ALU Augmentation for MPEG-4 Repetitive Padding

Georgi Kuzmanov Stamatis Vassiliadis
Computer Engineering Lab, Electrical Engineering Department,
Faculty of Information Technology and Systems, Delft University of Technology,
P.O. Box 5031, 2600 GA Delft, The Netherlands
E-mail:{G.Kuzmanov, S.Vassiliadis}@ET.TUDelft.NL

Abstract

In this paper we augment a general purpose ALU with an extra functionality - a repetitive padding operation. The proposed solution enables the processor to perform the time exhaustive MPEG-4 padding algorithm in real time. At trivial hardware costs of a few hundred 2x2 AND-OR (or equivalent) logical gates, we achieve an order of magnitude speed-up when compared to software running on a general purpose processor. Our approach allows the MPEG-4 software padding algorithm to run in real time for its most demanding profiles. As an example, a 64-bit pipelined implementation is discussed in details. The approach is general and fits into different architectural concepts and ALU operand widths. More specifically it is shown that 344 extra gates are necessary for a 64-bit implementation and at these expenses a processing speed of more than 7 million macroblocks per second (MB/s) can be achieved. This is an order of magnitude higher than the requirements of the most-demanding MPEG-4 profile levels. Speed and hardware estimations are also reported for 32 and 128-bit ALUs.

1. Introduction

Assuming MPEG standards, MPEG-4 [5, 6] is the first to deal with content-based coding of audio-visual scenes. To allow the efficient implementation of the standard, MPEG-4 defines several profiles. These profiles group the large set of required tools, according to the targeted classes of applications. Within each profile, a number of levels constrain the computational complexity and the required data bandwidth of the application.

In literature [8, 9], complexity analysis results indicate that the computational requirements of the highest profiles and levels of MPEG-4 are measured in billions of RISC-like instructions per second. These numbers will significantly exceed the capabilities of the general purpose processors, despite the near future technology achievements. The work presented in this paper is a part of a research activity, in which we focus on speeding MPEG-4 applications up. Our basic approach is to augment general-purpose Arithmetic-Logical-Units (ALU) with application specific functionalities. The specific functions are used to redefine the architecture and are implemented in hardware. Due to the different requirements of the MPEG-4 profiles and the large number of functionalities involved in it, the requirements for cost-effective implementations are essential. One important new feature in MPEG-4 is the *padding* technique, defined at all Levels in the Core and Main Profiles of the standard. Software profiling results, reported in [2, 3, 8, 14, 15], indicate that padding is a computationally demanding and time consuming process, which restricts the real time operation of the MPEG-4 codecs.

In this paper we propose an ALU augmentation circuitry, which enables a general purpose processor to perform the padding algorithm in real time. Thus we utilize the available resources of the ALU, and by exploiting the sub-word parallelism and dedicated instructions, we dramatically increase the computational power of the architecture. The proposed solution proved the following advantages:

- Real time processing for all MPEG-4 profiles and levels, utilizing the padding algorithm can be achieved
- Scalable implementation, tunable to different operand sizes
- General implementation for different architectural concepts

The implementation is estimated analytically and its influence on the performance and the hardware cost of augmented ALUs with different operand widths is discussed. Our results indicate high processing speed, achieved at trivial implementation costs. The achieved worst-case benefits for a 64-bit ALU example are:

- Approximately 30 times faster processing than the highest MPEG-4 requirements.

ISBN: 0-9669530-0-2

- Over 1000 times less MIPS (Millions of RISC-like Instructions per Second) than the software implementation on general purpose processors.
- Only 344 AND-OR gates extra hardware penalty.

Our results strongly suggest that software running on a processor incorporating our proposal will always meet the highest standard requirements.

The remainder of the discussion in this paper is organized as follows. Section 2 gives some background information and the motivation for the presented research, including a description of the repetitive padding algorithm. In Section 3, we describe the proposed implementation in details. Section 4 gives a quantitative evaluation of the design. Finally, the conclusions are presented in Section 5.

2. Background and Motivation

For content-based coding, MPEG-4 uses the concept of a Video Object Plane (VOP). A VOP is an arbitrarily shaped region of a frame, which usually corresponds to a semantic object in the visual scene. A sequence of VOPs in the time domain is referred to as a Video Object (VO). Each VOP is described by its *shape* and *texture*.
Shape is mainly represented in binary format. This format represents the shape as a bitmap, referred to as *binary alpha plane*. Each pixel in this plane takes one of two possible values, which indicate whether the pixel belongs to the object or not. The binary alpha plane is divided into 16x16-pixel blocks called *Binary Alpha Blocks (BAB)*.
The texture of a VOP represents its color by *macroblocks*. Each macroblock consists of one 16x16 array of luminance (grayscale) pixels and two 8x8 arrays of chrominance (color) pixels, which represent the full-color of the corresponding 16x16 area of a VOP.

As its preceding visual data compression standards, MPEG-4 adopts *motion compensation* techniques to exploit temporal redundancies in the encoded video sequences.In MPEG-4, this process includes a search algorithm for best matching between the macroblock to be encoded and an area of previously encoded frame.

The Repetitive Padding Algorithm. *The purpose of padding in MPEG-4 is to ensure more accurate block matching in motion compensation algorithms for arbitrary shaped visual objects.* The padding process defines the full-color values (luminance + chrominance) for pixels outside the shape of a VOP. In padding, two types of macroblocks are of interest. Macroblocks, which lie on the boundary of the VOP are referred to as boundary blocks. They are processed by the so called *repetitive padding*. Exterior macroblocks (completely outside the VOP) are padded using the *extended padding method.* Since repetitive padding is the most demanding padding algorithm, in this paper we will consider the padding of boundary macroblocks. The repetitive padding algorithm is described in [5, 13], but in literature some modifications can be met. In [4, 7, 11] new algorithms or algorithm modifications are proposed to redefine or even substitute the original repetitive padding. All of them, however, suggest software improvements of the coding efficiency and visual quality and do not focus on the real time performance. A hardware acceleration of the padding is discussed in [1]. In the same paper, the padding algorithm is modified to support specific instruction set extensions as the horizontal and vertical padding processes are divided into two phases each. These two phases consequently scan the lines/columns into two opposite directions and perform the padding operations. The proposed hardware solution there is a hardwired dedicated padding unit. In the present paper we use the standard repetitive padding algorithm (differentiates from [4, 7, 11]) on a general ALU (differentiates from [1, 4, 7, 11]), where a boundary block is separately processed horizontally, per scan-line basis and vertically - per columns. We still scan each line and column of a macroblock bidirectionally, but we do it in parallel (differentiates from [1]), thus saving a number of processing cycles. The standard repetitive padding algorithm, as defined in [5], is equivalent to the following steps:

1. Define any pixel outside the object boundary as a zero pixel. Make a duplicate binary alpha map.
2. Scan each horizontal line of a block. Each scan line is possibly composed of *zero* and *nonzero* line segments (according to the shape bits in the binary alpha map).
   (a) In zero segments, between an end point of the scan line and the end point of a nonzero segment, all zero pixels are replaced by the pixel value of the end pixel of nonzero segment.
   (b) In zero segments, between the end points of two different nonzero segments, all zero pixels take the average value of these two end points.

   Nonzero segments are not processed. All shape bits, corresponding to padded pixels are set in the duplicate binary alpha map.
3. Scan each vertical line of the block and perform the identical procedure as described for the horizontal line. The updated shape information from the duplicate binary alpha map is used.

Unlike its predecessors, MPEG-4 is much more demanding in terms of computational complexity with even more data intensive algorithms, which is illustrated in Table 1. While the computational complexity of the future general purpose processors may meet the demands of the lowest

 ISBN: 0-9669530-0-2

Simple Profile Levels, the challenge is still to meet the requirements of the most-demanding Core and Main Visual Profile of MPEG-4.

Table 1. Visual Profiles@Levels Definitions and Processing Speed in MacroBlocks per second [MB/s]

Profile	Level	Session Size	# VO	Max. MB/s	Boundary MB/s
Main	L4	1920x1088	32	489600	244800
	L3	CCIR 601	32	97200	48600
	L2	CIF	16	23760	11880
	L1	N.A.	N.A.	N.A.	N.A.
Core	L2	CIF	16	23760	11880
	L1	QCIF	4	5940	2970
Simple Scalable	L2	CIF	4	23760	N.A.
	L1	CIF	4	7425	N.A.
Simple	L3	CIF	4	11880	N.A.
	L2	CIF	4	5940	N.A.
	L1	QCIF	4	1485	N.A.

Motivation. A summary of the computational complexity of the QCIF, Core Profile Level 1 of MPEG-4 is reported in [8]. Since this is the lowest profile level, utilizing the padding algorithm, we shall consider its real-time requirements as the minimum for a hardware implementation. At this level, the computational power, reported for the software encoding of a single object is in the order of 4500 Million RISC-like Instructions Per Second (MIPS). Assuming a software performance optimization by a factor of up to 10, the total computational complexity is just within the computational capabilities of the contemporary general purpose processors(500-1000 MIPS). In the case of 4 video objects (see Table 1), however, the real-time software feasibility becomes problematic. Therefore, the need of a hardware acceleration of MPEG-4 is evident, even at this relatively low profile level. Further analysis of the requirements for the software implementation indicates that the padding algorithm occupies some 175 MIPS for a single video object, or around 700 MIPS for the maximum 4 video objects, stated at Level 1 of the Core profile (Table 1). A general purpose processor with such computational power is still very expensive to be dedicated just for the repetitive padding calculations. If we consider Table 1, we can estimate that the required speed of 5940 MB/s for the Core Profile Level 1 is approximately 82 times lower than the speed requirements of the highest - Main@Level4 Profile (489600 MB/s). A simple arithmetic estimation indicates that for the highest MPEG-4 profile level, the non-optimized software padding would require approximately 57 000 MIPS and when extremely optimized (10 times speed-up) - in the order of 6000 MIPS. Even for the significantly less complex decoder part of MPEG-4, the padding algorithm will require some 24 000 MIPS for non-optimized software implementation down to 2500 MIPS in dramatically optimized programming. These numbers, huge even for the expected technology levels of the near future, motivated our research to focus on a cost-effective hardware solution of the MPEG-4 algorithms and the repetitive padding in particular.

The large number of alternatively used algorithms in MPEG-4, however, makes the implementation of dedicated hardware units inefficient, since a lot of them may remain unutilized. A good example is the padding algorithm, which is not included in the Simple Profile Levels of MPEG-4. Thus, a multi-profile codec would not utilize a hardwired padding accelerator, when running at any of the Simple Profile levels. A promising solution of this problem is the reconfigurable implementation of hardware accelerators [10, 16].

Another possible approach to increase the efficiency of a general purpose architecture is adopted in this paper. We augment the ALU with additional circuitry that enriches its functionality with one more operation - the repetitive padding. Thus we utilize the available resources of the ALU, and by exploiting a sub-word parallelism and dedicated instructions, we dramatically increase the computational power of the architecture.

3. The Augmented ALU

Since padding is performed over horizontal and vertical pixel lines in identical manner and the width of the pixel data is relatively small (8 or 12 bits in MPEG-4), a wider general purpose ALU can not compute it efficiently. To improve the computational efficiency of such ALUs, we can exploit a sub-word data parallelism. In this section we augment a general purpose ALU, to make it capable to perform a padding operation. Since 8-bit integer chrominance and luminance data representations are the most frequently used, in this paper we assume the same data formats.

Pixel Processing. A single byte processing structure, which is dedicated to process each pixel of a block, is depicted in Figure 1. The same structure is used both for luminance and chrominance blocks padding. We pipeline the processing flow by dividing it into two stages. The first stage contains a Propagation Node (PN) and two multiplexers. The multiplexers are required to preserve the original functionality of the ALU. A byte adder and an output multiplexer build the second pipeline stage. The byte adder is a part of the original multi-byte ALU adder, with controllable carries between the bytes. The padding output multiplexer can be merged with the existing ALU output multiplexer and that is depicted in Figure 1 by the dash-lined arrow, from the logical part of the ALU (LU).

The function of the PN is to propagate the appropriate

ISBN: 0-9669530-0-2

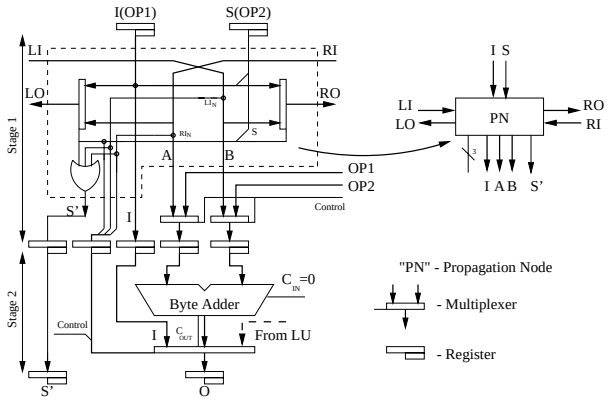


Figure 1. An ALU augmentation for a single pixel padding

values to its neighboring padding PNs and to supply data and control signals to the byte adder and the output multiplexer. Table 2 represents the control signals of the output multiplexer (SR(C_{OUT},ADD) means Shift Right with 1-bit the ADDer output together with its carry). Signal "Control" determines whether the ALU will perform padding, or its original operation(s).

Table 2. Truth Table for the Control Signals of the Output Multiplexer

Control	S	RI_N	RI_N	O
0	X	X	X	ADD
1	0	0	0	I
1	0	0	1	ADD
1	0	1	0	ADD
1	0	1	1	SR(C_{OUT},ADD)
1	1	X	X	I

The following equations describe the functionality of the PN:

$$LO = S \cdot |\overline{S,I}| \vee \overline{S} \cdot RI, RO = S \cdot |\overline{S,I}| \vee \overline{S} \cdot LI, \quad (1)$$

$$S' = S \vee LI_8 \vee RI_8; \quad (2)$$

LI, RI are left and right 9 bit input vectors;
LO, RO are left and right 9 bit output vectors;
I is the data input 8 bit vector;
S is the shape (input) bit before processing;
$S^{'}$ is a mask output bit after processing;
$|\overline{S,I}|$ denotes the concatenation of bit S and vector I;
and LI_N represents the N^{th} bit of vector LI.

The operation of the PN is as follows.

- If the input shape bit S is set (the pixel belongs to the object), then:
  1. The output O takes the value of the input I, i.e. the pixel keeps its color.
  2. The value of the input (pixel) I is propagated to the left and to the right (via outputs LO and RO) for further processing. The shape input bit S is propagated by the same multiplexers and occupies the most-significant bits of LO and RO.
  3. The output bit $S^{'}$ is set, meaning the pixel has been processed.

- If the input shape bit S is zero (the pixel does not belong to the object and has to be padded), then:
  1. The outputs A and B propagate to the byte adder the least significant 8-bits of RI and LI respectively. The propagated neighboring shape bit values (LI_8, RI_8) and bit S, are issued as control signals to the output multiplexer. The multiplexer redirects to O either the output of the adder or its right shifted value (see Table 2) i.e. the pixel takes the padded value.
  2. The LI value is propagated via RO and the RI - via LO including color and shape information.
  3. The output bit $S^{'}$ is set, meaning the pixel has been processed.

Line / Column Padding. To process a line or a column from a block by an n-byte ALU, we have to implement a chain of n PN (i.e. n-pixel parallel padding). A section of such processing circuitry for two neighboring pixels is depicted in Figure 2. This structure is scalable and can contain an arbitrary number of propagation nodes, depending on the ALU width (i.e. n elements for an n-byte ALU). It has the following features:

1. Operands are bypassing the first pipeline stage for a conventional ALU operation, thus preserving the original timing.
2. Operands are passed through the Propagation Nodes only when a padding operation is performed
3. The critical path of the operands through the PNs for an n-byte adder is equal to the delay of n multiplexers.
4. The critical path penalty for a read ALU operand is only one 2-1 multiplexer.
5. The ALU critical path penalty is a single 2-way AND element.

ISBN: 0-9669530-0-2

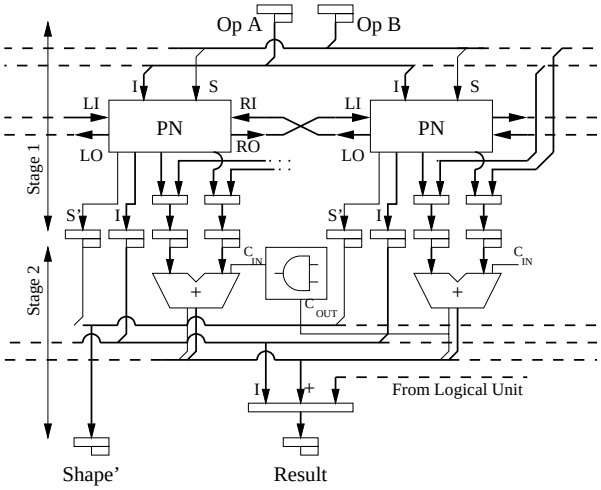
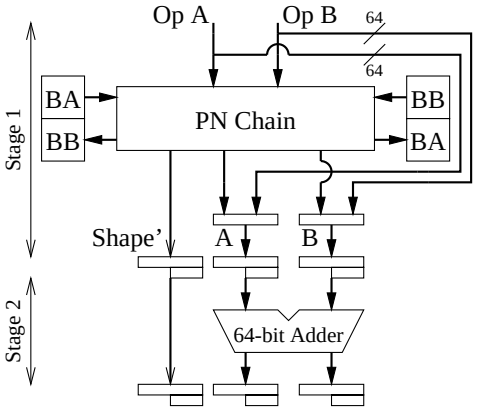
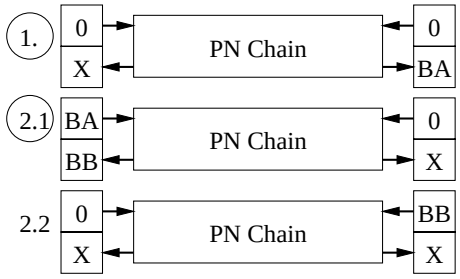


Figure 2. A Scan Line / Column Padding Augmentation of an ALU



(a) General View of the Pipeline Stages



BN - Buffer N; X-Don't care; 0-zero.

(b) Data Buffering by Cycles

Figure 3. Data Initialization and Buffering for Luminance Line / Column Processing by a 64-bit ALU.

The last two penalties may be avoided if special designs are employed (see [12]).

Putting Everything Together. Since a macroblock consist of one 16x16 luminance and two 8x8 chrominance blocks, it is efficient to implement structures processing 8 or 16 pixels simultaneously. This means that with 64 or 128-bit ALUs we will be able to pad an entire line or column of chrominance and/or luminance blocks in pipeline manner for two ALU cycles. However, the proposed structure is capable to perform padding even when it is implemented on smaller ALUs.

Here, we will explain in more details the padding process flow, performed by a 64-bit ALU. In the next section we will give generalized estimations for n-byte ALUs. First, for the proper circuit operation, the left-most and right-most inputs of the structure should be initialized. Figure 3(a) depicts a general view of the 64-bit initialization for luminance line / column padding. The BA and BB buffers are 8-bits wide and contain the initialization values, required by the unit to start the operation. In Figure 3(b), the cycle partitioning of the luminance line padding is illustrated. Since a luminance line (16x8-bit) can not be processed in one pass by a 64 bit (8x8-bit) ALU, we assume that the left-most half of the line is processed first. Depending on the right-most shape bit of this first half of the line, the full-line padding would require one or two more cycles.

If the right-most shape bit of the first half of the luminance line is "0", it means that data from the other half of the line is required to pad the first half. Thus the data, propagated to the right is stored into BA register. In the next cycle this data is used to fully pad the second half of the line, and the byte, which has to be propagated to the left is stored in buffer BB. Now, a third cycle is executed on the first half of the luminance line, with the proper right-to-left propagation value stored in the BB buffer.

If the right-most shape bit of the first half of the luminance line is "1", the pixel value to be propagated right is stored into buffer BA, and the first half of the luminance line is completely padded. Just one more cycle is required to pad the second half of the luminance line, provided the right propagation value is driven to the left input of the circuit from BA.

Since the right-most shape bit of the first half of the luminance line is available before the next operands (the other half of the padded line) are issued, we have branch determination (a perfect branch prediction) for the pipeline. Therefore, we can conclude that *a complete luminance line padding by a 64-bit padding augmented and pipelined ALU would take on average 2.5 cycles to perform. The padding*

ISBN: 0-9669530-0-2

of a chrominance line by the same ALU would take approximately one cycle for a long data sequence.

The processing flow is similar for a 32-bit ALU, but the required average cycle number is 5.5 (at least 4 at most 7 cycles), and the analysis is made on the right-most shape bits of each byte (4x8-bits) to be processed. Given the shape information of the whole line is available a priory we can still make a perfect branch prediction, i.e. we can predict the operands issue sequence perfectly. Note on Figure 3(b) that the left and right-most inputs of the whole luminance line are initialized with "0", including the propagation of the shape bit. This is also valid for the chrominance line processing and for all up or down scaling ALU implementations (say 32 or 128 bit ALUs). The next section gives more accurate estimation of the processing speed of several different implementations.

4. Evaluation

Speed estimation. We can easily evaluate the processing speed of the structure, given its operating speed [1]. Let's assume an $n \cdot 8$-bit padding augmented ALU like the one, depicted in Figure 2, operating at frequency F_n [Hz]. The values of *n* with practical significance are 4, 8, 16 (32, 64 or even 128-bit ALU). We state two more parameters of the particular implementation:
$\mathbf{N_n^{P8}}$ *and* $\mathbf{N_n^{P16}}$ *respectively denote the numbers of cycles, necessary to process an 8-pixel (chrominance) and a 16-pixel (luminance) line from a long data sequence.* Some potential values of these parameters are shown in Table 3.
Because of the data structures, imposed by MPEG-4, N_n^{P8}

Table 3. Values of N_n^{P8} and N_n^{P16}

ALU bits	Average		Worst Case	
	N_n^{P8}	N_n^{P16}	N_n^{P8}	N_n^{P16}
32-bit	2.5	5.5	3	7
64-bit	1	2.5	1	3
128-bit	0.5	1	0.5	1

is a variable for $n < 8$, and approximately constant for $n \geq 8$ (64-bit ALU). N_n^{P16} is a variable for $n < 16$, and a constant for $n \geq 16$ (128-bit ALU). Thus the processing of 16 pixels by any $n \cdot 8$ ALU will take $\frac{N_n^{P16}}{F_n}$ [seconds] and for a 256-pixel luminance block- $\frac{16 \cdot N_n^{P16}}{F_n}$ [s]. Identically, the processing of two 8x8-pixel chrominance blocks will take $\frac{16 \cdot N_n^{P8}}{F_n}$ [s] in the same ALU configuration. Since a macroblock consists of 256 luminance and 128 (2 x 64) chrominance pixels, padded vertically and horizontally, a whole macroblock will be padded for $\frac{32}{F_n} \cdot (N_n^{P8} + N_n^{P16})$ [s]. We can formulate the *Processing speed* of the ALU as follows:

$$Processing\_speed \approx \frac{F_n}{32 \cdot (N_n^{P8} + N_n^{P16})} \ [MB/s] \quad (3)$$

Assuming a realistic value of $F_n = 1GHz$ and using the data from Table 3 into Equation 3, we calculated the processing speed results, given in Table 4.

Table 4. Processing Speed at $F_n = 1GHz$

ALU bits	n	Speed [MB/s]	
		Average	Worst Case
32-bit	4	3 906 250	3 125 000
64-bit	8	8 928 600	7 812 500
128-bit	16	20 833 300	20 833 300

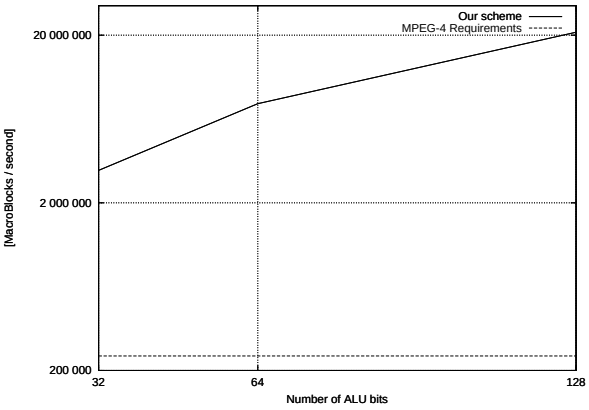


Figure 4. Processing Speed for Different ALU Operand Sizes and $F_n = 1GHz$ (note the logarithmic scale)

The most demanding profile level, *level 4* of the *Main* MPEG-4 profile, requires 244 800 Boundary MB/s (maximum 489600 MB/s) for a high resolution session type (1920 x 1088) and 32 objects (Table 1). These rates are an order of magnitude lower than the reported speed results for the feasible padding augmented ALU implementations (see Figure 4). The potentials of the structure indicate capabilities to meet even more-demanding future profiles of the visual data compression standards. Furthermore, a software implementation of the padding algorithm on a padding-augmented ALU would cost a neglectable number of MIPS (Millions of RISC-like Instructions per Second), compared to the MIPS numbers, discussed in the Section 2. Table 5 contains the estimated computational load of different padding augmented ALU sizes for Level4 of the Main MPEG-4 profile.

[1] In this paper we distinguish (data) processing speed, measured in [macroblocks/sec] (or [MB/s]) from the device operating speed (frequency), measured in [Hz].

ISBN: 0-9669530-0-2

Table 5. Computational load for Main Profile - Level 4 at $F_n = 1GHz$

ALU bits	n	[MIPS]	
		Average	Worst Case
32-bit	4	63	78
64-bit	8	27	31
128-bit	16	12	11

Hardware estimations. We choose the 2x2 AND-OR logic block as a basis for the hardware estimations. A one-bit 2 to 1 multiplexor is a 2x2 AND-OR gate. The hardware penalty for a single byte padding structure is: 2 x 9-bit multiplexers, 2 x 8-bit multiplexers and 1 OR gate. That makes 2x9+2x8+1=35 2x2 AND-OR gates. An n-byte implementation will cost $n \cdot 35$ AND-OR gates plus additional cost for the ALU multiplexer of $n \cdot 8$ gates, i.e. $n \cdot 43$ 2x2 AND-OR gates. Table 6 contains the exact values of the hardware penalties for different ALU sizes.

Table 6. Hardware estimation

ALU bits	n	Number of extra gates
32-bit	4	172
64-bit	8	344
128-bit	16	688

5. Conclusions

In this paper we introduced a scheme for general purpose ALU augmentation, which accelerates the MPEG-4 padding algorithm with orders of magnitude. We proposed a pipelined implementation of the idea, thus preserving the original functionality and timing scheme of the target ALU. At a trivial hardware cost of only a few hundred elementary 2x2 AND-OR logical gates, we could easily achieve a real-time performance at the most-demanding profile levels of MPEG-4. We proved that the proposed design is scalable by applying it on ALUs with different operand widths. The approach is general and can fit into different architectural concepts.

6. Acknowledgements

This research is supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs, the Technology Foundation STW (project AES.5021) and PHILIPS Research Laboratories, Eindhoven, The Netherlands.

References

[1] M. Berekovic, H.-J. Stolberg, M. B. Kulaczewski, P. Pirsh, H. Moler, H. Runge, J. Kneip, and B. Stabernack. Instruction set extensions for mpeg-4 video. *Journal of VLSI Signal Processing*, 23(1):27–49, October 1999.

[2] H.-C. Chang, L.-G. Chen, M.-Y. Hsu, and Y.-C. Chang. Performance analysis and architecture evaluation of MPEG-4 video codec system. In *IEEE International Symposium on Circuits and Systems*, volume II, pages 449–452, Geneva, Switzerland, 28-31 May 2000.

[3] H.-C. Chang, Y.-C. Wang, M.-Y. Hsu, and L.-G. Chen. Efficient algorithms and architectures for MPEG-4 object-based video coding. In *IEEE Workshop on Signal Processing Systems*, pages 13–22, 11-13 Oct 2000.

[4] E. A. Edirisinghe, J. Jiang, and C. Grecos. Shape adaptive padding for MPEG-4. *IEEE Transactions on Consumer Electronics*, 46(3):514–520, August 2000.

[5] ISO/IEC JTC11/SC29/WG11, N3312. MPEG-4 video verification model version 16.0.

[6] ISO/IEC JTC11/SC29/WG11 N4030. MPEG-4 overview, March 2001.

[7] A. Kaup. Object-based texture coding of moving video in MPEG-4. *IEEE Transactions on Circuits and Systems for Video Technology*, 9(1):5–15, February 1999.

[8] J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reissmann. The MPEG-4 video coding standard - a VLSI point of view. In *IEEE Workshop on Signal Processing Systems,(SIPS98)*, pages 43–52, 8-10 Oct. 1998.

[9] P. Kuhn and W. Stechele. Complexity analysis of the emerging MPEG-4 standard as a basis for VLSI implementation. In *SPIE Visual Comunications and Image Processing (VCIP)*, volume 3309, pages 498–509, San Jose, Jan. 1998.

[10] G. Kuzmanov, S. Vassiliadis, and J. van Eijndhoven. A Padding Processor for MPEG-4. In *12th Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC2001)*, Veldhoven, The Netherlands, 29-30 October 2001.

[11] J.-H. Moon, J.-H. Kweon, and H.-K. Kim. Boundary block-merging (BBM) technique for efficient texture coding of arbitrarily shaped object. *IEEE Transactions on Circuits and Systems for Video Technology*, 9(1):35–43, February 1999.

[12] M. Putrino, S. Vassiliadis, and E. Schwarz. Parallel binary byte adder / subtracter. *International Journal of Electronics*, 65(2):139–153, February 1988.

[13] Y. Q. Shi and H. Sun. *Image and Video Compression for Multimedia Engineering*. Boca Raton CRC Press, 2000.

[14] H.-J. Stolberg, M. Berekovic, P. Pirsch, H. Runge, H. Moller, and J. Kneip. The M-PIRE MPEG-4 codec DSP and its macroblock engine. In *IEEE International Symposium on Circuits and Systems*, volume II, pages 192–195, Geneva, Switzerland, 28-31 May 2000.

[15] S. Vassiliadis, G. Kuzmanov, and S. Wong. MPEG-4 and the New Multimedia Architectural Challenges. In *15th International Conference SAER'2001*, St.Konstantin, Bulgaria, 21-23 Sept. 2001.

[16] S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN rm-coded processor. In *11th International Conference on Field Programmable Logic and Applications (FPL)*, Belfast, Northern Ireland, UK, August 2001.

 ISBN: 0-9669530-0-2