A Sum of Absolute Differences Implementation in FPGA Hardware

Stephan Wong, Stamatis Vassiliadis, and Sorin Cotofana
Computer Engineering Laboratory,
Electrical Engineering Department,
Delft University of Technology,
{Stephan, Stamatis, Sorin}@CE.ET.TUDelft.NL

Abstract

In this paper, we propose a new hardware unit that performs a 16×1 *SAD operation. The hardware unit is intended to augment a general-purpose core. Further, we show that the* 16×1 *SAD implementation used can be easily extended to perform the* 16×16 *SAD operation, which is commonly used in many multimedia standards, including MPEG-1 and MPEG-2. We have chosen to implement the* 16×1 *SAD operation in field-programmable gate arrays (FPGAs), because it provides increased flexibility, sufficient performance, and faster design times. We performed simulations to validate the functionality of the* 16×1 *SAD implementation using the MAX+plus II (version 9.23 BASELINE) software from Altera and synthesis using the FPGA Express (version 3.4) software from Synopsis. Targeting the Altera's FLEX20KE family, synthesis of our* 16×1 *SAD unit produced the following results for area and clock frequency: 1699 look-up tables (LUTs) and 197 MHz, respectively.*

1 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. 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 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 describes this parallel hardware implementation of the SAD operation in field-programmable gate arrays (FPGAs).

Traditionally, the design of embedded multimedia processors were very much similar to the design of microcontrollers. 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 fueled by the need for faster time-to-market times. This resulted in the design of embedded processors utilizing 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 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 al-

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

lows 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 under consideration to be implemented in the same FPGA structure, e.g., Nios from Altera [1] and MicroBlaze from Xilinx [3].

In this paper, we have developed a VHDL model for a functional unit that is able to perform the 16×1 SAD operation as introduced in [15]. It is to be implemented in field-programmable gate arrays (FPGAs) and it is intended to augment a general-purpose processor core. As shown later in this paper, the proposed hardware unit can be easily extended to perform the 16×16 SAD operation. We performed simulations to validate its functionality using the MAX+plus II (version 9.23 BASELINE) software from Altera and synthesis using the FPGA Express (version 3.4) software from Synopsis. When our 16×1 SAD unit was synthesized on the FLEX20KE family of Altera, we obtained the following results for area and clock frequency: 1699 LUTs and 197 MHz, respectively.

2 Sum of Absolute Differences

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 an interesting one, 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 large and is referred to as **macroblock**. The MSE is calculated as follows:

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

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

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

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

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

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, 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. All the absolute operations of the SAD operation can be performed serially, per column in parallel, per row in parallel, or all 256 operations in parallel. While it is possible to perform all the operations serially, this approach is time-consuming and not efficient performance-wise. Performing the operations per row or per column in parallel are exactly the same with the only difference being the indexing of the pels. Considering that the completely parallel approach is a simple extension of the per-row or per-column approaches, we focus in this paper on the 16×1 SAD that processes all the pels in a row in parallel. An additional advantage of the per row parallel approach is because the pel data is stored in consecutive locations in the main memory. This alleviates the need for special reordering hardware. The complete SAD operation can be rewritten to:

$$SAD(x,y,r,s) = \sum_{j=0}^{15} SAD16_j(x,y,r,s)$$

with the $SAD16_j$ being defined as:

$$SAD16_j(x,y,r,s) = \sum_{i=0}^{15}|A_{(x+i,y+j)} - B_{((x+r)+i,(y+s)+j)}|$$

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

In the remainder of this paper, all data units A_i and B_i are considered to be unsigned 8 bits numbers. Subtraction of two unsigned numbers (e.g., $A - B$) is performed by adding A with a bit inverted B ($\overline{B} = 2^n - 1 - B$) and adding a 'hot' one: $A + (2^n - 1 - B) + 1 = 2^n + A - B$. Assuming that $B \leq A$, the resulting carry (2^n) of the addition can be ignored. The $SAD16_j$ operation can be performed in three steps:

- Compute $(A_i - B_i)$ for all 16×1 pel locations.
- Determine which $(A_i - B_i)$'s are negative, i.e., when no carry was generated and compute $(B_i - A_i)$ instead if this was the case.
- Add all 16 absolute values together.

This approach requires one addition in the first step and an occasional second addition in the second step. In [15], another approach was introduced to parallelize and speedup the $SAD16$ operation without the uncertainty of the second step. Its approach is briefly highlighted below:

- Determine the smallest of the two operands.
- Invert the smallest operand.

[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.

- Pass both operands to an adder tree.
- Add a correction term to the adder tree.
- Reduce the 33 addition terms to 2.
- Add the remaining two terms using an adder.

The first step is performed by computing $\overline{A} + B$. In case no carry was generated, this means that $B \not> A$ and thus B should inverted. Otherwise, A should be inverted. Next to passing the operands to an adder tree, an additional correction term must be added to counter the effects of using inverted values. The adder tree reduces the adder terms two terms which are then passed to an adder. For precise mathematical details of the approach, we refer to [15].

3 The VHDL implementation

In the previous section, we have highlighted the significance of motion estimation in video coding. An important metric used in motion estimation is the sum of absolute differences (SAD). The absolute difference operation can be implemented in several ways: serial, per column in parallel, per row in parallel, and fully parallel. In this paper, we focus on the $SAD16$ operation that performs the SAD on one row of a macroblock (16×1). All the input values are 8-bit unsigned binary numbers. By iteration or parallel execution of the $SAD16$ operation, the complete SAD operation for the 16×16 macroblock can be performed. In this section, we discuss the VHDL implementation of $SAD16$ operation using a method introduced in [15] and present the results afterwards. First, we discuss the steps necessary to perform the $SAD16$ operation in more detail:

- **Determine the smallest of the two operands** As suggested in [15], it is only necessary to determine whether $\overline{A} + B$ produces a carry or not.
- **Invert the smallest operand** If no carry was produced, B must be inverted, otherwise, A must be inverted. This is done by utilizing an exor.
- **Pass both operands to an adder tree** After inverting either A or B, the operands must be passed to an adder tree. Thus, the values ($\overline{A}$, B) or (A, $\overline{B}$) are passed further.
- **Add a correction term to the adder tree** Also discussed in [15], an additional correction term must be added to the adder tree which is 16 in this case.
- **Reduce the** 33 **addition terms to** 2 All 33 addition terms must be reduced to 2 terms before the final addition can be applied. This can be done using an 8-stage carry save adder tree using 243 carry save adders. This results in two term as shown in Figure 1.

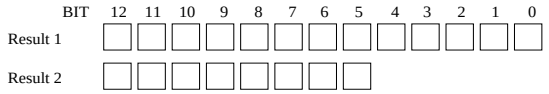

Figure 1. The resulting terms after the adder tree.

- **Add the remaining two terms using an adder** The final two addition terms are added using a 8-bit carry lookahead adder for the most significant bits. The result is a 13-bit unsigned binary number. However, as stated in [15], the most significant bit of this result can be disregarded resulting in a final 12-bit unsigned binary number.

In Figure 2, the first three steps are depicted. The determination wether the addition $\overline{A} + B$ generates a carry is performed without actually calculating the addition. Instead, this is achieved by only utilizing certain parts within a carry lookahead adder that calculate the carry. The resulting carry and inverted carry are fed to two exors that will invert the correct term.

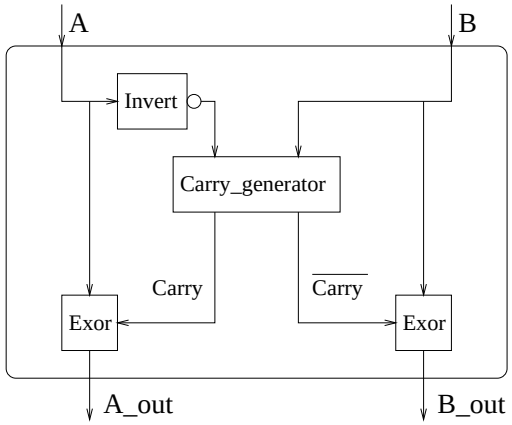

Figure 2. The first three steps.

The inversion of either As or Bs for all 16 absolute operations can be carried out in parallel and can be fed to an adder tree at the same time as shown in Figure 3.

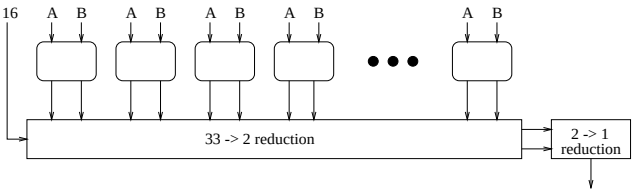

Figure 3. The $SAD16$ operation.

Figure 3 depicts the complete $SAD16$ operation that has been implemented in VHDL. Next to the parallel execution of the first three steps, the figure also depicts the addition of a correction term of 16, the $33 \rightarrow 2$ reduction tree, and the final $2 \rightarrow 1$ reduction. The implementation is synchronous and fully pipeline-able.

Before we present the results on this implementation, we discuss two methods of how we can utilize the $SAD16$ unit to implement the complete 16×16 SAD operation:

- When optimizing for speed, we replicate the $SAD16$ unit 16 times as depicted in Figure 4. In this figure, the $2 \rightarrow 1$ reduction in the $SAD16$ unit has been disabled, because it is a time-consuming operation compared to an additional reduction stage of the ensuing reduction tree. The $2 \rightarrow 1$ reduction takes several clock cycles while passing the two results directly to the $32 \rightarrow 2$ reduction tree only results in one additional clock cycle. The resulting two operands from this reduction tree are added using another $2 \rightarrow 1$ reduction.

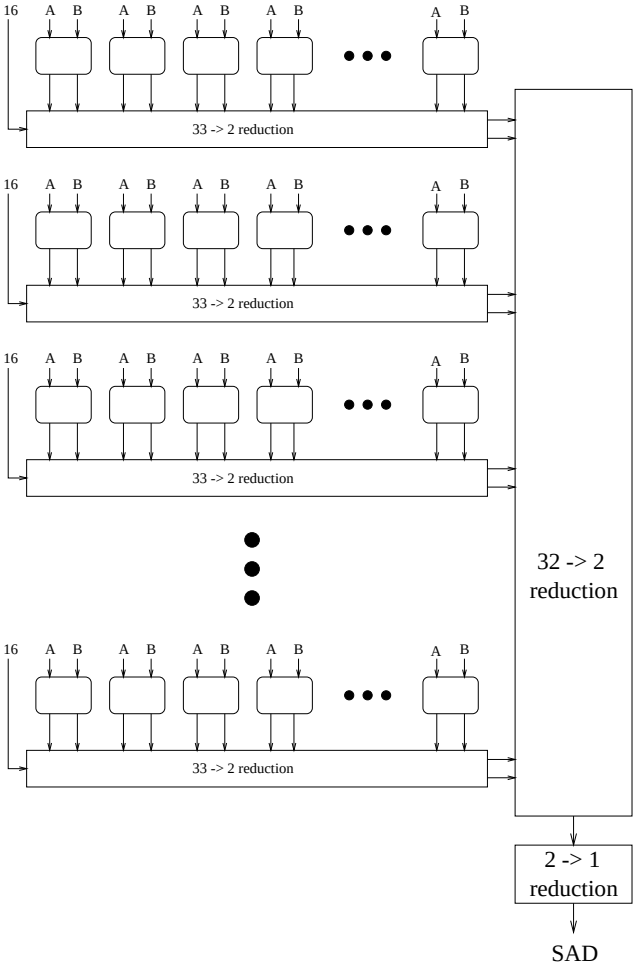

Figure 4. A 16×16 SAD.

- When optimizing for area, we can re-use the $SAD16$ unit as depicted in Figure 3. Instead of performing the $2 \rightarrow 1$ reduction for each row, the two terms are buffered until the whole macroblock has been processed. Then, all the buffered 32 terms are inserted into $33 \rightarrow 2$ together with a correction term of 0. This is followed by the final $2 \rightarrow 1$ reduction. We have to note that the bit widths of both the $33 \rightarrow 2$ reduction tree and the $2 \rightarrow 1$ must be extended to support larger binary numbers.

Assumptions In this section, we have discussed the $SAD16$ operation and how this can be implemented in hardware. VHDL code was written for the approach discussed in this section. Its functionality was validated using MAX+PLUS II, version 9.23 baseline software from Altera. Then, the VHDL model was synthesized using FPGA Express from Synopsis, build 3.4.0.5211 by targeting the FLEX20KE family from Altera. From the synthesis, we obtained results about area and clock frequencies.

Results The VHDL code describes a synchronous and pipelined design which takes 19 clock cycles to produce the first result. The results of the synthesis of the VHDL model are the following. The area of the $SAD16$ implementation is **1699 look-up tables (LUTs)**. The highest achievable frequency is **197 MHz**. At this clock frequency, the 19 clock cycles of the $SAD16$ unit translates into 96ns.

Finally, some estimations will be given regarding the number of clock cycles it takes to perform the 16×16 SAD using the $SAD16$ unit according to the two methods discussed earlier:

- When optimizing for speed, we note that there is only one additional $32 \rightarrow 2$ reduction tree (see Figure 4) when compared to the $SAD16$ unit depicted in Figure 3. Due to the fact that this reduction tree is of similar complexity as the $33 \rightarrow 2$ one, which takes 8 clock cycles, we only add 8 additional clockcycles resulting in a total of 27 clock cycles.
- When optimizing for area, producing all the pairs after the $33 \rightarrow 2$ reduction tree for all 16 rows takes $14 + 15$ clock cycles. 14 cycles are needed to obtain the first pair and all 15 subsequent pairs take one cycles each. Then, 8 clock cycles are needed to perform the $32 \rightarrow 2$ reduction and 5 for the last $2 \rightarrow 1$ reduction resulting in 42 clock cycles.

The second method is about 1.5 times slower than the first one, but it also requires considerably less area. We must keep in mind that the area utilized by the $SAD16$ unit in the second method is larger than 1699 LUTs. This is due to the fact that longer binary inputs must be supported resulting in a longer binary output. Furthermore, the second method is also less efficient performance-wise in the calculation of

subsequent 16×16 SADs that are usually required in motion estimation. This is due to the buffering of the intermediate results between the first stage results (after the $33 \rightarrow 2$) and the final stage which eliminates the intrinsic pipelined behavior of the proposed $SAD16$ unit.

4 Conclusion

In this paper, we have proposed the $SAD16$ unit which performs a 16×1 SAD operation. It is intended to augment a general-purpose processor core by speeding up the overall 16×16 SAD operation. We have shown that the $SAD16$ unit can be used to perform the complete 16×16 operation, either by replicating the unit 16 times or exploiting its pipeline characteristic. The $SAD16$ implementation produces its first result after 19 clock cycles. By replicating the $SAD16$ unit and adding another adder tree, the resulting fully parallelized implementation requires 27 clock cycles to produce the 16×16 SAD result. Another more area efficient method utilizing the pipelined $SAD16$ unit requires 42 clock cycles to perform the 16×16 SAD operation. We have chosen to implement the 16×1 SAD operation in field-programmable gate arrays (FPGAs), because it provides increased flexibility, sufficient performance, and faster design times. We have performed simulations to validate functionality of the $SAD16$ implementation using the MAX+plus II (version 9.23 BASELINE) software from Altera and synthesis using the FPGA Express (version 3.4) software from Synopsis. When our $SAD16$ unit was synthesized on the FLEX20KE family of Altera, we obtained the following results for area and clock frequency: 1699 look-up tables (LUTs) and 197 MHz, respectively.

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 24th Euromicro Conference*, 2000.

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