An Investigation on FPGA based SAD Hardware Implementations

Stephan Wong, Bastiaan Stougie, and Sorin Cotofana
Computer Engineering Laboratory,
Electrical Engineering Department,
Delft University of Technology,
Stephan@Dutepp0.ET.TUDelft.NL

Abstract— **In this paper, we argue that the utilization of field-programmable gate array (FPGA) structures can improve the performance of embedded systems based on programmable processor cores. Furthermore, in multimedia processing it is well-known that the sum-of-absolute-differences (SAD) operation is the most time-consuming operation when implemented in software running on such programmable processor cores. This is mainly due to the sequential characteristic of such an implementation. Therefore, in this paper we investigate several hardware implementations of the SAD operation and map the most promising one in FPGA. Our investigation shows that an adder tree based approach yields the best results in terms of speed and area requirements and has been implemented as such by writing high-level VHDL code. Due to the limited number of I/O pins of current commercially available FPGA chips, we opted to implement the SAD over multiple chips by utilizing a single design. The design was functionally verified by utilizing the MAX+plus II 10.1 Baseline software package from Altera Corp. and then synthesized by utilizing the LeonardoSpectrum software package from Exemplar Logic Inc. Preliminary results show that the design can be clocked at 380 Mhz. This result translates into a faster than real-time full search in motion estimation for the main profile/main level of the MPEG-2 standard.**

Keywords—**sum of absolute difference, field-programmable gate array, hardware synthesis.**

I. Introduction

In video coding, similarities between video frames can be exploited to achieve higher compression ratios. However, moving objects within a video scene diminish the compression efficiency of the straightforward approach that only considers pels[1] located at the same position in the video frames. In order to achieve higher compression efficiency, *motion estimation* was introduced in an attempt to accurately capture such movements. In the MPEG-1/2 multimedia standards, it is performed for every macroblock, i.e., an array of 16×16 pels, in the to be encoded frame by finding its 'best' match in a reference frame. The most commonly used metric to evaluate the match is the "sum of absolute differences" (SAD), which adds up the absolute differences between corresponding elements in the macroblocks. The SAD operation is very time-consuming due to the complex nature of the absolute operation and the subsequent multitude of additions. In [15], a parallel hardware implementation was proposed to speed up the SAD computation process. This paper also describes amongst others this parallel hardware implementation of the SAD operation and focus on their implementation in field-programmable gate arrays (FPGAs). The reasons to utilize FPGAs are discussed in the following.

[1]Pel stands for picture element and represents the smallest color data unit of a picture or video frame.

Traditionally, the design of embedded multimedia processors was very much similar to microcontroller design. This meant that for each targeted set of multimedia applications, an embedded multimedia processor needed to be designed in specialized hardware (commonly referred to as Application Specific Integrated Circuits (ASICs)). In the early nineties, we were witnessing a shift in the embedded processor design approach fuelled by the need for faster time-to-market times. In embedded processor, this resulted in the utilization of programmable processor cores augmented with specialized hardware units implemented in ASICs. Consequently, time-critical tasks were implemented in specialized hardware units while other tasks were implemented in software to be run on the programmable processor core [13]. This approach allowed a programmable processor core to be re-used for different sets of applications and only the augmented units need to be (re-)designed for specific application areas.

Currently, we are witnessing a new trend in embedded processor design that is again quickly reshaping the embedded processor design. Instead of implementing the time-critical tasks in ASICs, these tasks are to be implemented in field-programmable gate arrays (FPGA) structures or comparative technologies [4], [14], [16], [6]. The reasons for and the benefits of such an approach include the following:

- **Increased flexibility:** The functionality of the embedded processor can be quickly changed without requiring another roll-out of the embedded processor itself and design faults can be quickly rectified. It also allows for quick adaptation of new (possibly unforeseen) developments.

- **Sufficient performance:** The performance of FPGAs has increased tremendously and is quickly approaching that of ASICs [2]. This seems to be mainly due to the faster adaptation of new technological advancements by FPGAs than by ASICs.
- **Faster design times:** Faster design times are achieved by re-using intellectual property (IP) cores or by slightly modifying them. More importantly, high-level design languages (such as VHDL) can be used in the design process and thereby speeding it up significantly.

The mentioned advantages and enabling FPGA have even resulted in that programmable processor cores are implemented on the same FPGA structure, e.g., Nios from Altera [1] and MicroBlaze from Xilinx [3].

In this paper, we investigate several hardware implementation alternatives for the sum-of-absolute-differences (SAD) operation in terms of expected speed and area. The three implementation alternatives are based on a sequential adder, a systolic array of adders, and a pipelined adder tree, alternatively. The sequential adder based implementation yields the lowest area requirements, but it requires the most cycles to complete. The systolic array based implementation requires slightly less cycles to complete, but at the same time much more area is needed. The pipelined adder tree based implementation is the best approach since it requires less area than the systolic array based implementation and since it requires the lowest number of cycles to complete. This approach has been chosen to be implemented in FPGA hardware.

This paper is organized as follows. Section II discusses the background of multimedia processing and motion estimation in particular in which the SAD operation is being utilized. Section III discusses the implementation alternatives in detail and selects the most promising one. Section IV describes the chosen multi FPGA chip design and presents the generic design for all chips. Furthermore, the synthesis results of this design are presented. Finally, Section V concludes this paper with some remarks.

II. Background

Digital video compression entails the utilization of many coding techniques with the ultimate goal to reduce the size of the digital representation of a video sequence. The same techniques used to compress digital pictures, e.g., in the JPEG picture standard, can be applied to single video frames. Such techniques exploit the fact that colors in photographic images tend to only gradually change when traversed from one side to another. In the video coding case, the fact that subsequent video frames do not differ much can be similarly exploited in order to increase compression efficiency.

All coding techniques can be categorized into two main categories, namely lossy and lossless techniques. Lossy coding techniques remove pel information that the human eye is unable to perceive using coding techniques such as the discrete cosine transform and quantization. The information that has been removed in most cases cannot be exactly regained, but it usually can only be approximated. On the other hand, lossless coding techniques do not remove any information. Instead, it exploits redundancies, i.e., similarities, between pels found in and between video frames which results in the representation of pel information using fewer bits. A lossless coding technique is predictive coding which predicts *current* pel(s) using *reference* pel(s) and then store the difference(s) between the prediction and the current pel(s). Assuming redundancy between pels, the differences are usually small and can be coded using less bits than the coding of the original pels. Predictive coding can use pels from the same video frame as reference pels (intra-coding) or pels from other video frames (inter-coding). Inter-frame predictive coding can contribute to the overall compression efficiency, because consecutive video frames are usually similar, i.e., they do not differ much. In this sense, the reference pels can be found in a reference frame located at the same position as the current pels in the current to be coded frame. This approach can also be used to capture scene changes by choosing the reference frames in the near future of the current (to be encoded) frame instead from its past. However, such a straightforward approach has one major drawback. Objects in a video scene tend to move around resulting in poor compression performance of the straightforward inter-frame predictive coding method, because pels located at the same location in consecutive frames are now quite different.

Motion estimation has been introduced in an attempt to capture the motion of objects within a video scene. I.e., find the best match between the pel(s) in the current frame and the pel(s) in the reference frame. To this end, a search area within the reference frame must be traversed in order to find the best match. After finding the best match, the difference(s) between the pels must be coded together with the difference between the locations (motion vector). Motion estimation can be performed for single pels in the current frame, but it is rarely used, because the coding of motion vectors for single pels reverses the gains of predictive coding. Therefore, block-based motion estimation is the most commonly used form in which a search is performed in the reference frame for a block of pels in the current frame.

Two key issues are associated with motion estimation in general, namely the size of the search area and which metric to use for determining the ‘best match’. The first issue is a trade-off, because a limited search area reduces the possibility of finding a ‘best match’ and an exceedingly large search area results in many unnecessary computations. In order to reduce the number of computations, many search area traversing methods have been proposed in literature [11], [7], [9], [8]. The second issue relates to finding a metric that will guarantee a good coding performance. Two of such metrics are the *mean square error* (MSE) and the *mean absolute difference* (MAD).

Considering that block-based motion estimation is most commonly used in multimedia standards such as MPEG-1 [12], MPEG-2 [5], and Px64 [10], we briefly highlight the block-based forms of the MSE and the MAD metrics. Such a block is usually 16×16 in size and is referred to as **macroblock**. The MSE is calculated as follows:

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

with $0 \leq x$,$y <$ framesize
with (r,s) being the motion vector
with $A_{(x,y)}$ being a current frame pel at (x,y)
with $B_{(x,y)}$ being a reference frame pel at (x,y)

Due to the square operation on the differences, this metric is less commonly used. Instead, the MAD is used more often and it is calculated as follows:

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

with $0 \leq x$,$y <$ framesize
with (r,s) being the motion vector
with $A_{(x,y)}$ being a current frame pel at (x,y)
with $B_{(x,y)}$ being a reference frame pel at (x,y)

The vector (x,y) denotes the location of the to be encoded macroblock in the current frame. Both x and y are multiples of 16 due to the blocksize is 16×16. The (motion) vector[2] (r,s) denotes the location of the macroblock to be used as a prediction in the reference block relative to the location of the to be coded macroblock in the current frame. Due to the computational simplicity of the MAD and reasonable accuracy, it is being used more often than the MSE. The MAD can be rewritten to:

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

The division by 256 in (binary) computer arithmetic is translated into an easy shifting the final SAD result by 8 bits. Therefore, we are focusing solely on the SAD in the remainder of this paper.

III. SAD IMPLEMENTATION ALTERNATIVES

In this section, we investigate several design alternatives in implementing the ”sum of absolute differences” operation. We can observe that the SAD operation can be divided into two stages. In the *absolute* stage, all the $|A_k - B_k|$’s calculated (possibly in parallel) before these results are summed up in the *sum* stage.

The *absolute* stage: All data values A_k and B_k are considered to be unsigned 8-bit numbers. In a straightforward implementation approach, the data values A_k and B_k are converted to a number representation that accommodates negative values allowing the values to be subtracted from each other. In the case that the result of the subtraction is negative, the result must be changed to a positive value. The discussed implementation approach has two main disadvantages. First, arithmetic encompassing negative numbers require more bits to represent the same range of positive values. Furthermore, additional logic is needed to perform boundary checks. Second, there is an occasional delay incurred by the last step (negative $\rightarrow$ positive) leading to an extension of all data-paths since the delay can not be pre-determined.

Before we discuss the next approach, we have to note that the subtraction of two unsigned numbers (e.g., $A_k - B_k$) is performed by adding A_k with a bit inverted B_k ($\overline{B_k} = 2^n - 1 - B_k$) and adding a ‘hot’ one: $A_k + (2^n - 1 - B_k) + 1 = 2^n + A_k - B_k$. Assuming that $B_k \leq A_k$, the resulting carry (2^n) of the addition can be ignored. In the case that no carry was generated, B_k was greater than A_k and the addition yields an incorrect addition.

Utilizing the previously discussed subtraction of two unsigned numbers, it is possible to maintain the bit length of the values A_k and B_k. However, the effort of performing an addition in order to determine whether a carry is generated is wasted in the case that no carry was generated. Therefore, we propose to implement a carry generator to calculate the carry based on the well-known carry-

[2]Contrary to x and y, are r and s not multiples of 16 as the granularity of the search area is on the pel level.

propagate algorithm. We have to note that no actual addition is being performed. Based on the result of the carry generator, the right value (A_k or B_k) is bit inverted and added to the remaining value (B_k or A_k) together with a 'hot' one. While this approach is already faster than the straightforward approach (since only *addition* is needed), the performance of the overall SAD operation can be further improved by delaying the summation to be included in the *sum* stage. This approach is depicted in Figure 1. We have to note that in the *sum* stage all the 'hot' ones must taken care of. This can be done by counting all the needed 'hot' ones and adding this count (256) as an additional summation term in the *sum* stage.

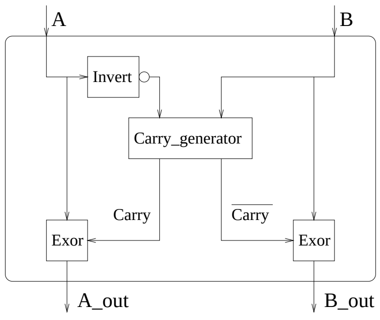

Fig. 1. Utilizing a carry generator in the *absolute* stage.

In conclusion, the second approach proved to be beneficial in terms of area (not requiring more bits) and speed (no addition is required). Furthermore, this approach also allows a tighter integration of the *absolute* stage with *sum* stage since we have substituted the subtraction in the MAD definition with an addition.

The *sum* stage: In this stage, all the K summation terms (X_k) outputted by the *absolute* stage must be summed up. To this end, we propose three different methods which we have termed: sequential addition, systolic array of adders, and pipelined adder tree.

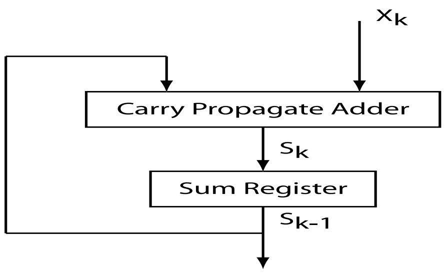

Fig. 2. Sequential addition with carry propagate adder.

A possible implementation of sequential addition is depicted in Figure 2. In this figure, the values from the *absolute* stage are summed utilizing a carry propagate adder (CPA), e.g., a carry look-ahead adder or a ripple carry adder, and a sum register. The precision of the sum register can be pre-determined, because the bit-length of input values and the number of addition terms are known beforehand. However, this also dictates the length of the carry propagate adder which is much slower due to the longer bit length of its inputs. The amount of cycles to calculates the result is: $K\times$ length of CPA (in bits).

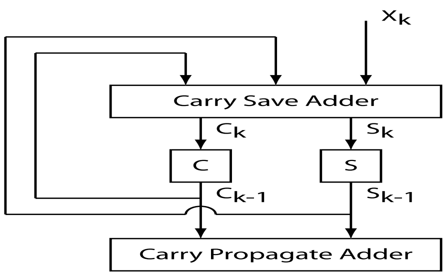

Fig. 3. Sequential addition with carry save adder.

Another possible implementation of sequential addition is depicted in Figure 3. In this figure, a carry save adder is used to calculate the intermediate sum value (block S) and carry value (block C). Since such a carry save adder performs a 3-to-2 reduction, in each cycle a new addition term can be added. After the last X_K term has been entered, the final sum and carry values are added in the carry propagate adder. This implementation takes "$K+$ length of CPA" cycles to calculate the result. Both possibilities of a sequential addition require only a small area, but vary in speed in terms of cycles.

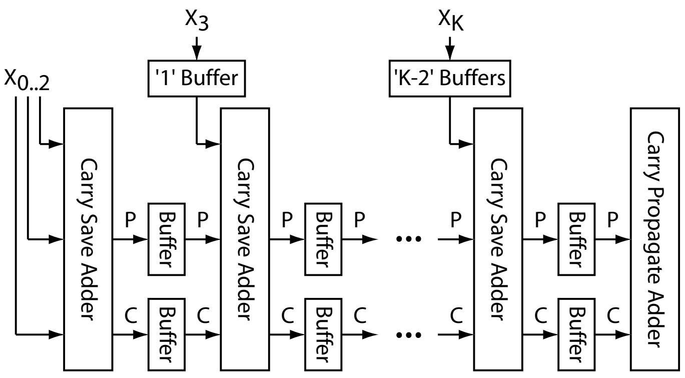

Fig. 4. Addition utilizing a systolic array.

A possible implementation of addition utilizing a systolic array is depicted in Figure 4. In this systolic array, the intermediate result flows through that array. In addition, at each stage a new X_k is being added to the intermediate result. As a result, all the X_k values must be buffered $k-2$ cycles and thereby requiring considerable amounts of area. On the other hand, an advantage of this approach is that the implementation is pipeline-able. This allows input values to be put into the array at each cycle and will produce a result in each cycle (after a certain startup time).

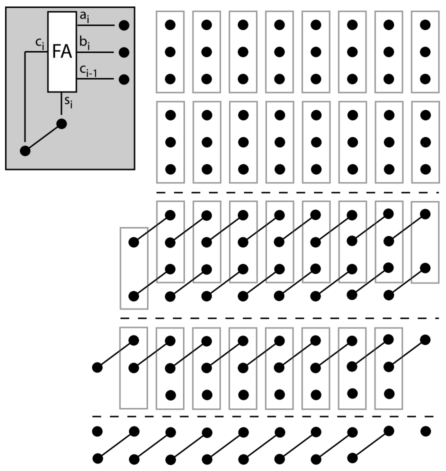

Fig. 5. An example adder tree in dot notation.

The third approach is based on the pipelined adder tree which utilizes full adders in order to add three bits (of the same weight) and produce a sum bit and a carry bit (called 3-to-2 reduction). By iteratively applying this method, all the summation terms X_k can be reduced to two intermediate summation terms. These two intermediate summation terms are then added by utilizing a carry propagate adder. A simplified adder tree which starts with six summation terms is shown in Figure 5. In this figure, each bit is represented with a black dot and the full adder is represented by a gray box. This method has several advantages. First, it allows optimizations within the adder tree since the adder tree is fixed. Second, it is pipeline-able. Third, it requires considerably less area since no buffering of the input is required. Fourth, it has a considerably smaller latency than the other two implementation methods to produce the first result. Finally, it allows the additional summation terms from the *absolute* stage when utilizing the method depicted in Figure 1 to be easily included in the adder tree.

In conclusion, based on our preliminary estimations based on number of cycles and area, it is best to combine the carry generator based implementation (for the *absolute* stage) and the adder tree based implementation (for the *sum* stage).

IV. VHDL IMPLEMENTATION AND SYNTHESIS RESULTS

In the previous section, we have selected to implement the SAD operation based on the carry generator method (depicted in Figure 1) for the *absolute* stage and the adder tree for the *sum* stage. In this section, we discuss two possible multi-chip designs. Multi-chip designs are needed, because currently available chips only have $\pm$ 1000 I/O pins which is not enough to encompass a fully parallel design of the SAD operation. Such a parallel design has the following I/O pin requirements: $(512$ inputs $\times$ 8 bits$)$ $+$ 16 output bits $+$ 1 clock signal $=$ 4113 pins. At the time of this investigation, the Altera STRATIX EP1S80 was the largest commercially available FPGA chip in terms of I/O pins $(= 1234)$ and served as the basis for our investigation into multi-chip designs.

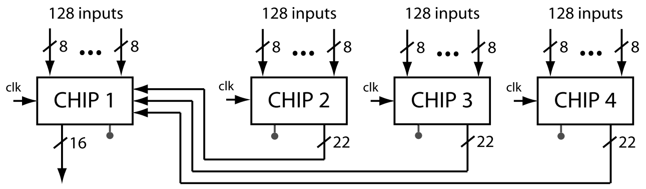

Fig. 6. A 4-chip design.

A 4-chip design is depicted in Figure 6. By distributing the input pins over four chips, the I/O pin requirements can be significantly reduced. Furthermore, we have opted to implement one generic design (depicted in Figure 6) for all chips which significantly reduced the design time. The generic design is such that two modes are supported. The first mode performs the operations needed in the *absolute* stage, i.e., utilizing the carry generator (CG), till the point just after ”Adder Tree 1”. This mode is used by chips 2 through 4. The second mode (employed by chip 1) also starts calculating the *sum* stage, but continues with ”Adder Tree 2” by utilizing the results from the other three chips. The I/O pin requirements are as follows: (64 carry generators (CGs) $\times$ 2 inputs $\times$ 8 bits) $+$ (1 output to other chip $\times$ 22 bits) $+$ 3 input from other chips $\times$ 22 bits $+$ 16-bit SAD output $+$ 1 clock $=$ 1129 pins.

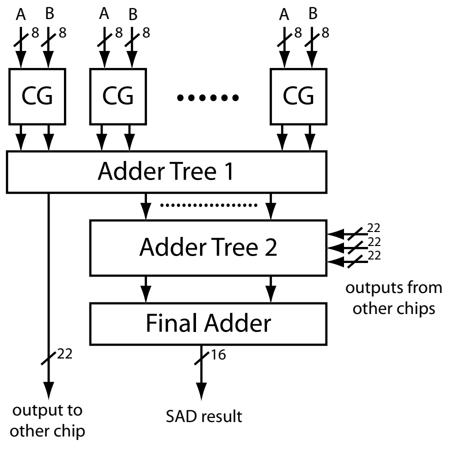

Fig. 7. The generic chip’s internal organization.

The number of cycles to calculate the SAD result is 29 clock cycles. The functionality of the design has been verified by utilizing the MAX+plus II 10.1 Baseline software package from Altera Corp. The synthesis results after running LeonardoSpectrum are presented in Table I.

```
*********************************************
Device utilization for EP1S80F1508C
*********************************************
```

Resource	Used	Avail	Utilization
IOs	1129	1213	93.08%
LCs	7765	79040	9.82%
Memory Bits	0	7427520	0.00%
DSP block 9-bit elems	0	176	0.00%

Clock Frequency Report

Clock	: Frequency
CLK	: 380.7 Mhz

TABLE I
SYNTHESIS RESULTS OF THE GENERIC CHIP IMPLEMENTATION.

We can observe in the table that our implementation utilizes 93% of the available I/O pins. Furthermore, since our design only requires 9% of the chip area, we envision that more functionality (like DCT, IDCT) be included on the same chip by multiplexing the I/O pins. Finally, the chip can be clocked at a frequency of 380 Mhz. We can note that this implementation on the STRATIX FPGA is I/O bound since the frequency corresponds to the data arrival time of 2.63 ns. Assuming that the memory is fast enough to provide the needed data, our design is able to support full search for the main profile/main level (720×576 resolution) in the MPEG-2 standard. The full search algorithm requires 30 frames/second $\times$ 1620 macroblocks/frame $\times$ 1620 SAD operations/macroblock $=$ 78782000 SAD operations/second. This translates into that every 12.70 ns one SAD operation must be performed which is much larger than 2.63 ns which is the (cycle) time needed to produce a SAD result in our pipelined design.

V. CONCLUSIONS

In this paper, we have argued that the sum-of-absolute-differences (SAD) operation is commonly used in video encoding schemes in order to determine the 'closeness' of two macroblocks (a 16×16 array of pels). We have established that the SAD operation can be divided into two stages, namely *absolute* and *sum*. For each stage, several implementation alternatives can be identified. Based on expected speed and area estimates we have selected to implement the SAD operation utilizing a carry generator in the *absolute* stage and an adder tree in the *sum* stage. Furthermore, in order to implement a fully parallel design the I/O pin requirements exceed what is provided by current commercially available field-programmable gate array (FPGA) structures. Therefore, we have chosen to implement the SAD by utilizing 4 chips. We have to note that only a single generic design was utilized for all 4 chips. The synthesis results show that 1129 I/O pins are required and 7765 LCs are utilized which translates into an area utilization of about 9%. Finally, the presented pipelined implementation is able to perform faster than real-time full search in motion estimation for the main profile/main level of the MPEG-2 standard.

REFERENCES

[1] Nios Embedded Processor. http://www.altera.com/products/devices/excalibur/exc-nios_index.html.

[2] Virtex-II 1.5V FPGA Family: Detailed Functional Description . http://www.xilinx.com/partinfo/databook.htm.

[3] Xilinx MicroBlaze. http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=microblaze.

[4] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pages 23–40, March 1999.

[5] B. G. Haskall, A. Puri, and A. N. Netravali. *Digital Video: An introduction to MPEG-2*. Digital Multimedia Standard Series. Chapman and Hall, 1996.

[6] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium of Field-Programmable Custom Computing Machines*, pages 24–33, April 1997.

[7] 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.

[8] S. Kappagantula and K. Rao. Motion Compensated Predictive Coding. In *Proc. Int. Tech. Symp. SPIE*, San Diego, CA, August 1983.

[9] T. Koga, K. Linuma, A. Hirano, Y.Iijima, and T. Ishiguro. Motion-Compensated Interframe Coding for Video Conferencing. In *NTC 81 Proceeding*, pages G5.3.1–5, New Orleans, LA, December 1981.

[10] M. Liou. Overview of the px64 kbit/s Video Coding Standard. *Communications of the ACM*, 34(4):59–63, April 1991.

[11] B. Liu and A. Zaccarin. New Fast Algorithms of the Estimation of Block Motion Vectors. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(2):148–157, April 1993.

[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, 1996.

[13] S. Rathnam and G. Slavenburg. An Architectural Overview of the Programmable Multimedia Processor, TM-1. In *Proceedings of the COMPCON '96*, pages 319–326, 1996.

[14] R. Razdan and M. Smith. A High-Performance Microarchitecture with hardware-programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, November 1994.

[15] S. Vassiliadis, E. Hakkennes, S. Wong, and G. Pechanek. The Sum-Absolute-Difference Motion Estimation Accelerator. In *Proceedings of the 24^{th} Euromicro Conference*, 2000.

[16] R. Wittig and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. In *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, April 1996.