Alternatives in FPGA-based SAD Implementations

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

Abstract—**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 programmable processor cores. This is mainly due to the sequential characteristic of such an implementation. 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. 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, *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. In this paper, we focus on the implementation of the SAD operation in FPGA hardware due to the reasons presented in the following paragraph. Implementations of other multimedia operations can be found in [3], [4], [7].

In the past, the design of embedded multimedia processors was very much similar to microcontroller design by utilizing 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. This resulted in the utilization of programmable processor cores augmented with specialized hardware units. 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 [5]. 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 (again) or for specific application areas. Currently, we are witnessing a new trend in embedded processor design that implemented the time-critical tasks in field-programmable gate arrays (FPGA) structures or comparative technologies [2], [10], [9], [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 and design faults can be quickly rectified.
- **Sufficient performance:** The performance of FPGAs has increased tremendously and is quickly approaching that of ASICs [1].
- **Faster design times:** Faster design times are achieved by re-using intellectual property (IP) cores and utilization of high-level hardware description languages (e.g., VHDL).

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 alternatives are based on a sequential adder, a systolic array of adders, and a pipelined adder tree. The sequential adder based implementation yields the lowest area requirements, but it requires the highest number of cycles to complete. The systolic array based implementation requires slightly less cycles to complete, but 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 implementation alternatives in detail and selects the most promising one. Section III describes the chosen multi-chiplet design and presents the generic design for all chiplets. Furthermore, the synthesis results of this design are presented. Finally, Section IV concludes this paper with some concluding remarks.

II. SAD implementation alternatives

In this section, we investigate several design alternatives in implementing the "sum of absolute differences" (SAD).

$$SAD(x, y, r, s) =$$

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

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)

We can observe that the SAD operation can be divided into two stages. In the *absolute* stage, all the $|A_k - B_k|$'s are calculated (possibly in parallel) before these results are summed up in the *sum* stage.

The *absolute* stage: All values A_k and B_k are considered to be unsigned 8-bit numbers. In a straightforward approach, the 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 requires 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.

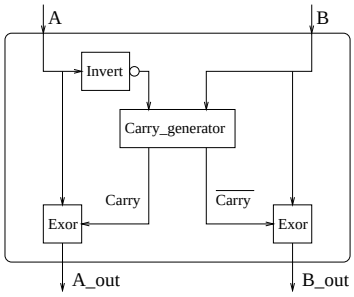

Fig. 1. Utilizing a carry generator in the *absolute* stage.

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 discussed subtraction of two unsigned numbers, it is possible to maintain the bit length of the values A_k and B_k. Instead of performing the addition, we propose to implement a carry generator to calculate the carry based on the well-known carry-propagate algorithm. Based on the result of the carry generator, the correct 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. This approach is faster than the previous approach (since only *'addition'* is needed) and 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.

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.

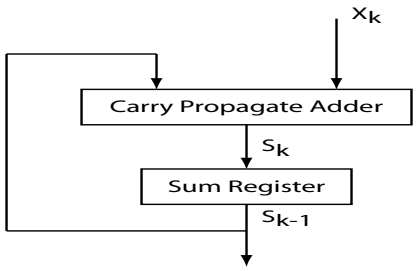

Fig. 2. Sequential addition with carry propagate adder.

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).

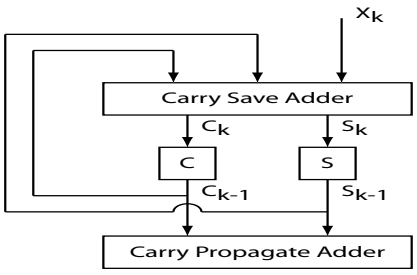

Fig. 3. Sequential addition with carry save adder.

Another 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, a new addition term can be added in each cycle. After entering the last term (X_K), the final

sum and carry values are added in the carry propagate adder. This implementation produces the result after "$K+$ length of CPA" cycles. Both implementation possibilities of a sequential addition require only a small area, but vary in speed in terms of cycles.

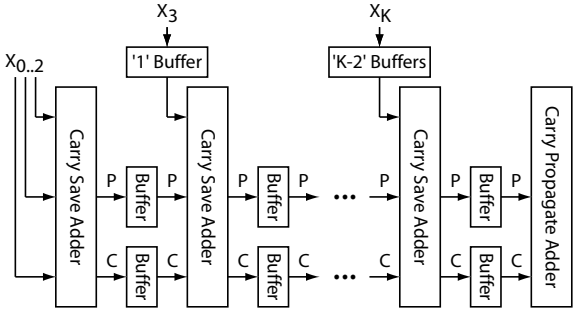

Fig. 4. Addition utilizing a systolic array.

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).

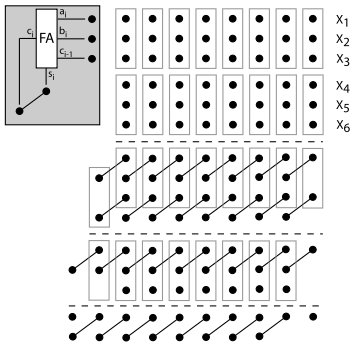

Fig. 5. An example adder tree in dot notation.

The third approach is based on a scheme proposed in [8] and utilizes a pipelined adder tree. By iteratively applying full adders, 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. An example 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 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).

III. 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-chiplet designs, 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 chiplets can be mapped to multiple FPGA chips or a single FPGA chip in the future. A completely 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-chiplet designs.

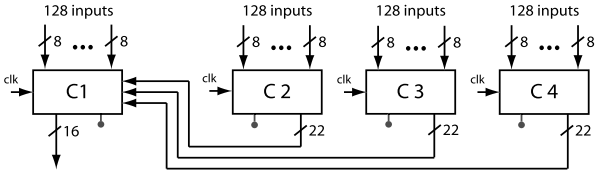

Fig. 6. A 4-chiplet design.

A 4-chiplet design is depicted in Figure 6. By distributing the input pins over four chiplets, the I/O pin requirements can be significantly reduced. Furthermore, we have opted to implement one generic design (depicted in Figure 6) for all chiplets 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 chiplets 2 through 4. The second mode (employed by chiplet 1) also starts calculating the *sum* stage, but continues with "Adder Tree 2" by utilizing the results from the other three chiplets. The I/O pin requirements are as follows: (64 carry generators (CGs) $\times$ 2 inputs $\times$ 8 bits) + (1 output to other chiplet $\times$ 22 bits) + 3 input from other chiplets $\times$ 22 bits + 16-bit SAD output + 1 clock = 1129 pins.

The number of cycles to calculate the SAD result is 29 clock cycles. The functionality of the design has been ver-

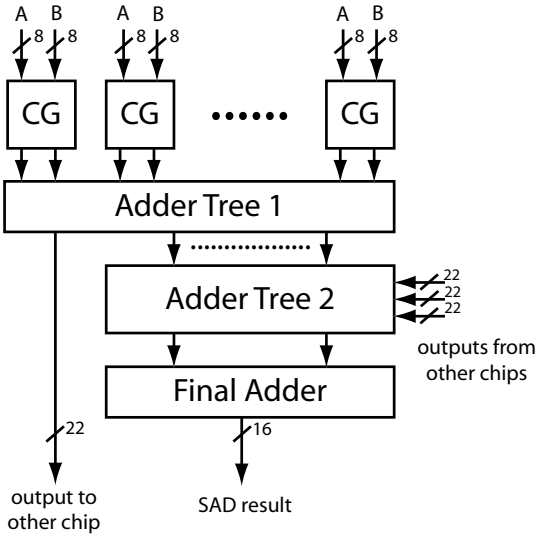

Fig. 7. The generic chiplet's internal organization.

ified 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 CHIPLET 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.

IV. CONCLUSIONS

In this paper, we considered for implementation the sum-of-absolute-differences (SAD) operation which 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 chiplets. We have to note that only a single generic design was utilized for all 4 chiplets. 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] Virtex-II 1.5V FPGA Family: Detailed Functional Description . http://www.xilinx.com/partinfo/databook.htm.

[2] 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.

[3] G. Kuzmanov and S. Vassiliadis. Reconfigurable Repetitive Padding Unit. *Proceedings of the 12th Great Lakes Symposium on VLSI*, pages 98–103, April 2002.

[4] J. Nikara, S. Vassiliadis, J. Takala, M. Sima, and P. Liuha. Parallel Multiple-Symbol Variable-Length Decoding. In *Proceedings of the IEEE International Conference on Computer Design (ICCD2002)*, pages 126–131, Freiburg, Germany, September 2002.

[5] 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.

[6] S. Seng, W. Luk, and P. Cheung. Run-time Adaptive Flexible Instruction Processors. *Proceedings of the Conference on Field Programmable Logic 2002 (FPL2002)*, pages 545–555, September 2002.

[7] M. Sima, S. Cotofana, S. Vassiliadis, J. T. van Eijndhoven, and K. Vissers. MPEG Macroblock Parsing and Pel Reconstruction on an FPGA-augmented TriMedia Processor. In *IEEE International Conference on Computer Design (ICCD2001)*, pages 425–430, September 2001.

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

[9] S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-coded Processor. In *Proceedings of the Conference on Field Programmable Logic 2001 (FPL2001)*, pages 275–285, 2001.

[10] 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.