SAD implementation in FPGA hardware

Stephan Wong, Stamatis Vassiliadis, and Sorin Cotofana
Computer Engineering Laboratory,
Electrical Engineering Department,
Delft University of Technology,
{Stephan, Stamatis, Sorin}@Dutepp0.ET.TUDelft.NL

Abstract— **In this paper, we propose a new unit intended to augment a general-purpose core that is able to perform a 16×1 SAD operation. We show that the 16×1 SAD implementation can easily be extended to perform the complete 16×16 SAD operation. The 16×16 SAD operation 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, good-enough 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 3.5) software from Synopsis. When our 16×1 SAD unit was synthesized by targeting the FLEX20KE family of Altera, we obtained the following results for area and clock frequency: 1699 LUTs and 197 MHz, respectively.**

Keywords— **Sum of absolute differences, motion estimation, video coding, field programmable gate arrays.**

I. Introduction

In video coding, similarities between two video frames can be exploited to achieve higher compression ratios. However, moving objects in the video scene diminish the compression efficiency of the straightforward approach that only looks at the macroblocks at the same position in the two video frames. As a result, *motion estimation* was introduced to capture such movements in order to maintain compression efficiency. It determines the 'best' match between a 16×16 macroblock in the current (to be coded) frame and a macroblock in the reference frame. For this purpose, the "sum of absolute differences" (SAD) is used as the main metric. This is achieved by determining the absolute differences between the corresponding elements in the macroblocks and then adding up the differences. The SAD operation is very time-consuming in that a lot of additions must be performed and due to the complexity of the absolute operation. In [1], a possible parallel hardware implementation was introduced to speed up the SAD computation process.

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. This meant that time-critical tasks were implemented in specialized hardware units while other tasks were implemented in software to be run on the programmable processor core [2]. This approach allowed a programmable processor core to be reused 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 [3], [4], [5], [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.
- **Good-enough performance:** The performance of FPGAs has increased tremendously and is quickly approaching that of ASICs [7]. 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.

Due to these benefits, the implementation of the processor core is also considered to be implemented in the FPGA structures. One example can be found in [8].

In this presentation, we have developed a VHDL model

for a functional unit, to be implemented in field programmable gate arrays (FPGAs) that augments a general-purpose processor, that is able to execute the 16×1 SAD operation as introduced in [1] and which 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 3.5) 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.

II. The sum of absolute differences

Digital video compression entails the utilization of many coding techniques with the ultimate goal to reduce the total size of the digital representation of a video sequence. The same techniques used to compress digital pictures can be applied to single video frames, but redundancies found between video frames can now also be 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[1] information that the human eye is unable to perceive using, e.g., quantization of discrete cosine transform coefficients. Lossless coding techniques only exploit redundancies, i.e., similarities, between pels found in and between video frames which results in representing the pel information using less 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 is possible, 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 frame instead from the 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 try 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 searched 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 a too large search area results in too many unnecessary computations. In order to reduce the number of computations, many search area traversing method have been proposed in literature [9], [10], [11], [12]. 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[13], MPEG-2[14], and px64[15], we briefly highlight the block-based forms of the MSE and the MAD metrics. The most commonly used blocksize is 16×16 and it is usually referred to as the **macroblock**. The MSE is calculated as follows:

$$MSE(x,y,r,s) =$$
$$\frac{1}{256}\sum_{i=0}^{15}\sum_{j=0}^{15}\left(A_{(x+i,y+j)} - B_{((x+r)+i,(y+s)+j)}\right)^2$$
$$with\ \ 1 \le x,y \le (framesize - 16)$$
$$with\ A_{(x,y)}\ being\ a\ current\ frame\ pel\ at\ (x,y)$$
$$with\ B_{(x,y)}\ being\ a\ reference\ frame\ pel\ at\ (x,y)$$

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:

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

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

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, because the blocksize is 16×16. The (motion) vector (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 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 in a macroblock and subsequent summation can be performed serially, per column in parallel, per row in parallel, or all 256 operations in parallel. While it possible to perform all the operations serially, it is time consuming and performance-wise not efficient. Performing the operations per row or per column in parallel are exactly the same with the only difference being the indexing of the pels. Also considering that we can easily extend this to a fully parallel approach, we perform a complete 16×16 SAD by performing 16 separate 16×1 SADs by processing all the pels in a row in parallel. The resulting SAD 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:

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

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 [1], 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
- 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 the precise details of the previous approach, we refer to [1].

III. 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) for which many types of parallelization can be performed. In this paper, we focus on the $SAD16$ which performs the SAD on one row of an macroblock (16×1). By iteration or parallel execution of the $SAD16$ operation, the complete SAD for the 16×16 macroblock can be performed. In this section, we discuss the VHDL implementation of $SAD16$ using a method introduced in [1] and present the results afterwards. First, we discuss the steps in performing the $SAD16$ in more detail:

- **Determine the smallest of the two operands:** As suggested in [1], 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 [1], an additional correction tree must be added to the adder tree.

• **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 a 8-stage carry save adder tree using 243 carry save adders.

• **Add the remaining two terms using an adder** The final two addition terms are added using a 8-bit carry lookahead adder. The result is a 12-bit number.

In Figure 1, the first three steps are depicted. The carry generator generates a carry without actually performing an addition. This is done by only utilizing a small part of the carry lookahead components within a carry lookahead adder.

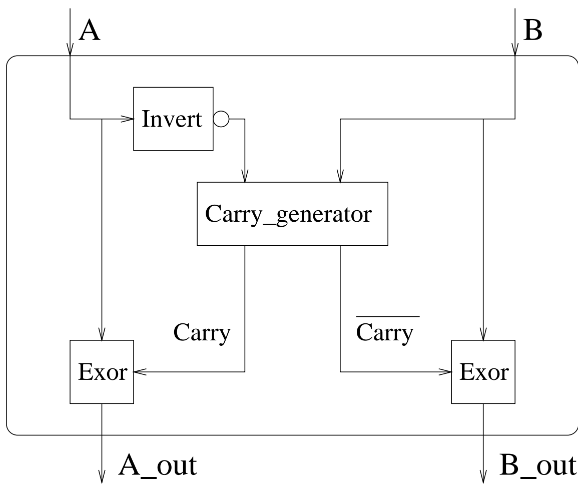

Fig. 1. The first three steps.

The inversion of either the As or Bs for all 16 absolute operations can be carried out in parallel and this is depicted in Figure 2. Also shown in this figure is the correction term of 16.

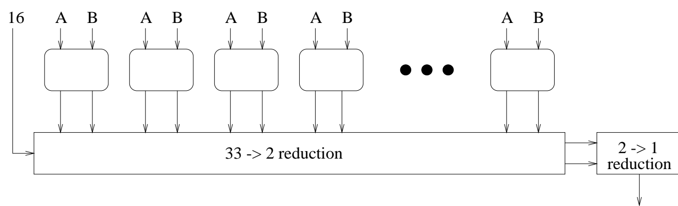

Fig. 2. The SAD16 operation.

It is this structure that has been implemented in VHDL. Before we show the results of this implementation, Figure 3 depicts how a 16×16 SAD can be performed in hardware using the SAD16 implementation. In this figure, the $2 \rightarrow 1$ reduction in the SAD16 operation using a carry lookahead adder is not used anymore. This is because the carry lookahead adder takes several clock cycles to perform. 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 carry lookahead header. Thus, execution time and the area of 15 $2 \rightarrow 1$ reduction units are saved.

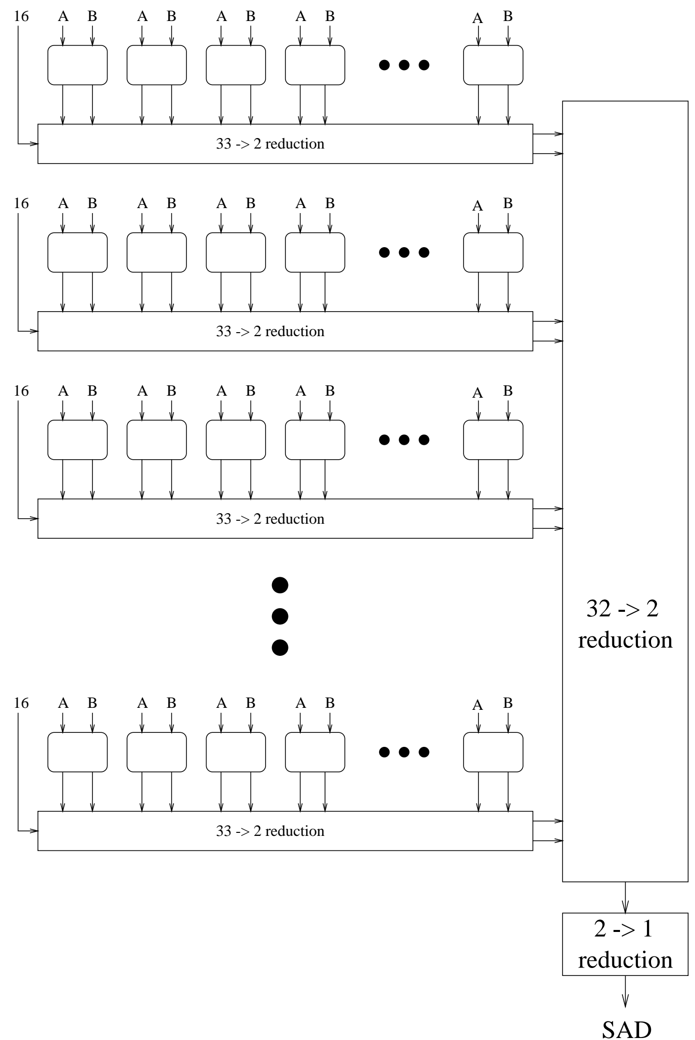

Fig. 3. A 16×16 SAD.

Assumptions In this section, we have discussed the SAD16 operation and how this can be implemented in hardware. For the approach discussed in this section, a VHDL code was written. 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 produces

the first result. The results of the synthesis of the VHDL model are the following:

- The area of the SAD16 implementation is 1699 LUTs.
- The highest achievable frequency if 197 MHz. At this clock frequency, the 19 clock cycles of the $SAD16$ unit translates into 96ns.

Assuming that we want to perform a complete 16×16 SAD operation, the $SAD16$ implementation needs to be replicated 16 times without the $2 \rightarrow 1$ reduction (see Figure 3). This is followed by another $32 \rightarrow 2$ reduction tree which will take another 8 clock cycles. The complexity of the ensuing $2 \rightarrow 1$ reduction is similar to the $2 \rightarrow 1$ reduction used in the $SAD16$ implementation and will not add additional clock cycles. The resulting number of clock cycles to compute the 16×16 SAD is 27 cycles.

Another method to perform the 16×16 SAD is by reusing the already discussed pipelined $SAD16$ unit. Actually, it must be slightly modified to accomodate the larger bitsizes of the intermediate results just after the $SAD16$ operation. The $32 \rightarrow 2$ reduction can be easily performed by putting zero on first input of the $33 \rightarrow 2$ reduction tree. Given that the $2 \rightarrow 1$ reduction takes 5 cycles and that our $SAD16$ implementation is pipelined, it takes $14 + 15$ cycles to produce all the number pairs after the $33 \rightarrow 2$ reduction tree (shown in Figure 2). 14 cycles are needed to perform the first result and all 15 subsequent results 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. This results in 42 clock cycles, but requires much less area than the previous approach.

IV. 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 can produce 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, good-enough 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 3.5) 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 LUTs and 197 MHz, respectively.

References

[1] S. Vassiliadis, E.A. Hakkennes, Stephan Wong, and G.G. Pechanek, "The Sum-Absolute-Difference Motion Estimation Accelerator," in *Proceedings of the 24th Euromicro Conference*, 2000.

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

[3] D.C. 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*, March 1999, pp. 23–40.

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

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

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

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

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

[9] Bede Liu and Andr Zaccarin, "New Fast Algorithms of the Estimation of Block Motion Vectors," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 3, no. 2, pp. 148–157, April 1993.

[10] Jaswant R. Jain and Anil K. Jain, "Displacement Measurement and Its Applications in Interframe Image Coding," *IEEE Transactions on Communications*, vol. COM-29, no. 12, pp. 1799–1808, December 1981.

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

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

[13] Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg, and Didier J LeGall, *MPEG Video Compression Standard*, Digital Multimedia Standard Series. Chapman and Hall, 1996.

[14] Barry G. Haskall, Atul Puri, and Arun N. Netravali, *Digital Video: An introduction to MPEG-2*, Number ISBN 0-412-08411-2 in Digital Multimedia Standard Series. Chapman and Hall, 1996.

[15] Ming Liou, "Overview of the px64 kbit/s Video Coding Standard," *Communications of the ACM*, vol. 34, no. 4, pp. 59–63, April 1991.