem is as follows. By determining the smallest of both operands and subtracting it from a constant, which is greater or equal than the maximum value of a pixel, 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 16×1 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.

Pass proper operands 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 the constant $(2^8 - 1 = 255)$.

The above two steps can be carried out in parallel for 16 pels. This results in 32 eight-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.

Reduce the 33 *rows to* 2: The resulting 32 rows are passed to the adder tree together with the correction-term. These 33 rows are reduced to 2 rows by using a counter scheme, see for example [13].

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

Figure 2 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. Figure 3 shows a graphical representation of a 16×1 unit, that is a unit operating 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 16×16 pel blocks. In this case,

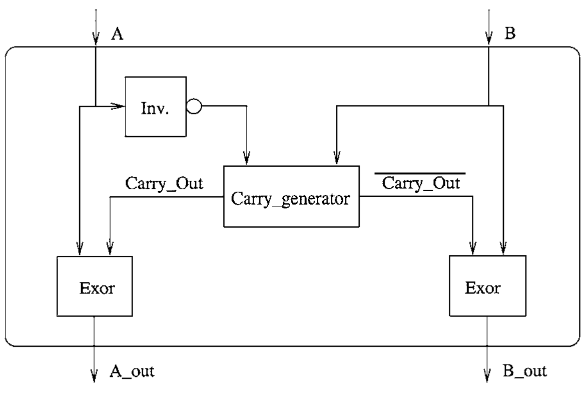


Figure 2. First two computation steps of the SAD.

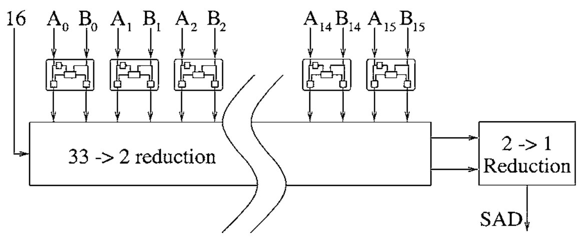

16
A_0 B_0 A_1 B_1 A_2 B_2 A_{14} B_{14} A_{15} B_{15}
33 -> 2 reduction
2 -> 1 Reduction
SAD

Figure 3. 16×1 block SAD computation.

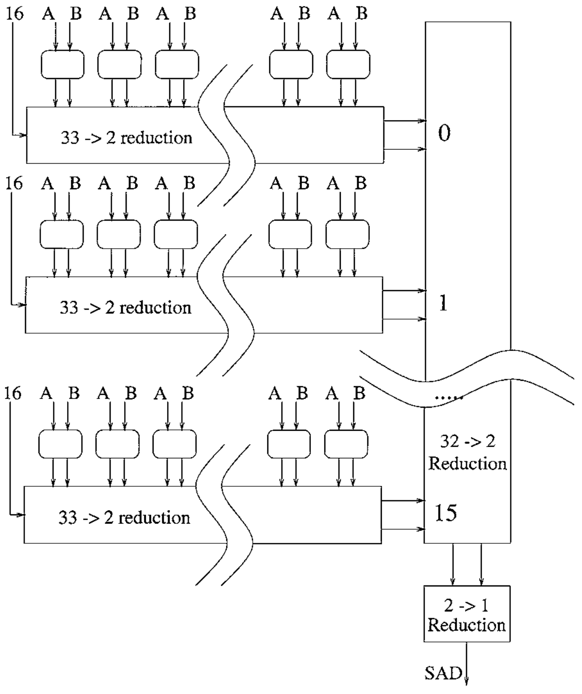


Figure 4. A 16×16 pel SAD computation unit.

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 Fig. 4.

4. Paeth Prediction and Coding

This section describes an execution unit capable of computing the Paeth Predictor [14], as used in the Portable Network Graphics (PNG) standard. The hardwired predictor/codec is capable of computing three different quantities:

```
int predict ( int $a$, $b$, $c$ )
{
int $p$, $pa$, $pb$, $pc$
$p = a + b - c$                                  /* this is the initial estimate */
$pa$ = abs( $p - a$ )                          /* distance of each member to the */
$pb$ = abs( $p - b$ )                                       /* initial estimate */
$pc$ = abs( $p - c$ )
if ( $pa \leq pb$ ) and ( $pa \leq pc$ ) return ( $a$ )                   /* return */
else if ( $pb \leq pc$ ) return ( $b$ )                 /* element nearest to p, */
return ( $c$ )                                        /* in a,b,c tie-break order */
}
```

Figure 5. The PNG Paeth encoding routine.

- the Paeth predictor of three inputs,
- the difference of the current pixel and the Paeth predictor of the other three inputs (Coding),
- the sum of the coded input and the Paeth predictor of the other three inputs (Decoding).

The Paeth predictor is normally computed in software. The routine used for this is defined in the PNG standard and shown here as Fig. 5.

To compute the Paeth predictor on a SunSparc 10 processor, 21 instructions are needed, including 6 branches. To improve the execution speed, we propose a hardwired Paeth prediction unit, which computes the Paeth predictor of a set of three input values in at most two machine cycles. As the predictor is selected from the input values, and the critical path is the control of the output selectors, we can also precompute the difference or the sum of a fourth input (d) with each of the three inputs. This means that we are also able to compute the coded or decoded value within the same machine cycles. We compute the predictor by directly computing the distances of the initial estimator (p) to each input, and selecting the input which has the smallest distance. If we use the codec structure, we select the current pixel +/− the inputs. The proposed scheme operates as follows:

- Direct computation of the distance of each of the inputs a, b and c to the initial estimate.
- Compare these distances using Carry-generators.
- Select the input with the lowest distance.

For the codec unit, we precalculate three temporal results, which are the sum (decoding) or difference (encoding) of the current pixel and each of the inputs, and select one of these precalculated values. Using this scheme, the critical path is not affected, and yet the number of executed operations is increased.

-	c	b	-
-	a	d	·

Figure 6. The definition of the position of the a, b, c and d input pixels.

Figure 6 gives the naming conventions of the Paeth predictor within the PNG standard. The pixels denoted with "-" have been dealt with in the past and are no longer of interest, the pixel denoted with "d" is the current pixel and the pixels denoted with "." will be transmitted in the future.

In order to propose a hardware implementation of the routine in Fig. 5, we rewrite it. In Fig. 7 we can

```
int predict ( int a, b, c )
{
int $pas$, $pbs$, $pcs$
bool $Test\_1$ $Test\_2$ $Test\_3$
$pas = ( b - c )$
$pbs = ( a - c )$
$pcs = ( a + b - 2c )$
$Test\_1 = ( |pas| \leq |pbs| )$
$Test\_2 = ( |pas| \leq |pcs| )$
$Test\_3 = ( |pbs| \leq |pcs| )$
If $Test\_1$ and $Test\_2$ return ( $a$ )
else if $Test\_3$ return ( $b$ )
return ( $c$ )
}
```

Figure 7. Simplified Paeth-Algorithm.

distinguish three steps. These steps will be exposed in the hardware implementation. In the first step, we compute pas, pbs and pcs. These are the signed variants of pa, pb and pc, denoted as 10 bit, two's complement intermediate numbers. In the second step, we compute Test_1,Test_2 and Test_3. The third step is the selection of the right input as the output.

The computation of pas, pbs and pcs is done by an adder circuit [15]. As the range of the unsigned bytes a, b and c is from 0 to $(2^8 - 1)$, the variable pcs can range from $0 + 0 - 2 * (2^8 - 1)$ to $2^8 - 1 + 2^8 - 1 - 2 * 0$, which is from $-2^9 + 2$ to $2^9 - 2$. This range is just covered by a 10-bit two's complement number, which ranges from -2^9 to $2^9 - 1$. Although pas and pbs are representable as 9-bit two's complement numbers, we also represent them as 10-bit two's complement numbers to facilitate the subsequent comparisons and to preserve the regularity of the unit. In binary notation, this leads to the following additions:

$$pas = (b - c) = (00b + 11\bar{c} + 1) \tag{1}$$

$$pbs = (a - c) = (00a + 11\bar{c} + 1) \tag{2}$$

$$pcs = (a + b - 2c) = (00a + 00b + 1\bar{c}1 + 1) \tag{3}$$

As can be concluded from the previous three formulas, a sign-extension takes place to make all the input numbers 10 bits long. The negative value of $-c$ is computed using an inversion and the addition of a hot-one.

The computation of pcs involves a 3 to 1 addition, where one of the operands is shifted one position to the left (multiplied by 2). This is accommodated by using an extra level of Full-Adders, which performs a carry-save addition of the three operands, resulting in a sum and a carry word. These are then added in a 2–1 binary adder.

After the computation of pas, pbs and pcs, there are several ways to compute *Test_1*, *Test_2* and *Test_3*. To compute *Test_1* we have to find out whether $|pas| \leq |pbs|$. We can use a carry-based comparison of pas and pbs. This means that we add them in some form and that the resulting carry reflects whether the inequality was true or false.

We first have to adjust the signs of pas and pbs. In order to compare them, we check whether $|pbs| - |pas| \geq 0$. In order to facilitate this, we have to make sure that pbs has a positive sign and pas has a negative sign. If the sign is opposite, we invert the operand and add a hot-one to the result. This hot one is taken care off in the addition. If both pas and pbs are inverted, there are two hot ones. The carry-generator needs a special

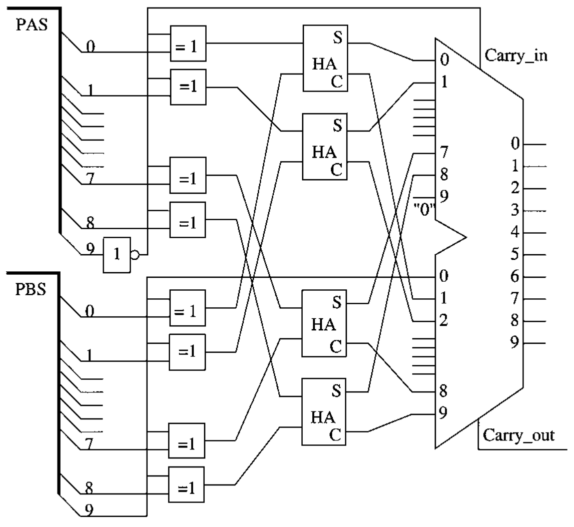


Figure 8. The computation of Test_1.

structure to accommodate this. This is implemented as a layer of half-adders, as shown in Fig. 8. It should be noted that in the figure bit number 9 is the Most Significant Bit, the Sign-bit. The outputs 0 to 9 are not used, and need not be computed. They are only shown for clarity.

The test $|pbs| - |pas| \geq 0$ is now modified to $pb_{pos} + pa_{neg} \geq 0$. The test for carry_out is basically the test for result $\geq 2^{10}$. We have to keep in mind that the sign-bit of pas is interpreted as a positive number here, with value 2^9 in stead of -2^9. We are therefore essentially adding 2^{10}. The binary summation is therefore: $pa_{neg} + 2^{10} + pb_{pos} \geq 2^{10}$ So if $pb_{pos} + pa_{neg} \geq 0$ the binary addition $pb_{pos} + pa_{neg}$ generates a carry_out. Figure 8 gives a graphical representation of the unit which computes Test_1 from pas and pbs. Test_2 and Test_3 are computed using similar logic.

A possible extension of this unit is the extension to a Paeth codec, which has not only the a, b and c input, but also uses the to be encoded or decoded pixel, d. When the prediction is known, the coding is simply subtracting the prediction from the actual data, that is $Coded\_Data = Actual\_Data - Prediction = d - pred$. Decoding is done by adding the prediction and the received coded data, that is $Original\_Data = Coded\_Data + Prediction = d + pred$ These operations are done modulo 256, as defined in the standard [14]. The program-notation for this is given in Fig. 9. A graphical representation of the implementation of this optimized execution unit is shown in Fig. 10.

We have presented a sample implementation of a 4-input Paeth codec, capable of coding and decoding

```
int codec ( int $a$, $b$, $c$, $d$, $mode$)
{
int $pas$, $pbs$, $pcs$
unsigned int $Res\_a$, $Res\_b$, $Res\_c$
bool $Test\_1$ $Test\_2$ $Test\_3$

$pas = ( b - c )$
$pbs = ( a - c )$
$pcs = ( a + b - 2c )$
$Res\_a = (d + (1 - 2 * mode) * a)$
$Res\_b = (d + (1 - 2 * mode) * b)$
$Res\_c = (d + (1 - 2 * mode) * c)$

$Test\_1 = ( |pas| \leq |pbs| )$
$Test\_2 = ( |pas| \leq |pcs| )$
$Test\_3 = ( |pbs| \leq |pcs| )$

If $Test\_1$ and $Test\_2$ return ( $Res\_a$ )
else if $Test\_3$ return ( $Res\_b$ )
return ( $Res\_c$ )
}
```

Figure 9. The optimized Paeth Codec Algorithm.

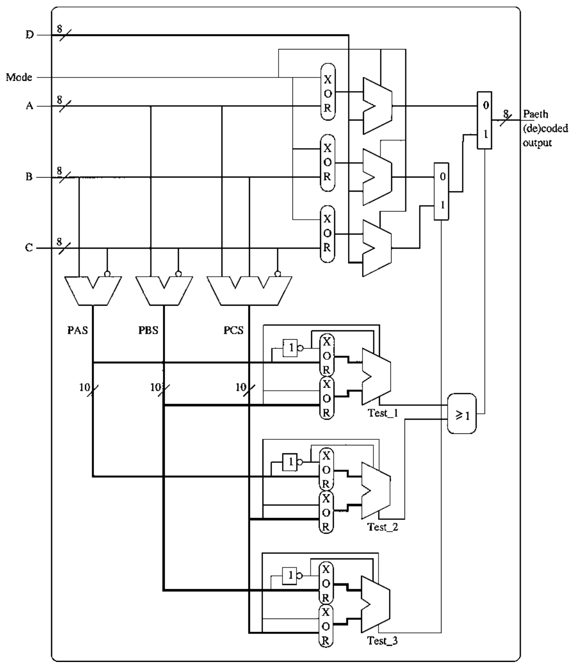


Figure 10. Optimized implementation of the Paeth Codec.

images using the Paeth Predictor as described in the PNG standard.

The critical path of the codec unit is the control of the output multiplexers. It is basically two levels of binary adders long. The path to the data-input of the multiplexers is only one level of adders long. We estimate that the speed of this unit equals that of a standard two-cycle multiply unit or requires two ALU cycles. This estimation is based on the fact that the critical path is basically two times an adder delay, which is bounded by an ALU cycle time.

5. Median, Max, Min, and Mean

In this section we will introduce an extension of the Paeth unit so that is can additionally compute the Median of three inputs, used in video-deinterlacing. We will show that using the same paeth extended logic, it is trivial to compute also the maximum and minimum of the three inputs. Furthermore, we introduce an extension of the Add-Multiply-Add unit, in order to compute the Mean of three inputs.

The median is defined as the middle value of a sorted list of input values. For the median computation we propose a hardware unit also described in [16] which calculates the median in two steps as follows:

Step 1: (*Compute three inequations* ($a \geq b$, $b \geq c$, $a \geq c$)

As indicated earlier we compute the three inequations: $a \geq b; b \geq c; a \geq c$ in the first step. As the Paeth computation uses the ten bits wide two's complement notation of both $b - c$ and $a - c$ as intermediate results, we will use the same hardware. As an example we show that the testing of $b \geq c$ is equivalent to the testing whether $b - c \geq 0$. This is a simple test on the sign-bit of the result of the subtraction. That is the inequation holds true if the sign-bit is zero. The carry-out of the addition is the inverse of this sign-bit. That means that the carry-out is one if-and-only-if the sign-bit is zero. The computation of the first inequation is performed in a similar way, we compute the ten-bit, two's complement number $a - b$ and use the carry-out of that subtraction.

To summarize, we only have to add one carry-chain generator to the Paeth logic to perform the computation of the $a \geq b$ inequation. For the other two inequations the logic is already there. Let $Test_1$, $Test_2$ and $Test_3$ represent the result of $a \geq b$, $b \geq c$ and $a \geq c$ respectively:

$$Test_1 \Longleftrightarrow a \geq b \Longleftrightarrow 00a + 11\bar{b} + 1 \geq 2^{n+2} \quad (4)$$

$$Test_2 \Longleftrightarrow b \geq c \Longleftrightarrow 00b + 11\bar{c} + 1 \geq 2^{n+2} \quad (5)$$

$$Test_3 \Longleftrightarrow a \geq c \Longleftrightarrow 00a + 11\bar{c} + 1 \geq 2^{n+2} \quad (6)$$

Note that the input-numbers are extended with two bits to ten bit inputs. This stems from the paeth unit, which

Table 2. Requirements for Min, Max and Median.

$a \geq b$ $Test_1$	$b \geq c$ $Test_2$	$a \geq c$ $Test_3$	Min	Median	Max
True	True	True	c	b	a
True	True	False	–	–	–
True	False	True	b	c	a
True	False	False	b	a	c
False	True	True	c	a	b
False	True	False	a	c	b
False	False	True	–	–	–
False	False	False	a	b	c

needs the 10-bit result of $b - c$ and $a - c$. We could also pick the 9th bit of this subtraction, thereby reducing the critical path a little.

Step 2: (*Select the median*) Based upon the three resulting carries, we select one of the three operands as the result, using Table 2. The Table 2 has six columns. The first three describe the eight output combinations of the comparisons and the last three columns determine which of the operands needs to be chosen as the minimum, median and maximum respectively. Note that two out of the eight combinations cannot occur, as there is no set of a, b, and c which for which these conditions hold true. One would need $a \geq b$, $b \geq c$ and $a \ngeq c$. From the first two conditions, one can deduce that $a \geq c$, which means that if $Test_1$ and $Test_2$ are True, $Test_3$ would also have to result in True.

In order to implement the median(a, b, c) instruction we note that the two steps require three carry generators and the implementation of the logic implied from Table 2. All median requirements can be performed from the logic depicted in Fig. 11. This can be proven by the following: The first part of the logic (the carry-generators) determine which of the three conditions ($Test_1$, $Test_2$ and $Test_3$) hold true. That is carry generator 1 determines $Test_1$ ($a \geq b$), carry generator 2 determines $Test_2$ ($b \geq c$) and carry generator 3 determines $Test_3$ ($a \geq c$).

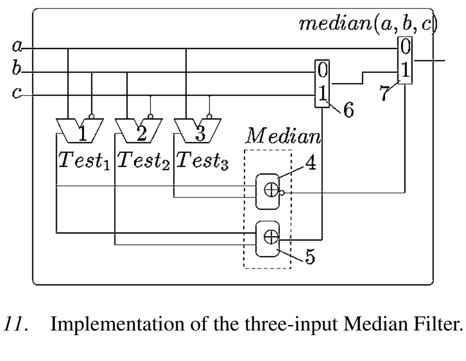


Figure 11. Implementation of the three-input Median Filter.

Clearly, once we have computed the three inequations, it is trivial to select the maximum and the minimum of the three inputs. In order to do so, we extend the unit with an extra set of multiplexers, which controls the control-signals to the output multiplexers. To compute the min and max of three inputs, we need to determine which input to select on the basis of the three computed test signals. These selection requirements are deduced form Table 2 which determines which operand to select as min and max output. In order to compute the mean of three numbers and avoid the division operation, we perform the following: Assume that the input numbers, a, b and c are in two's complement notation and that each number is representable in 8 bits, then:

Step 1: Compute X and Y as the carry-save sum of $A + B + C$. In this step, three eight-bit numbers are converted into two nine-bit numbers, using a carry-save addition. Both are sign extend to ten-bit, two's complement numbers.

Step 2: Compute $((X + Y)*341 + 511)$ using a ten-bit input Add-Multiply-Add unit described in Chapter 2. The result is a 21-bit number in two's complement notation. Note that $\frac{341}{1024} \approx \frac{1}{3}$[1], and $\frac{511}{1024}$ is as close as we can come to $\frac{1}{2}$ using 10-bit 2's complement numbers.

Step 3: Shift the result 10 positions to the right. (discarding the lower 10, and the upper three bits).

Proof of correctness: Assume the result of the mean-operation is P. This implies that the sum of A, B and C equals either $3P$, $3P + 1$ or $3P - 1$ (as the mean is equal to one third of this sum, rounded to the nearest whole number).

If we undo step 3, the rounding, we observe that the result of step 2 is $P * 1024 + rest$, where $0 \leq rest \leq 1023$. This means we have to prove that rest remains within its range for all P we can expect as output. As the range of the mean of any number of inputs is equal to the range of the input numbers, we have to prove that for all values P can take in two's complement, the rest is within bounds. Consequently we have to prove whether rest remains within bounds for all P in the output range, taking the rounding into account. That is $rest = (SUM) * 341 + 511 - P * 1024$, where SUM can be $P + 1$, P or $P - 1$. is $P * 1024 + rest$ equals $(3P + 1) * 341 + 511$ for all input combinations.

There are three cases to consider:
Case 1: Sum = 3P

$$P * 1024 + rest = 3P * 341 + 511$$
$$P + rest = 511$$
$$rest \in \{0...1023\}$$
$$P \in \{-511...512\}$$

Case 2: Sum = 3P + 1

$$P * 1024 + rest = (3P + 1) * 341 + 511$$
$$P + rest = 511 + 341 = 852$$
$$rest \in \{0...1023\}$$
$$P \in \{-171...852\}$$

Case 3: Sum = 3P −1

$$P * 1024 + rest = (3P - 1) * 341 + 511$$
$$P + rest = 511 - 341 = 170$$
$$rest \in \{0...1023\}$$
$$P \in \{-853...170\}$$

The intersection of these three sets is $P \in \{-171 ...170\}$. This covers the range of the input values, $\{-128...127\}$, so the result will be correct for all input combinations.

Figure 12 give a graphical representation of the Mean3 Unit, implemented using a Carry-Save adder and the Add-Multiply-Add unit. It is noted that there is additional hardware to perform a carry-save operation may produce a critical path delay problem (the extra logic required is the XOR of 3 inputs). If such a problem occurs than an additional cycle should be added in order to perform the mean operation.

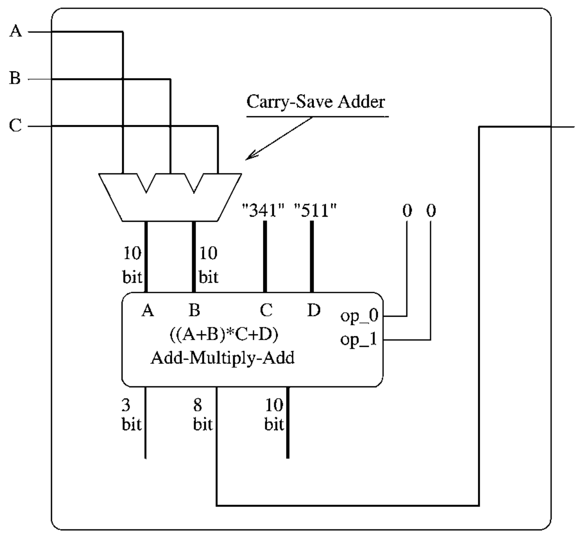


Figure 12. Implementation of the three-input Mean unit using an Add-Multiply-Add unit.

Note: This subsection's primary concern was in accommodating the mean of three inputs with the modification of the add-multiply-add unit. However, this is not strictly necessary as if we have only an AMA unit available, we can still use this method, using three instructions. The first instruction would be an add of A = A + B. The second would be the AMA operation as (A + C)*341 + 511. The third operation is a right-shift over 10 positions. Note however that this also will require modifications, as we need a 10-bit AMA unit for 8-bit inputs. Special care has to be taken not to generate an overflow, as the intermediate result is a 20-bit number.

6. Putting it all Together

In building a multimedia extension to a general purpose processor, it might be more convenient to follow the general processor paradigm which has few units with each unit used by multiple instructions. That is it is of interest to combine the units we have proposed into one single execution unit.

We begin by noting that all units operate on more than the usual two operands (with the exception of some of the simpler instructions of the Add-Multiply-Add unit and the SAD instruction, but the SAD would use any extra operands supplied). This suggests combining the units in one general unit, which has at least four inputs. If two instruction(slots) are needed to specify the operands for the unit, we could think of delivering two results as well. This would only need some extra logic on the output-stage. Note that not all combinations would be possible. Another solution to this problem is the specification of register pairs. This register addressing mode implies that only two source registers are specified, and that the other two source registers are implicit. For example the machine instruction ama r2, r6, r5 would specify that r5 gets the result of (r2 + r3)*r6 + r7. That is whenever a register is specified as a source register, its "upper neighbor" is also specified implicitly. The "upper neighbor" of a register is the subsequent register. The basic data-types the proposed architecture supports are:

unsigned byte (*8 bit*) This is frequently used for pixel-data, where it contains one color-component, or the luminance of a pixel.

Table 3. Instruction set.

1 input	2 input	3 input	4 input
Add-Multiply-Add unit			
Negate	Add	Add-Multiply	Add-Multiply-Add
	Subtract	Subtract-Multiply	Add-Multiply-Subtract
	Multiply	Multiply-Add	Subtract-Multiply-Add
		Multiply-Subtract	Subtract-Multiply-Subtract
		Add-Add	
		Add-Sub	
		Sub-Add	
		Sub-Sub	
		Mean	
Sum-Absolute-Difference unit			
	SAD_2	SAD_Accumulate	SAD_4
Paeth unit			
		Paeth_PNG	Paeth_PNG_Encode
			Paeth_PNG_Decode
Minimum Maximum Median unit			
		Min	Min encode
			Min decode
		Max	Max encode
			Max decode
		Median	Median encode
			Median decode

signed byte (8 *bit*) This is used for intermediate results.
unsigned half-word (16 *bit*) This is also used for pixel-data in very-high color-depths. The PNG standard supports this, but states that the 16 bits should be treated as two independent bytes.
signed half-word (16 *bit*) This is used for audio samples.
unsigned word (32 *bit*) This is used for intermediate results.
signed word (32 *bit*) This is used for intermediate results.
unsigned double-word (64 *bit*) This is used for intermediate results.
signed double-word (64 *bit*) This is used for intermediate results.

The base of our representation is 2 and the bit enumeration is from high order to low (that is the most significant bit has the highest number and the least significant bit has the number 0). We note that all signed numbers are represented as two's complement numbers. The instructions we support can be found in Table 3, from which it can be observed that the instructions have been divided into 4 categories and every category into instructions. Furthermore the instructions are divided according to the number of inputs they require. More specifics on the instructions can be found in [17].

A possible implementation of the unit is describe here using a general dataflow. As control logic is dependent on various other factors, such as decoding, accessing registers, structural hazards etc, it is not discussed here. Also no specific unit logic implementations details, such as adders, multipliers and carry-logic etc, are presented and they are left to individual designers.

The general dataflow of the execution unit we propose is described in Fig. 13. The unit is designed to assume four inputs denoted by the source registers **rs1a**, **rs1b**, **rs2a**, and **rs2b**. Control logic signals are added to determine the correct flow of data. It comprises a number of blocks, denoted as sub-units, which are described in the text to follow.

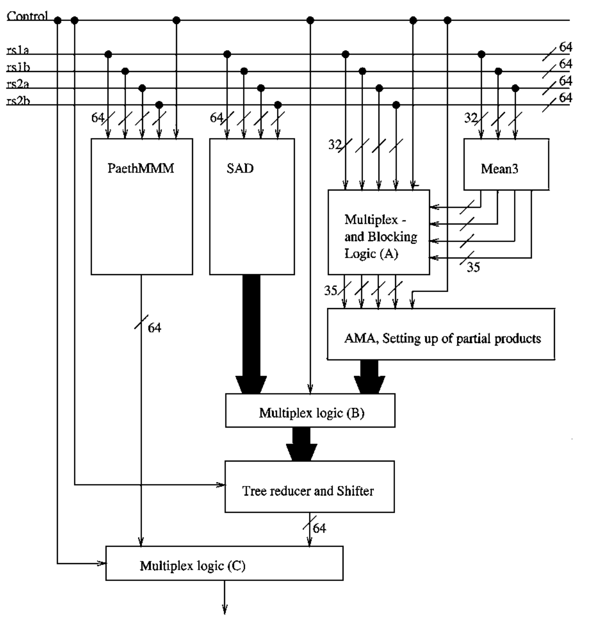


Figure 13. The combined execution unit.

The sub-unit Paeth MMM, computing the Paeth, Median, Minimum and Maximum related operations is described in Fig. 14. In this figure the data inputs are called A, B, C and D. These are connected to **rs1a**, **rs1b**, **rs2a** and **rs2b** respectively. The three control signals, op_1, op_2 and mode are to used to control the operation of the unit. These three inputs are decoded from the opcode. All data-inputs A, B, C, and D are 64 bits wide. These inputs can be split into several smaller parts, namely 8 times 8 bits, 4 times 16 or two times 32 bits. For clarity, this is not shown in the figure. Furthermore, the inputs can be signed or unsigned. The unit was originally designed for unsigned inputs. A trivial adaptation makes it possible for it to operate on signed inputs in two's complement as well. We should note that this is not defined for the Paeth operations, as the PNG standard [14] explicitly states that all operations are carried out on bytes, which are to be interpreted as unsigned.

We show the operation for an eight bit slice of the Paeth MMM unit. This means that 8 of these units are present and operate in parallel. In order to operate on larger data-types, several signals have to be chained together. Note also that we only show the operation for unsigned data. The operation of the Paeth MMM unit is as follows: Carry-generator 1 and Adders 2 and 3 are used for the comparison of inputs A, B, and C. Based on these results, *MTest_1*, *MTest_2* and *MTest_3*, the dashed block denoted as Min, Max and Median determine which of the inputs is to be selected. At the same time, adders 2, 3 and 4 compute the intermediate results *pas*, *pbs* and *pcs*. These are compared in blocks 8–10, which determines which of them has the lowest absolute value, as described in Section 4. This results in three similar test-signals, *Test_1*, *Test_2* and *Test_3*. The dashed block denoted as Paeth determines which of A, B, and C is to be selected as the Paeth Predictor. Selectors 11 and 12 are used to determine which of the four functions is shown on the outputs and selectors 13 and 14 select the actual value. This can be the value A, B or C if input D is blocked, or (D − A), (D − B), (D − C) in case of encoding or (D + A), (D + B), (D + C) in case of decoding. Using these precalculated values it is possible to compute the Paeth-encoded value in the same number of cycles needed as for the Paeth predictor. Note that these additions and subtractions are done modulo 2^n, in accordance with the PNG specification [14]. Finally we state that in case of an instruction is executed on the Paeth MMM unit, the bottom multiplexer (C) of the combined execution unit (Fig. 13) needs to select its left input.

We will assume the Mean3 instructions to work on 32-bit values only. In order to accommodate unsigned numbers, we will internally use 33 bits. That is we sign-extend with a zero if we operate on unsigned numbers and with the proper sign if we operate on signed numbers. As we need two extra bits in order to get the required precision, we need an Add-Multiply-Add unit which is 35 bits wide. The Mean3 part of Fig. 13 is represented in more detail in Fig. 15. It supplies the internal Add-Multiply-Add unit with the intermediate results of adding **rs1a**, **rs1b** and **rs2a**, and the constants $2^{35}/3$ and $2^{34} - 1$. These four inputs are used in an add-multiply-add instruction, and the result is shifted to the right over 35 bits.

The Add-Multiply-Add logic needed for the MEAN3 instruction is also available for general use. The 35 bit wide unit can support both signed and unsigned 32-bit words. For unsigned operation, we sign-extend the 32-bit inputs with zero's and for signed operation we sign-extend the 32-bit inputs with their own sign-bit. After that we can treat the inputs as signed, 35 bit values.

The Add-Multiply-Add sub-unit is represented in Fig. 1. The inputs of the Add-Multiply-Add unit come from multiplexer (A), which chooses between the output of the the Mean3 unit, depicted in Fig. 15, or the direct inputs of the total unit. This multiplexer can also

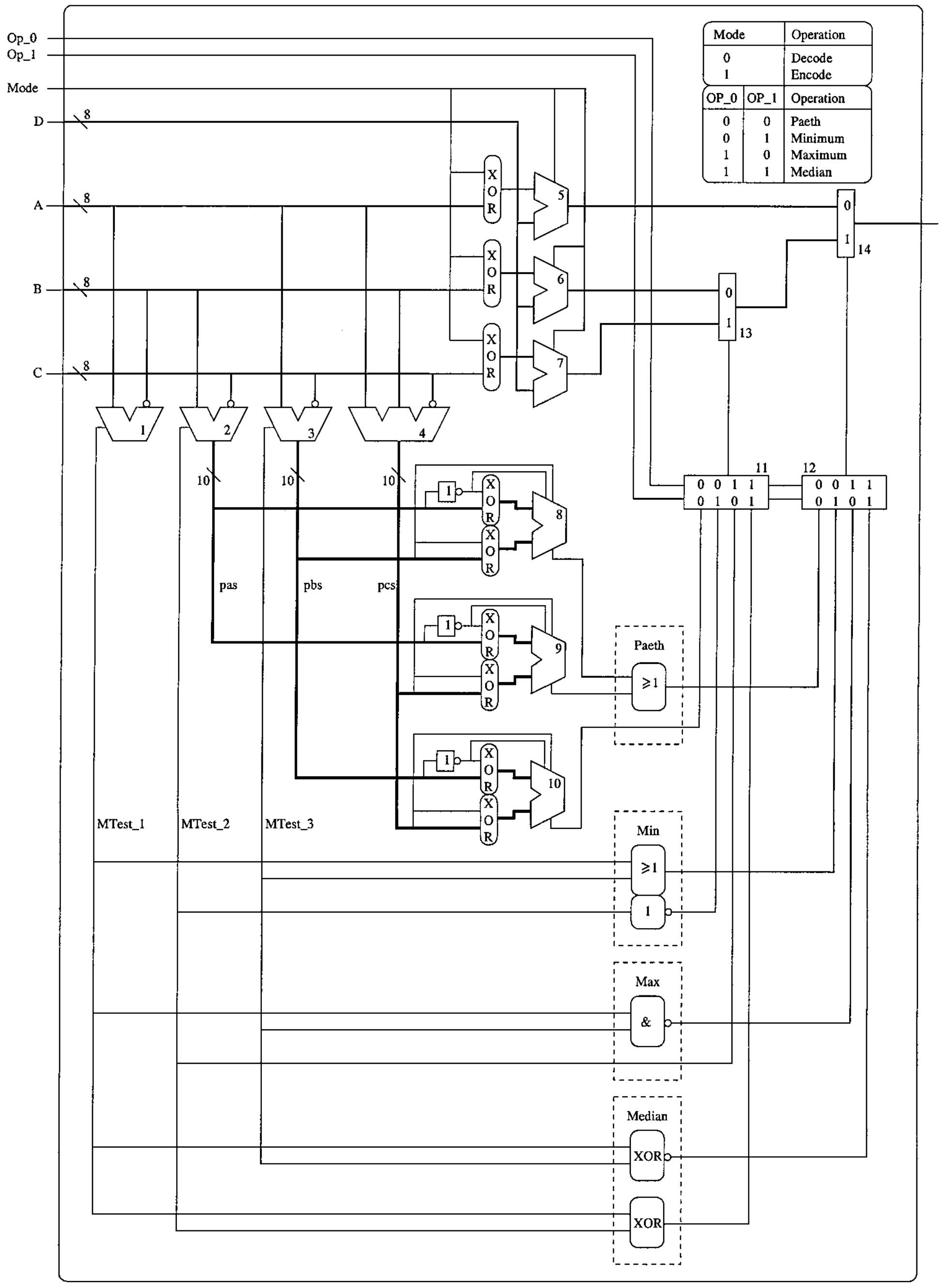


Figure 14. The total Paeth MMM unit.

block one or more of the inputs, thereby enabling the computation of several simpler expressions, such as Add-Add and Multiply-Add.

The Add-Multiply-Add unit can operate in two different ways, explained in Section 2. The Inversion Selection Technique uses the fact that $-X$ is almost equal to $\bar{X}$. To be precise, $-X = \bar{X} + 1$. Therefore we can rewrite for instance $(A - B) * C + D$ as $(A * C) + (B * -C) + D$, which can be rewritten as $(A * C) + (B * (\bar{C} + 1)) + D == (A * C) + (B * \bar{C}) + B + D$. As a given bit of C can only be 1 or 0, and the corresponding bit of $\bar{C}$ is 0 respectively 0, we

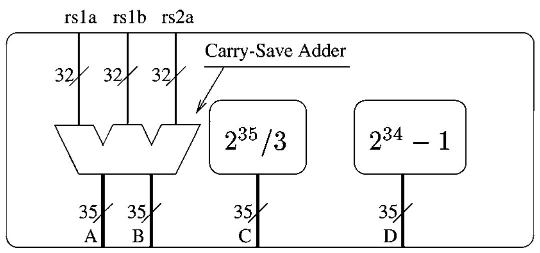


Figure 15. Graphical representation of the Mean-part of Fig. 13.

are essentially multiplexing A and B instead of adding them together.

The SAD operation takes place on 16 unsigned byte pixels in parallel, therefore using 32 bytes as input. For each pair of input bytes, it is determined which of them is the smallest. This is done by inverting the first operand and adding the result to the second operand. The carry out of this addition determines which of them is the smallest. After that the smallest of each pair of inputs is inverted. Note that the figure shows the configuration for unsigned bytes as input. By coupling unit 0 and 1 it is possible to operate on 8 couples of halfwords as well. In that case the carry-out of unit 0 is used as carry-in for unit 1. The carry-out of unit 1 is then used to determine which of the operands to invert. The constant 16 should be changed to 8 in that case.

7. Final Remarks and Conclusions

The present section is dedicated to some final remarks regarding our investigation. In brief the following has been achieved.

- We have shown that a number of true data dependencies that have been identified to be present in large percentages in embedded system applications [11] can be captured by an unique expression: $(A \pm B) * C \pm D$. Consequently, assuming two's complement representation, we propose two schemes for the implementation of such an expression. Both schemes require no more machine cycles than the multiplication of two numbers.
- For a very frequent motion estimation operation, the Sum of Absolute Differences (SAD), we proposed a vector instruction and we investigated possible implementations for such an instruction. Assuming a machine cycle comparable to the cycle of a two cycle multiply, we have shown that for a block of 16×1 or 16×16, the SAD operation can be performed in 3 or 4 machine cycles respectively.
- We propose a hardwired solution for the paeth predictor for the Portable Networks Graphics standard and proposed a hardware Paeth codec, capable of computing three different quantities: the Paeth predictor of three inputs, the difference of the current pixel and the Paeth predictor of the other inputs (Coding), and the sum of the coded input and the Paeth predictor of the other three inputs (Decoding), within two cycles, where a cycle is comparable to a general purpose ALU cycle. Depending on the mode of operation, the proposed mechanism produces the predictor or the (de/en)-coded pixel value.
- We have shown that without additional pipeline stage time penalties an extension of the Paeth unit is possible so that is can additionally compute the Median of three inputs. This median is used in video-deinterlacing, which is needed when displaying normal video on a non-interlaced computer screen or a modern, high-end television set. We have also shown that the median operation can be computed by itself in one machine cycle.
- We have introduced a number of new instructions and shown that these instructions can be implemented with trivial additions to the multiply (multiply-add) hardware together with other well known instructions, for example multiply,multiply-add, multiply-subtract etc. We have also shown that the remaining instructions can be executed by a new execution unit that is no more complex than a traditional ALU design.
- As a minor contribution we have shown that using the same paeth extended logic, it is trivial to compute also the maximum and minimum of the three inputs. Furthermore, we introduced an extension of the Add-Multiply-Add unit, whereby the Add-Multiply-Add unit can compute the Mean of three inputs.

Note

1. There is no power of 2 which is dividable by 3, so we have to approximate. Note however that the result is always rounded correctly.

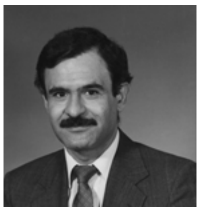
References

1. K. Aono, M. Toyokura, T. Araki, A. Ohtani, H. Kodama, and K. Okamoto, "A Video Digital Signal Processor with a Vector-Pipeline Architecture," *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 12, pp. 1886–1894, December 1992.
2. P.A. Ruetz, P. Tong, D. Bailey, D.A. Luthi, and P.H. Ang, "A High-Performance Full-Motion Video Compression Chip Set,"

IEEE Transactions on Circuits and Systems for Video Technology, Vol. 2, No. 2, pp. 111–122, June 1992.

3. K. Herrmann, M. Seifert, K. Gaedke, H. Jeschke, and P. Pirsch, *Architecture and VLSI Implementation of a RISC Core for a Monolithic Video Signal Processor*, VLSI Signal Processing. New York: IEEE, 1994, pp. 368–377.
4. S. Rathnam and G. Slavenburg, "An Architectural Overview of the Programmable Multimedia Processor, TM-1," in *Proceedings of COMPCON '96*, IEEE, 1996, pp. 319–326, Los Alamitos, 25–28 February 1996.
5. K. Guttag, R.J. Gove, and J.R. van Aken, "A Single-Chip Multi-Processor for Multimedia: The MVP," *IEEE Computer Graphics and Applications*, Vol. 12, No. 6, pp. 53–64, November 1992.
6. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, Vol. 16, No.4, 1996, pp. 42–50.
7. R.L. Sites and R. Witek, *Alpha AXP Architecture: Reference Manual*, 2nd edn., Digital Press, Burlington, 1995.
8. P.M. Kogge, *The Architecture of Pipelined Computers*, Advanced computer science series. McGraw-Hill Book Company, New York, 1981.
9. R. Montoye, E. Hokenek, and S. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM Journal of Research and Development*, Vol. 34, No. 1, 1990, pp. 59–70.
10. S. Vassiliadis, J. Phillips, and B. Blaner, "Interlock Collapsing ALU's," *IEEE Transactions on Computers*, Vol. 42, No. 11, 1993, pp. 825–839.
11. F. Onion, A. Nicolau, and N. Dutt, "Compiler Feedback in ASIP Design," Technical Report 94-2, Department of Information and Computer Science, University of California, September 1994.
12. J.L. Mitchell, W.B. Pennebaker, C.E. Fogg, and D.J. LeGall, *MPEG Video Compression Standard*, Digital Multimedia Standard Series. Chapman and Hall, New York, 1996.
13. L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, Vol. 34, 1965, pp. 349–356.
14. T. Boutell and T. Lane, "PNG (Portable Network Graphics) Specification," version 1.0. ftp://ftp.uu.net/graphics/png/documents/png-1.0-w3c.ps.gz.
15. S. Waser and M.J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, CBS College Publishing, 1982.
16. T. Doyle and P. Frencken, "Median Filtering of Television Images," in *Proceedings of the International Conference on Consumer Electronics, Digest of Technical Papers*, June 1986, pp. 186–187.
17. E.A. Hakkennes, "Multimedia Hardware Accelerators," Ph.D. Thesis, Delft University of Technology, December 1999.

Edwin A. Hakkennes received his Ir. degree (M.Sc) in electrical engineering in 1995, and his Dr. degree (Ph.D) in 1999, both from Delft University of Technology, The Netherlands. From 1995 to 1999 he was at Delft University of Technology, as a research and teaching assistant. His research interests are Computer Engineering, in particular the design of execution units for special purpose and application domain specific processors. In 2000 he joined X Integrated Circuits in Rotterdam, The Netherlands, as design engineer.
e.hakkennes@et.tudelft.nl



Stamatis Vassiliadis is a professor in the Electrical Engineering department of Delft University of Technology (T.U. Delft), The Netherlands. He has also served in the faculties of Cornell University, Ithaca, NY and the State University of New York (S.U.N.Y.), Binghamton, NY. He worked for a decade with IBM in the Advanced Workstations and Systems laboratory in Austin TX, the Mid-Hudson Valley laboratory in Poughkeepsie NY and the Glendale laboratory in Endicott NY. In IBM he has been involved in a number of projects regarding computer design, organizations, and architectures and the leadership to advanced research projects. He has been involved in the design and implementation of several computer systems including for example the IBM 9370 model 60. A number of his inventions have been implemented in commercially available systems and processors including the IBM POWER II, the IBM AS/400 Models 400, 500, and 510, Server Models 40S and 50S, and the IBM AS/400 Advanced 36. For his work he received numerous awards including 23 levels of Publication Achievement Awards, 15 levels of Invention Achievement Awards and an Outstanding Innovation Award for Engineering/Scientific Hardware Design in 1989. Six of his 67 patents have been rated with the highest patent ranking in IBM and in 1990 he was awarded the highest number of patents in IBM. Dr. Vassiliadis is an IEEE fellow. His research interests include computer architecture, parallel embedded systems, hardware design and functional testing of computer systems, parallel processors, computer arithmetic, EDFI for hardware implementations, neural networks, fuzzy logic and systems, and software engineering.
s.vassiliadis@et.tudelft.nl