# Customized Nios II multi-cycle instructions to accelerate block-matching techniques

Diego González [a], Guillermo Botella[a,*], Carlos García[a],
Anke Meyer Bäse[b], Uwe Meyer Bäse[c], Manuel Prieto-Matías[a]

[a]Dept. Computer Architecture, Universidad Complutense de Madrid. 28040 Madrid (Spain)
[b]Dept. Computer Scientific, Florida State University, Tallahassee, Florida 32310-6046 USA
[c]Dept. Electrical and Computer Engineering, Florida State University,
Tallahassee, Florida 32310-6046 USA

diegogonzalez@grupobme.es, {gbotella, garsanca, mpmatias}@ucm.es,
{umeyerbaese, ameyerbaese}@fsu.edu

## ABSTRACT

This study focuses on accelerating the optimization of motion estimation algorithms, which are widely used in video coding standards, by using both the paradigm based on Altera Custom Instructions as well as the efficient combination of SDRAM and On-Chip memory of Nios II processor. Firstly, a complete code profiling is carried out before the optimization in order to detect time leaking affecting the motion compensation algorithms. Then, a multi-cycle Custom Instruction which will be added to the specific embedded design is implemented. The approach deployed is based on optimizing SOC performance by using an efficient combination of On-Chip memory and SDRAM with regards to the reset vector, exception vector, stack, heap, read/write data (.rwdata), read only data (.rodata), and program text (.text) in the design. Furthermore, this approach aims to enhance the said algorithms by incorporating Custom Instructions in the Nios II ISA. Finally, the efficient combination of both methods is then developed to build the final embedded system. The present contribution thus facilitates motion coding for low-cost Soft-Core microprocessors, particularly the RISC architecture of Nios II implemented in FPGA. It enables us to construct an SOC which processes 50×50 @ 180 fps.

Keywords: Computer Vision, Optical Flow, MPEG Compression, Block Matching algorithm, NIOS II, FPGA, Custom Instructions, Embedded Systems.

## 1. INTRODUCTION

There is an increasing need to process multimedia information in Real-Time, and portable consumer electronic devices encourage research into green processing. There are many algorithms and systems that are specifically aimed at handling multimedia task such as motion compensation and coding, which are widely used in the video coding standards. Furthermore, motion estimation is still the focus of much research, and there exist several structures and implementations using FPGAs [1-6], GPUs [7-8] and embedded systems [9] in the framework of Real-Time implementation such as surveillance, tracking, navigation and automotive applications [10-11].

There has been much research into the paradigm of motion estimation for standard video coding in the last 30 years, as it avoids the use of temporal redundancy in video data for storage and transmission [12-14]. Motion estimation for multimedia purposes is achieved through Block Matching Techniques [15-19] that analyse the Macro Blocks (blocks of pixels, or MBs) of the so-called reference frame to estimate the closest block to the current one in the current frame. We can define the motion vector as an offset from the current frame's MB coordinates to the MB coordinates in the reference frame. Figure 1 show that the procedure of coding the frame processed with motion estimation in video, either for the predictor or for the entropy coding, is a crucial part of the MPEG-4, H.264/6 and, nowadays, the H.265 standards. With regards to Block Matching, there are three main techniques:

Real-Time Image and Video Processing 2015, edited by Nasser Kehtarnavaz, Matthias F. Carlsohn,
Proc. of SPIE-IS&T Electronic Imaging, SPIE Vol. 9400, 940002 · © 2015 SPIE-IS&T
CCC code: 0277-786X/15/$18 · doi: 10.1117/12.2077104

FST (Full Search Technique) [15], which matches all possible blocks within a search window in the reference frame to find the closest block to the one fixed in the current frame (Figure 2). The closest is the one with the minimum Summation of Absolute Differences (SAD):

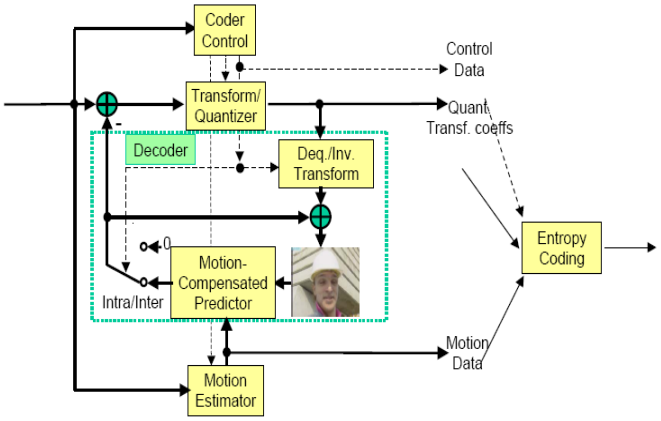


Fig. 1. Motivation. Part of the MPEG-4 scheme.

It(x, y) represents the pixel value at the coordinate ($x$, $y$) in the frame $t$ and the ($u$, $v$) is the displacement of the candidate Macro Block (MB). If there is a block of size 64×64, the FST algorithm requires 4096 subtractions and 4095 additions to calculate the SAD expression. The required number of checking blocks is $(1+2d)^2$ while the search window is limited to within ± d pixels, and currently a power of two is used for it.

$$SAD(x,y;u,v)=\sum_{x=0}^{31}\sum_{y=0}^{31}\left|I_t(x,y)-I_{t-1}(x+u,y+v)\right|, \tag{1}$$

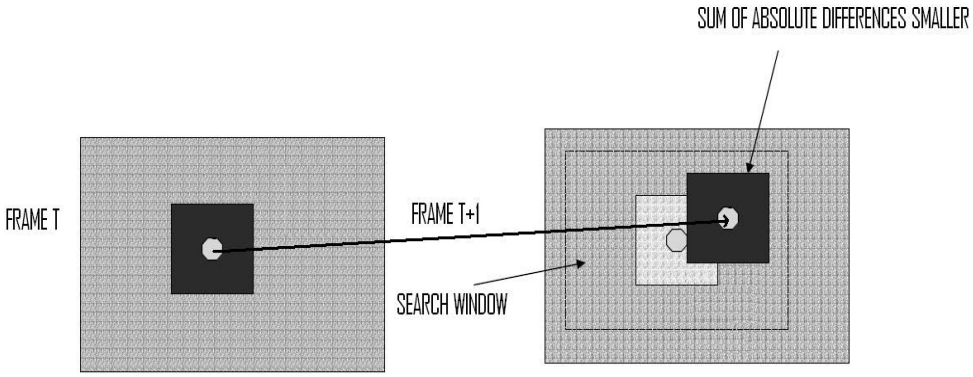


Fig. 2. Full Search Technique.

TSST (Three Steps Search Technique) [20-22] selects nine candidate points, including the center point and eight checking points on the boundary of the search centre movement ratio (Figure 3), moving forward to the matching point with the minimum SAD and reducing the step size by half in each of its three steps. The last step stops the search process with the optimal MV and the minimum SAD that can be obtained.

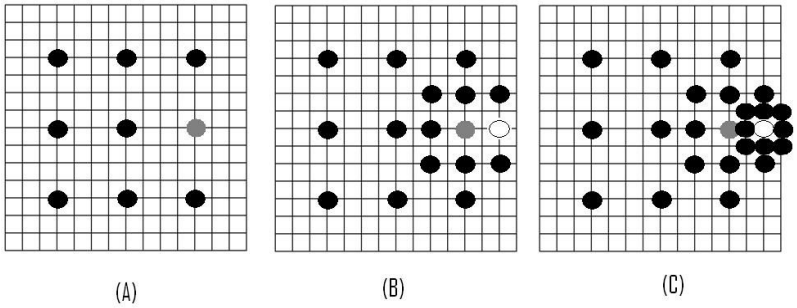


Fig. 3. Three Step Search Technique, (a) First step, (b) Second step, (c) Third step.

2DLOG (Two Dimensional Logarithmic Search) [23] uses a cross pattern search (+) in each step until the step size is one pixel, the initial step size being d/4 (Figure 4). The step size is reduced by half only when the minimum point of the previous step is the centre one or the current minimum point reaches the search window boundary. If none of these two conditions is met, the step size remains the same.

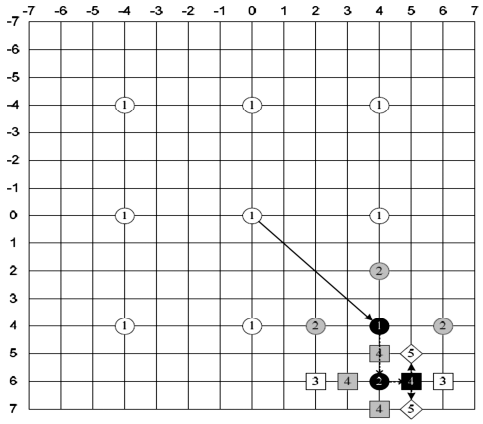


Fig. 4. 2DLog. Search path of 2DLOG search algorithm.

The organization of the paper is as follows. In Section 2, an overview of the NIOS II processor is given, accompanied by its custom instruction types (Section 2.1) and its different memory architectures (Section 2.2), while Section 3 describes the methodology used to accelerate the algorithms presented in this work. The final results from the memory configuration chosen and custom instruction paradigm acceleration are shown in Section 4. Finally, Section 5 contains our conclusions and lines for future work.

## 2. NIOS II PROCESSOR

NIOS II [24-25] is a 32 bit general soft core embedded processor which allows the acceleration of time-critical software algorithms by adding custom instructions to its instruction set. It belongs to a family of three members, namely Fast, Economy and Standard; each one optimized for a specific price and performance range.

The Nios II/f Fast CPU is optimized for maximum performance [26-27], offering a performance of up to 220 DMIPS in the Stratix II family of FPGAs, placing it squarely in the ARM 9 [28] class of processor. This performance allows it to meet the constraints of using custom instructions, high bandwidth switch fabric, and hardware accelerators. It supports fixed and variable cycle operations. The NIOS II/e Economy CPU is optimized for lowest cost, achieving a smaller FPGA footprint, and the NIOS II/s Standard CPU is a trade-off solution between processing performance and logic element usage, reaching over 120 DMIPS while consuming only 930 LEs (Stratix II [29-30]).

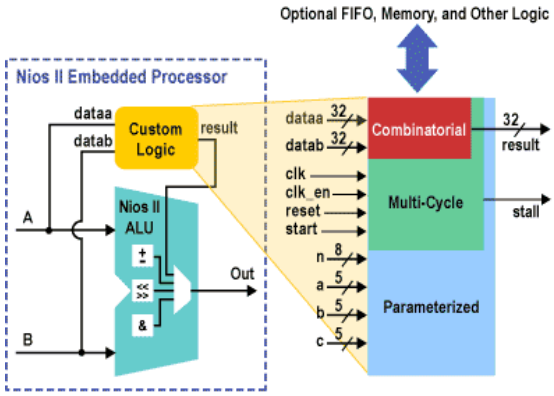


Fig. 5. NIOS II Embedded Processor [13].

## 2.1 NIOS II CUSTOM INSTRUCTIONS

Our first task in this work is to accelerate the algorithms by using the paradigm of the NIOS II custom instructions [31], which are custom logic blocks adjacent to the ALU in the processor's data path (Figure 5). They allow the designer to reduce a complex sequence of standard instructions to a single instruction implemented in hardware. The NIOS II processor uses *gcc* built-in functions to map to custom instructions [32], so it is feasible to use the macro directly in your C or C++ application code, thus avoiding writing assembly code and making one's progress along the steep learning curve to access custom instructions considerably faster. The NIOS II processor supports different types of custom instructions, of which there are four kinds that can be used to meet the application's constraints and requirements.

The first type is a combinational custom instruction that is described as a logic block that completes its logic function in a single clock cycle. The second and third types are multi-cycle or sequential custom instructions consisting of a logic block that requires two or more clock cycles to complete an operation. Finally, the fourth type is an extended custom instruction, which allows a single custom logic block to implement several different operations using an index to specify which operation the logic block has to perform.

## 2.2 MEMORY SYSTEM DESIGN AT EMBEDDED MACHINE VISION SYSTEMS

Another task addressed in this work is to perform an optimization in terms of memory use. Initially we carried out the whole algorithmic design without using the custom instruction paradigm, but by choosing an efficient combination of memory types to achieve a faster design. According to NIOS II specifications [34] there are four types of memories that we could use in our NIOS II processor based design: On-Chip memory, External SRAM, Flash Memory and SDRAM. Due to the limitations of our low-cost Altera DE2 board [35-36], here we can only use On-chip memory and SDRAM, which are described below.

On-Chip memory: On-Chip is embedded inside the FPGA. Therefore, this memory type is the fastest and provides the lowest possible latency. On-Chip memory has a large number of good characteristics, such as transaction pipelining and no additional circuit-board wiring required. Some kinds of On-Chip memories are characterized by dual-port mode accessing with different ports for reading and writing, which allows reading over one port while writing is performed over the other. With regards to drawbacks, it raises volatility and has limited capacity because designed memory capacity depends only on the specific FPGA device. Due to its advantages and disadvantages, On-Chip memories are mainly used for storing boot code or LUT (Look-Up Tables).

SDRAM: SDRAM is similar to SRAM, but it must be refreshed periodically to keep its data. The devices that employ SDRAM are usually low cost and of high-capacity, although a specific hardware controller is needed for it to operate. Since SDRAM organises the memory space in columns, rows and banks, the controller occupies a major part of the

interface. The complexity of the interface means always using an SDRAM controller which drives the timing, addresses multiplexing and refreshes every cycle. Thus SDRAM provides a large capacity at low cost, and its power consumption is lower compared with SRAM. It is feasible to share SDRAM buses to connect many SDRAM devices and external memories of other families such as flash or SRAM. SDRAM latency is always greater than regular external SRAM or FPGA on-chip memory. However, while first-access latency is high, the pipelining of consecutive access increases the global throughput. SDRAM can achieve higher clock frequencies than SRAM, thus improving performance.

## 3. METHODOLOGY

The methodology can mainly be divided into two different sections. The first one presents the parameters for the designs achieved through the use of NIOS II custom instructions, and the second one presents the improvements through every valid and possible combination of On-Chip and SDRAM memory in a design using NIOS II.

### 3.1 NIOS II CUSTOM INSTRUCTIONS

In this part profiling is performed (using the well-known *codeblocks* tool [33]) of the three presented algorithms in order to directly address the time leak point, which is where we can improve performance by replacing source code with custom instructions. For a better comprehension of our profiling, Figures 6-8 show the *FST* [15], *TSST* [20-22] and *2DLOG* [23] flow charts, and Figures 9-11 contain the *CopyBlock*, *GetBlock* and *GetCost* flow charts.

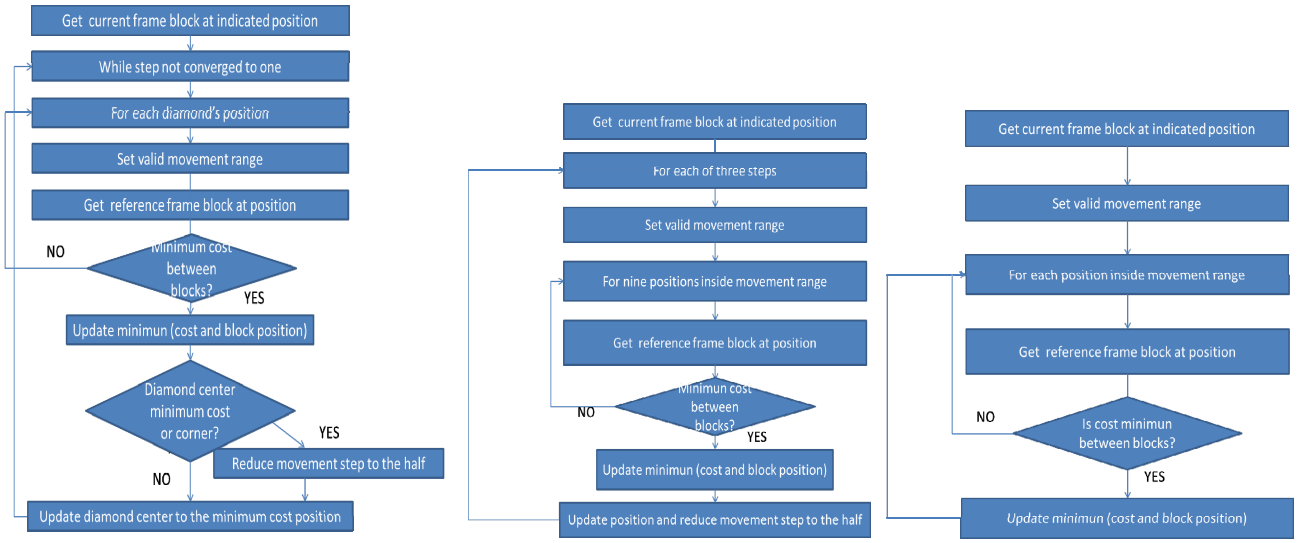


Fig. 6(left) FST, 7 (center) TSST and 8 (right) 2DLOG flow charts.

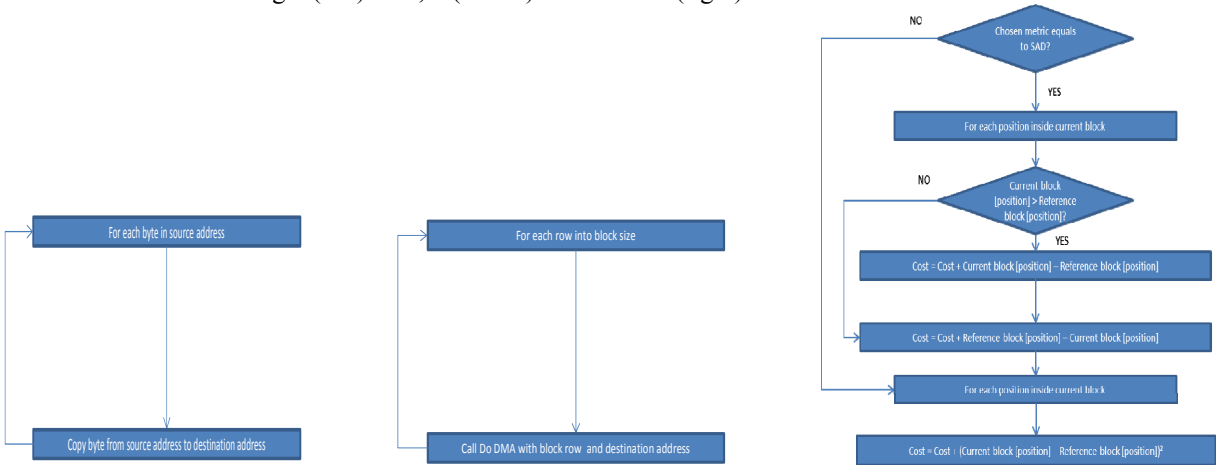


Fig. 9(left) *CopyBlock* ,10 (center) *GetBlock* and 11 (right) *GetCost* charts.

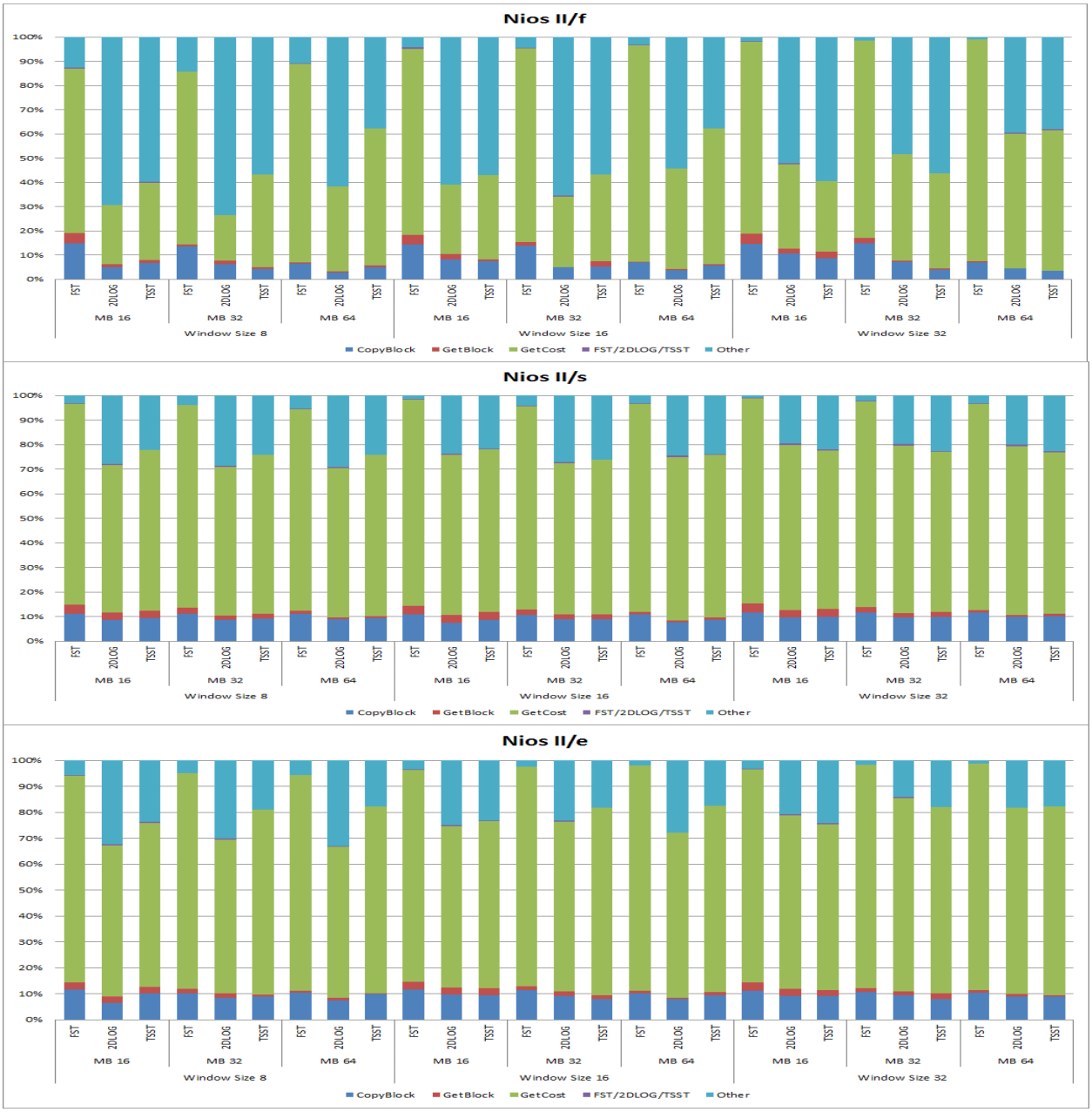


Fig. 12. Profiling of the different algorithms over all Nios II processors for each Macro-block (MB) and Window size.

Figure 12 shows the profiling results obtained over complete executions of our motion estimation process when choosing FST, TSST and 2DLOG accordingly. From the profiling analysis we can decide, as candidate for replacement, to replace the *GetCost* function with a specific custom instruction. Regarding the custom instruction types and the candidate function structure to be replaced, we have decided to approach it with a multi-cycle custom instruction.

In our first approach [27-28] the combinational custom instruction was accelerated, due to its speed (only one clock cycle), its easiness, and the *GetCost* function structure. Nevertheless, the multi-cycle custom instruction is implemented here due to its inherent advantages with regards to parallelism and pipelining, thus providing a more sophisticated custom instruction than the combinational one. The extended custom instruction was discarded because only one kind of

operation was needed between each pair of pixels (calculating SAD). The internal register file custom instruction derives from the multi-cycle one, and that one is tackled as previously mentioned. Finally, the external interface custom instruction has been discarded due to the fact that there is no need to communicate to external interfaces.
We therefore employ the multi-cycle type to replace the *GetCost* function source code, as was indicated by the profiling. As explained above, the *GetCost* function is used to calculate the SAD between two macroblocks, one from the reference frame, and the other one from the current frame.
When replacing the *GetCost* source code by the combinational custom instruction, which was carried out previously [27], the latter is called for every pair of pixels, one from each macroblock, and the local SAD calculated is accumulated to later give the total SAD between the required pair of macroblocks. On the other hand, when replacing the *GetCost* source code by the multi-cycle custom instruction, the latter is called for every group of eight pixels, four from each macroblock, and its main feature, the multi-cycle architecture, is used to calculate an accumulated SAD for that group. In this way, executing the multi-cycle custom instruction produces few accumulated SADs, which are then added to obtain the total SAD between the required pair of macroblocks.

In Figure 13, we present all the results obtained for every possible combination between the executed algorithm (FST, 2DLOG, and TSST), the selected macroblock size (16, 32, and 64), and the window size used (8, 16, and 32), for the well-known Foreman test bench video coding test sequence [33] using just the baseline case ("CI OFF"), the mono cycle ("CI Co ON") and multi-cycle custom instruction ("CI Mu ON"). They will be shown in comparison with the combinational custom instruction case and the base case, which we consider to be the *GetCost* source code, translated directly to Nios II processor instructions.
In this way, it is possible to analyze at a glance how much time we are saving with our improvement using our designed custom instruction. The best case achieves an improvement of 76.08% when executing the FST algorithm on the Nios II/e processor using a window search of 32 and a macroblock size of 32. On the other hand, the worst case remains without improvement when executing the TSST algorithm on the Nios II/f processor using a window size of 32 and a macroblock size of 16.

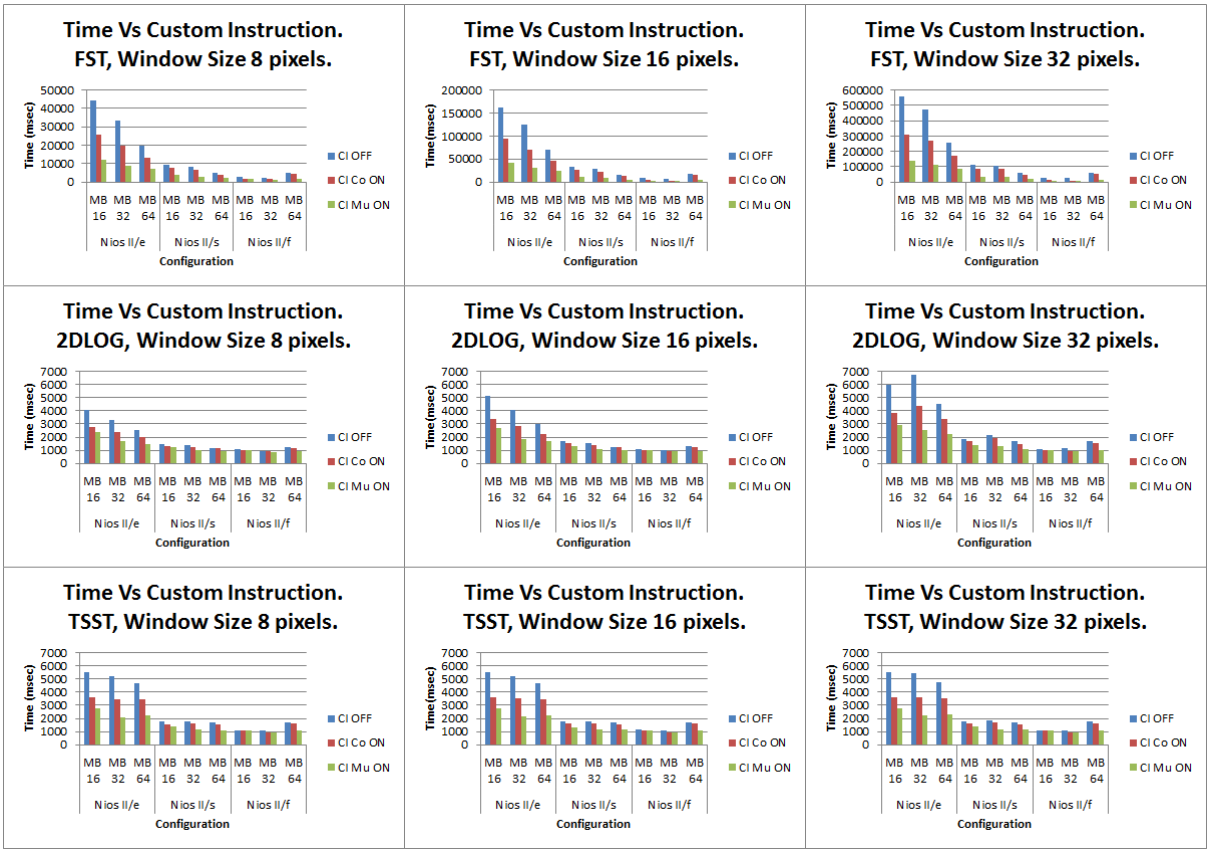

Fig. 13. Throughput for each algorithm, MacroBlock and processor without/with using Custom Instruction in *GetCost*.

### 3.2 MEMORY SYSTEM DESIGN

We can deal with different memory combinations by taking advantage of each kind of memory mentioned above in order to achieve a rapid design. In order to achieve that, we have improved our design by testing it with every permitted memory combination, as shown in Table 1 (using on-chip and SDRAM memories only). We have chosen the on-chip memory because it is the fastest available memory on the FPGA, and SDRAM due to its big capacity and its lower cost with a good performance. The system is fitted with a fixed window size of 32 pixels. We make a comparison of the results obtained for each one of the tested algorithms. Here we only present the allowed memory combinations (Table 4) which deliver a valid design (figures 14-16) that produces a correct program output on our testing platform (Altera DE2 board) [35], which incorporates a Cyclone II EP2C35F672C6 chip [36].
Figures 14-16 represent the throughput in every mentioned combination for the three motion estimation algorithms, which helps the designer to characterize every combination. Configuration types 2 and 3 give the optimized performance since program memory is allocated using On-Chip memory. The second most optimized configuration group is numbered as 6-8, where the Stack is configured to be On-Chip memory. We consider the baseline case (number 1) to be just using SDRAM in every single parameter of the microprocessor design.

Table 1. Memory System Design Configuration (MSDC). SD=SDRAM; OC=On-Chip Memory.

| Design Number | Processor Reset Vector | Processor Exception Vector | Stack | Heap | Read/ Write Data (.rwdata) | Read Only Data (.rodata) | Program (.text) |
|---|---|---|---|---|---|---|---|
| 1 | SD | SD | SD | SD | SD | SD | SD |
| 2 | SD | SD | SD | SD | SD | SD | OC |
| 3 | SD | SD | SD | SD | SD | OC | OC |
| 4 | SD | SD | SD | SD | OC | SD | SD |
| 5 | SD | SD | SD | SD | OC | OC | SD |
| 6 | SD | SD | OC | SD | SD | SD | SD |
| 7 | SD | SD | OC | SD | OC | SD | SD |
| 8 | SD | SD | OC | SD | OC | OC | SD |
| 9 | OC | OC | SD | SD | SD | SD | SD |
| 10 | OC | OC | SD | SD | SD | OC | SD |
| 11 | OC | OC | SD | SD | OC | SD | SD |
| 12 | OC | OC | SD | SD | OC | OC | SD |
| 13 | OC | OC | OC | SD | SD | SD | SD |
| 14 | OC | OC | OC | SD | SD | OC | SD |
| 15 | OC | OC | OC | SD | OC | SD | SD |
| 16 | OC | OC | OC | SD | OC | OC | SD |

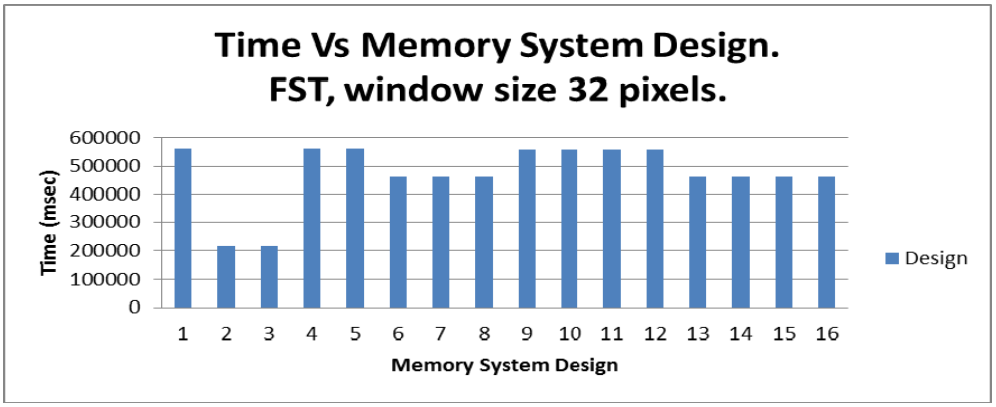


Fig. 14. Throughput obtained for each MSDC (FST).

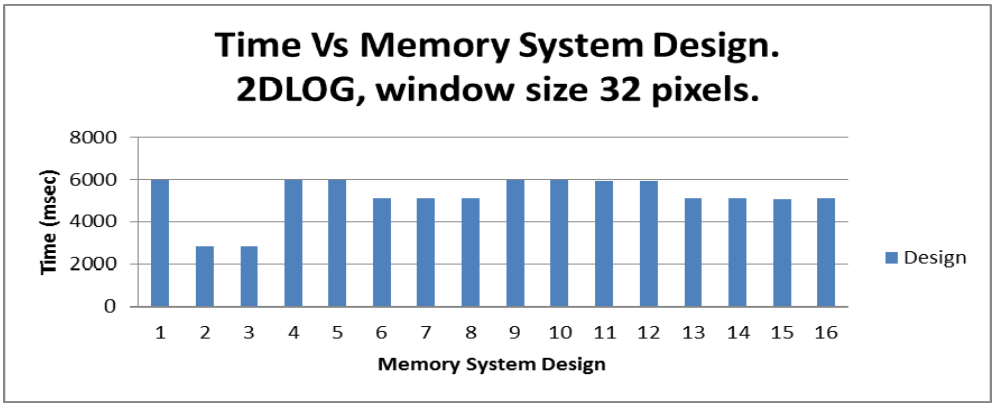


Fig. 15. Throughput obtained for each MSDC (2DLOG).

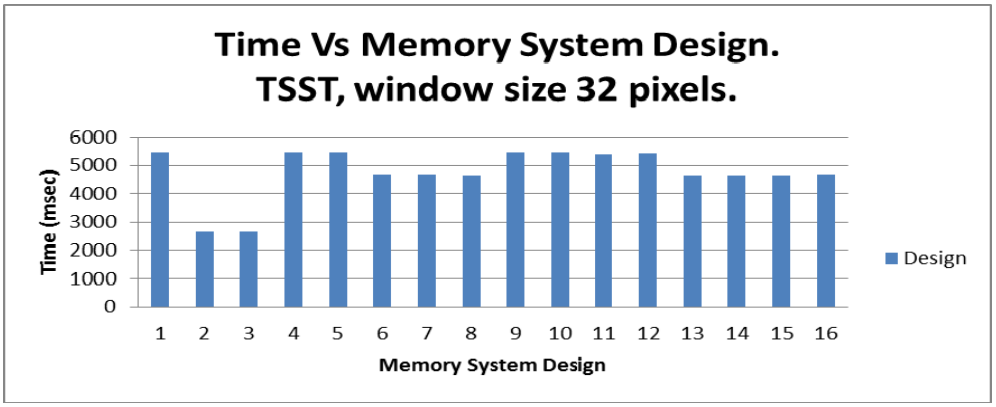


Fig. 16. Throughput obtained for each MSDC (TSST).

As we can see, the group formed of configurations 2 and 3 achieves the best performance since the program text is allocated using On-chip memory. The second best group of configurations is formed of designs 6 to 8 and 13 to 16, where the stack is configured to be On-chip. The third group is formed of the remaining configurations (1, 4, 5, and 9 to 12), where stack and program text are both allocated in SDRAM. The baseline case, design 1, is constructed using just SDRAM in every single configuration parameter of the memory system design.

## 4. FINAL RESULTS: CUSTOM INSTRUCTIONS AND MEMORY CHOICE

In the two approaches that we have implemented and described in this paper, we built the embedded system by putting all results together (multi-cycle custom instruction and memory combination in the architecture of the embedded system) in order to enhance the overall performance results

We can see in Figures 17-19 the performance obtained for every single memory combination with and without Custom Instruction and classified by processor type. In order to be able to make a comparison, we also show the enhancement obtained with respect to the monocycle custom instruction in our previous approaches [37-38]. Additionally, we present the performance compared against the reference design (number 1). Color columns represent the improvement when custom instruction is not enabled (CI OFF), and superposed gray columns show the performance when custom instruction is enabled (CI ON).
The results obtained when running the Nios II/e processor are translated into three groups of configurations. The best performance group is formed of configurations 2 and 3 (up to nearly 60%) due to the fact that the program text is allocated in the On-chip memory. The second best performance group is formed of configurations 6 to 8 (up to nearly 35%) due to the fact that the stack is allocated in the On-chip memory. The third performance group is formed of configurations 1, 4, and 5 (up to nearly 3%), due to both program text and stack are allocated in the SDRAM memory.
Focusing on the Nios II/s processor, the results obtained are translated again into two main groups of configurations. The best performance group is formed of configurations 6 to 8 (up to nearly 75%) due to the stack being allocated in the On-chip memory, and the other one is formed of configurations 1 to 5 (up to nearly 5%) due to the fact that using the instruction cache of this processor and allocating program text in the On-chip memory does not have any effect.
Regarding the Nios II/f processor, the best performance group is formed of configurations 6 to 8 (up to nearly 60%) due to the fact that the stack is allocated in the On-chip memory, and formed of configurations 1 to 5 (up to nearly 3%) due to the fact that using the instruction cache of this processor and allocating program text in the On-chip memory barely has any effect.

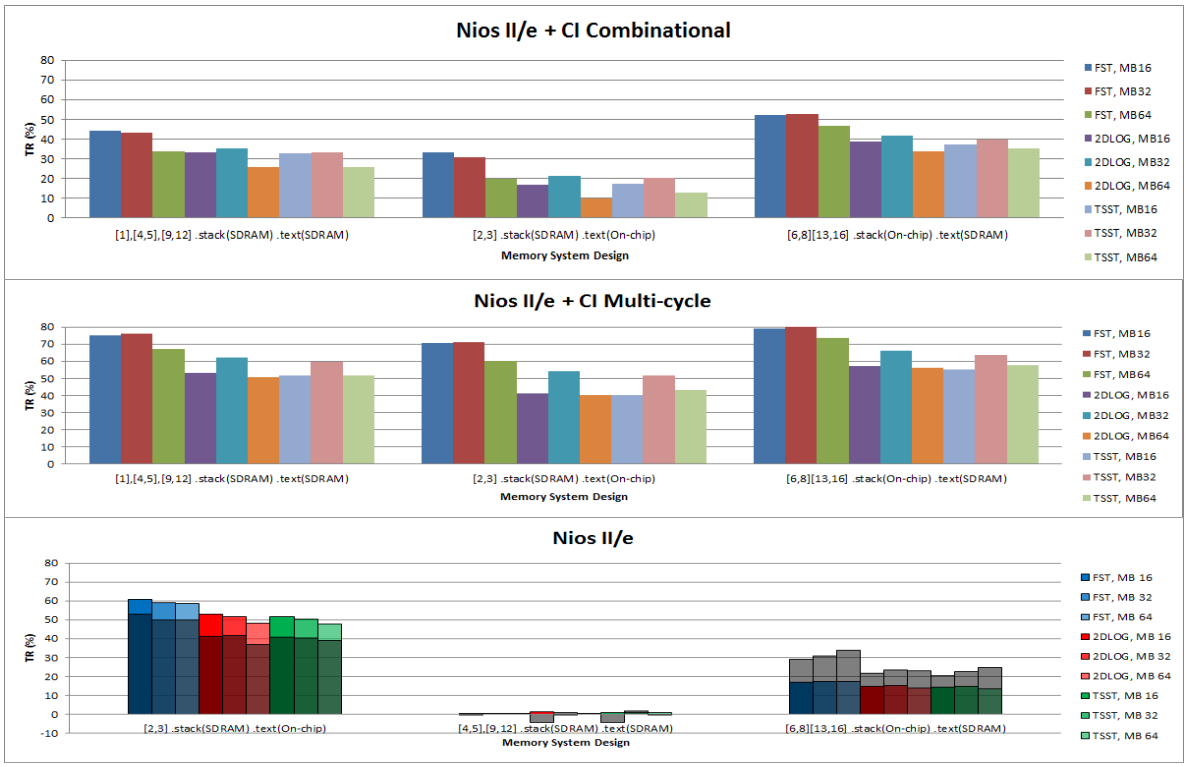


Fig. 17. Time Reduction (%) Vs Custom Instruction+ Memory Optimization for combinational custom instruction [37] and multi-cycle custom instruction regarding Nios II / e. Also is shown the enhancement regarding baseline.

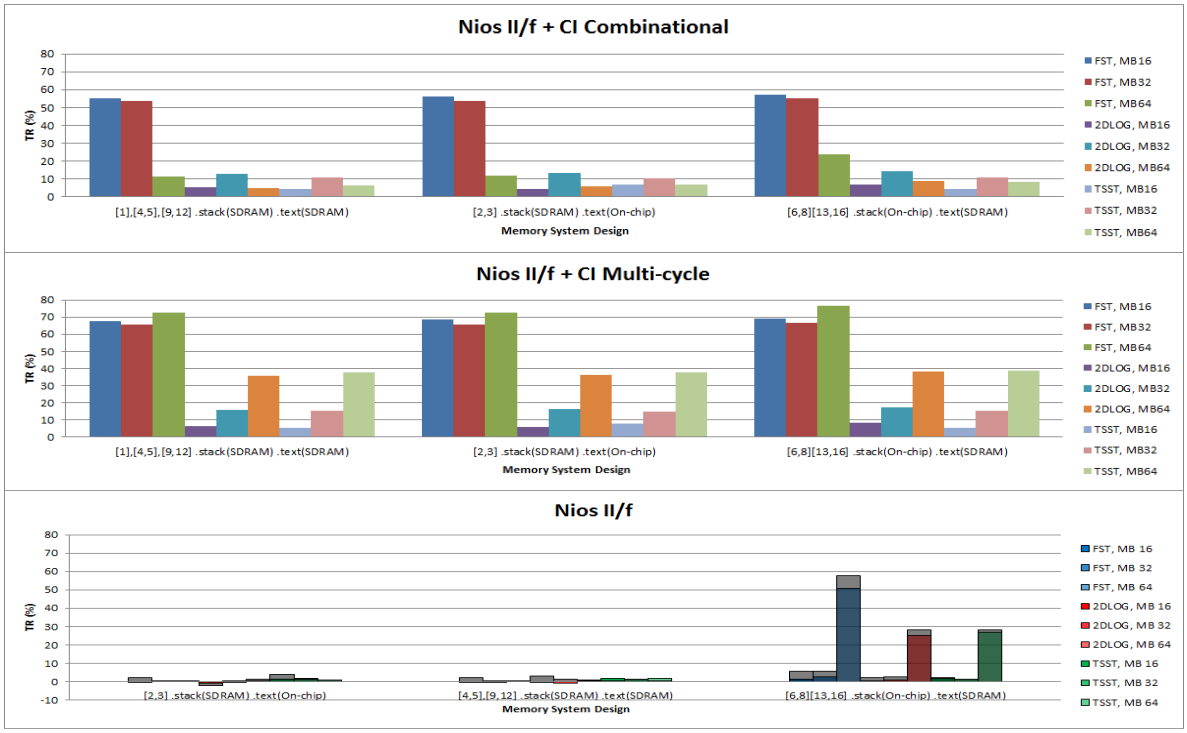


Fig. 18. Time Reduction (%) Vs Custom Instruction+ Memory Optimization for combinational custom instruction [37] and multi-cycle custom instruction regarding Nios II / f. Also is shown the enhancement regarding baseline.

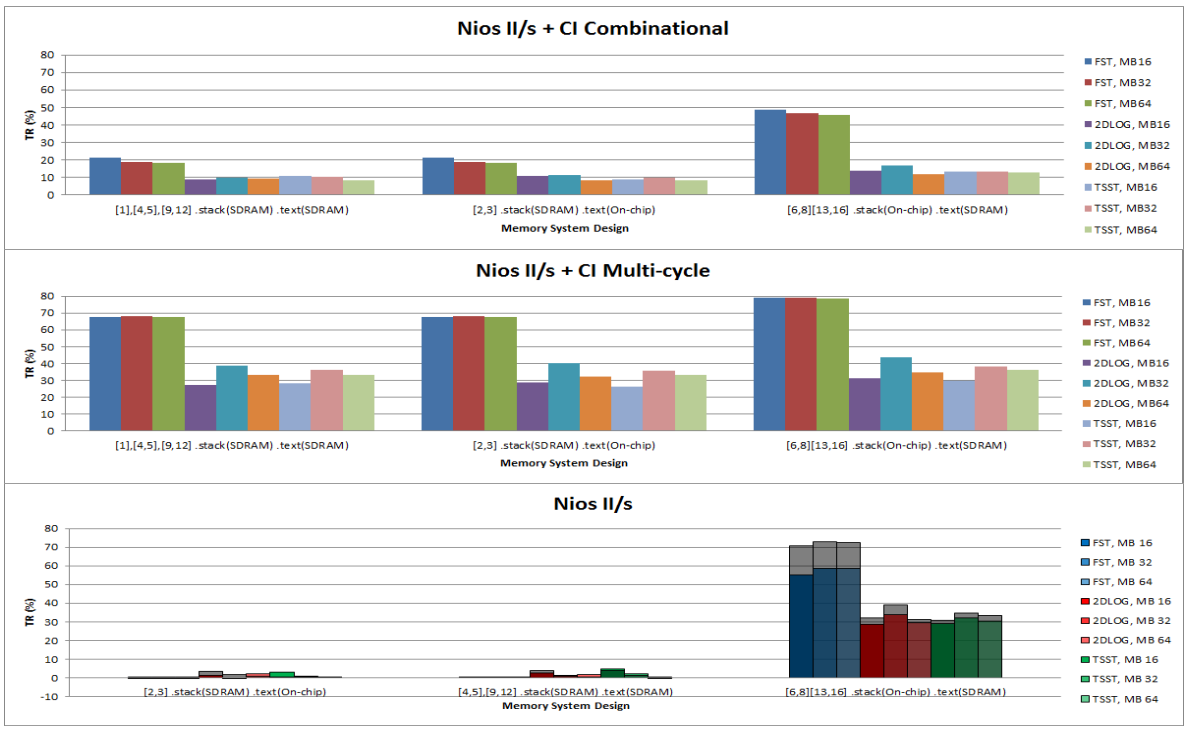


Fig. 19. Time Reduction (%) Vs Custom Instruction+ Memory Optimization for combinational custom instruction [37] and multi-cycle custom instruction regarding Nios II / s. Also is shown the enhancement regarding baseline.

It does not matter what the selected macroblock size or the executed algorithm is, the Nios II/e processor always achieves better results when activating the custom instruction with the group formed of configurations 6 to 8 and 13 to 16, between slightly more than 30% and 50% depending on the macroblock size and the algorithm executed, due to the stack being allocated in the On-chip memory. In the case of the group formed of configurations 2 and 3, it achieves improvements of between more than 10% and more than 30% when turning our custom instruction on, depending also on the macroblock size and the technique used. Although the group formed of configurations 1, 4, and 5 spends more time executing our combinatorial custom instruction or not, when using it the improvement achieved compared with the use of the *GetCost* source code is greater than the one achieved with the group built of configurations 2 and 3. Indeed, it is between more than 20% and more than 40% depending on the said factors.
Regardless of the macroblock size or the executed algorithm, the Nios II/s processor always achieves a better performance when turning our custom instruction on with the group formed of configurations 6 to 8 and 13 to 16, due to the fact that the stack is allocated in the On-chip memory. This group achieves improvements from slightly more than 10% to nearly 50% when turning our custom instruction on, depending on the algorithm executed and the selected macroblock size. On the other hand, there are improvements which vary from nearly 10% to slightly more than 20% depending on those factors when activating our custom instruction instead of translating the source code into the processor instruction set, due to the fact that using the instruction cache of this processor and allocating program text in the On-chip memory does not have any effect.
Regarding the case of turning our custom instruction on, with the Nios II/f processor, all the designs are inside the same group of configurations regardless of the selected macroblock size. This is due to the fact that only the FST algorithm using macroblock sizes of 16 and 32 calls the custom instruction many more times. Therefore, the designs present an improvement from nearly zero up to more than 50% depending on the algorithm executed and the macroblock size.
Regarding all the Nios II processor types, we can conclude that classification of the memory system designs grouped by the improvements achieved when turning our custom instruction on corresponds directly with the classification performed when grouping the memory system designs by their achieved improvement compared with design number 1, except when dealing with the Nios II/f processor. In spite of this, the best performance group of the memory system designs does not always correspond to the best performance group of memory system designs when turning our custom instruction on, as happens when running on the Nios II/e processor. Although on the Nios II/s processor there is a direct relation between configurations grouped by achieved improvement compared with design number 1 and configurations grouped by achieved improvement when turning our custom instruction on.

Regarding the resources consumed, we show in Table 2 the comparison when the custom instruction is used (Nios II /f) or no custom instruction is used.

Table 2. FPGA Resources measured with Quartus tool [39] with a window's size of 32 pixels.

| Method | Logic Cells | Embedded Multipliers | M4K Rams | Memory bits |
|---|---|---|---|---|
| No Custom Instruction. | 2210 (7%) | 0 | 0 | 412272 (85%) |
| Custom Instruction (mono cycle). | 3970 (12%) | 4 | 98 (96%) | 99200 (20%) |
| Custom Instruction (multi-cycle). | 4962 (15%) | 4 | 99 (97%) | 128960 (26%) |

## 5. CONCLUSIONS

This work outlines a low-cost system, mapped using very large scale integration technology, which accelerates software algorithms by converting them into custom hardware logic blocks, and shows the best combination of On-chip memory and SDRAM for the Nios II processor.

The average performance obtained is about 45% for the full set of parameters: window and macroblock sizes, algorithms and processor architecture used. The maximum throughput using this design represents an improvement of about 75% (window and macroblock sizes of 32, FST, Nios II/e processor). With the optimization of using the memory types available in the design, an improvement of 60% was achieved in the execution time. Finally, considering the combination of both techniques, an improvement of 80% was reached on average, and 90% for the optimum case. This means a considerable improvement with respect to its monocycle counterpart which corresponds to a throughput of 450 Kpps (Kilo pixel per second) or just a SoC which processes 50×50 @ 180 fps if using the 2DLOG technique with a macroblock size of 32 running on the Nios II/f processor. We are actively working on characterizing the power and energy consumption for mono and multi-cycle instructions, releasing a dense feedback between accuracy and efficiency

## 6. ACKNOWLEDGMENTS

This work has been partially supported by Spanish Project TIN 2012/32180.

## 7. REFERENCES

[1] Botella, G., Martín, H. J., Santos, M., & Meyer-Baese, U. (2010). FPGA-based multimodal embedded sensor system integrating low-and mid-level vision. Sensors (Basel, Switzerland), 11(8), 8164-8179.

[2] Botella, G., García, A., Rodríguez-Álvarez, M., Ros, E., Meyer-Baese, U., & Molina, M. C. (2010). Robust bioinspired architecture for optical-flow computation. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 18(4), 616-629.

[3] Botella, G., Meyer-Baese, U., & García, A. (2009). Bio-inspired robust optical flow processor system for VLSI implementation. Electronics letters, 45(25), 1304-1305.

[4] Botella, G., Ros, E., Rodríguez, M., García, A., & Romero, S. (2006, May). Pre-processor for bioinspired optical flow models: a customizable hardware implementation. In Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean (pp. 93-96). IEEE.

[5] Botella, G., Meyer-Baese, U., García, A., & Rodríguez, M. (2012). Quantization analysis and enhancement of a VLSI gradient-based motion estimation architecture. Digital Signal Processing, 22(6), 1174-1187.

[6] Meyer-Baese, U., Botella, G., Romero, D. E., & Kumm, M. (2012, May). Optimization of high speed pipelining in FPGA-based FIR filter design using genetic algorithm. In SPIE Defense, Security, and Sensing (pp. 84010R-84010R). International Society for Optics and Photonics.

[7] García, C., Botella, G., Ayuso, F., Prieto, M., & Tirado, F. (2013). Multi-GPU based on multicriteria optimization for motion estimation system. EURASIP Journal on Advances in Signal Processing, 2013(1), 1-12.

[8] Ayuso, G Botella, C García, M Prieto, F Tirado, GPU-based acceleration of bio-inspired motion estimation model. Concurrency and Computation: Practice and Experience. 25, 1037 –1056 (2013). doi:10.1002/cpe.2946

[9] Igual, F. D., Botella, G., García, C., Prieto, M., & Tirado, F. (2013). Robust motion estimation on a low-power multi-core DSP. EURASIP Journal on Advances in Signal Processing, 2013(1), 1-15

[10] Mota, S., Ros, E., Díaz, J., Botella, G., Vargas, F., & Prieto, A. (2003, September). Motion driven segmentation scheme for car overtaking sequences. In Proceedings of 10th International Conference on Vision in Vehicles (VIV'2003).

[11] Díaz, J., Ros, E., Mota, S., Botella, G., Cañas, A., & Sabatini, S. (2003). Optical flow for cars overtaking monitor: the rear mirror blind spot problem. Ecovision (European research project).

[12] Marpe, D.; Wiegand, T.; Sullivan, G.J. The H.264/MPEG4 advanced video coding standard and its applications. IEEE Commun. Mag. 2006, 44, 134–143.
[13] ITU-T Recommendation H.264 (draft). International standard for advanced video coding; 2003.
[14] ITU-T Recommendation H.264 & ISO/IEC 14496-10 (MPEG-4) AVC. Advance Video Coding for Generic Audiovisual Services; 2005.
[15] Konrad, J. Estimating motion in image sequences. IEEE Signal Process Mag. 1999, 16, 70–91.
[16] Kappagantula, S.; Rao, K.-R. Motion compensated interframes image prediction. IEEE Trans. Commun. 1985, 33, 1011–1015.
[17] Kuo, C.-J.; Yeh, C.-H.; Odeh, S.-F. Polynomial Search Algorithms for Motion Estimation. In Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, Orlando, FL, USA, 11 July 2012; pp. 813–818.
[18] Zhu, S.; Ma, K.-K. A new diamond search algorithm for fast block-matching motion estimation. IEEE Trans. Image Process. 2000, 9, 287–290.
[19] Zhu, S. Fast Motion Estimation Algorithms for Video Coding. M.S. thesis, Nanyang Technology University: Singapore, 1998.
[20] Koga, T.; Iinuma, K.; Hirano, A.; Iijima, Y.; Motion-Compensated Interframe Coding for Video Conferencing. In Proceedings of the IEEE National Telecommunications Conference, New Orleans, LA, USA, 15 November 1981.
[21] Liu, B.; Zaccarin, A. New fast algorithms for estimation of block motion vectors. IEEE Trans. Circuit. Syst. Video Technol. 1993, 3, 148–157.
[22] Li, R.; Zeng, B.; Liou, M.-L. A new three-step search algorithm for block motion estimation. IEEE Trans. Circuit. Syst. Video Technol. 1994, 4, 438–422.
[23] Jain, J.-R.; Jain, A.-K. Displacement measurement and its application in interframes image coding. IEEE Trans. Commun. 1981, 29, 1799–1808.
[24] [on-line] http://www.altera.com/devices/processor/nios2.
[25] Chu, P. Embedded SoPC Design with NIOS II Processor; Wiley: Hoboken, NJ, USA, 2012.
[26] [on-line] Nios II Performance Benchmarks. http://www.altera.com/literature/ds/ds_nios2_perf.pdf.
[27] [on-line] Altera: Nios Processor. Available online: http://www.altera.com/literature/lit-nio.jsp.
[28] [on-line] ARM. Architecture for the Digital World. http://www.arm.com/products/processors/classic/arm9/.
[29] [on-line] http://www.altera.com/devices/fpga/stratix-fpgas/stratix-ii/stratix-ii/st2-index.jsp.
[30] [on-line] Altera. http://www.altera.com/devices/fpga/stratix-fpgas/stratix-ii/stratix-ii/st2-index.jsp.
[31] [on-line] http://www.altera.com/devices/processor/nios2/benefits/performance/ni2-acceleration.html.
[32] [on-line] http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf.
[33] [on-line] https://launchpad.net/codeblocks..
[34] [on-line] Yushin, C. CIPR Sequences. Available online: http://www.cipr.rpi.edu/resource/sequences/ .
[35] [on-line] http://www.altera.com/education/univ/materials/boards/de2/
[36] [on-line] Altera. Cyclone II FPGAs at Cost That Rivals ASICs. Available online: http://www.altera.com/devices/fpga/cyclone2/ cy2-index.jsp.
[37] González, D., Botella, G., García, C., Prieto, M., & Tirado, F. (2013). Acceleration of block-matching algorithms using a custom instruction-based paradigm on a Nios II microprocessor. EURASIP Journal on Advances in Signal Processing, 2013(1), 118.
[38] González, D., Botella, G., Meyer-Bäse, A., & Meyer-Bäse, U. (2013, May). Optimization of block-matching algorithms using custom instruction-based paradigm on NIOS II microprocessors. In Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series (Vol. 8750).
[39] [on-line] Quartus II tool. Available online: http://www.altera.com/education/univ/software/quartus2/unv-quartus2.html