EINDHOVEN UNIVERSITY OF TECHNOLOGY

Application specific instruction-set processor template for motion estimation in video applications.

Citation for published version (APA):
Peters, H., Sethuraman, R., Beric, A., Meeuwissen, P., Balakrishnan, S., Alba Pinto, C. A., Kruijtzer, W., Ernst, F., Alkadi, G., Meerbergen, van, J., & Haan, de, G. (2005). Application specific instruction-set processor template for motion estimation in video applications. *IEEE Transactions on Circuits and Systems for Video Technology*, *15*(4), 508-527. https://doi.org/10.1109/TCSVT.2005.844462

DOI:
10.1109/TCSVT.2005.844462

Document status and date:
Published: 01/01/2005

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 25. Apr. 2025

Application Specific Instruction-Set Processor Template for Motion Estimation in Video Applications

Harm Peters, Ramanathan Sethuraman, *Member, IEEE*, Aleksandar Berić, Patrick Meuwissen, Srinivasan Balakrishnan, *Member, IEEE*, Carlos Antonio Alba Pinto, *Member, IEEE*, Wido Kruijtzer, *Member, IEEE*, Fabian Ernst, *Member, IEEE*, Ghiath Alkadi, Jef van Meerbergen, *Senior Member, IEEE*, and Gerard de Haan, *Senior Member, IEEE*

***Abstract*—The gap between application specific integrated circuits (ASICs) and general-purpose programmable processors in terms of performance, power, cost and flexibility is well known. Application specific instruction-set processors (ASIPs) bridge this gap. In this work, we demonstrate the key benefits of ASIPs for several video applications. One of the most compute- and memory-intensive functions in video processing is motion estimation (ME). The focus of this work is on the design of a ME template, which is useful for several video applications like video encoding, obstacle detection, picturerate up-conversion, 2-D-to-3-D video conversion, etc. An instruction-set suitable for performing a variety of ME functions is developed. The ASIP is based on a very long instruction word (VLIW) processor template and meets low-power and low-cost requirements still providing the flexibility needed for the application domain. The ME ASIP design consumes 27 mW and takes an area of 1.1 mm^2 in 0.13 μm technology performing picturerate up-conversion, for standard definition (CCIR601) resolution at 50 frames per second.**

***Index Terms*—Application specific instruction-set processor (ASIPs), hardware, intellectual property, motion estimation (ME), parallelism.**

I. Introduction

VIDEO processing applications have an ever increasing demand for processing power [1]. On the one hand, emerging applications like 3-DTV [2] and smart cameras [3], [4] are inherently complex while, on the other hand, traditional applications like video format conversion [5] and video compression [6] demand higher performance to achieve a better picture quality. Currently, two contrasting implementations are often considered to achieve high performance video processing: application specific integrated circuits (ASICs) and general-purpose programmable processors (e.g., ARM, TriMedia, TI's C6X). The characteristics of these devices are:

1) ASICs optimally meet performance and power requirements, but lack flexibility. The design entry is in a hardware description language like VHDL, which causes relatively long design times and also makes late specification changes difficult to handle. This may affect time-to-market adversely;
2) general-purpose programmable processors are highly flexible, but have significant overhead in achieving the performance requirements for a power budget. The advantage is that the application entry is in a high-level language like C, which results in shorter time-to-market.

We foresee that next generation video processing devices are likely to be application specific instruction-set processors (ASIPs).[1, 2, 3] They will leverage the commonalities between traditional and new video applications, while bridging the gap between ASICs and general-purpose programmable processors in terms of power, area, flexibility, design/application entry and short time-to-market.

ASIPs, tuned to an application domain, can be based on any processor architecture template such as a very long instruction word (VLIW) architecture [7], or a vector processor architecture [8]. In this work, we use the VLIW architecture template. It is interesting to note that the choice of the ASIP template architecture greatly depends on the characteristics of the application domain. For instance, the motion estimation (ME) is efficiently implementable on the VLIW architecture template. Among the available tool flows for ASIP design, namely A|RT [7], LISA [9] and CHESS [10], we use the A|RT-based tool flow in this work.

ASIPs based on a VLIW processor architectural template have the following characteristics [7], [11].

1) *Instruction-Set*: ASIPs accelerate application specific functions. The data-path of these ASIPs consists of standard functional units [like arithmetic logic units (ALUs) and address calculation units (ACUs)] and application specific units (ASUs) with different levels

Manuscript received November 14, 2003; revised March 24, 2004. This work is supported in part by the European Commission under the IST-2001-34410 CAMELLIA project. This paper was recommended by Associate Editor R. Chandramouli.

H. Peters, R. Sethuraman, P. Meuwissen, S. Balakrishnan, C. A. Alba Pinto, W. Kruijtzer, F. Ernst, and G. Alkadi are with Philips Research Laboratories, Eindhoven 5656AA, The Netherlands (e-mail: harm.peters@philips.com; ramanathan.sethuraman@philips.com; patrick.meuwissen@philips.com; srinivasan.balakrishnan@philips.com; carlos.alba.pinto@philips.com; wido.kruijtzer@philips.com; fabian.ernst@philips.com; ghiath.alkadi@philips.com).

A. Berić is with the Department of Electrical Engineering, University of Technology, Eindhoven 5600MB, The Netherlands (e-mail: a.b.beric@tue.nl).

J. van Meerbergen and G. de Haan are with Philips Research Laboratories, Eindhoven, The Netherlands. They are also with the Department of Electrical Engineering, University of Technology, Eindhoven, The Netherlands (e-mail: jef.van.meerbergen@philips.com; g.de.haan@philips.com).

Digital Object Identifier 10.1109/TCSVT.2005.844462

[1]ARC [Online] Available: http://www.arc.com.

[2]Tensilica [Online] Available: http://www.tensilica.com.

[3]Silicon Hive [Online] Available: http://www.siliconhive.com.

1051-8215/$20.00 © 2005 IEEE

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

of processing granularity (like 8-point 2-D discrete cosine transform (DCT), 3×3 pixel filtering). Design space exploration of application-specific functions can be performed using high-level-synthesis tools (HLS) to realize efficient designs.

2) *Flexibility*: ASIPs are flexible within an application domain. For instance, a carefully designed ASIP to address ME [1] can be programmed for different video applications while benefiting from the instruction-set that accelerates ME functionality. The design and application entry is in a high-level language like C, starting from a sequential description of the application.

3) *Performance/Power/Area*: ASIPs offer performance, power and area that are comparable to ASICs. ASIP implementations are orders of magnitude superior in terms of performance, power and area compared to general-purpose programmable processors for applications in their domain.

These key features of ASIPs directly translate to lower cost and shorter time-to-market. Hence, ASIPs are promising candidates for the next generation video signal processing architectures. We will demonstrate the versatility of ASIPs through a case study involving a ME ASIP template suitable for an application set consisting of video encoding [6], low-speed obstacle detection using smart cameras [3], temporal up-conversion [5], and 2-D-to-3-D video conversion [2].

As the starting point of the design, we use a behavioral description in C-language for the application set, the reference code. In the next step, we perform the hardware/software partitioning by determining the compute- and control-intensive tasks of the application set. One of the common compute-intensive tasks of the application set is the ME consisting of kernels that perform SAD and bilinear interpolation. These kernels are mapped onto hardware, while the rest of the application tasks can be realized as software. The software tasks can be mapped onto a general-purpose programmable processor (ARM,[4] MIPS[5]), while the hardware task will be mapped onto an ASIP. The partitioned application set has been simulated extensively, and compared with the reference code, since both should have exactly the same behavior. The simulations were performed at three different abstractions: partitioned C-code, register transfer language (RTL) (generated by a high-level synthesis tool [7]) and netlist (generated by gate-level synthesis tool).[6] All three levels of simulation were carried out using a bit- and cycle-true communication protocol [12] for the communication between hardware and software tasks. From the C-language description of the hardware task we automatically derive the hardware description language (VHDL) of the VLIW-based ASIP through the toolset A|RT [7].

This paper is organized as follows. Section II describes the application set considered in this work with emphasis on the ME parts of the applications. In Section III, we present the design methodology that includes hardware/software co-design and ASIP design methodology. Section IV deals with the choice of instruction-set for ME functionality and provides the design details to support the instruction set. In Section V, we present the results of the ASIP design. We draw our conclusions in Section VI.

[4]ARM [Online] Available: http://arm.com.

[5]MIPS [Online] Available: http://www.mips.com.

[6]Cadence Tools [Online] Available: http://www.cadence.com.

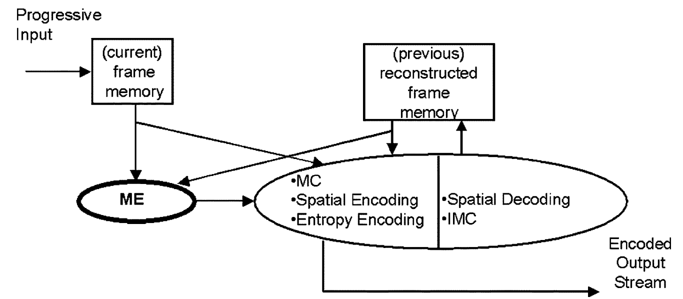

Fig. 1. Video encoding.

II. Video Applications

This section describes four different video applications that use ME, namely video encoding [6], low speed obstacle detection [3], [4], picturerate up-conversion [5], and 2-D-to-3-D video conversion [2]. These applications typically work on progressive video material and most of the applications use luminance-based ME (exception 2-D-to-3-D video conversion).

A. Video Encoding

Video encoding [6] plays a key role in enabling video processing on mobile multimedia systems, e.g., mobile videophone. Video encoding yields a compact representation of a signal by exploiting spatial and temporal correlation.

Fig. 1 illustrates the process. The ME unit determines the best match of a block in the current frame with blocks, shifted over the candidate motion vectors, in the previous reconstructed frame. The difference between the block under consideration in the current frame and the best matching block in the reconstructed frame is computed (MC). This is followed by spatial encoding (DCT/Q) and entropy encoding of the motion compensated block, via zigzag, run-length and variable length coding respectively. An embedded decoder reconstructs the spatially encoded data and performs an inverse motion compensation (IMC) before storing the data in the reconstructed frame memory. This data is needed for encoding the next frame.

The ME unit uses blocks of size 16×16 pixels and scans the current frame in a block-based left-to-right, top-to-bottom scan order. Candidate motion vectors for a block include temporal, i.e., predicted by blocks of the previous frame, with and without random updates, spatial, i.e., from the neighboring blocks in the current frame, random and zero vectors (modified 3DRS algorithm [13]). The best vector can be full-, half- or quarter-pixel refined. These vectors are restricted to a search area window, the size of which is dependent on the application.

B. Low-Speed Obstacle Detection

In low-speed obstacle detection (LSOD) [14], [15], a camera is placed behind the windshield of a car, facing forward. The

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

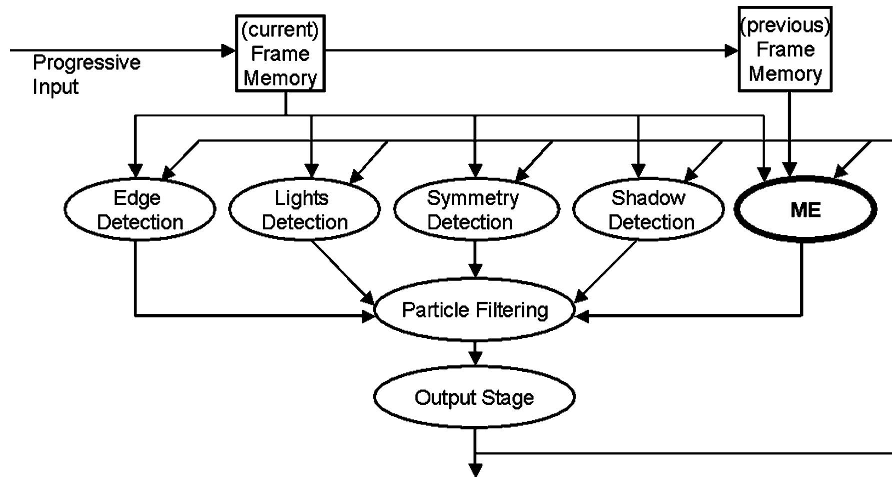

Fig. 2. Low-speed obstacle detection.

goal of this application is to detect any vehicle located few meters in front of the car in urban conditions, i.e., where the car speed is not over 40 km/h. The system will calculate the distance between the car and the detected vehicle. This application may help the car driver in different traffic conditions in order to decrease stress. It may also help to deal with the complexity of night and bad weather situations including also cut-in scenarios.

The LSOD combines the results of five different algorithms: vertical edge detection, shadow detection, lights detection, symmetry detection and vehicle motion segmentation. The main algorithm is represented in Fig. 2.

The vehicle **motion segmentation** block, which is of interest to this work, uses a ME algorithm to detect moving targets in front of the car. The ME detects vehicles and objects that are moving into the scene (cutting the way) or (de) accelerating. The ME works with blocks of 16×16 pixels. It scans a (current) frame three times in three different ways, i.e., left-to-right top-to-bottom, right-to-left bottom-to-top, and random block access. Candidate motion vectors for a block include temporal (from blocks of the previous frame) with and without random updates, spatial (from the neighboring blocks in the current frame), random and/or zero vectors. These vectors are restricted to a search area window.

C. Picture-rate Up-Conversion

The refresh rate of today's TV displays ranges from 50 to 100 Hz, whilst the source picturerate can be 50 or 60 Hz for video material and 24, 25, or 30 Hz for movie material. Clearly, a high quality conversion of signals from one format to another is of great importance. The simplest up-conversion algorithms like picture repetition, produce visual artifacts (judder, blur) and, hence, are not suitable for high quality up-conversion. Recent up-conversion algorithms [5] are based on ME and compensation to achieve high quality up-conversion.

The motion compensation is based on the motion vector field generated by the 3DRS motion estimator [13]. After ME, to every pixel identified with spatial position $\boldsymbol{x}$ and temporal position n, a best matching motion vector candidate is assigned. The best matching motion vector candidate, displacement vector $\boldsymbol{D}(\boldsymbol{x}, n)$, is the vector from the candidate set, that offers the lowest match error. Based on the displacement vector field calculated at the temporal position $n+\alpha$, $0 \leq \alpha \leq 1$ as well as the luminance values of the pixels available at the time instances n and $n+1$, new pixels can be interpolated at the time instance $n+\alpha$. Fig. 3 illustrates the creation of the pixel 'e' in the interpolated image at time instance $n+\alpha$.

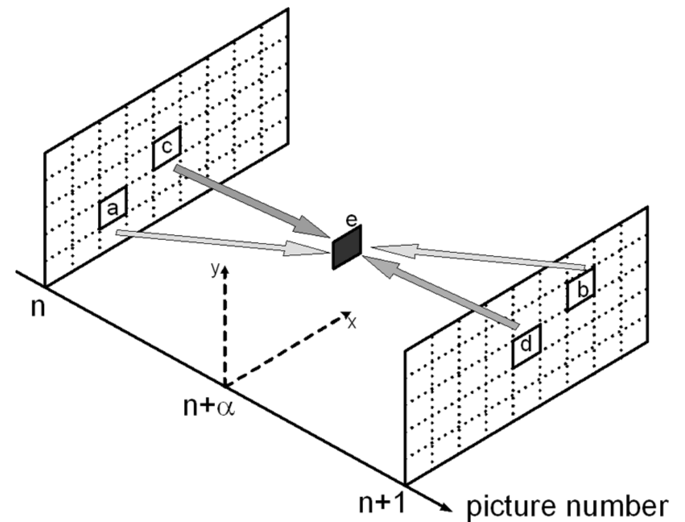

Fig. 3. Illustration of motion compensation in picturerate up-conversion.

The motion estimator performs one ME scan, scanning the image from left to right, top to bottom using five motion vector candidates per processed block (two spatial candidates, temporal, null and random candidate) [16], [17]. The criterion used for ME is the SAD, which offers a good compromise between computational complexity and quality. The dimension of the SAD window size is set to 8×8 pixels. Depending on their expected reliability, penalties are added to the match errors of motion vector candidates given by the SAD criterion.

The architecture of the up-converter is depicted in Fig. 4. The input frames are written into the current frame memory at the input frame rate f_1 and read at the output frame rate f_2. The previous frame is stored in the previous frame memory. The previous frame (time instance n in Fig. 3) and the current frame

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

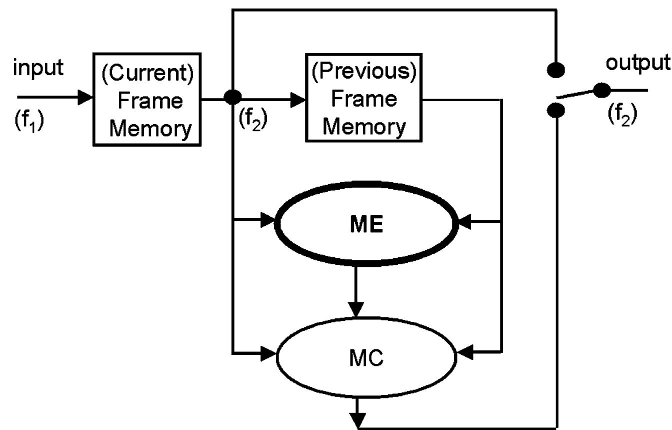

Fig. 4. Block diagram of the up-conversion module. Frame memories containing current and previous frames, are used for frequency conversion from f_1 to f_2 and for providing the delayed image.

(time instance $n+1$ in Fig. 3) are used by the ME and compensation in order to generate the new interpolated frame (time instance $n+\alpha$ in Fig. 3).

D. 2-D-to-3-D Video Conversion

While traditional video processing algorithms view an image as a set of blocks, emerging advanced video signal processing algorithms like 2-D-to-3-D video conversion [2], view an image as a set of segments of arbitrary size and shape (see Fig. 5). As segments are content-dependent and relate to meaningful entities in the image, they provide two advantages: Reasoning on objects and their relations in an image (e.g., segment A is *in front* of segment B), and tracking segments and their properties throughout the video sequence (e.g., for temporal filtering).

From an algorithmic point of view, this requires segment-based motion estimation (SBME). SBME is similar to the block-based ME of the previous section with two main differences: The domain of a motion vector is a segment instead of a block, and the image is scanned multiple times to obtain the accuracy required for the application.

However, a straightforward implementation of SBME suffers from an inefficient use of data memory bandwidth due to the irregular addressing caused by arbitrary shapes. To overcome the above problem, we have introduced modifications to the SBME algorithm to realize a practical and highly structured SBME algorithm (see Fig. 6).

Given a segmentation of a video frame (namely, for the current frame) and its next frame, a sketch of the modified SBME algorithm implemented in our design is as follows:

Step 0) (*Segmentation refinement*): lay a block (say, 16×16 pixels) grid on the segmented current video frame. Ensure that each block contains contributions from not more than four segments to restrict the number of hardware resources required to complete all calculations for such a block within a guaranteed time limit; this can be achieved by reassigning pixels belonging to small segments to large segments based on nearest neighbor and ordering criteria. Tests on a large set of sequences have shown that in practice this limit is rarely exceeded.

Fig. 5. Segmentation of a frame in Renata video sequence. White lines indicate segment boundaries.

Step 1) (*Choice of candidate motion vectors*): In the 3DRS block-based motion algorithm, spatial candidate motion vectors for a block are selected from a fixed set of blocks in the neighborhood. For SBME, spatial candidates are derived from neighboring segments. For this purpose, all neighboring segments are sorted completely based on a certain metric (e.g., number of pixels in the shared segment boundary, the color difference with the current segment, etc.). Then, the motion vectors of the current segment in the previous scan (for first scan: motion vector from corresponding segment in previous frame) as temporal candidate. Zero motion vector (for the first scan), and motion vectors of N closest neighbors (where, N is typically between 4–6) with 50% probability for a random update are chosen as candidate motion vectors for each segment; this is similar to the strategy used in the 3DRS-type (block-based) ME [13] described in the previous sections.

Step 2) (*ME kernel*): this step is decomposed into the following substeps:

1) evaluate the ME criterion (SAD) for each candidate MV per (sub)segment of a block and for all blocks of the current frame;

2) accumulate the evaluated criterion over all subsegments of a segment for each candidate MV and for all segments of the current frame;

3) normalize the evaluation criterion of all segments with respect to their sizes, then choose the best candidate MV for each segment of the current frame;

Note that for step 1, all blocks of the current frame can be processed in a regular way; the operations of step 2 and step 3 are done on the level of (sub)segments and are thus computationally much less complex.

Step 3) (*Convergence check*): perform global convergence check. Go to Step 1 if not yet converged. Other-

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

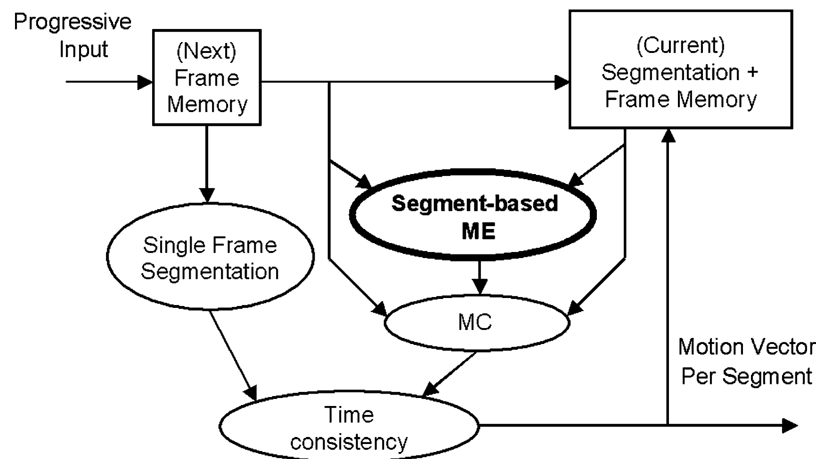

Fig. 6. Time-consistent segmentation being part of 2-D-to-3-D video conversion.

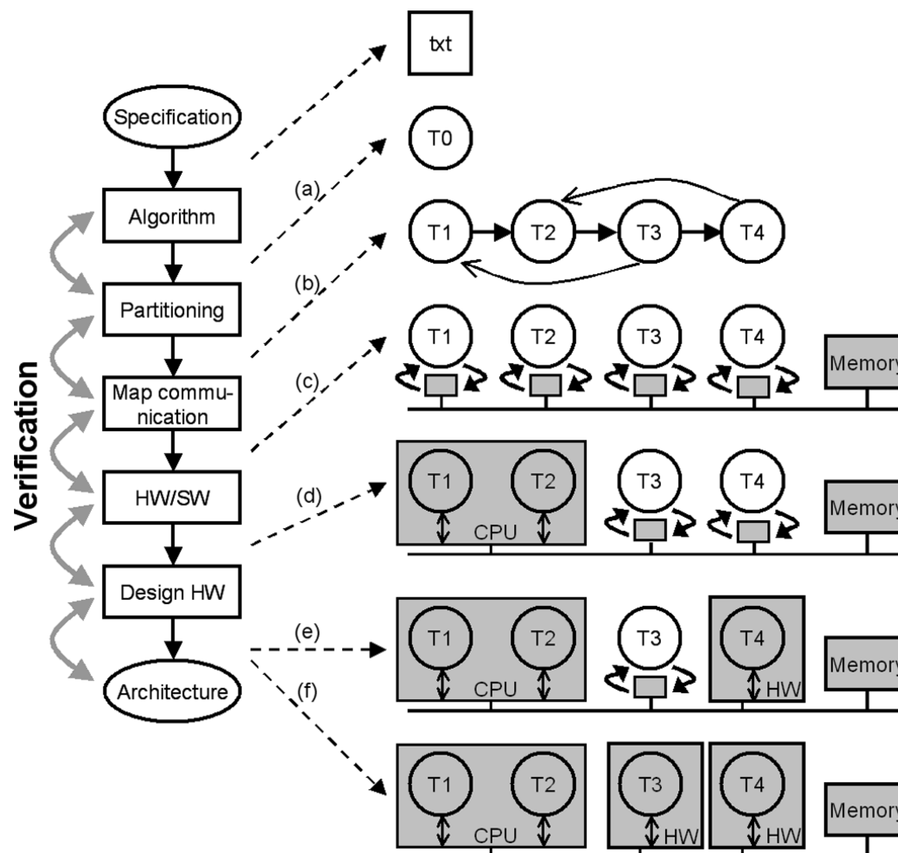

Fig. 7. Levels of design abstraction.

wise, provide the best matching MV per segment. Multiple scans are required for two reasons. First of all, the application requires accurate motion vectors. Furthermore, in block-based ME, some spatial candidates arise from neighboring blocks, which have already been updated in the current frame. With the approach in Step 2, *all* spatial candidates for the segments are motion vectors of the *previous* scan, as the segments can not be processed sequentially. This slows the convergence.

In further sections we will focus on step 2.1, because this part of the algorithm is executed on the ME ASIP presented in this paper.

III. Design Methodology

In Section III-A, we present the hardwaresoftware co-design methodology used in this work, while in Section III-B the ASIP design method using A|RT tool-flow is described.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

A. HardwareSoftware Co-Design Method

State-of-the-art hardware/software implementations trade efficiency (performance, cost, power) for flexibility and time-to-market for cost. An efficient hardware/software design flow then would start from a high-level specification (i.e., C-language) and converge toward a silicon/software implementation in several steps and iterations. One of the major tasks in this design flow is to ensure that hardware and software tasks communicate with each other correctly. This can be achieved by using a modular, flexible and scalable heterogeneous multiprocessor architecture template with shared memory and an efficient/transparent protocol for communication [12]. The hardwaresoftware co-design (and verification) methodology followed in this work can be divided into several steps. The various steps will be explained based on the video encoding application.

The first step consists of partitioning the video encoder application into hardware and software tasks. With a C-language based behavioral description as starting point the application was profiled on a CPU (e.g., ARM) to obtain an estimate of the computational load. This provided the required clock-frequency for a softwareonly solution on a programmable processor and the breakdown of the computational load for different functional modules of the encoder. The main strategy was to implement the encoding standard and control-intensive parts in software (e.g., ARM) and the compute-intensive parts in hardware. This step refers to moving from level "a" to "b" in Fig. 7.

The second step consists of implementing the communication primitives in tasks. The input parameters to the hardware task consist of two parts namely frame constants (e.g., frame size) and run-time parameters (e.g., coordinates of current block). Since it is not efficient to read the frame constants from shared memory for each block, an initialization mode was added to the task in which the frame constants are stored locally in the task. Only the run-time parameters that vary across blocks are communicated which reduces bandwidth on the bus. Via system simulations the communication is further optimized resulting in using two busses one for the control data and another for the pixel data. The resulting system architecture is shown in Fig. 8. This step refers to moving from level "b" to "c" in Fig. 7.

The third step focuses on improving the performance of the implementation. The result of the previous step consists of a system without concurrency. A software task starts the hardware task and wait for completion of the hardware task (and vice versa), thus using the resources ineffectively. By pipelining the software and hardware tasks concurrency can be implemented. To fully pipeline and optimize the design some dependencies between software and hardware tasks need to be broken. Extensive simulations were carried out, e.g., for video encoding verify that the compression ratio and SNR are comparable to the original C description. This step refers to moving from level "c" to "d" in Fig. 7. Note that, cycle-true models for the software tasks were used for system simulations in order to determine the real-time performance of the software tasks executed on the CPU.

The last step relates to the design of the hardware processor, which corresponds to moving from level "d" to "e" in Fig. 7.

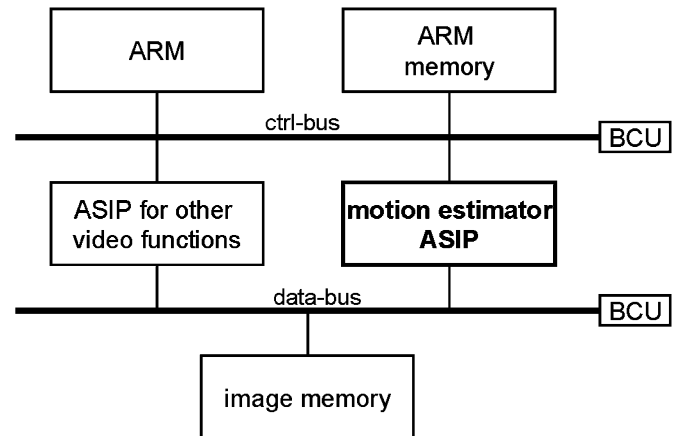

Fig. 8. Target system architecture showing motion estimator as hardware processor communicating via two busses (control and pixel data respectively).

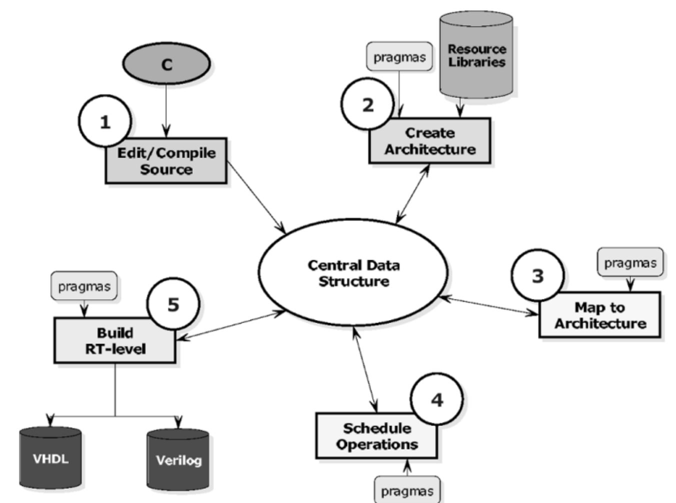

Fig. 9. Design flow of A|RT-Designer.

The design method for realizing hardware processors is explained in Section III-B. The target system architecture depicted in Fig. 8 is based on a scalable heterogeneous multiprocessor architecture template [12], wherein the processors run concurrent tasks. The tasks synchronize on data and buffer-space availability and may use on-chip buffers as communication buffers to reduce bandwidth to off-chip memory. The processors are chosen such that a good tradeoff between flexibility and efficiency is achieved. For example, functions that are well known and are not subject to change can be implemented in an efficient way (with minimal flexibility). On the other hand, functions that need to be adapted to the context in which they would operate can be implemented with maximum flexibility (with less concern for efficiency). Therefore, the template architecture supports highly flexible cores (CPUs) to highly efficient cores (ASICs or ASIPs).

The communication protocol used in this work can handle data streams of infinite length [12]. The protocol is based on the model of process networks [18], wherein the overall function is decomposed into a number of parallel processes communicating via point-to-point channels with first-in first-out (FIFO)

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

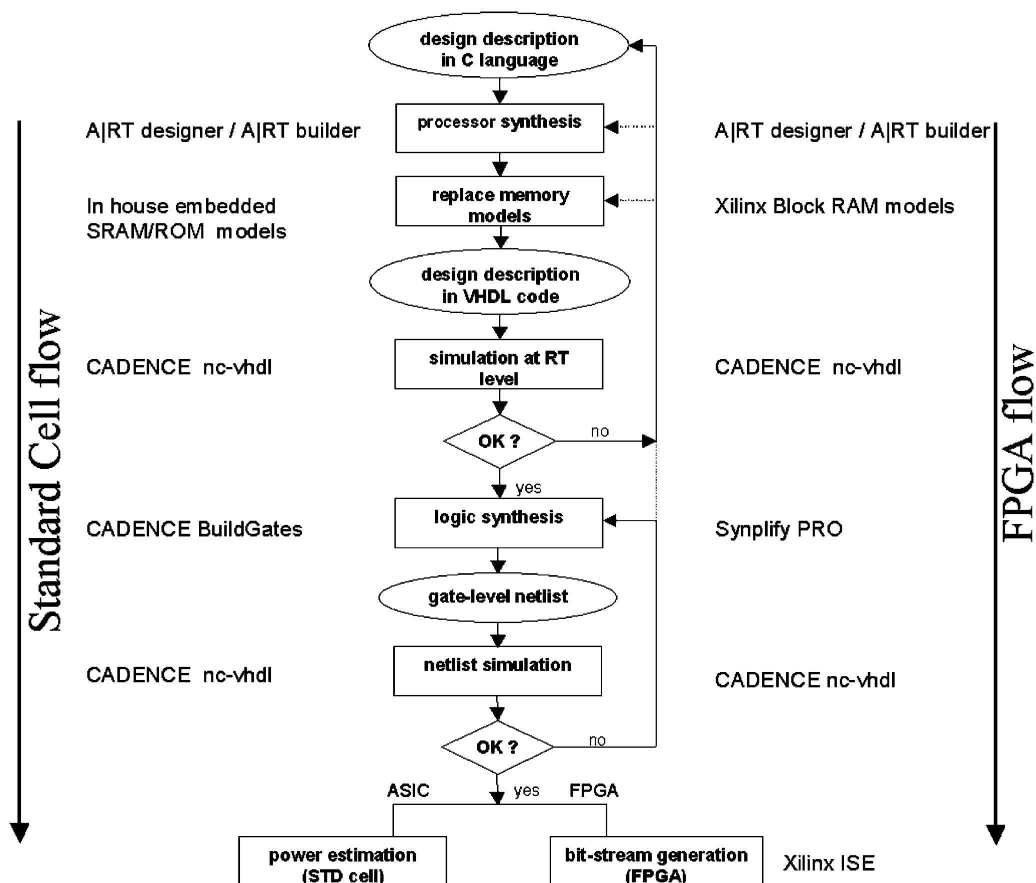

Fig. 10. Diagram showing design flow, including tool names, for both standard cell (left) and FPGA (right) technology streams. A C-language application program is compiled onto a VLIW-based ASIP architecture resulting in a synthesisable description. After replacing memories with realistic memory models and subsequent verification, logic synthesis and netlist verification is done. Depending on the technology stream either power estimation (standard cell) or bit stream generation (FPGA) is performed.

behavior. A process can block either due to nonavailability of data in the input channel or due to nonavailability of space in the output channel. Synchronization of processes is derived from the status of the FIFO and takes place on a per-token basis. The amount of data associated with a token can vary from zero bytes to an entire video frame. In the context of shared memory implementations, there is no physical transfer of data required. Only synchronization primitives for data pointers are needed. For performance reasons all memory usage is allocated at startup.

B. ASIP Design Method

The ASIP for ME presented in this work was designed using the A|RT tools [7]. In this ASIP design method, A|RT-Builder is used for designing the ASUs, while A|RT-Designer is used for generating the VLIW ASIP which uses the ASUs apart from standard functional units like ALUs and ACUs.

A|RT-Builder takes a C-based functional specification of an algorithm as input, and creates a data path with resources of word sizes derived from the specification in a bit accurate way. The output of A|RT-Builder is a synthesizable RTL description in either VHDL or Verilog.

The A|RT-Designer tool assists designers in the development of a hardware processor, customized for the algorithm that has to be executed on this architecture. The generated processor consists of a set of data path resources, controlled by a VLIW type controller. This configurable controller architecture is scalable for parallelism as well as performance. The main strength of A|RT-Designer is that it facilitates the exploration of several alternative architectures, and allows to determine the optimal architecture for the design. This will result in a reduced development time and an increased productivity. In Fig. 9 the internal design flow of A|RT-Designer is shown.

The starting point of A|RT-Designer is a C-based algorithm and this description is compiled to an internal representation during the first step.

In the second step (architecture creation) the architecture is composed of standard and application specific resources from one or more libraries. Typically standard resources (like ALU, ACU, multplier (MULT), constant ROM/RAM) are selected from a default A|RT-library, while ASUs (created with A|RT-Builder) are selected from a user defined library.

In the third step (mapping to architecture) the algorithm is mapped onto the defined architecture. Variables and constants are mapped on available memory resources (register files, RAM, ROM), followed by assigning operations to the data path resources, which are then translated into register transfers. Multiplexers are provided at the inputs of the registers in case more busses are connected to the same input or if two variables with different types are transferred to that input over a bus connected to it.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

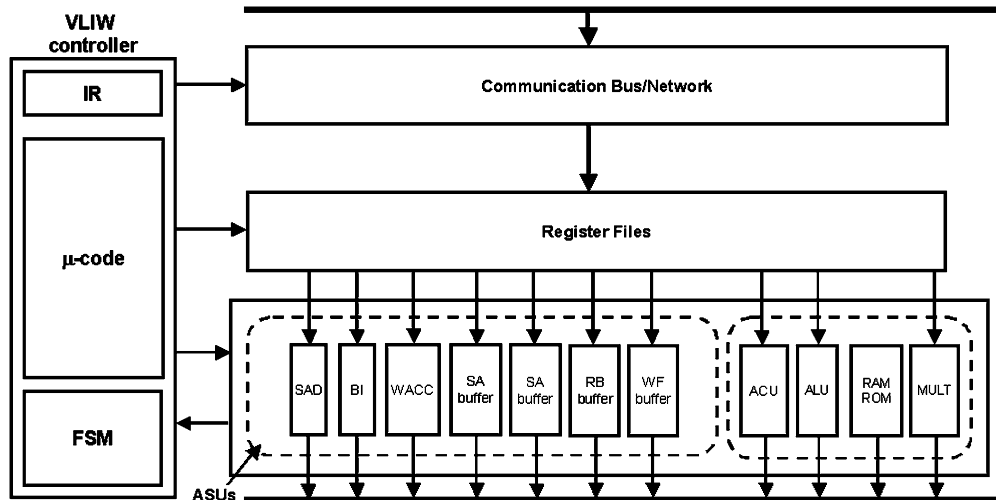

Fig. 11. Application specific instruction-set processor for ME, showing standard and application specific resources.

The scheduling operations step (fourth step) performs two major tasks: scheduling and register assignment. Scheduling involves ordering the register transfers along a time axis. Scheduling is done by means of a basic scheduler and a set of optimization techniques for improving the basic schedule. Register assignment entails assigning variables to the fields of register files in such a way that the size of the register files is minimized. The schedule and the register usage can further be refined with advanced optimizations like loop folding, peephole and lifetime optimizations [7]. The goal of the scheduler is to minimize the global machine cycle count and to keep the necessary number of registers as low as possible.

The final step generates the complete design, data path and controller, in synthesizable VHDL or Verilog.

Fig. 10 illustrates the standard cell flow and field programmable gate array (FPGA) tool flow for synthesizing and verifying the motion estimator ASIP design. The tool flow includes the A|RT tools for generating RTL descriptions from C-based algorithmic specifications. Further, two technology streams are exercised, namely an FPGA stream for proof-of-concept on silicon, and an ASIC stream to obtain power/area/performance numbers based on netlist simulations. The design of the motion estimator ASIP based on the above flow is presented in the next section, while the results of both streams are presented in Section V.

IV. Motion Estimator ASIP Design

Fig. 11 depicts the motion estimator VLIW ASIP generated by A|RT. The architecture consists of standard resources like an ALU, ACU, MULT, constant-ROM, RAM, instruction-memory holding microcode, VLIW controller and ASUs. The ASUs are tailored to accelerating the inner kernels of the ME. The following ASUs can be identified.

1) Search area buffer (SA buffer): contains pixel data for the complete search area.
2) Reference block buffer (RB buffer): contains (current) block being motion estimated.
3) Weight factor buffer (WF buffer): similar to RBB but holds subsegment mask data.

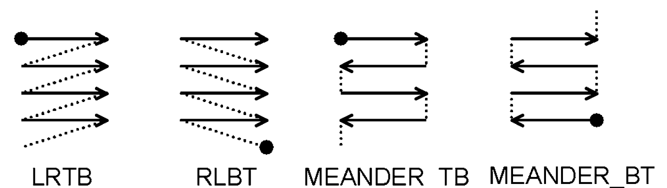

Fig. 12. Supported scan orders. (1) LRTB: left to right top to bottom. (2) RLBT: right to left, bottom to top. (3) MEANDER_TB: meandering, top to bottom. (4) MEANDER_BT: meandering bottom to top.

4) Bi-linear interpolator (BI): supports half and quarter pixel refinements.
5) Sum of absolute differences (SAD): determines criterion for ME.
6) Weight accumulator (WACC): normalization of SAD values.

The ASUs are discussed in full detail in Section IV-A, while examples of pseudo code depicting the software on the ASIP are described in Section IV-B.

A. Application Specific Functional Units

1) SA Buffer: In order to have a predictable system design, the complete search area is stored in the SA buffer. By restricting the motion vector candidates to the search area, this approach results in improved performance and reduced power dissipation.

The SA buffer offers the following functionalities:

1) **Scan order**: scanning consecutive blocks (e.g., 8×8 pixels) of a frame could be done in different fashions as illustrated in Fig. 12.
2) **Subpixel accuracy**: based on coordinate of requested block, subpixel accuracy is detected and the appropriate block together with the additional pixels to support interpolation is delivered.
3) **Block size**: the frame memory could be organized in different block sizes (e.g., 8×8 pixels). Since the SA buffer holds parts of frame memory data, the block organization should be identical. Currently, 8×8 and 16×16 block sizes are supported.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

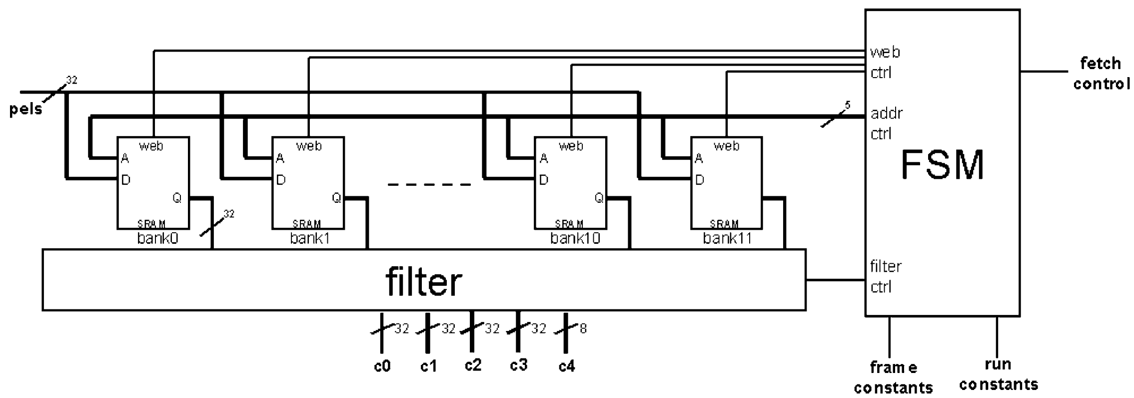

Fig. 13. Block diagram of SA buffer showing a 12-kbits memory system partitioned in 12 banks (each 32 words of 32 bits) to store 32 pixel-lines of 48 eight pixels per line, together with filter and control circuitry.

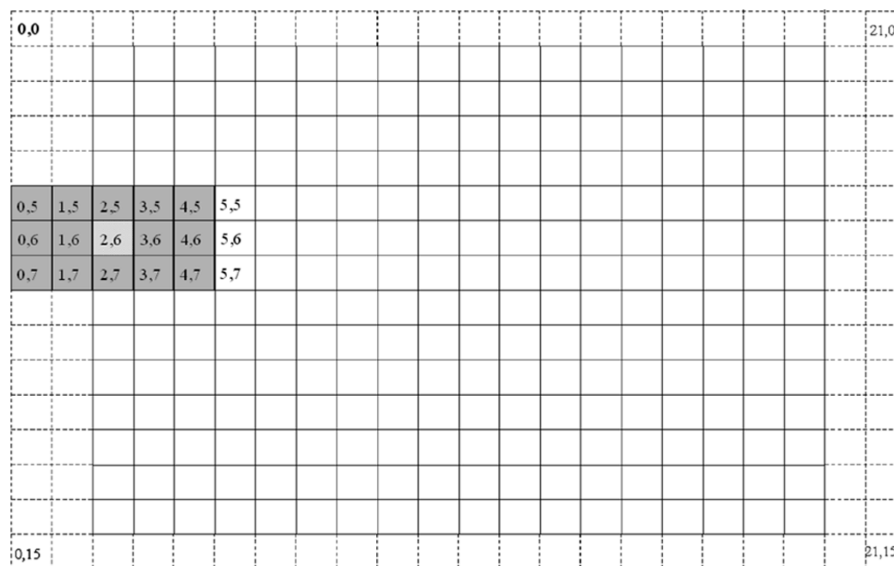

Fig. 14. Frame memory organized in 8×8 pixel blocks together with search-area window (of 5×3) centered on the block (2, 6). Special processing for blocks in border region (dashed) can be applied.

4) **SAD window size**: this refers to the block size being read, which could be different from the block size while filling. Currently, 4×4, 8×8, and 16×16 block sizes are supported.
5) **Subsample formats**: currently only luminance pixels are retrieved from frame memory. However, the following formats of frame storage are supported: 4:0:0 (four luminance components), 4:2:0 (four luminance and two chrominance components), 1:0:0 (single luminance component).
6) **Address calculation unit**: based on the supplied subsample format and frame memory address-limits, the necessary frame memory address calculations are done.
7) **Intra-mode** (video encoding): The SA buffer has a feature to deliver a user set-able constant pixel value for every pixel to bypass the buffer content. This could be useful when one of the two SAD inputs needs a constant pixel value (e.g., performing summation with SAD engine could be done by setting bypass value to zero). This feature is useful for the video encoding application.

Fig. 13 shows the architecture of the SA buffer. This architecture illustrates the use of customized memory system that exposes data parallelism. The memory system is organized as 12 banks. Each contains 32 pixel-lines of 32 bits or four 8-bit luminance pixels per memory location. Each single memory location can be accessed individually when filling. During reading, a number of banks are being selected and the resulting bank outputs are concatenated and filtered. One filtered pixel-line can be delivered every clock cycle.

The SA buffer is controlled via the following signal clusters.

1) **Frame constants**: these are input signals, which are constant for a complete frame and are usually loaded in internal registers before processing a new frame. Frame constants are *frame size* (number of lines per frame and number of pixels per line), *scan order*, *block size*, *SAD window size*, *frame memory address-limits*, and *subsample format*.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

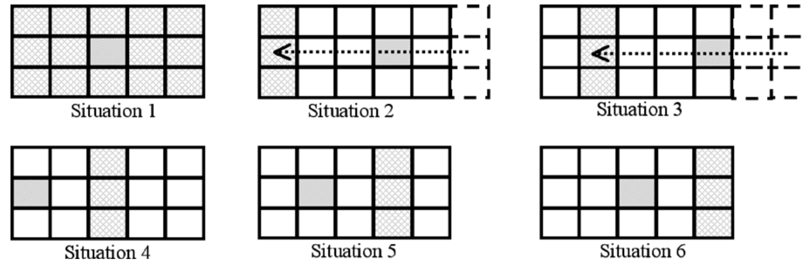

Fig. 15. SA buffer filling: Scenarios for different 8×8 block positions. The grey blocks denote the center block of a search area window. The shaded blocks are the blocks to be filled-in when the center block moves to next position within frame memory (scan order LRTB, block size 8×8, search area size 3×5). Note that, every column within the search area window resides in two banks.

```
sa_buffer_fill_init(x, y);
sa_buffer_fill_init_out(nr_blocks);
for (cnt=0; cnt<nr_blocks; cnt++) {
  sa_buffer_fill_run();
  sa_buffer_fill_init_out(ip);
  for (i=0; i<16; i++) {
    pels = memory_read(ip);     //external memory read
    sa_buffer_fill_write(pels);
    ip++;
  }
}                                                   filling
---------------------------------------------------------------
sa_buffer_read_init( x, y, intra, mean );
sa_buffer_read_run();
sa_buffer_read_run();
for (cnt=0;cnt<16; cnt++) {// in case of 16-by-16 block
  sa_buffer_read_out(c4, c3, c2,c1, c0);
  ..
}                                                   reading
```

Fig. 16. Pseudocode showing the filling and reading of SA buffer. Every instruction is single cycle.

2) **Run constants**: these are input signals, which are related to the block being processed and are loaded into internal registers at the start of a new block. Run constants are *block coordinates* (x, y), *four-pixel-line* input in case of filling and *intra-mode* control.

3) **Fetch control**: these are two output signals, which controls reading from frame memory. The *number of* 8×8 *blocks* to be read from frame memory and the *start-address* of first 8×8 block in frame memory control the access of frame memory data.

Consider the situation (see Fig. 14) where the search area is at an arbitrary position in the frame (note that, the zero motion vector corresponds to the center of the search area). After filling the SA buffer it holds the complete search area. For ease of implementation we simplify the processing of border blocks by either adding artificial border blocks or skipping the border blocks all together. The illustration in Fig. 14 corresponds to processing the first block of a new line (scan order is LRTB).

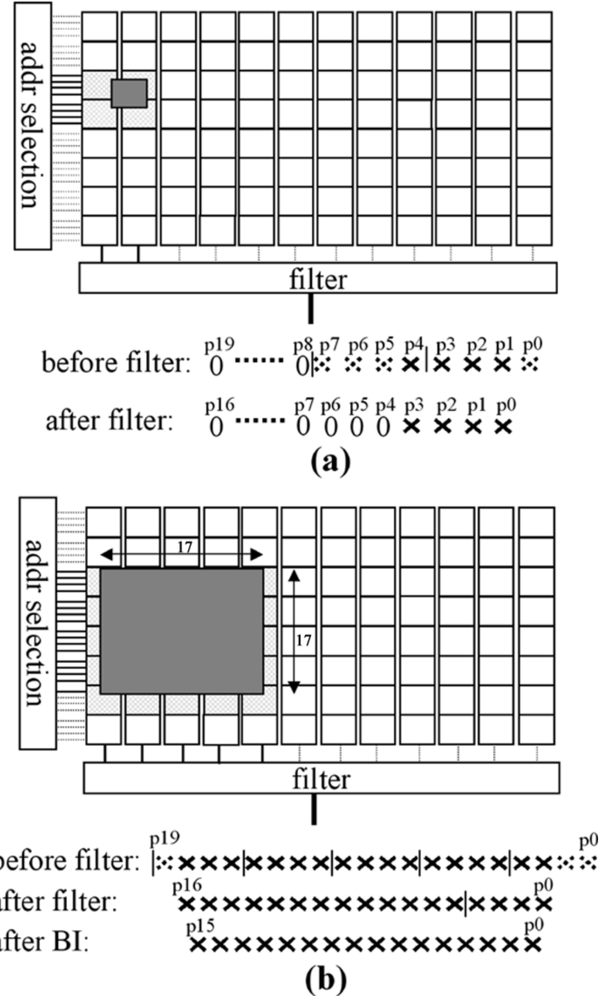

Fig. 17. SA buffer reading for two scenarios. (a) SAD window size 4×4 pixels motion vector in pixel accurate. Four pixel lines are subsequently output (four active pixels ordered from right to left, non active pixels are set to zero). (b) SAD window size 16×16 pixels motion vector subpixel accurate. 17 pixel lines are subsequently output (17 active pixels).

Fig. 15 shows different situation related to filling. In case a new line is detected (situation 1) the complete search area needs to be filled in (five columns, each contains three 8×8 blocks) and the reference defined as block-pointer is set to the center block (note that, this situation is illustrated in Fig. 14). When the frame block-pointer moves to its neighboring block (situation 2), only one column of search area window (three 8×8 blocks) is required. The block-pointer moves to next block position within the buffer since this new position becomes the center of search area window. The leftmost column of the buffer (position of which is wrapped around horizontally) will be invalidated and filled with the new column contents. This process is repeated until the complete block-line has been traversed. Besides the start position (new line situation) there are five different situations regarding the block-pointer as can be seen in Fig. 15.

In the process described above there is only horizontal wrap around present in the buffer since the scan order is LRTB. The

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

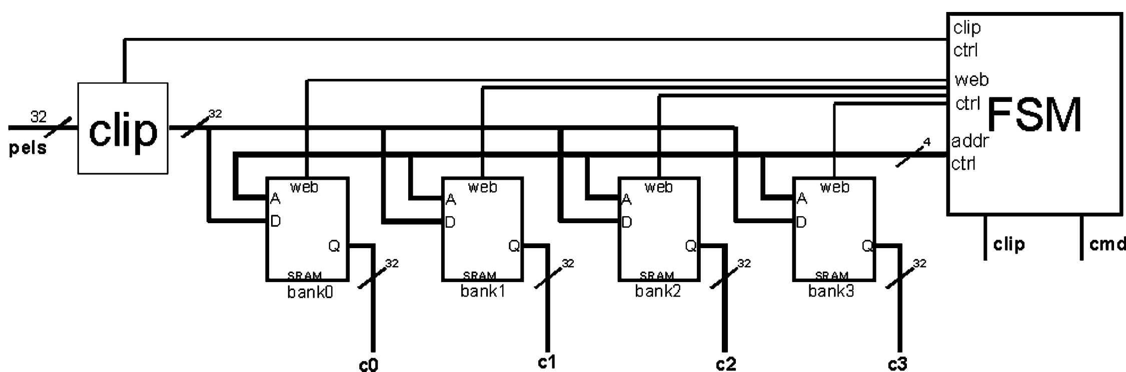

Fig. 18. Architecture of RB buffer showing the memory organization unfolded in four individual memories (16 words of 32 bits each).

block-pointer does not move in the vertical direction. However, for the case of meandering scan order (e.g., MEANDER_TB, MEANDER_BT as shown in Fig. 12), the block-pointer also moves in the vertical direction. Furthermore, the filling is dependent on the block size. When the block size is 16×16 pixels, the step size of the block-pointer and frame block-pointer will be two 8×8 block columns which implies two 8×8 block columns need to be filled when moving to a neighboring 16×16 block position.

In Fig. 16 the pseudo code for filling is shown. Via an initialization instruction *sa_buffer_fill_init* the final state machine (FSM) is initialized, and the coordinate of the center block of the search area window is loaded. The SA buffer determines the number of blocks to be filled in to make the search area complete. The required number of blocks is an output of the instruction *sa_buffer_fill_init_out*. For every requested block, the block start address calculation is issued through the instruction *sa_buffer_fill_run* and the resulting address is retrieved through the instruction *sa_buffer_fill_init_out*. Based on this frame address, four pixels (packed in 32 bits) are fetched from the frame memory and stored via instruction *sa_buffer_fill_write*. The above process of fetching pixel data from the frame memory and storing is repeated until all the pixels pertaining to all the requested blocks are filled in [direct memory access (DMA) is not implemented yet].

In Fig. 16 the pseudo code for reading is shown. Via the initialization instruction *sa_buffer_read_init* the candidate vector (length (x, y) relative to center block) is loaded and the appropriate memory block (rectangular region) is selected. Using the output instruction *sa_buffer_read_out* a line of 17 pixels is delivered each cycle. Note that, reading the SA buffer is a pipelined operation with a latency of three cycles, therefore two explicit run instructions *sa_buffer_read_run* are required. In case of intra-mode the latter process is not executed, but the supplied constant pixel value for every pixel (mean) is delivered. The intra-mode is controlled via the initialization instruction *sa_buffer_read_init*.

The correct memory block is selected by reading from a number of memory banks (horizontal) at a number of subsequent address locations (refer to Fig. 13 for block diagram). The required banks depend on the candidate vector (x), SAD window size, and subpixel accuracy mode. Consecutive lines are read by incrementing the address. The number of lines read is dependent on candidate vector (y), SAD window size, and

```
rb_buffer_init(clip);            //clip = 1; enable clipping
for (i=0; i<64; i++) {
  pels = memory_read(ip);   //external  memory read
  rb_buffer_fill_run(pels);
  ip++;
}                                                      filling
---------------------------------------------------------------
rb_buffer_init(0);
rb_buffer_read_run();
for (i=0; i<16; i++) {
  rb_buffer_read_out(c0, c1, c2, c3);
  ...;
}                                                      reading
```

Fig. 19. Pseudocode showing the filling and reading of RB buffer. Every instruction is single cycle.

subpixel accuracy mode. In Fig. 17 two different read scenarios are shown.

2) RB Buffer: The RB buffer functional unit is used to store the reference block. The RB buffer has been designed to support 16×16 pixels SAD evaluations and therefore delivers 16 pixels per clock cycle, which is a single pixel line of a block. By exploiting this parallelism, it takes 16 clock cycles to read the complete reference block.

Fig. 18 shows the architecture. The RB buffer is organized as four banks; each contains 16 pixel-lines of 32 bits, or four 8-bit luminance pixels per memory location. While reading, the four bank outputs are concatenated to deliver 16 pixels in parallel. The memory banks are in read-mode by default. The RB buffer can perform pixel value clipping, which is part of the video encoding process. When clipping mode is enabled every incoming pixel value will be clipped to a minimum of 1 and a maximum of 254.

In Fig. 19 the pseudo code of the filling and reading is shown. Via the initialization instruction *rb_buffer_init* the FSM is initialized, and the clipping control signal is evaluated (clipping is only effective during writing). Using the fill instruction *rb_buffer_fill_run*, four pixels are filled in the appropriate memory bank every cycle. Using the read instruction *rb_buffer_read_out* a line of 16 pixels is delivered per cycle. The instruction *rb_buffer_read_run* addresses the pipelined read operation.

3) WF Buffer: The WF buffer is used to store the subsegment mask and pixel weight data of the reference block. The subsegment mask consists of 2 bits per pixel. It indicates to

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

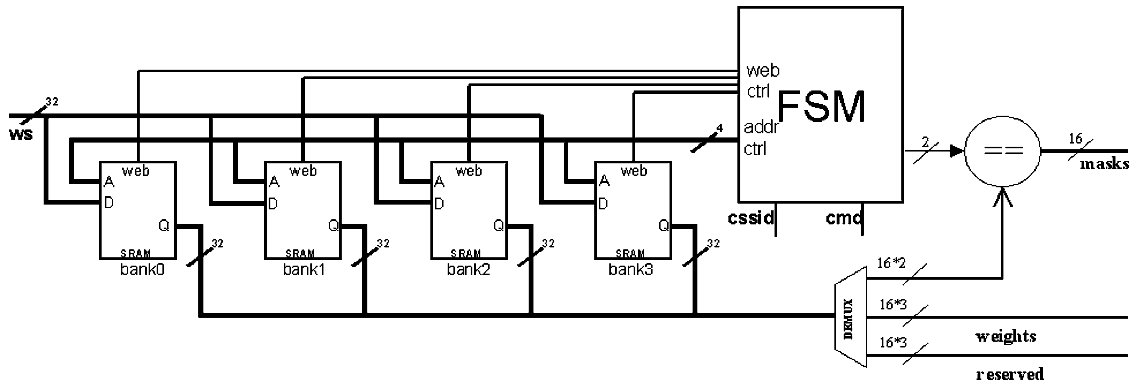

Fig. 20. Architecture of WF buffer showing the memory organization (identical to RB buffer), and additional logic to generate the subsegment mask and weight bits using the current subsegment ID.

```
wf_buffer_init();
for (i=0; i<64; i++) {
  ws = memory_read(ip);       // external memory read
  wf_buffer_fill_run(ws);
  ip++;
}                                                filling
```

```
wf_buffer_init();
wf_buffer_read_run(cssid);  // set current sub-segment ID
for (i=0; i<16; i++) {
  wf_buffer_read_out(masks, weights, reserved);
  ...;
}                                                reading
```

Fig. 21. Pseudocode showing the filling and reading of the WF buffer. Every instruction is single cycle.

which of the (up to) 4 subsegments of the block the corresponding pixel belongs. The 3 bits per pixel weight value will be used in future implementations to improve the quality of the detected motion vectors. For example, it can be used to emphasize the SAD contribution of pixels near edges, whilst suppressing the contribution of pixels in homogeneous areas. This yields more reliable SAD results, and thus the chance of detecting the true motion is increased. We have reserved 3 additional bits per pixel for future use. In total 8 bits per pixel are stored for weight factor related values.

Since the size and the number of bits per pixel are equal to the size of the RB buffer, the WF buffer is based on the same design. In particular, the memory layout and the filling strategy are identical (see Fig. 18).

Fig. 20 shows the architecture of the WF buffer. Since the design is very similar to the design of the RB buffer (see Fig. 18), we will focus on the differences between the two. First of all, the WF buffer does not require the clipping functionality. Secondly, a demultiplexer has been added to separate the three fields stored for each of the 16 pixels (2 bits subsegment mask, 3 bits pixel weight, and 3 bits reserved). Also, a comparator has been added to compare the 16 subsegment mask values with the current subsegment ID. The result is a binary mask (one bit per pixel) that is used by the SAD and weight factor accumulation units (WACC) to determine which pixels contribute to the current subsegment; only these pixels are included in the SAD and weight calculations.

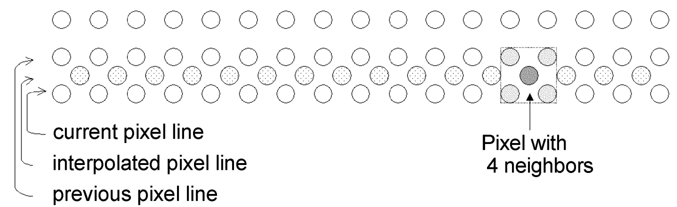

Fig. 22. Illustration of the bilinear interpolation. Picture shows the creation of the interpolated pixel based on its four neighboring pixels.

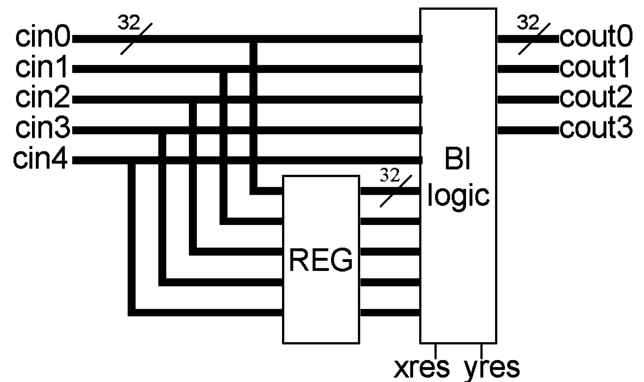

Fig. 23. Organization of the BI ASU. The register holds a complete pixel-line and via combinatorial logic (BI logic) the weight averaging is performed based on resolution setting.

```
bi_init(xres, yres);
for (i=0;i<2;i++) {
  sa_buffer_read_out(cin0,cin1,cin2,cin3,cin4);
  bi_in(cin0,cin1,cin2,cin3,cin4);
}
for (i=0;i<15;i++) {
  bi_out(cout0,cout1,cout2,cout3);
  sa_buffer_read_out(cin0,cin1,cin2,cin3,cin4);
  bi_in(cin0,cin1,cin2,cin3,cin4);
}
bi_out(cout0,cout1,cout2,cout3);
```

Fig. 24. Pseudocode showing the bilinear interpolation calculation for a 16×16 block. Every instruction is single cycle.

In Fig. 21, the pseudo code of the filling and reading actions is shown (similar to RB buffer).

4) BI: The BI unit is used for generating corresponding pixels for the SAD calculation in case subpixel accuracy of

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

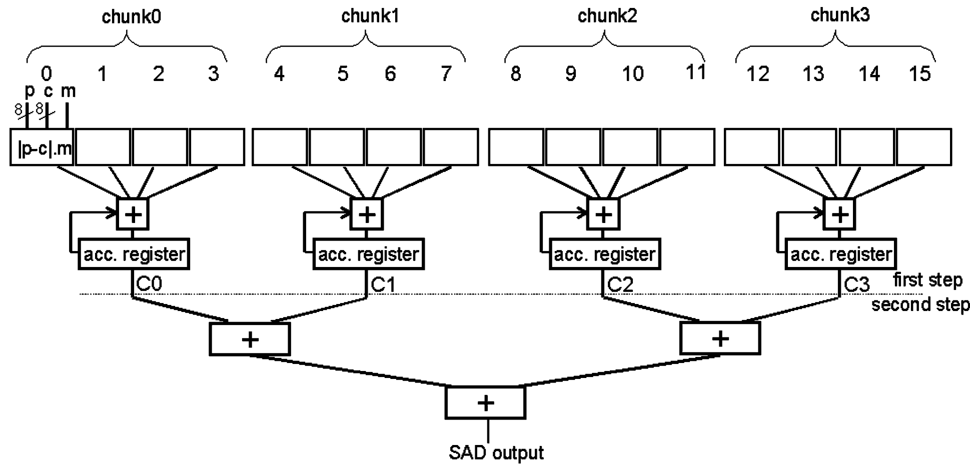

Fig. 25. Organization of the SAD functional unit. The output is formed by finding partial sums of differences between pixels belonging to the same line but to different frames. Each line can be divided in four chunks, where each chunk consists of four pixels.

motion vectors is required. Each interpolated pixel is generated by taking the weighted average value of its four nearest neighboring pixels. The weights are determined by the fractional values of the x and y components of the motion vector candidate currently being evaluated, $D_{\rm x}$ and $D_{\rm y}$, respectively.

The position of the SAD window's top-left pixel is determined by the truncated value of the D_x and $D_{\rm y}$. In order to properly interpolate pixels located at the right-most column and lowest row of the SAD window, one additional column and one additional line are needed.

The BI is pixel line organized and its functionality is illustrated in Fig. 22. Based on two successive pixel lines of width 17 pixels, it generates 16 interpolated pixels in one clock cycle. Because of the pixel line fashion storage, one pixel line of width 17 pixels is required. In case that SAD window size is set smaller than 16 pixels it is assumed that nonrelevant pixels are set to zero to reduce power consumption.

Fig. 23 shows the architecture featuring a register to hold a complete pixel-line (17 pixels) and combinatorial logic to perform the BI calculations per pixel. The interpolated pixel position (resolution) can be set to 0, 1/4, 1/2, and 3/4 in both the horizontal and vertical direction.

In Fig. 24, the pseudo code of the BI kernel using the BI ASU instructions is shown. Via an initialization instruction *bi_init* the desired interpolated pixel position is set. The next step is to supply the block in a line-by-line manner, wherein in each clock cycle one pixel line from the block is supplied via the input instruction *bi_in*. During each clock cycle a single line is output via an output instruction *bi_out*. (Note that 17 lines of 17 pixels are supplied, while the result is 16 lines of 16 pixels)

5) SAD: The SAD functional unit is used to calculate the SAD of every motion vector candidate. It compares a block within the current frame and the corresponding block within the previous frame shifted by the motion vector candidates. The SAD function can be formally described by

$$\mathrm{SAD} = \sum_{i=1}^{\mathrm{hsad}} \sum_{j=1}^{16} \left|\mathrm{prev}(i,j) - \mathrm{curr}(i,j)\right| .\mathrm{mask}(i,j)$$

```
sad_asu_init();
for (i=0;i<16;i++) {
  sa_buffer_read_out(pc0,pc1,pc2,pc3)
  rb_buffer_read_out(cc0,cc1,cc2,cc3)
  wf_buffer_read_out (mask,weights,reserved)
  sad_asu_in(pc0,pc1,pc2,pc3,cc0,cc1,cc2,cc3,mask);
}
sad_asu_out(sad);
```

Fig. 26. Pseudocode showing the calculation of SAD value for a 16×16 pixel block. Every instruction is single-cycle.

where, $\mathrm{prev}(i,j)$ and $\mathrm{curr}(i,j)$ identify the spatial position of pixels located in previous and current block, respectively and hsad indicates the height of the SAD window in pixel lines (4, 8, or 16). The $\mathrm{mask}(i,j)$ denotes a binary segmentation mask and via this it is determined which pixels contribute to the current subsegment and are therefore included in SAD calculations.

Fig. 25 shows the architecture. The partial SAD calculation is designed as four separate SAD subblocks each being capable of calculating the SAD of a chunk of four pixels in a single clock cycle. The maximal width of the SAD window of 16 pixels is supported and the number of clock cycles required to calculate the SAD of a given motion vector candidate is equal to $h_{\rm SAD}+1$. In case the requested width of the SAD window is less than the maximally supported width, the unused SAD subblocks are not triggered since the unused pixels are set to zero by the SA buffer. This reduces the power dissipation.

In Fig. 26, the pseudo code is shown. Via the initialization instruction *sad_asu_init* the four partial sum registers are cleared. The next step is to supply the current and previous blocks in a line-by-line manner, wherein, in each clock cycle one pixel line from current and previous blocks are supplied via the input instruction *sad_asu_in*. The final SAD value is then retrieved via an output instruction *sad_asu_out*.

6) Weight Factor Accumulator: The weight factor accumulator (WACC) is used to calculate the accumulated weight of all pixels that are included in the SAD calculation. Currently, this accumulated weight is only used to normalize the SAD result with respect to the area of the corresponding segment. Hence, only binary weights indicating whether a pixel belongs to the current subsegment or not, have to be accumulated (see Fig. 27

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

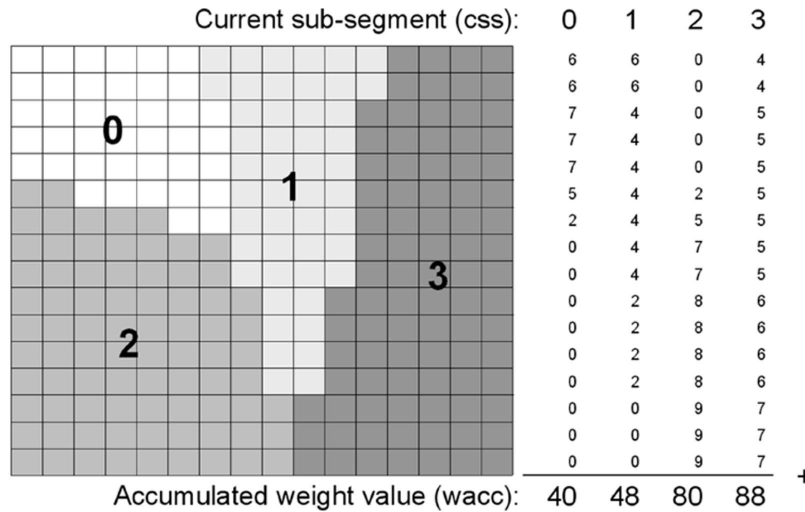

Fig. 27. Example of binary weight accumulation.

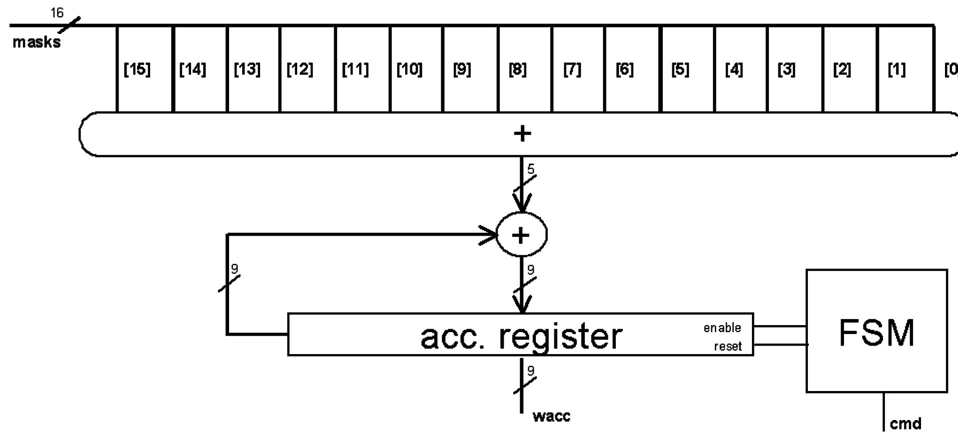

Fig. 28. Architecture of the weight factor accumulator.

for an example). Note that the sum of the weight values for all subsegments in a 16×16 block is always equal to 256 (the total area of the entire block). When the 16 × 3-bit weight outputs of the weight factor buffer are used to improve the ME quality (see the description of the weight factor buffer for more details), these weights first have to be multiplied with the corresponding subsegment weight mask bits before being accumulated.

Fig. 28 shows the architecture of the weight factor accumulator. Note that the upper adder counts the number of ones in the 16-bit weight mask. This value is accumulated for all lines in a block, using the lower adder and the accumulator register. Since the maximum result that can occur is 256, a precision of 9 bits is required. The accumulator register is reset to zero during initialization.

In Fig. 29 some pseudo code is shown (similar to sum of absolute difference unit).

B. Pseudo Code

The motion estimator ASIP presented in this work supports four different applications as described before. Depending on the application a minimum set of functional units is required. Table I shows the minimum required functional units per application. Obviously, the SAD functional unit is the minimum functionality needed to evaluate motion vectors. The SAD functional unit operates on either a SA/RB-buffer pair or a SA/SA-buffer pair (for up-conversion). A BI unit generally

```
wacc_asu_init();
for (cnt=0;cnt<16;cnt++) {
  wfb_read_out(masks, weights, reserved);
  wacc_asu_in(masks);
}
wacc_asu_out(wacc);
```

Fig. 29. Pseudocode showing the usage of the weight factor accumulator. Every instruction is single cycle.

TABLE I
INSTRUCTION-SET: MINIMUM FUNCTIONAL UNITS REQUIRED PER APPLICATION

	SA buffer	RB buffer	WF buffer	BI	SAD	WACC
video encoding	1	1	0	1	1	0
low speed obstacle detection	1	1	0	1	1	0
2D-to-3D video conv.	1	1	1	1	1	1
picture-rate up-conv.	2	0	0	1	1	0

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

```
For each block of block line:
  Fill RB buffer
  Fill WF buffer
  Fill SA buffer with complete search- area
  For all segments:
    For all vectors:
      Read candidate vector
      Initialize Application Specific Units
      Read first block line from SA buffer
      Supply first block line to BI
      For 16 block lines
        Read block-line from RB buffer          : P
        Read weight factors from WF buffer      : W
        Read block line from SA buffer
        Supply block line from SA buffer to BI
        Read interpolated line from BI          : N
        Supply results P, W, N to SAD unit
        Supply W to WACC unit
      Read block-SAD result from SAD unit
      Read accumulated weigh factor from WACC unit
      Store block-SAD result and accumulated weigh factor result
```

Fig. 30. Pseudocode of the 2-D-to-3-D algorithm. The synchronization granularity between the software task and the VLIW ASIP is an entire block line. The (up to) 4 subsegments within a macroblock are processed in sequential iterations. For every motion vector the innermost loop is called. The innermost loop iterates over the 16 lines within a macroblock. Note that bilinear interpolation requires 17 lines from SA buffer.

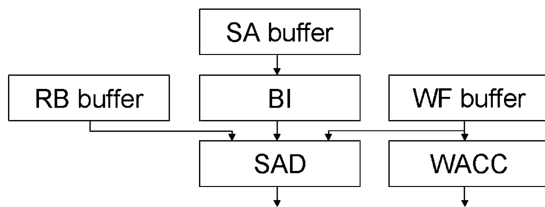

Fig. 31. Flow chart of the pixel processing performed by the functional units.

operates on a SA buffer output. For SBME, additional hardware assist is required. In the next paragraphs the 2-D-to-3-D video conversion and picturerate up-conversion applications are detailed out.

Fig. 30 details the pseudocode of the motion estimator process for the 2-D-to-3-D video conversion application. This pseudocode corresponds to the inner kernels of the algorithm; it contains mainly instructions to functional units.

The pseudocode consists of four nested loops each with a specific purpose. The outer loop iterates over all the 16×16 blocks within a block-line. At the start of a block iteration, the three (SA, RB, and WF) buffers are filled with all the data, required in further processing steps.

The first inner loop iterates over the (up to) four subsegments within a block. Currently the different subsegments are processed in consecutive passes. The next inner loop iterates over all candidate motion vectors. For every loop iteration the appropriate motion vector data is read and the functional units are initialized. The innermost loop iterates over the 16 lines of a block and produces the results for an entire block. The innermost loop mainly contains instruction calls to different functional units. The flow of data between the functional units is shown in Fig. 31. All functional units operate in a single-instruction/multiple-data (SIMD) fashion on either 16 pixels in parallel (RB buffer, WF buffer, SAD, and WACC) or 17 pixels in parallel (SA buffer and BI).

In Fig. 32 the schedule of the innermost loop, obtained via loop unrolling [7] is shown. During the first two cycles the first two block lines are fetched from the SA buffer and supplied to the BI functional unit (note that an additional line is required to perform bilinear interpolation, the SA buffer supports this requirements). Then during 15 cycles RB buffer, WF buffer, and BI outputs are fed into the consuming functional units (SAD and WACC). In cycle 17 the result of the last line is fed into the consuming functional units. Because of internal pipelining of the functional units an additional cycle is needed to produce the results.

After the SAD and weight values have been calculated in the inner loop, they are sent to the control task, which runs in software. The software task then performs the accumulation of the SAD results for entire segments and selection of the best candidate for each segment, respectively (substeps 2 and 3 as described in Section II-D).

In Fig. 33 pseudocode for the temporal up-conversion application is shown. Apart from 8×8 block processing this application uses a different set of functional units (two SA buffers, two BI units and a SAD unit). In Fig. 34 the flow of data between the functional units is shown while in Fig. 35 the desired schedule is depicted.

V. Results

The four applications are partitioned in a hardware and a software task respectively, as discussed in Section III. ME is implemented in hardware, using the VLIW architecture template described in Section IV. In order to prove the concept an FPGA realization is implemented of which the results are described in Section V-A. The results of standard-cell netlist simulations are presented in Section V-B.

A. *FPGA for Proof of Concept*

In order to prove the concept, we used the RAPIDO prototyping methodology [19]. Within this methodology a software and hardware task exist and execute in parallel. The prototyping setup consists of two parts: an off-the-shelf PC and the PCI-based Nallatech[7] FPGA board as shown in Fig. 36. The complete VLIW-based ASIP design is mapped on the FPGA board and represents the hardware task. The remainder of the application is realized as a software task and is executed on the PC. Communication between tasks is achieved by using communication calls as discussed in Section III-A.

Table II presents the FPGA device utilization for each of the four applications. During synthesis, dedicated block RAMs were used for mapping the SA buffer, RB buffer, and WF buffer respectively (12/4/4) instead of synthesizing these memories to logic gates.

B. *Standard-Cell Implementation*

The primary objective of this work was to design a motion estimator ASIP template that can be used by four different applications. This essentially involved integrating the four motion estimator applications into a single solution. We therefore

[7]Nallatech Ltd. [Online] Available: http://www.nallatech.com.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

Cycle ->	0	1	2≤N≤16	17	18
SA buffer	output $SA_{line\ 0}$	output $SA_{line\ 1}$	output $SA_{line\ N}$		
BI	input $SA_{line\ 0}$	input $SA_{line\ 1}$	output $BI_{line\ N-2}$ input $SA_{line\ N}$	output $BI_{line\ 15}$	
RB buffer			output $RB_{line\ N-2}$	output $RB_{line\ 15}$	
WF buffer			output $WF_{line\ N-2}$	output $WF_{line\ 15}$	
SAD			input $BI_{line\ N-2}$ $RB_{line\ N-2}$ $WF_{line\ N-2}$	input $BI_{line\ 15}$ $RB_{line\ 15}$ $WF_{line\ 15}$	output
WACC			input $WF_{line\ N-2}$	input $WF_{line\ 15}$	output

Fig. 32. Illustration of the optimal schedule for a single block processing. It takes 19 (0-18) cycles in total to process a 16×16 block. This includes bilinear interpolation.

For each block of block line:
*Fill **SA buffer 1***
*Fill **SA buffer 2***
For all vectors:
Read candidate vector
*Initialize **Application Specific Units***
*Read first block line from **SA buffer 1***
*Supply first block line of SA buffer 1 to **BI***
For 8 block lines
*Read block line from **SA buffer 1***
*Supply block line from SA buffer 1 to **BI***
*Read interpolated line from **BI 1**: F1*
*Read block line from **SA buffer 2**: F2*
*Supply results F1, F2 to **SAD unit***
*Read block-SAD result from **SAD unit***
Store block-SAD

Fig. 33. Pseudocode of the temporal up-conversion algorithm showing three nested loops. Within the outer loop, first both SA buffers are filled before a inner loop iterating over the number of candidate vectors is called. The innermost loop iterates over all eight block lines using effectively only half the SAD function.

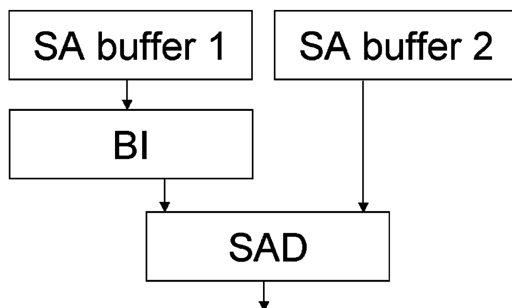

Fig. 34. Flow chart of the pixel processing performed by the functional units.

combined the C description of the four applications into a unified description, and compiled this unified description toward an ASIP template via the A|RT tool. Taking the unified description through the complete design flow is necessary for the allocation of instruction-encoding bits in the VLIW instruction word, for all the functional units used across all the applications.

The current version of the A|RT toolset produces VHDL that always clocks all registers and memories, whether the corresponding functional units are active or not. In order to reduce the power consumption of our design, we created scripts to automatically apply clock gating to all memories and registers at RTL level. Furthermore, ASUs, which are not active in a particular cycle, are clock gated. This reduces the average power consumption of the ASIP by 20%.

Table III summarizes the synthesis results of the clock-gated design for each of the four individual applications. Netlists for each of the designs were synthesized using the Cadence synthesis tools (Fig. 10). The area and power figures are listed for each component in the design. Area and power figures under the heading core includes the standard resources from the A|RT template together with their associated registers and multiplexers, a generic RAM, and a ROM for holding program constants. The power and area figures of each of the ASUs are listed individually. Power dissipation was obtained using a proprietary Philips power analysis tool.

The two major contributors to power dissipation in the ME ASIPs of each application are the processor core and the instruction memory. A compact encoding of the instruction memory can possibly be used to lower the power dissipation. Reducing the power dissipation of the core would however involve applying circuit level techniques for power reduction.

Table IV summarizes the synthesis results of the integrated ASIP that can run all the four applications. Due to a limitation in the A|RT toolset which prevented certain critical software optimizations in the resulting ASIP, the power numbers for 2-D-to-3-D video conversion application could not be obtained.

Interesting comparisons can also be made between the power consumption figures of the integrated ASIP and the ASIP implementations optimized for a single application. A major contributor to the increased power dissipation is the fact that instruction words in the ASIP become wider because of the extra bits included for encoding operations for all the functional units in the template. This indicates that a technique like instruction memory partitioning can be used to lower the power dissipation of the ASIP [20], [21]. The core on the other hand has a power dissipation that is 40% more on the average compared to that of the ASIP tailored to a single application. A large contributor to this increase is the larger register files associated with each of the ASUs in the ASIP template.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

Cycle ->	0	1	2≤N≤8	9	10
SA buffer 1	output $SA1_{line\ 0}$	output $SA1_{line\ 1}$	output $SA1_{line\ N}$		
BI	input $SA1_{line\ 0}$	input $SA1_{line\ 1}$	output $BI_{line\ N-2}$ input $SA1_{line\ N}$	output $BI_{line\ 7}$	
SA buffer 2			output $SA2_{line\ N-2}$	output $SA2_{line\ 7}$	
SAD			input $BI_{line\ N-2}$, $SA2_{line\ N-2}$	input $BI_{line\ 7}$, $SA2_{line\ 7}$	output

Fig. 35. Illustration of the optimal schedule for a single block processing. It takes 11 (0-10) cycles in total to process an 8×8 block; this includes bilinear interpolation.

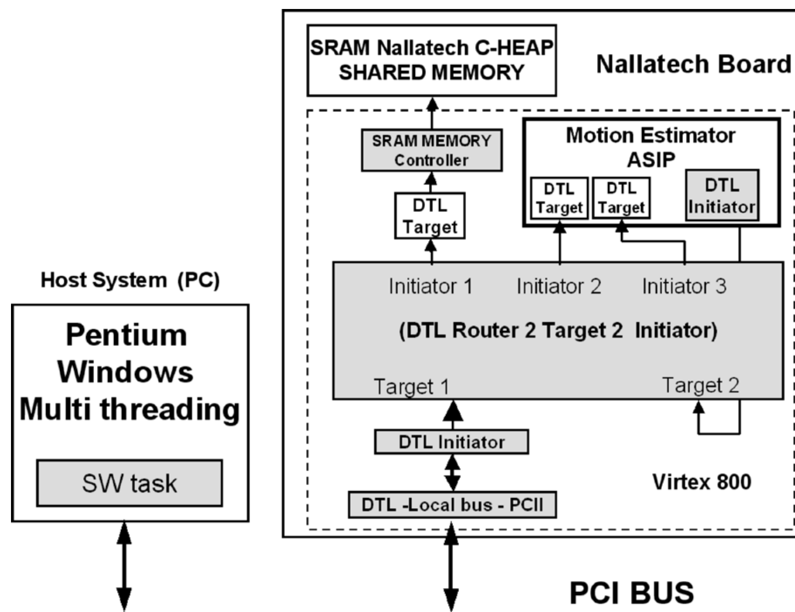

Fig. 36. Prototyping setup. The block on the right shows the Nallatech board mounting a Xilinx device (Virtex 800), and the left block shows the Host PC on which the rest of the application runs. Via a RAPIDO router the ASIP is connected to PCI bus and on-board shared memory.

TABLE II
UTILIZATION OF XCV800-BG432-6 FPGA DEVICE OPTIMISED FOR EACH OF THE FOUR APPLICATIONS. EVERY SLICE CONTAINS TWO 4-INPUT LUTS AND TWO FLIP FLOPS. BUFFERS ARE MAPPED ON BLOCK-RAM MODULES (12 FOR SA BUFFER, 4 FOR RB BUFFER, AND 4 FOR WF BUFFER)

Application	Total slices	IO-pins	Block RAMs
video encoding	7570 (80%)	97 (30%)	16 (57%)
low speed obstacle detection	6555 (69%)	97 (30%)	16 (57%)
2D-to-3D video conv.	9335 (99%)	97 (30%)	20 (71%)
picture-rate up-conv.	9060 (96%)	97 (30%)	24 (85%)

VI. CONCLUSION

A methodology to design and implement a ME ASIP has been presented. The applicability of the design over a wide application range has been demonstrated. The ASIP approach clearly demonstrates the benefits of alternative space/time computing solutions as opposed to the traditional approach of ASICs and general-purpose programmable processors.

An important secondary goal was to demonstrate the advantages of a C-based design entry in the design of ASIPs. A C-based design entry level allowed us to rapidly adapt our algorithms to the architecture and vice versa. In addition it allowed easy and efficient hardwaresoftware partitioning. The design of the ASIP was realized in 10 man months.

The compute intensive parts of the motion estimator are executed by ASUs. The power dissipation of the ASUs for the three benchmarked applications in the integrated ASIP ranges from 45% to 52% of the total power dissipation of the processing core. Also, we have built some configurability into the ASUs without detrimentally affecting performance or power dissipation. These ASUs not only include data paths that implement kernels but also customized memory system that can expose high bandwidth to the kernels.

The estimated area of the integrated ASIP is 4.031 mm^2 in 0.18 μm technology with a power dissipation ranging from 9 to 141 mW depending on the application. If we optimize the de-

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

TABLE III
SYNTHESIS (TOP) AND PREAYOUT NETLIST POWER SIMULATION (BOTTOM) RESULTS FOR ASIP OPTIMISED FOR A SINGLE APPLICATION. WORST CASE SYNTHESIS CONDITIONS IN 0.18 μm TECHNOLOGY. TYPICAL CASE CONDITIONS FOR POWER ESTIMATION

Area [mm^2]	Total	Core	IM	DM
video encoding	1.596	1.448	0.103	0.045
low speed obstacle detection	1.599	1.449	0.103	0.045
2D-to-3D video conv.	2.763	1.670	0.103	0.990
picture-rate up-conv.	2.584	1.986	0.103	0.495
SA buffer	0.655			
RB buffer	0.138			
WF buffer	0.138			
BI	0.286			
SAD	0.092			
WACC	0.027			

Power [mW]	video encoding CIF@30fps	low speed obstacle detection 384*288@25fps	2D-to-3D conversion CIF@18fps	up-conversion SD@38fps
Total	7.09	5.16	73.56	90.31
Core	5.41	4.02	55.91	72.72
SA buffer 1	3.21	2.57	26.44	22.27
SA buffer 2	-	-	-	22.31
RB buffer	0.51	0.39	3.73	-
WF buffer	-	-	1.69	-
BI	0.26	0.19	3.74	0.74
SAD	0.25	0.18	3.72	0.72
WACC	-	-	2.19	-
IM	1.65	1.12	16.62	16.7
RAM/ROM	0.03	0.02	1.03	0.89

TABLE IV
SYNTHESIS (TOP) AND PRELAYOUT NETLIST POWER SIMULATION (BOTTOM) RESULTS FOR INTEGRATED ASIP. SAME SIMULATION AND PROCESSING CONDITIONS AS BEFORE

Area [mm^2]	Total	Core	IM	DM
Integrated ASIP	4.031	2.430	0.611	0.990

Power [mW]	video encoding CIF@30fps	low speed obstacle detection 384*288@25fps	up-conversion SD@38fps
Total	13.41	9.42	141.34
Core	8.33	6.11	91.73
SA buffer 1	2.69	2.26	21.48
SA buffer 2	-	-	21.66
RB buffer	0.51	0.39	-
WF buffer	-	-	-
BI	0.26	0.19	2.10
SAD	0.25	0.17	2.04
WACC	-	-	-
IM	5.03	3.27	48.76
RAM/ROM	0.05	0.04	0.85

sign to support a particular application we can reduce the power consumption of the ASIP by about 50%.

First experiments based on improvements on the picturerate up-conversion application show that the ASIP design dissipates 27 mW and occupies an area of 1.1 mm^2 in 0.13 μm technology, for standard definition (CCIR601) video resolution at 50 frames per second [22].

REFERENCES

[1] G. de Haan, *Video Processing for Multimedia Systems*, 3rd ed. Eindhoven, The Netherlands: University Press, May 2003.

[2] F. Ernst, P. Wilinski, and K. van Overveld, "Dense structurefrom-motion: an approach based on segment matching," in *Proc. ECCV*, Copenhagen, 2002, Paper no. LCNS 2531, pp. II:17–II:23.

[3] W. Wolf, B. Ozer, and T. Lu, *Smart Cameras as Embedded Systems*. Piscataway, NJ: IEEE Press, 2002.

[4] J. Jachalsky, M. Wahle, P. Pirsch, S. Capperon, W. Gehrke, W. M. Kruijtzer, and A. Nuñez, "A core for ambient and mobile intelligent imaging applications," in *Proc. IEEE Int. Conf. Multimedia and Expo (ICME)*, 2003, pp. II–1–4.

[5] G. de Haan, "Video format conversion," *J. Society Inform. Display (SID)*, vol. 8, no. 1, pp. 79–87, 2000.

[6] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards, Algorithms and Architectures*, 2nd ed. Norwell, MA: Kluwer, 1997.

[7] AlRT Designer and AlRT Builder Tools. Adelante Technologies. [Online]. Available: http://www.adelantetechnologies.com

[8] V. Aue, J. Kneip, M. Weiss, M. Bolle, and G. Fetweis, "A design methodology for high performance Ics: wireless broadband radio basedband case study," in *Proc. EuroMicro Symp. Digital System Design*, Sep. 2001, pp. 16–20.

[9] A. Hoffmann, A. Nohl, G. Braun, O. Schliebusch, T. Kogel, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIP) using a machine description language," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 20, no. 11, pp. 1338–1354, Nov. 2001.

[10] D. Lanneer, J. van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens, "CHESS: retargetable code generation for embedded DSP processors," in *Code Generation for Embedded Processors*, P. Marwedel, Ed. Norwell, MA: Kluwer, 1995.

[11] B. Ramakrishna Rau and M. S. Schlansker, "Embedded computer architecture and automation," *IEEE Computer*, vol. 34, pp. 75–81, Apr. 2001.

[12] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busa', R. P. Llopis, K. Goosens, and P. Lippens, "C-HEAP: a heterogeneous multiprocessor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation Embedded Syst.*, vol. 3, pp. 233–270, 2002.

[13] G. de Haan, P. Biezen, H. Huijgen, and A. Ojo, "True motion estimation with 3-D recursive search block-matching," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 3, no. 10, pp. 368–379, Oct. 1993.

[14] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filter for on-line nonlinear/non-Gaussian Bayesian tracking," *IEEE Trans.Signal Process.*, vol. 50, no. 2, pp. 174–188, Feb. 2002.

[15] R. Wittebrood and G. de Haan, "Real-time recursive motion segmentation of video data on a programmable device," *IEEE Trans. Consumer Electron.*, vol. 47, pp. 559–567, Aug. 2001.

[16] A. Berić, G. de Haan, R. Sethuraman, and J. van Meerbergen, "A technique for reducing complexity of recursive motion estimation algorithms," in *Proc. IEEE Workshop Signal Processing Syst.*, Aug. 2003, pp. 195–200.

[17] A. Berić, G. de Haan, J. van Meerbergen, and R. Sethuraman, "Toward an efficient high quality picturerate up-converter," in *Proc. IEEE Int. Conf. Image Processing*, Sep. 2003, pp. 363–366.

[18] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing*, J. Rosenfeld, Ed. Amsterdam, The Netherlands: North-Holland Publishing Co., 1974.

[19] N. Busa', G. Alkadi, M. Verberne, R. Peset LLopis, and S. Ramanathan, "RAPIDO: a modular, multiboard, heterogeneous multiprocessor, PCI-bus based prototyping framework for the validation of SoC VLSI designs," in *Proc. IEEE Workshop Rapid System Prototyping*, 2002, pp. 159–165.

[20] A. Macii, E. Macii, and M. Poncino, "Improving the efficiency of memory partitioning by address clustering," in *Design Automation Test in Europe Conf. Exhibition*, 2003, pp. 18–23.

[21] L. Benini, L. Macchiarulo, A. Macci, and M. Poncino, "Layout-driven memory synthesis for embedded systems-on-chip," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 10, no. 2, pp. 96–105, Apr. 2002.

[22] A. Berić, R. Sethuraman, H. Peters, J. van Meerbergen, G. de Haan, and C. Alba Pinto, "A 27 mw 1.1 mm^2. motion estimator for picturerate up-converter," in *Proc. 17th IEEE Int. Conf. VLSI Design*, Jan. 2004, pp. 1083–1088.

[23] J. L. Hennesy and D. A. Patterson, *Computer Architecture: A quantitative approach*. San Mateo, CA: Morgan Kaufmann, 1996.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

Harm Peters received the B.Sc. degree in electronics at Hogeschool Eindhoven, Eindhoven, The Netherlands, in 1992.

He worked for four years on research of Integrated Circuit failure analysis at Philips Research, Eindhoven. In 1996, he moved to Philips Semiconductors, where he specialized on signal integrity aspects and robust integrated circuit design. In 2000, he returned to Philips Research working on architectural synthesis and hardware design for video signal processing architectures. His interests include design methodology, processing architectures, and hardware design of complex systems.

Ramanathan Sethuraman (M'00) received the M.Sc. and Ph.D. degrees in electrical engineering from the Indian Institute of Science, Bangalore, India, in 1992 and 1997, respectively.

Currently, he is a Senior Scientist in the Embedded Systems Architectures on Silicon Group of Philips Research Eindhoven, Eindhoven, The Netherlands. His research interests include embedded system design, low-power VLSI systems, hardware/software codesign, VLSI systems for multimedia, VLSI signal processing, and RISC/VLIW processor architectures. He has published more than 35 articles and filed ten patents.

Dr. Sethuraman received the Best M.Sc. and Ph.D. thesis Award from the Indian Institute of Science.

Aleksandar Berić received the degree in electrical engineering from the University of Belgrade, Belgrade, Serbia in 2001. He is currently working toward the Ph.D. degree at the University of Eindhoven, Eindhoven, The Netherlands.

His fields of interest are algorithm/architecture codesign of video processing functions, motion estimation algorithms, low-power VLSI systems, and processor architectures.

Patrick Meuwissen received the M.Sc. degree in information technological science from Eindhoven University of Technology, Eindhoven, The Netherlands, in 1997.

Currently, he is a Senior Scientist in the Embedded Systems Architectures on Silicon Group of Philips Research Eindhoven, The Netherlands. His research interests include system architecture, embedded system design, hardware/software codesign, and VLSI Systems for Multimedia.

Srinivasan Balakrishnan (M'00) received the Ph.D. degree in computer science from the Supercomputer Education and Research Centre, Bangalore, India, in 2000.

Since July 2001, he has been a Senior Scientist in the Embedded Systems Architectures on Silicon Group of Philips Research Eindhoven, Eindhoven, The Netherlands. His interests include the design of uniprocessor and multiprocessor systems, application specific architectures, vector processing, and compilation techniques for vector machines.

Carlos Antonio Alba Pinto (M'00) received the B.Sc. degree in electronics engineering at the Catholic University of Lima, Lima, Peru in 1994, and the M.Sc. degree in microelectronics from the Federal University of Rio Grande do Sul, Porto Alegre, Brazil in 1996. He received the Ph.D. degree from Eindhoven University of Technology, Eindhoven, The Netherlands, in 2002.

Since 2001, he has been a member of the Embedded Systems Architectures on Silicon Group of Philips Research Eindhoven, The Netherlands. His research interests include embedded systems' architectures design for video signal processing and smart imaging applications.

Wido Kruijtzer (M'01) received the M.Sc. degree in electrical engineering from Eindhoven University of Technology, Eindhoven, The Netherlands, in 1995.

Currently, he is a Senior Scientist in the Embedded Systems Architectures on Silicon Group of Philips Research Eindhoven, The Netherlands. He is active in the field of embedded systems architectures, with a focus on design technology. His research interests include system-level design methodologies, embedded systems architectures, and embedded processors. He regularly serves on program committees of major conferences in his field, such as DATE.

Fabian Ernst (M'01) received two M.Sc. degrees (*cum laude*) in applied mathematics and systems engineering, policy analysis, and management in 1994 and 1997, respectively, and the Ph.D. degree (*cum laude*) in 1999 all from Delft University of Technology, Delft, The Netherlands.

Since 1999, he has been working as a Senior Scientist at Philips Research Eindhoven, Eindhoven, The Netherlands. His research interests include signal processing, image and video processing, and wave propagation and scattering.

Dr. Ernst is a member of the Society of Exploration Geophysicists (SEG) and the European Association of Geoscientists and Engineers (EAGE).

Ghiath AlKadi received the B.Sc. degree in analogue electronics from the Technical University of Applied Science, Damascus, Syria, and the B.Sc. degree in computer science at the Technical University of Applied Science, Rotterdam, The Netherlands. He is currently working toward the M.S. degree in computer science at Eindhoven University, Eindhoven, The Netherlands.

Currently, he is a Research Engineer in the Embedded Systems Architectures on Silicon Group of Philips Research Eindhoven, Eindhoven, The Netherlands. His research interest includes prototyping methodology for system-on-chip and streaming video applications.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.

Jef van Meerbergen (M'87–SM'92) received the M.S. degree in electrical engineering and the Ph.D. degree from the Katholieke Universiteit Leuven, Belgium, in 1975 and 1980, respectively.

In 1979, he joined Philips Research Eindhoven, Eindhoven, The Netherlands, where he started to design MOS digital circuits, domain-specific processors, and general-purpose digital signal processors. He was the Project Leader of the Sigma-Pi project which delivered the first general purpose DSP within Philips. In 1985, he started working on application-driven high-level synthesis in the context of a European project in close cooperation with Imec. Initially, this work was targeted toward DSP applications and resulted in the AR|T system which is used to design audio, video, and communication functions. Later, the application domain shifted toward high-throughput streaming applications for which the Phideo compiler was developed. This compiler was used for the design of feature box ICs for 100-Hz conversion for TV (Melzonic, Falconic) and for MPEG2 encoding (I.McIC). The Phideo paper received the best paper award at the 1997 ED&TC conference. His current interests are in design methods, heterogeneous multiprocessor systems, reconfigurable architectures and Networks-on-Silicon. He is a part-time professor at the Eindhoven University of Technology, Eindhoven, The Netherlands.

Dr. van Meerbergen is a Philips Research Fellow and Associate Editor of *Design Automation for Embedded Systems Journal*.

Gerard de Haan (M'95–SM'97) received the B.Sc., M.Sc., and Ph.D. degrees from Delft University of Technology, Delft, The Netherlands, in 1977, 1979, and 1992, respectively.

He joined Philips Research, Eindhoven, The Netherlands, in 1979. Currently, he is a Research Fellow in the Group Video Processing and Visual Perception of Philips Research Eindhoven and a Professor at the Eindhoven University of Technology, Eindhoven, The Netherlands. He has a particular interest in algorithms for motion estimation, scan rate conversion, and image enhancement. His work in these areas has resulted in several books, more than 100 scientific papers, more than 50 patents, and several commercially available ICs.

Dr. de Haan was the First Place Winner in the 1995 and 2002 ICCE Outstanding Paper Awards program, the Second Place Winner in 1997 and 1998, and the 1998 recipient of the Gilles Holst Award. In 2002, he received the Chester Sall Award from the IEEE Consumer Electronics Society. The Philips "Natural Motion Television" concept, based on his Ph.D. dissertation, received the European Innovation Award of the Year 95/96 from the European Imaging and Sound Association, its successor "Digital Natural Motion" received a Wall Street Journal Europe Business Innovation Award 2001.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on August 26, 2009 at 04:02 from IEEE Xplore. Restrictions apply.