Zero-overhead loop controller that implements multimedia algorithms

Article *in* IEE Proceedings - Computers and Digital Techniques · August 2005

DOI: 10.1049/ip-cdt:20041187 · Source: IEEE Xplore

Nikolaos Kavvadias
Silexica
43 PUBLICATIONS **305** CITATIONS

Spyridon Nikolaidis
Aristotle University of Thessaloniki
239 PUBLICATIONS **1,661** CITATIONS

Zero-overhead loop controller for implementing multimedia algorithms

Nikolaos Kavvadias and Spiridon Nikolaidis

Section of Electronics and Computers, Department of Physics,

Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece

Email: nkavv@skiathos.physics.auth.gr

Abstract

Multimedia algorithms in their majority consist of regular repetitive loop constructs. In this paper, a novel control unit design for implementing such loop intensive algorithms is described. The proposed architecture, termed as zero-overhead loop controller (ZOLC) exploits the regularity of computations, which is a common characteristic of multimedia algorithms in order to efficiently support the corresponding datapaths. The ZOLC controls the operations in datapath modules by activating/deactivating their corresponding controlling FSMs. Algorithmic flow dependencies, which determine the appropriate loop sequencing are mapped on a look-up table (LUT). For another algorithm to execute, LUT context and FSM configurations only have to be reprogrammed, assuming a generic datapath. Thus, partial reconfiguration possibilities for implementing multimedia algorithms on programmable platforms can be exploited. As proof-of-concept, implementations of algorithms of the multimedia domain are investigated to evaluate the performance of the proposed unit, against other methods of control. Also, a full-search motion estimation processor employing the ZOLC is synthesized. It is proven that the ZOLC provides flexibility by supporting various algorithms of the multimedia field with performance improvements up to 2.1 over conventional control methods.

1 Introduction

The popularity of multimedia systems used for computing and exchanging information is rapidly increasing. Especially during the last decade, many computer architectures optimized for multimedia processing to increase the performance achievements have been proposed. Furthermore, with the emergence of portable multimedia applications (mobile phones, laptop computers, video cameras, etc) the power consumption has been promoted to a major design consideration [1]. Consequently, there is great need for power and performance optimized architectures that can introduce large savings compared to conventional approaches.

Two general implementation approaches exist, to meet this demand. The first is to use general-purpose instruction set processors. This choice offers programmability but requires increased power while achieving relatively poor performance. The second is to involve Application-Specific ICs (ASICs) or Application Specific Instruction set Processors (ASIPs) to efficiently match the application profile. This solution leads to high performance and reduced power consumption, however with reduced flexibility. In recent work, research efforts focus on performance enhancing mechanisms either to provide efficient hardware support for overhead computations as address calculations, or to exploit the execution flow characteristics of the targeted applications for minimizing power and/or execution cycles due to branching operations [2].

In this work, an architectural solution named zero-overhead loop controller (ZOLC) is presented that can be used for eliminating the branching overheads in the execution of structured algorithms for any combination of loops. The novel characteristics are single-cycle branching against five or more cycles in general-purpose processors and flexibility compared to the limitations of controllers found in literature. For the loop statement indexing, initial, step and final values are kept in local register files and only a single process unit is employed for calculating the control schedule for the whole loop structure. As the controller runs, full access to the values of the indices is provided, at any time, so that any function requirement, e.g. address generation, can be satisfied. Also loop statements with variable index bounds are supported.

The remainder of this paper is organized as follows. Section 2 overviews previous research regarding the overhead operations related to looping. In Section 3, the special characteristics of the multimedia applications are studied. The zero-overhead loop controller is presented in Section 4. Details for the hardware design of case studies from the considered application set are given in Section 5. In Section 6, results against other control methods are unfolded. Finally, Section 7 summarizes the paper.

2 Related work

In recent literature, two different approaches for treating the looping overhead in signal processing applications can be distinguished. The first one involves the use of a small instruction buffer or cache, respectively termed as loop buffer and loop cache, to hold frequently executed program loops. In the first pass of a loop, instructions from main memory are fetched, decoded and copied to the loop buffer, while in subsequent passes, the previously decoded instructions are used. When executing from the loop buffer, the program memory and instruction decoder can remain idle, which has a positive impact on the system power consumption [3],[4]. The second method regards reducing the cycle overheads due to looping by using either zero-overhead loop instructions or specialized hardware units employing control mechanisms for updating the index values and switching between loops. This concept can provide the means for better performance with expected gains in the system energy consumption.

Techniques that reside in the first category have been proposed in [5],[6],[7]. The loop cache fill mechanism is controlled by either a state machine for detecting a single loop [5] or a stack-based controller supporting nested loops [7]. Generally, this method is not able to reduce total execution time, since the branch instructions cannot be surpassed. However, power is reduced due to the smaller capacitance switched when the accesses are made to the internal buffer layer.

The second approach is often encountered in commercial DSP processors [8],[9] where hardware mechanisms are provided for zero overhead switching between loops. In [10], a microprogrammed control unit that accounts for nested loops is presented, however performance comparison results against other loop branching approaches are not mentioned for any application. The DSP56300 [9] supports seven levels of nesting using a system stack. There is a 5-cycle overhead for preparing a loop for this type of hardware control, which may be important for small number of iterations or for linear loops placed at a certain nest level. In our work such overheads are eliminated.

An alternative implementation is found in [2],[11] where a hardware unit is used to handle loop nesting up to five levels which suffices for the studied benchmarks. Its main advantage is that successive last iterations of nested loops are performed in a single cycle. In contrast to our approach, only fully-nested structures are supported and the area requirements for handling the loop increment and branching operations grow proportionally to the considered number of loops which implies high levels of redundancy. In addition, the design complexity of the priority encoder that determines which of the loop counters to increment should exaggerate for a greater maximum number of loops. Also, this unit cannot be efficiently used with any datapath since a certain parallelism is assumed to perform several operations per cycle [11].

In our approach, a zero-overhead loop controller is presented for handling loop structures in a much more efficient way. To our knowledge, it is the first time that a control method is proposed for accommodating complex loop structures with any combination of nested or linear loops. Only a single unit is utilized for calculating the indices for all loops present in the algorithm, instead of dedicated units for each loop resulting in significant area savings compared to [11]. The presented method can be applied to loop structures with loop parameter values changing at run-time. This particular case cannot be serviced by any of the previously discussed methods. With the proposed architecture we succeed in improving performance figures over other control methods under reasonable hardware cost.

3 Multimedia algorithm characteristics

Application programs of the multimedia domain spend about 90% of their execution time in small data-processing kernels comprising of loop statements [12]. More complicated algorithmic structures are often the outcome of the application of transformations on the multimedia code. A methodology for the derivation of such transformations has been proposed in [13],[14] in order to reduce power consumption of data-dominated applications. These are data-reuse transformations that introduce additional loops in the original algorithm code, which imply the existence of memory hierarchy layers closer to the processor.

A general form of a multimedia algorithm is shown in Fig. 1. It consists of a loop structure where data processing tasks are positioned. We distinguish two types of data processing tasks. The first type noted as star in Fig. 1 corresponds to tasks, at the innermost or closing position of a loop, with the final task, *bwd0*, marking the exiting position of the loop-intensive segment of the program. The loop indices are updated during the execution of these tasks, which are designated as backward (bwd) tasks. Bwd tasks are always situated in a loop terminating position even when no processing task is implied. Tasks of the second type are quoted with a circle and are placed in non-terminating positions of a loop. Such tasks are termed as forward (fwd), can take part in control flow decisions and have no effect on the loop indices. In a complete case when all possible tasks exist in an algorithm with n loops, there is a maximum of $2n+1$ total tasks. There may be up to n fwd tasks and $n+1$ bwd tasks.

From Fig. 1 it can be seen that a data processing task corresponds to a sequence of operations defined in the algorithm. In the case of a programmable processor, these operations are implemented as instructions, which can be single- or complex multi-cycle, as it is common for ASIPs.

4 Architecture of the zero-overhead loop controller

4.1 Architecture template for implementing multimedia algorithms

A generic block diagram of the proposed architecture for implementing multimedia algorithms is shown in Fig. 2. On the control path, reside the ZOLC and a set of distributed FSMs for controlling the execution of the data processing tasks of the algorithm on distributed datapath modules. A distributed topology is not mandatory, however, by its introduction, the applicability of our concept is even more understandable.

The main task of the ZOLC is to direct the datapath controlling procedure so that the appropriate FSM undertakes control of the datapath. The ZOLC determines the current loop, activates the appropriate FSM and updates the loop indices. The task sequencing information is stored in a LUT. On completion of a data processing task, an entry is selected from the LUT to address the succeeding data processing task and the loop parameter blocks, based on which task has completed and the status of the current loop. The initial, final and step loop parameters are used to calculate the current index value and determine if a loop has terminated. By employing these mechanisms, the cycle overheads regarding task switching are eliminated.

As shown in Fig. 2, the ZOLC uses the loop parameters and exploits loop dependencies described in the LUT, which is incorporated in the *loop_count_unit*, whereas the *index_calculation_unit* updates the index values (*indices*). With the update of the final value of the current index, a loop termination signal (*loop_end*) is activated. The *loop_end* signal and the completion signal from the currently active FSM are involved in the activation procedure for the appropriate local FSM.

4.2 Detailed view of the zero-overhead loop controller

A detailed block diagram of the ZOLC is shown in Fig. 3. The loop parameters are stored in their corresponding memories. These are available to the *for_structure_unit* as indicated by the bold connection in Fig. 3, which combined with the *index_control_unit* and *loop_end_mem* register form the *index_calculation_unit*. In the *for_structure_unit* the indices are calculated for the running loop. The loop sequencing information is mapped on the LUT placed in the *loop_count_unit*. The local FSMs to control the bwd and fwd tasks are positioned outside the ZOLC. To support an algorithm with loop parameters that are not statically determined, their updated values (*initial_upd*, *step_upd*, *final_upd*) are stored back in the respective storage modules as calculated at run-time.

In Fig. 4, the address and data word formation of the LUT are given. A LUT data word consists of fields to specify a loop (*loop_addr*), identify the task type (*FSMsel*), and select the specific fwd FSM of a given loop (*fwdsel*), e.g. *fwd1_0* or *fwd1_1* in Fig. 1. The data word combined with *gloop_end* defines the next address word to the LUT and selects the succeeding task. *gloop_end* is generated in the *loop_count_unit* by ANDing the *loop_end* signal, with bwd FSM termination signal (*FSMbwd*). When switching from a fwd task, the value of *gloop_end* is irrelevant.

The *for_structure_unit* is exhibited in Fig. 5. This unit calculates the current index by adding the step to the initial or intermediate value. Index update is performed only on the completion of a bwd FSM. Multiplexers MUX1 and MUX2 are used for passing correctly the initial, or stepped values as needed. With the beginning of the execution of a task, the *loop_end* signal applied to MUX2 passes the *initial* value for the current loop and on the following iteration, the *mux_sel* signal on MUX1 activated by the *index_control_unit*, selects the *initial* to be added to the *step* value to produce the next index value. The operation of the *index_control_unit* is to acknowledge the status of the running loop to the *for_structure*. The following index values are computed by iteratively adding the *step* value to these results. A local register file with one read and one write port stores the loop indices and is controlled by *load* signal issued from the corresponding bwd FSM (its *FSMbwd* signal). In case multiple indices are needed during the same cycle for data or address computations, a register bank configuration can be used instead. The output of the comparator unit (*loop_end_next*) is active when the subsequent iteration value is the final for the specified loop. The *loop_end_next* signal is stored in the *loop_end_mem* status register and is accessible as *loop_end* signal, on the following iteration of the same loop.

The implementation of the task FSMs is not related to the design of the ZOLC. We have developed C or HDL models for these, to obtain execution performance measurements or logic synthesis metrics in Section 6.

4.3 Description of the ZOLC generation tool

To execute a different algorithm implementation, the *loop_count_unit*, which is the only application-dependent component of the ZOLC, should be updated with the corresponding loop sequencing information. For this purpose, an automated generation tool has been created and is described in Fig. 6. This tool, named *lcugen*, produces a synthesizable RTL description of the *loop_count_unit*, which is parameterized on the width of the LUT fields and can adjust its interface in case of fwd task control flow decisions or if the *fwdsel* field is not actually needed. *lcugen* also generates the top-level VHDL model for the ZOLC, which only requires modifications according to the *loop_count_unit* interface signals.

First, the control flow graph (CFG) of the application described in ANSI C is generated using the "do_il2cfg" pass of the MachSUIF compiler [15]. Then, natural loop analysis [16] is performed on the CFG using the control flow analysis library of MachSUIF, to extract the loops in the algorithm. The loop analysis report contains the loop nesting depth and three additional boolean flags for determining: a) if a loop begins at the specified node (*begin_node*), b) if a loop ends at the specified node (*end_node*), c) if an exit from the loop is possible from that node (*exit_node*). From these results, the control flow of the algorithm can be mapped to its data processing task graph (DPTG). Also, the user can specify with an annotation file, which tasks should be disregarded or implement loop entry decisions, and as a result the DPTG is correspondingly rearranged.

lcugen generates a representation of the *loop_count_unit* by interpreting the edge list for the DPTG which is a weighted graph, in the following forms: the synthesizable VHDL code for the LUT, a visual representation of the DPTG in VCG format [17] or the FSM implementation for the entire *loop_count_unit* in VHDL.

5 Case study algorithm implementations

The proposed architecture was used as the control unit for the implementation of the Full-Search Motion Estimation (*fsmeorg*) algorithm. ASIC-like design was used for the datapath units. Motion estimation is used in MPEG video compression [18] for removing the temporal redundancy in a video sequence, which is determined by the similarities present amongst consecutive pictures. Compression is achieved by encoding only the displacement values of pixel blocks (motion vectors) between successive frames. The calculation of the motion vector is performed by means of a matching or distance criterion, a cost function for minimizing the prediction error [19]. We adopt the Sum of Absolute Differences (SAD) computation since it is considered suitable for VLSI implementation.

In addition to the *fsmeorg* algorithm, an application set consisting of three more benchmarks was used for verification and also the performance evaluation of algorithm implementations using the ZOLC. A dedicated processor for full-search motion estimation with data-reuse transformations applied (*fsme_dr*) was designed. Also, cycle-accurate C models incorporating the proposed ZOLC, based on the block-based matrix multiplication, *matmult*, and the row-column decomposition DCT algorithm, *rcdct*, implementations, were simulated to obtain performance results. For the *fsmeorg* and *fsme_dr* algorithms, the VHDL models of the processors were also run and the actual outputs were verified against the reference software implementations.

5.1 Overview of the case study processors

In Fig. 7, the pseudocode of the *fsmeorg* algorithm is shown. It consists of three double nested *for* loops, that incorporate the data processing tasks of the algorithm. The outer (*x,y*) loops select the block from the current picture for which the minimum motion vector is calculated. By iterating the (*i,j*) couple, each time a reference block is selected from the reference window. Initially, the *dist* variable is cleared, in order to accumulate the distance metric for the selected block. For each position in the search region, the distance kernel is executed, and this is performed for all (*k,l*) pixels in the current picture block.

Four distinct tasks are served in the algorithm flow, which are denoted as *fwd2(0)*, *fwd4(0)*, *bwd4* and *bwd6*. The *fwd2(0)* and *fwd4(0)* tasks correspond to initializing the *min* and *dist* variables. Task *bwd6* implements the SAD criterion by accumulating the absolute difference of two input pixels from the current picture and reference picture. In task *bwd4*, the SAD value is acclaimed as the new minimum if it is smaller than the current value stored in the *min* register. The corresponding (*i,j*) determines the reference block displacement and constitutes the associated motion vector.

Regarding the other benchmarks, *fsme_dr* consists of a loop structure with 20 loops and contains forward tasks that implement control flow decisions. The *matmult* algorithm comprises of 5 fully nested loops and the *rcdct* has an aggregate of 18 loops, with a maximum loop depth of 5.

A design for the motion estimation processor is implemented in order to show a complete case study that uses the ZOLC. It is not our intention to compare this rather obvious datapath design against optimized hardware solutions as those found in recent work on reconfigurable architectures for media processing [20]. For serving the data processing tasks of the algorithm, the corresponding local controlling FSMs and datapaths have been designed. The specific operations executed at each state of these FSMs are described in Table 1.

An overall view of our motion estimator design is shown in Fig. 8. In the control path, the algorithm dependencies have been recorded as context information in the *loop_count_unit* of the ZOLC. The ZOLC provides the *FSMsel* and *loop_addr* signals that select the corresponding local controlling FSM for the specified data processing task (in this case, the *fwdsel* signal is not needed). Also, the loop indices are made available from the ZOLC to the single-cycle *address_generator* unit that is triggered by *bwd4* and *bwd6* FSMs to calculate source and destination pixel addresses. Input data for the motion estimation kernel are read from the frame memories and fed to the *SAD_and_minimum* unit. Updated motion vectors are written back to the motion vector RAM blocks.

The *fsme_dr* algorithm makes efficient use of a customized memory hierarchy to exploit temporal locality in the data accesses [13],[14]. As reported in [13],[14] the optimal memory hierarchy consists of two individual hierarchies for the current and reference picture memories. For the current picture, the current block (*CB*) memory layer is introduced, whereas for the reference picture, the corresponding memory hierarchy incorporates memory layers for a reference window (*RW*), and a line of candidate blocks (*PB line*). This combination has been derived for a pre-characterized ASIC process based on a power-sensitive selection criterion.

The application of the data-reuse transformations introduces 14 additional loops in the algorithm description to form a total of 20 loops as shown in Fig. 9. The same architecture for the ZOLC is used as for the *fsmeorg* processor. Changes were only required for the contents and interface signals of the LUT. Also, additional tasks are involved compared to *fsmeorg*, that implement control-flow decisions as well as memory transfers from higher to lower memory layers (inter-copy reuse [13]) or in context of a single memory layer (intra-copy reuse).

The datapath for the *matmult* and *rcdct* algorithms would be designed considering a similar interface to the ZOLC.

6 Performance evaluation of the zero-overhead loop controller - Results

For the purpose of performance evaluation, we compare the efficiency of the ZOLC against five different control methods used for reducing looping overheads, which are indicated in Table 2. It is assumed that these methods are applied to the same datapath specializations in order to generate comparable results. The overhead cost in cycles for implementing the looping was determined from the instruction cycle timings reported in literature for all the considered control methods. Data and control hazards are regarded as resolved which in fact favors all implementations except the ZOLC. Variation *delay_slots* has two architected delay slots, 100% utilized, while architecture *branch_taken* follows a branch taken policy with a misprediction cost of two cycles [12]. *zol* supports zero-overhead operation for innermost loops only [8]. A 2-cycle overhead is required for loading the loop counter register and setting up a block of instructions or a single instruction for zero-overhead operation. For *dsp56300* there is a 5-cycle overhead for preparing the program control unit modules to operate in zero-overhead mode [9]. *mbreeze* assumes that the looping hardware of [11] is used. A conservative estimate of a 100-cycle penalty is associated between the detection and start of a Breeze instruction [11]. In Table 2, the percentage increase in the number of cycles is given for the aforementioned control methods against *zolc*. *zolc* provides up to 2.1 times better performance against these control architectures. Only *mbreeze* achieves similar performance figures to *zolc*, however *zolc* is more flexible and can be applied in a general context.

The ZOLC and MediaBreeze architectures implement sequencers for loop-intensive applications, following different approaches: ZOLC stores task switching context in a LUT, while MediaBreeze interprets a special instruction to accelerate a fully-nested loop computation. For this reason only the corresponding modules from these architectures are synthesized, which are the *for_structure* and MediaBreeze looping unit respectively, to derive gate count and maximum clock frequency estimates. The remaining units cannot be directly compared since the complexity of several MediaBreeze units is not detailed. The designs are targeted to the TSMC 0.18um standard cell library using the MentorGraphics' LeonardoSpectrum tool. For the MediaBreeze hardware, a technology-optimized carry select adder is designed for both additions and increments. MediaBreeze supports looping only for zeroth initial and unitary step values, which implies that for most algorithms extensive modifications have to be applied on the application code.

In Table 3, the area and clock frequency metrics for the two architectures are contrasted for a maximum number of 5, 8 and 16 loops. The MediaBreeze looping unit requires twice the hardware cost of the *for_structure*, while providing lower performance than its register file configuration. If a register bank is used in our design, performance is decreased, since input demultiplexers are required for the write enable and write-back index signals and an output multiplexer for the intermediate index. A RAM-based register file consists of compact six-transistor cells and highly optimized decoding logic, while not requiring additional circuitry with the drawback of providing a single index per cycle. Also, a large amount of signals are decoded in the MediaBreeze instruction decoder, which may additionally degrade the looping unit timing characteristics.

Finally, two versions of the ZOLC (one unpipelined and one with two-stage pipelining) and the complete processor for the *fsmeorg* application have been synthesized. Table 4 shows the corresponding metrics compared to ARM7TDMI and ARM946E synthesizable implementations [21]. The gate count is about 30K for the motion estimation processor excluding the frame and motion vector RAMs, which is significantly lower than of the ARM processors. A maximum clock frequency of 141MHz for the motion estimation processor using the unpipelined implementation of the ZOLC is achieved, which compares to the 100MHz clock frequency of the ARM7 and 160MHz of the ARM9 processor. It should be noted that the motion estimator designs provide 7 to 12 times speedup in machine cycles against ARM7 due to fast address generation and elimination of branching overheads. For the ZOLC, the critical path was found to be that from the loop parameter RAMs, across the *index_control* and *for_structure* unit and terminating to the *loop_end_mem* unit. It is quite satisfactory that the *loop_count_unit* operates fast enough not to be included in the critical path of the architecture. Also, the ZOLC can be further pipelined into a *loop_count_unit* operation stage and an index update stage, and then the maximum clock frequency is increased to about 250MHz with a minimal additional cost in area.

7 Conclusions

In this paper, a zero-overhead loop controller for implementing multimedia algorithms is introduced. The special characteristics of loop-intensive algorithms are exploited in order to provide for efficient handling of the loop branching operations. The presented architecture is able to execute structured algorithms for any combination of loops, with no cycle overheads incurred for task switching. While it operates, indices of all loops are accessible so that data or address requirements can be satisfied. The proposed architecture is documented in VHDL and its cycle efficiency is tested against established methods of control. Also, hardware characteristics are compared against other specialized architectures for loop branching. Overall, performance improvements up to 2.1 are reported against other methods of control for the same datapaths. Finally, an automation tool has been implemented for generating the VHDL description for the entire ZOLC, adapted to the application in mind.

References

[1] A. P. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*, Kluwer Academic Publishers, Boston, 1995.

[2] D. Talla, L. K. John, and D. Burger, “Bottlenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements,” IEEE Transactions on Computers, Vol. 52, No. 8, pp. 1015-1031, August 2003.

[3] R. S. Bajwa, M. Hiraki, H. Kosima, D. J. Gorny, A. Shridhar, K. Seki, and K. Sasaki, “Instruction Buffering to Reduce Power in Processors for Signal Processing,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 5, No. 4, pp. 417-425, December 1997.

[4] N. Bellas, I. N. Hajj, C. D. Polychronopoulos, and G. Stamoulis, "Architectural and Compiler Techniques for Energy Reduction in High-Performance Microprocessors," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 8, No. 3, pp. 317-326, June 2000.

[5] L. H. Lee, W. Moyer, and J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops," Proceedings of the International Symposium on Low Power Electronics and Design, August 1999, San Diego, CA.

[6] A. Gordon-Ross, S. Cotterell, and F. Vahid, "Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example," IEEE Computer Architecture Letters, January 2002.

[7] C. T. Wu and T. T. Hwang, "Instruction Buffering for Nested Loops in Low Power Design," Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), 26-29 May 2002, Scottsdale Princess Resort, Scottsdale, Arizona.

[8] G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns and V. Cao, "Effective Exploitation of a Zero Overhead Loop Buffer," ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99), pp. 10-19, May 1999, Atlanta, USA.

[9] Motorola Inc., *DSP56300 24-Bit Digital Signal Processor Family Manual*, Revision 3.0, December 2000.

[10] B. W. Bomar, "Implementation of Microprogrammed Control in FPGAs," IEEE Transactions on Industrial Electronics, Vol. 49, No. 2, pp. 415-422, April 2002.

[11] D. Talla, "Architectural Techniques to Accelerate Multimedia Applications on General-Purpose Processors," Ph.D. thesis, University of Texas at Austin, August 2001.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1996.

[13] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology*, Kluwer Academic Publishers, Boston, 1998.

[14] S. Kougia, A. Chatzigeorgiou, N. Zervas and S. Nikolaidis, "Analytical Exploration of Power Efficient Data-reuse Transformations on Multimedia," International Conference on Acoustics, Speech and Signal Processing, Utah, May 2001.

[15] M. D. Smith and G. Holloway, "An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization," Technical report, Harvard University, Cambridge, MA, 2000.

[16] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.

[17] G. Sander, "Graph layout through the VCG tool," Technical Report Sep 26-34, Technical University of Munich, September 26, 1995.

[18] International Organization of Standardization, Working Group on Coding of Moving Pictures and Audio, MPEG-4 Video Verification Model Version 18.0, Pisa, January 2001.

[19] P. Kuhn, *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation*, Kluwer Academic Publishers, Boston, 1999.

[20] S. Wong, S. Vassiliadis, and S. Cotofana, "SAD implementation in FPGA hardware," in Proceedings of the 12th Annual Workshop on Circuits, Systems, and Signal Processing (PRORISC2001), 2001.

[21] ARM Ltd., http://www.arm.com

Data-processing task	State number	Operations performed at specified state
fwd2(0)	1	Load initial value to min register
fwd4(0)	1	Clear dist register
bwd6	1	Address formation for the input and output data structures
	2	Pixels p1, p2 are read from the frame memories
	3	The absolute difference of p1, p2 is calculated and the result is accumulated in dist register
bwd4	1	Comparison between the dist and min values
	2	Store the corresponding motion vector

Table 1. Datapath operations per state for the local controlling FSMs of the *fsmeorg* application

Benchmark	Description	# cycles	% increase in number of cycles compared to zolc				
		zolc	delay_slots	branch_taken	zol	dsp56300	mbreeze
fsmeorg	Full-search motion estimation	70128467	41.01	42.48	6.10	8.09	-2.08
fsme_dr	Full-search motion estimation with data-reuse transformations	50759199	52.29	53.73	8.87	11.72	0.62
matmult	Block-based matrix multiplication	1940451	50.18	52.98	18.77	22.52	-0.74
rcdct	Row-column decomposition 2D-DCT	6565753	42.37	45.14	13.50	17.20	-1.19

Table 2. Performance comparison of the ZOLC against other control methods for the benchmarks

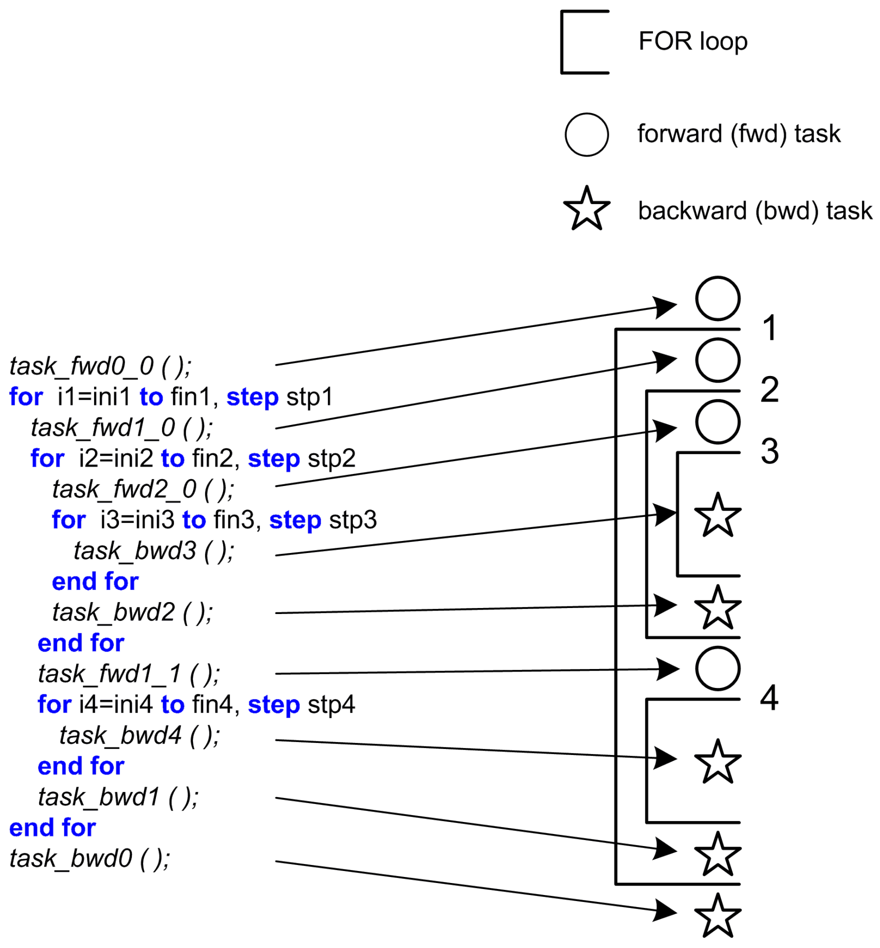
Hardware unit	Number of loops	Number of gates	Clock frequency
Mediabreeze looping	5	10837	279.7 MHz
Mediabreeze looping	8	17394	272.2 MHz
Mediabreeze looping	16	34807	233.5 MHz
for_structure (register bank)	5	6716	234.9 MHz
for_structure (register bank)	8	7827	223.7 MHz
for_structure (register bank)	16	14649	185.2 MHz
for_structure (register file)	16	3812	315.2 MHz

Table 3. Synthesis results for equivalent hardware blocks of the ZOLC unit and MediaBreeze architecture

Processor	Technology	Pipeline stages	Number of gates	Clock frequency
ARM7TDMI-S	0.18um	3	50K	100 MHz
ARM946E-S	0.18um	5	200K	160 MHz
zolc_unit	TSMC 0.18um (6LM)	1 (no pipelining)	15176	164.3 MHz
zolc_unit_pipe	TSMC 0.18um (6LM)	2	15669	253.5 MHz
fsmeorg processor	TSMC 0.18um (6LM)	2	31818	141.4 MHz

Table 4. Synthesis results for the ZOLC and the entire processor for the *fsmeorg* application

List of Figures



```
task_fwd0_0 ( );
for  i1=ini1 to fin1, step stp1
  task_fwd1_0 ( );
  for  i2=ini2 to fin2, step stp2
    task_fwd2_0 ( );
    for  i3=ini3 to fin3, step stp3
      task_bwd3 ( );
    end for
    task_bwd2 ( );
   end for
   task_fwd1_1 ( );
   for i4=ini4 to fin4, step stp4
     task_bwd4 ( );
   end for
   task_bwd1 ( );
end for
task_bwd0 ( );
```

Figure 1. General form of a multimedia algorithm

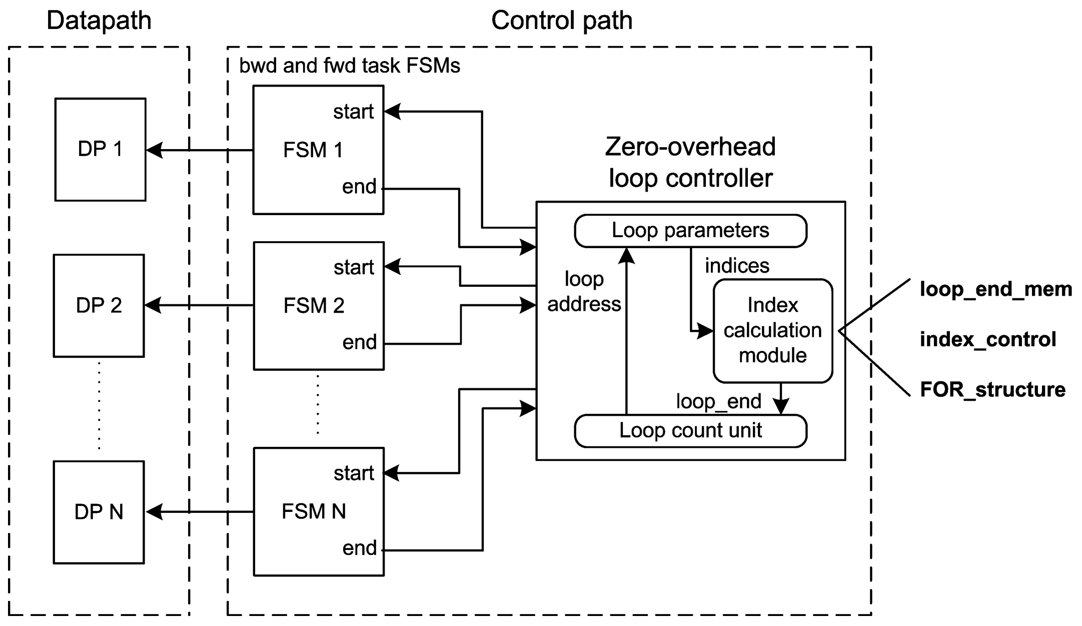


Figure 2. General template of the proposed architecture for implementing multimedia algorithms

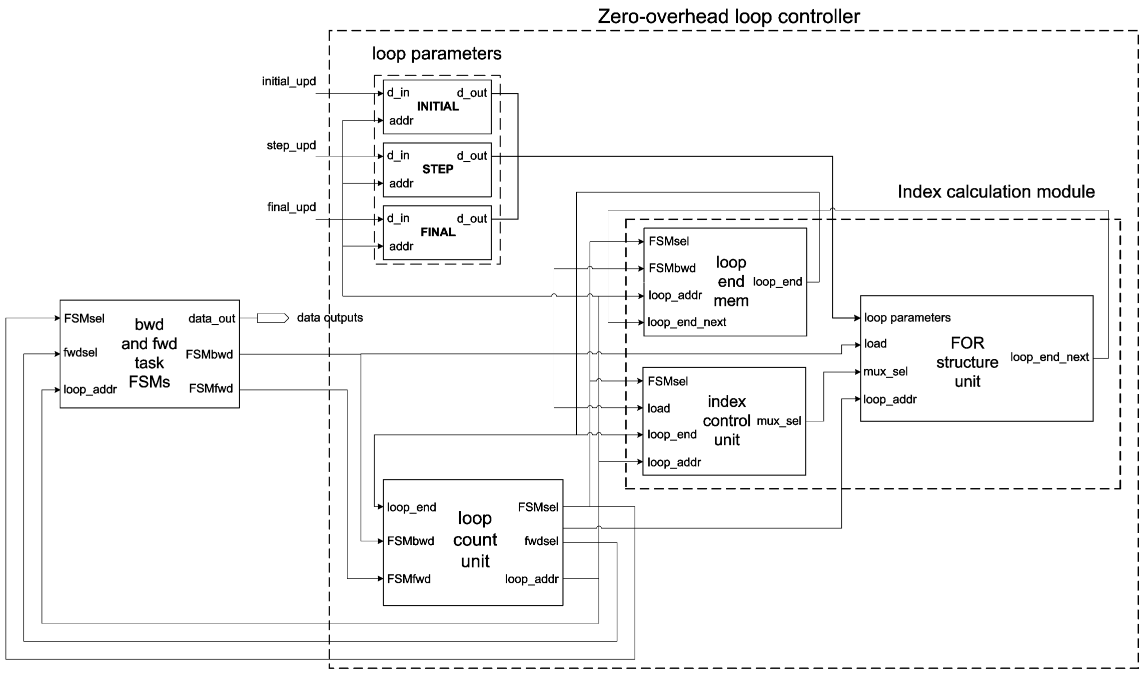
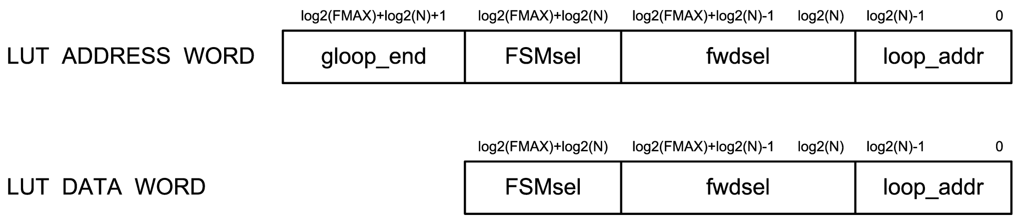


Figure 3. Zero-overhead loop controller architecture



N: number of loops in the algorithm

FMAX: maximum fwdsel field value

Figure 4. LUT address and data word formation

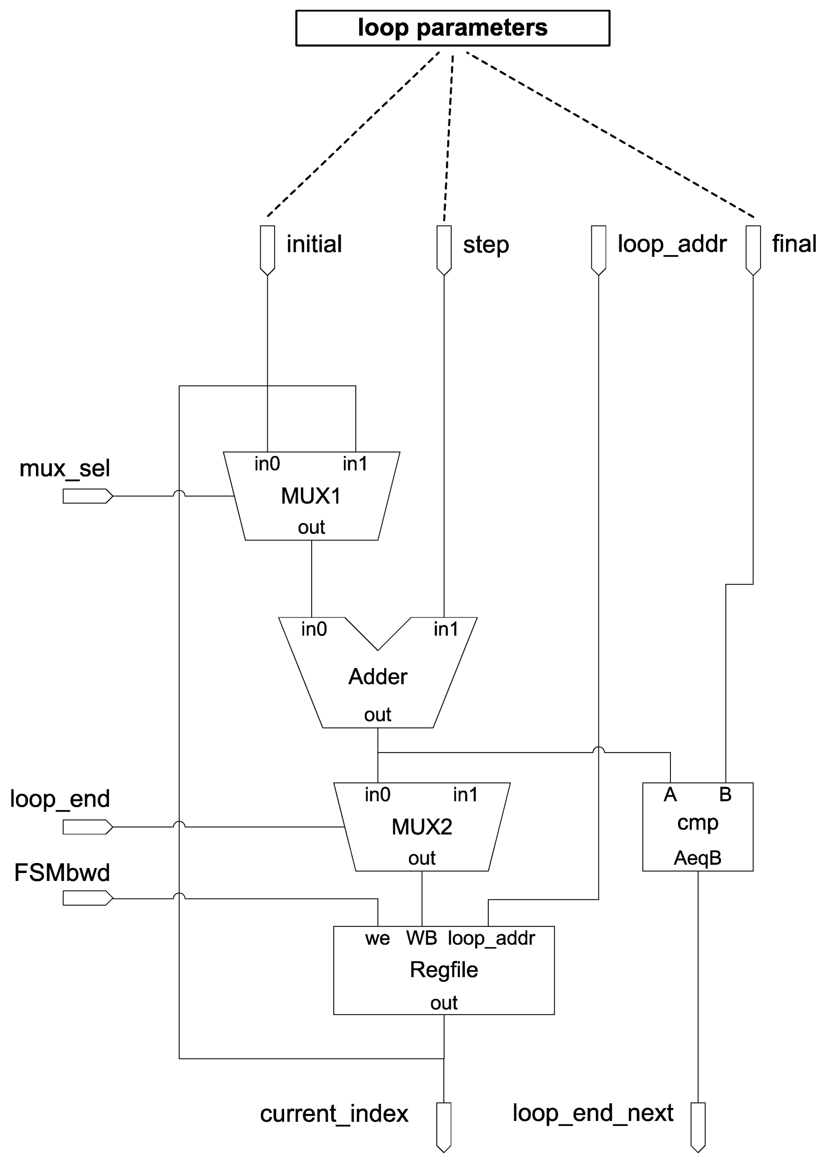


Figure 5. *for_structure_unit* block diagram

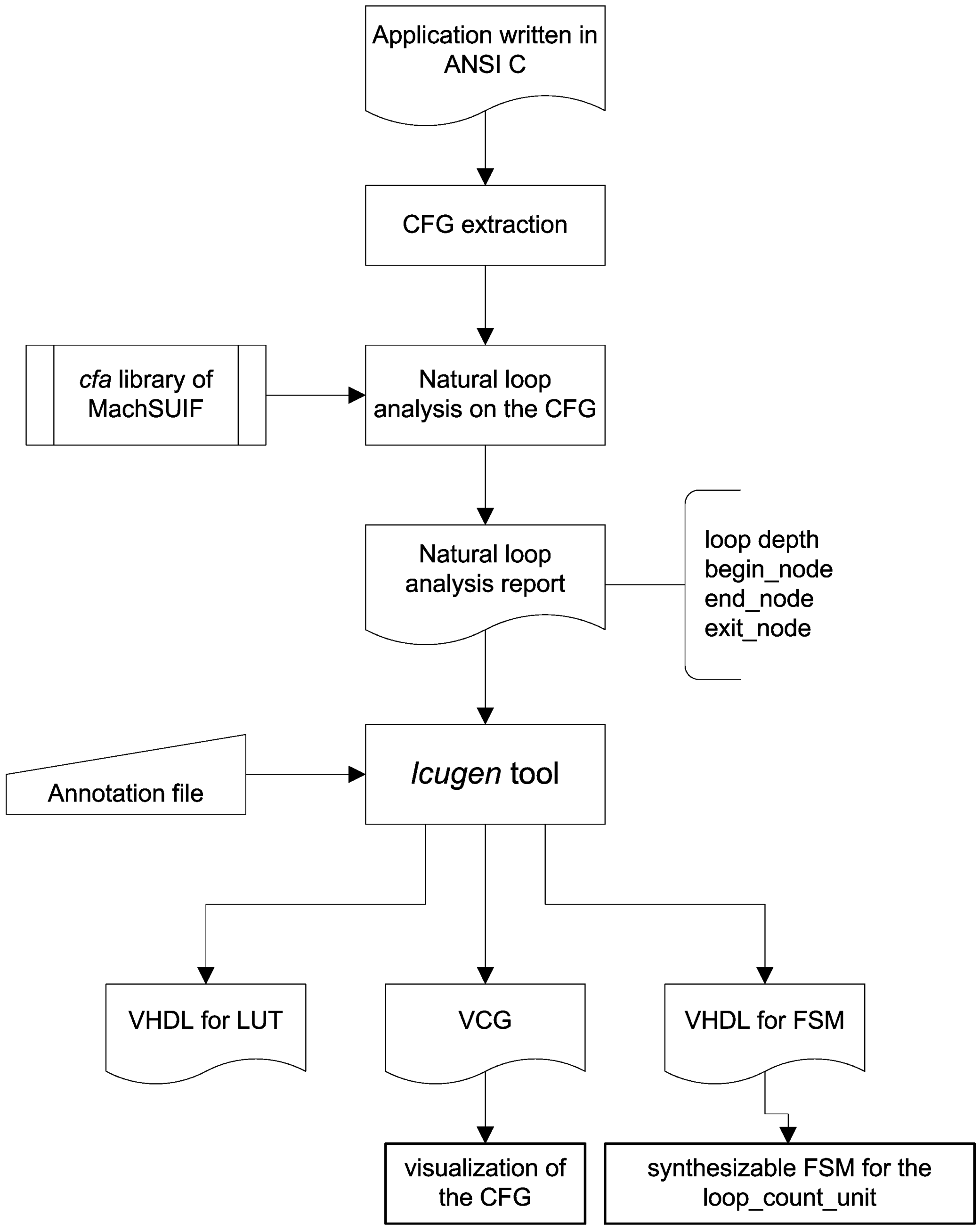
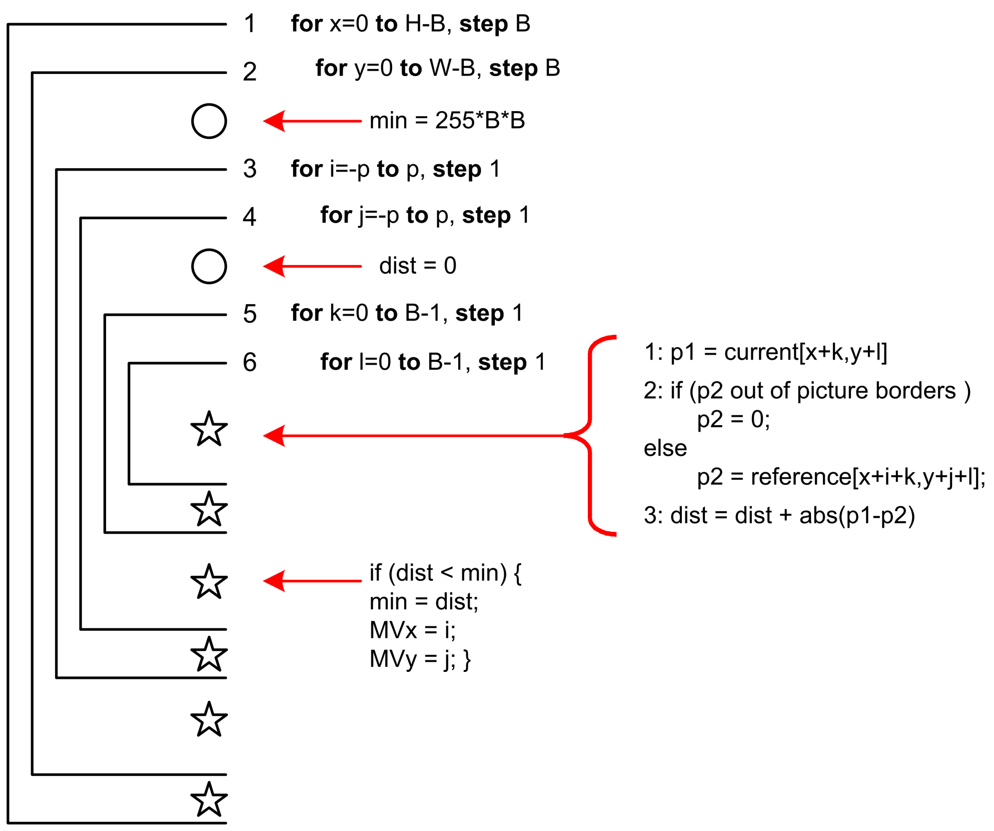


Figure 6. Overview of the process for generating the *loop_count_unit* using the *lcugen* tool



```
1   for x=0 to H-B, step B
2       for y=0 to W-B, step B
            min = 255*B*B
3   for i=-p to p, step 1
4       for j=-p to p, step 1
            dist = 0
5   for k=0 to B-1, step 1
6       for l=0 to B-1, step 1
            1: p1 = current[x+k,y+l]
            2: if (p2 out of picture borders )
                   p2 = 0;
               else
                   p2 = reference[x+i+k,y+j+l];
            3: dist = dist + abs(p1-p2)

            if (dist < min) {
            min = dist;
            MVx = i;
            MVy = j; }
```

Figure 7. Pseudocode flow for the Full Search Motion Estimation algorithm

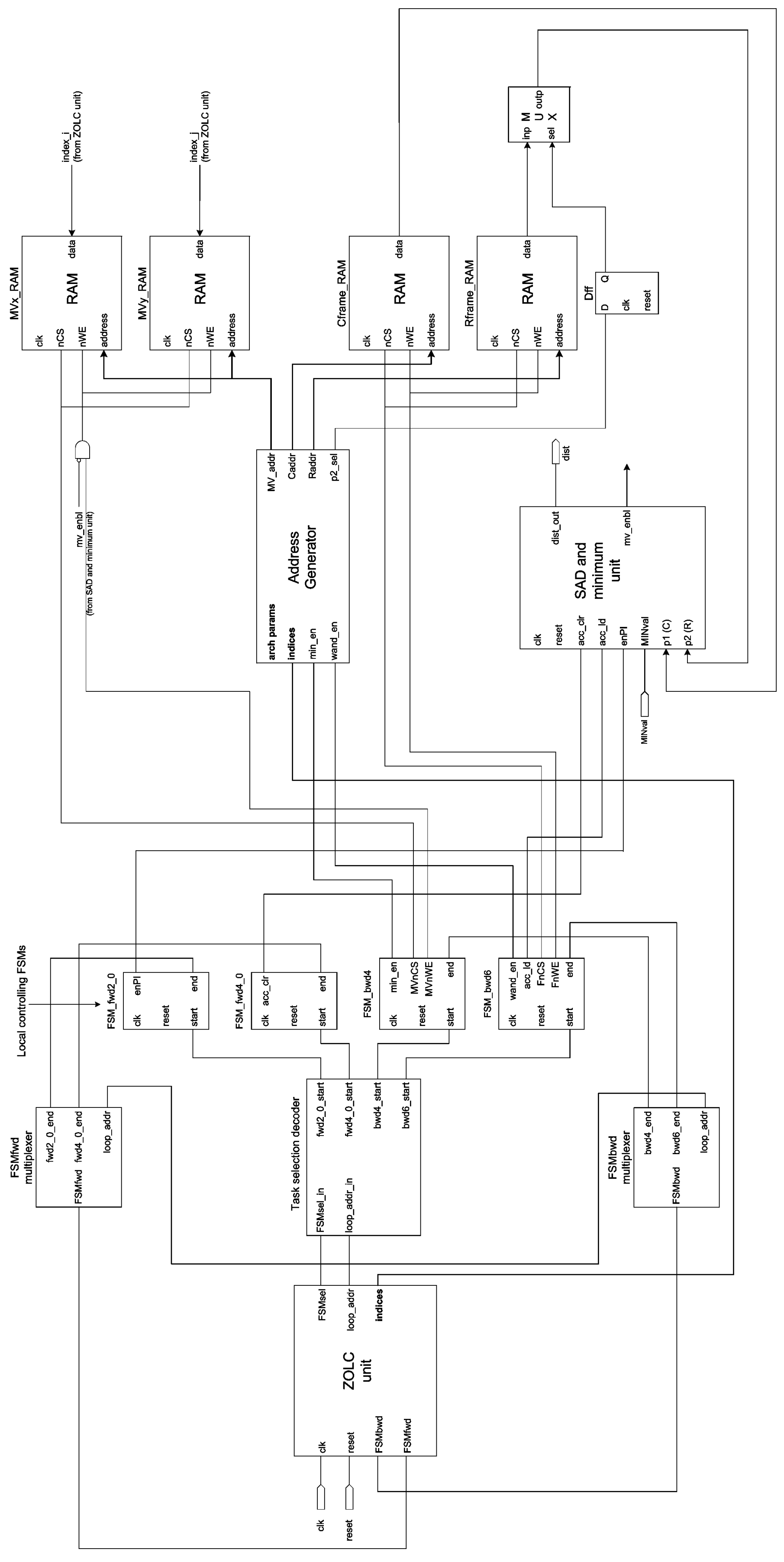


Figure 8. Block diagram view of the motion estimation processor

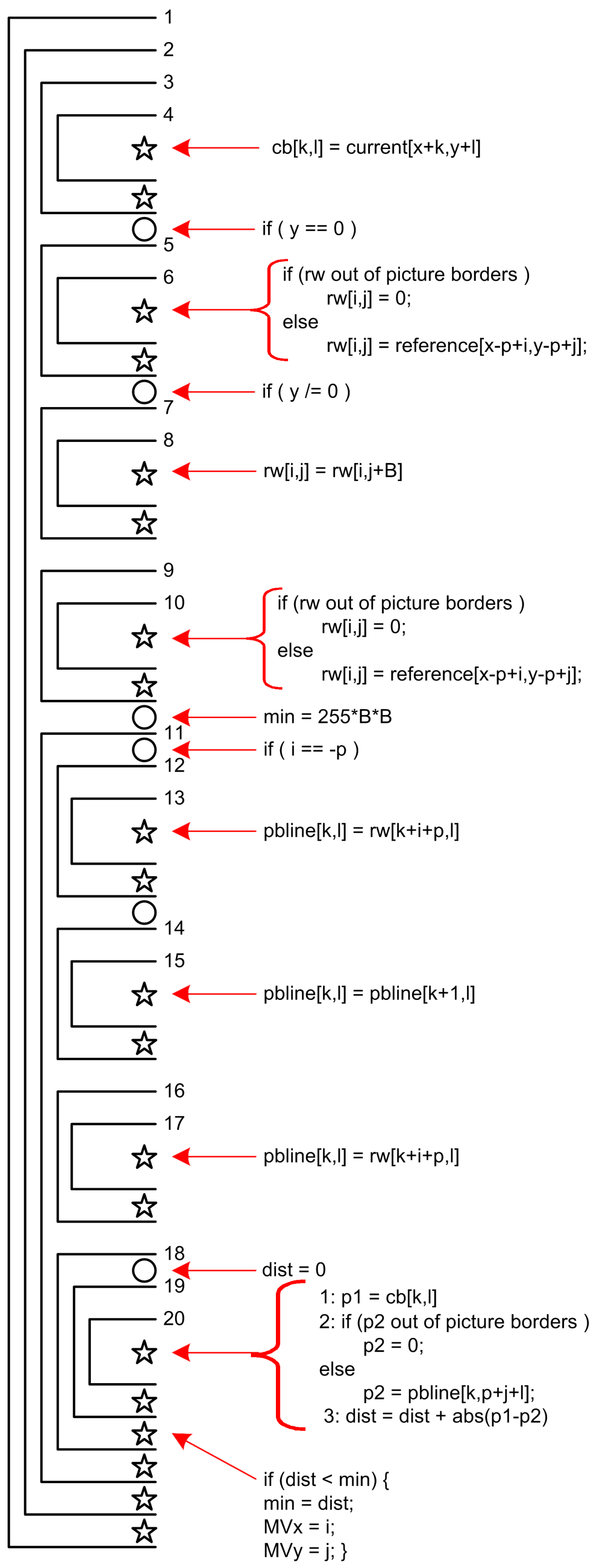

1
2
3
4
cb[k,l] = current[x+k,y+l]
if (y == 0)
5
6
if (rw out of picture borders)
rw[i,j] = 0;
else
rw[i,j] = reference[x-p+i,y-p+j];
if (y /= 0)
7
8
rw[i,j] = rw[i,j+B]
9
10
if (rw out of picture borders)
rw[i,j] = 0;
else
rw[i,j] = reference[x-p+i,y-p+j];
min = 255*B*B
11
if (i == -p)
12
13
pbline[k,l] = rw[k+i+p,l]
14
15
pbline[k,l] = pbline[k+1,l]
16
17
pbline[k,l] = rw[k+i+p,l]
18
dist = 0
19
20
1: p1 = cb[k,l]
2: if (p2 out of picture borders)
p2 = 0;
else
p2 = pbline[k,p+j+l];
3: dist = dist + abs(p1-p2)
if (dist < min) {
min = dist;
MVx = i;
MVy = j; }

Figure 9. Algorithmic flow for the *fsmeorg* algorithm after the application of data-reuse transformations