Performance Improvement of Multimedia Kernels by Alleviating Overhead Instructions on SIMD Devices

Asadollah Shahbahrami[1,2] and Ben Juurlink[1]

[1] Computer Engineering Laboratory,
Delft University of Technology, 2628 CD Delft, The Netherlands
{a.shahbahrami,b.h.h.juurlink}@tudelft.nl
[2] Department of Computer Engineering, Faculty of Engineering,
University of Guilan, Rasht, Iran

Abstract. SIMD extension is one of the most common and effective technique to exploit data-level parallelism in today's processor designs. However, the performance of SIMD architectures is limited by some constraints such as mismatch between the storage and the computational formats and using data permutation instructions during vectorization. In our previous work we have proposed two architectural modifications, the extended subwords and the Matrix Register File (MRF) to alleviate the limitations. The extended subwords, uses four extra bits for every byte in a media register and it provides additional parallelism. The MRF allows flexible row-wise as well as column-wise access to the register file and it eliminates data permutation instructions. We have validated the combination of the proposed techniques by studying the performance of some multimedia kernels. In this paper, we analysis each proposed technique separately. In other words, we answer the following questions in this paper. How much of the performance gain is a result of the additional parallelism? and how much is due to the elimination of data permutation instructions? The results show that employing the MRF and extended subwords separately obtains the speedup less than 1 and 1.15, respectively. In other words, our results indicate that using either extended subwords or the MRF techniques is insufficient to eliminate most pack/unpack and rearrangement overhead instructions on SIMD processors. The combination of both techniques, on the other hand, yields much more performance benefits than each technique.

1 Introduction

Multimedia extensions are one of the most common approach to exploit Data-Level Parallelism (DLP) in multimedia applications on General-Purpose Processors (GPPs). With this approach, multiple data items are packed into a wider

This research was supported in part by the Netherlands Organization for Scientific Research (NWO).

media register which can be processed using a Single Instruction and Multiple Data (SIMD) instruction. These extensions can improve the performance of several multimedia applications. Nevertheless, they have some limitations. First, there is a mismatch between the computational format and the storage format of multimedia data. Because of this many data type conversion instructions are used in SIMD implementations. Second, existing SIMD computational instructions cannot efficiently exploit DLP of the 2D and interleaved multimedia data. In order to vectorize 2D and interleaved multimedia data, many rearrangement instructions are needed.

In our previous work, two architectural enhancements, the Matrix Register File (MRF) and extended subwords techniques have been proposed to overcome the above limitations [16]. Extended subwords use registers that are wider than the packed format used to store the data. Extended subwords avoid data type conversion instructions. The MRF allows to load data stored consecutively in memory to a column of the register file, where a column corresponds to corresponding subwords of different registers. This technique avoids the need of data rearrangement instructions. The MMX multimedia extension [12] has been modified by the proposed techniques that was called the Modified MMX (MMMX) architecture. The MMMX architecture have been validated by studying the performance of several important multimedia kernels. Our results show that the performance benefits by employing both proposed techniques is higher than just using the extended subwords technique. In other words, those multimedia kernels which employ the MRF and extended subwrods techniques obtain more speedups than just using the extended subwords technique. However, we did not determine how much of the performance gain is a result of employing the extended subwords technique and how much is due to the employing the MRF. Our goal in this paper is to analysis each technique separately.
We make the following contributions compared to other works.

– We have applied the proposed techniques on a wide range of multimedia kernels.
– In order to determine the performance benefits of each proposed technique, we analysis each technique separately. In other words, we have enhanced the MMX architecture with extended subwords (MMX + ES) and with an MRF (MMX + MRF) separately.
– Our results indicate that using either extended subwords or the MRF techniques is insufficient to eliminate most pack/unpack and rearrangement overhead instructions. In addition, using the MRF is both unuseful and causes performance loss. The MMMX architecture that employs both proposed techniques, on the other hand, yields much more performance benefits.

This paper is organized as follows. In Section 2, we present background information related to the multimedia extensions and their performance bottlenecks. In Section 3, we describes the MMMX architecture that features the extended subwords and the MRF techniques. We discuss several multimedia kernels selected for performance evaluation in Section 4 followed by performance evaluation in Section 5. We analysis each proposed technique separately in Section 6. Finally, conclusions are given in Section 7.

2 Background

We present a short explanation of the multimedia extensions in this section.

2.1 GPPs Enhanced with Multimedia Extension

In order to increase the performance of multimedia applications, GPPs vendors have extended their ISAs. These ISA extensions use the Subword Level Parallelism (SLP) concept [10]. A subword is a smaller precision unit of data contained within a word. In SLP, multiple subwords are packed into a word and then whole word is processed. SLP is used in order to exploit DLP with existing hardware without sacrificing the general-purpose nature of the processor. In SLP, a register is viewed as a small vector with elements that are smaller than the register size. This requires small data types and wide registers.

The first multimedia extensions are Intel's MMX [12], Sun's Visual Instruction Set (VIS) [17], Compaq's Motion Video Instructions (MVI) [3], MIPS Digital Media eXtension (MDMX) [8], and HP's Multimedia Acceleration eXtension (MAX) [10]. These extensions supported only integer data types and were introduced in the mid-1990's. 3DNow [1] was the first to support floating-point media instructions. It was followed by Streaming SIMD Extension (SSE) and SSE2 from Intel [13]. Motorola's AltiVec [4] supports integer as well as floating-point media instructions. In addition, high-performance processors also use SIMD processing. An excellent example of this is the Cell processor [7] developed by a partnership of IBM, Sony, and Toshiba. Cell is a heterogeneous chip multiprocessor consisting of a PowerPC core that controls eight high-performance Synergistic Processing Elements (SPEs). Each SPE has one SIMD computation unit that is referred to as Synergistic Processor Unit (SPU). Each SPU has 128 128-bit registers. SPUs support both integer and floating-point SIMD instructions. Table 1 summarizes the common and distinguishing features of existing multimedia instruction set extensions [15].

2.2 Performance Bottlenecks

SIMD architectures generally provide two kinds of SIMD instructions. The first are the SIMD computational instructions such as arithmetic instructions. The second are the SIMD overhead instructions that are necessary for data movement, data type conversions, and data reorganization. The latter instructions are needed to bring data in a form amenable to SIMD processing. These instructions constitute a large part of the SIMD codes. For example, Ranghanathan et al. [14] indicated that the SIMD implementations of the MPEG/JPEG codecs using the VIS ISA require on average 41% overhead instructions such as packing/unpacking and data re-shuffling. In addition, the dynamic instructions count

Table 1. Summary of available multimedia extensions. *Sn* and *Un* indicate *n*-bit signed and unsigned integer packed elements, respectively. Values *n* without a prefix *U* or *S* in the last row, indicate operations work for both signed and unsigned values. [1] Note that 68 instructions of the 144 SSE2 instructions operate on 128-bit packed integer in XMM registers, wide versions of 64-bit MMX/SSE integer instructions.

GPP with Multimedia Extension ISA Name	**AltiVec/VMX**	**MAX-1/2**	**MDMX**	**MMX/ 3DNow**	**VIS**	**MMX/ SIMD**	**SSE**	**SSE2**	**SPU ISA**
Company	Motorola/IBM	HP	MIPS	AMD	Sun	Intel	Intel	Intel	IBM/Sony/Toshiba
Instruction set	Power PC	PARISC2	MIPS-V	IA32	P. V.9	IA32	IA64	IA64	-
Processor	MPC7400	PA RISC	R1000 PA8000	K6-2	Ultra Sparc	P2	P3	P4	Cell
Year	1999/2002	1995	1997	1999	1995	1997	1999	2000	2005
Datapath width	128-bit	64-bit	64-bit	64-bit	64-bit	64-bit	128-bit	128-bit	128-bit
Size of register file	32x128b	(31) /32x64b	32x64b	8x64b	32x64b	8x64b	8x128b	8x128b	128x128b
Dedicated or shared with	Dedicated	Int. Reg.	FP Reg.	Dedicated	FP Reg.	FP Reg.	Dedicated	Dedicated	Dedicated
Integer data types:									
8-bit	16	-	8	8	8	8	8	16	16
16-bit	8	4	4	4	4	4	4	8	8
32-bit	4	-	-	2	2	2	2	4	4
64-bit	-	-	-	-	-	-	-	2	2
Shift right/left	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multiply-add	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Shift-add	No	Yes	No	No	No	No	No	No	No
Floating-point	Yes	No	Yes	Yes	No	No	Yes	Yes	Yes
Single-precision	4x32	-	2x32	4x16 2x32	-	-	4x32	4x32	4x32
Double-precision	-	-	-	1x64	-	-	-	2x64	2x64
Accumulator	No	No	1x192b	No	No	No	No	No	
# of instructions	162	(9) 8	74	24	121	57	70	144[1]	213
# of operands	3	3	3-4	2	3	2	2	2	2/3/4
Sum of absolute-differences	No	No	No	Yes	Yes	No	Yes	Yes	Yes
Modulo addition/ subtraction	8, 16, 32	16	8, 16	8, 16 32	16, 32	8, 16 32, 64	8, 16 32, 64	8, 16 32,64	8, 16 32,64
Saturation addition/ subtraction	U8, U16, U32 S8, S16, S32	U16, S16	S16	U8, U16 S8, S16	No	U8, U16 S8, S16	U8, U16 S8, S16	U8, U16 S8, S16	- -

of the EEMBC consumer benchmarks running on the Philips TriMedia TM32 shows that over 23% of instructions are data alignment instructions such as pack/merge bytes (16.8%) and pack/merge half words (6.5%) [5]. The execution of this large number of the SIMD overhead instructions decreases the performance and increases pressure on the fetch and decode steps.

To illustrate where overhead instructions are needed in the SIMD implementations of multimedia kernels, we explain it in more detail. Data reordering and data type conversion instructions are used after loading the input data and before storing the outputs. For example, in case of the RGB-to-YCbCr color space conversion, 35 and 6 instructions are needed in each loop iteration to convert 8 pixels from the band interleaved format to the band separated format and unpack the packed byte data types to packed 16-bit word data types, respectively. In addition, 12 instructions are needed to pack the unpacked results and store in memory. On the other hand, the number of SIMD computational instructions is 78. This means that the number of overhead instructions is significant compared to the number of SIMD computational instructions. As another example, matrix transposition is a very common operation in multimedia applications. 2D multimedia algorithms such as the 2D Discrete Cosine Transform (DCT) consists of two 1D transforms called horizontal and vertical transforms. The horizontal transform processes the rows while vertical transform processes the columns. SIMD vectorization of the vertical transform is straightforward, since the corresponding data of each column are adjacent in memory. Therefore, several columns can be processed without any rearranging of the subwords. For horizontal transform on the other hand, corresponding elements of adjacent rows are not continuous in memory (in a row-major storage format). In order to employ SIMD instructions, data rearrangement instructions are needed to transpose the matrix. This step takes a significant amount of time. For example, transposing an 8×8 block of bytes, requires 56 MMX/SSE instructions, if the elements are two bytes wide, then 88 instructions are required. Consequently, it is important either to eliminate, to alleviate, or to overlap these instructions with other SIMD computational instructions.

3 MMMX Architecture

The MMMX architecture is MMX enhanced with extended subwords, the MRF, and a few general-purpose SIMD instructions that are not present in the MMX and SSE extensions. The employed techniques in the MMMX architecture are discussed briefly in the following section. More detail about this architecture can be found in [15].

3.1 Extended Subwords

Image and video data is typically stored as packed 8-bit elements, but intermediate results usually require more than 8-bit precision. As a consequence, most 8-bit SIMD ALU instructions are wasted. In the SIMD extensions, the choice is either to be imprecise by using saturation operations at every stage, or to loose parallelism by unpacking to a larger format. Using saturation instructions produces inaccurate results. This is because saturation is usually used at the end of computation. It is more precise to saturate once at the end of the computation rather than at every step of the algorithm. For instance, adding three signed 8-bit values $120 + 48 - 10$, using signed saturation at every step produces 117 and using signed saturation at the last step produces 127.

SIMD architectures support different packing, unpacking, and extending instructions to convert the different data types to each other. For example, the MMX/SSE architectures provide `packss`{`wb,dw,wb`} and `punpck` {`hbw,hwd,hdq, lbw,lwd,ldq`} instructions for data type conversions.

To avoid the data type conversion overhead and to increase parallelism, extended subwords are employed. This means that the registers are wider than the data loaded into them. Specifically, for every byte of data, there are four extra bits. This implies that MMMX registers are 96 bits wide, while MMX has 64-bit registers. Based on that, the MMMX registers can hold 2×48-bit, 4×24-bit, or 8×12-bit elements.

3.2 The Matrix Register File

The ability to efficiently rearrange subwords within and between registers is crucial to performance. To overcome this problem, a matrix register file is employed, which allows data loaded from memory to be written to a column of the register file as well as to a row register. In the MMMX architecture, the MRF provides parallel access to 12-, 24-, and 48-bit subwords of the row registers that are horizontally located. This is similar to conventional SIMD architectures, which provide parallel access to 8-, 16-, and 32-bit data elements of media registers. In addition, the MRF provides parallel access to 12-bit subwords of the column registers that are vertically arranged.

Figure 1(a) depicts a block diagram of a register file with one write port (Port C) and two read ports (Port A and Port B). The input and output of this block diagram is based on eight 96-bit registers. Figure 1(b) illustrates the combination

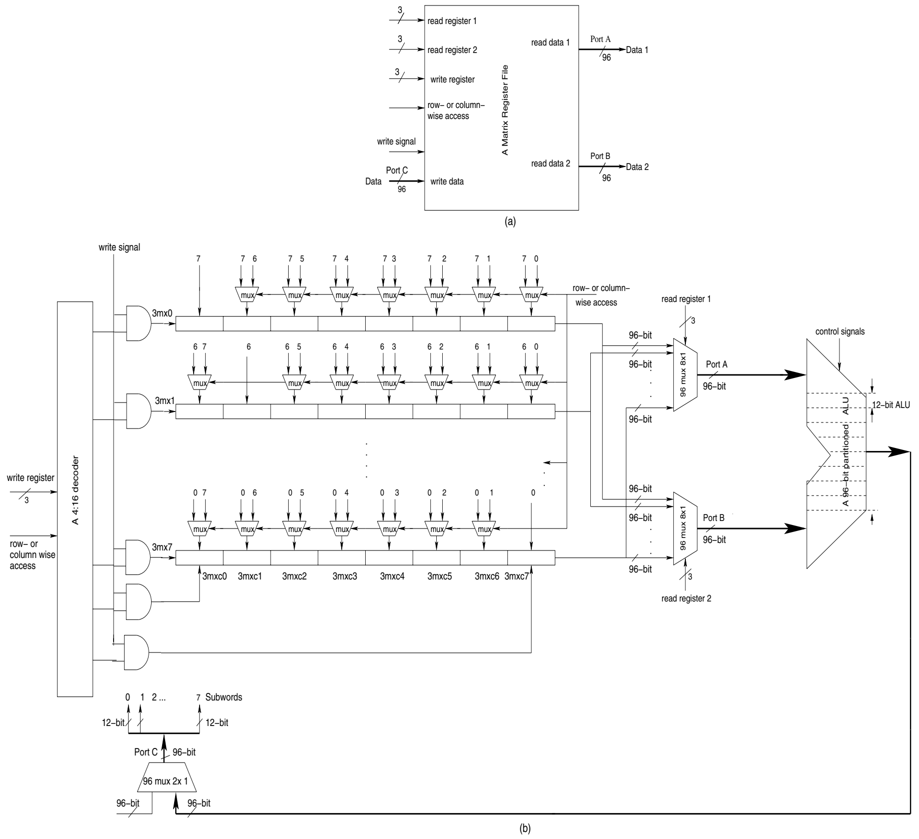


Fig. 1. (a) A register file with eight 96-bit registers, 2 read ports, and 1 write port, (b) the implementation of two read ports and one write port for a matrix register file with 8 96-bit registers as well as a partitioned ALU for subword parallel processing

of the MRF with a 96-bit partitioned ALU for the MMMX architecture. The partitioned ALUs have been designed based on the subword adder. Multiplexers have been used in subword boundaries to propagate or prevent the subword carries in the carry chain [6]. There are eight 12-bit adders. These adders operate independently for 12-bit data. They can also be coupled to behave an four pairs of two adders to perform four 24-bit operations, or combined into two groups of four adders for two 48-bit format.

3.3 MMMX Instruction Set Architecture

The MMMX architecture has different load/store, ALU, and multiplication instructions, which some of them are discussed in the remainder of this section.

The `fld8u12` instruction loads eight unsigned bytes from memory and zero-extends them to a 12-bit format in a 96-bit MMMX register. The `fld8s12` instruction, on the other hand, loads eight signed bytes and sign-extends them to a 12-bit format. These instructions are illustrated in Figure 2 for little endian. The `fld16s12` instruction loads eight signed 16-bit, packs them to signed 12-bit

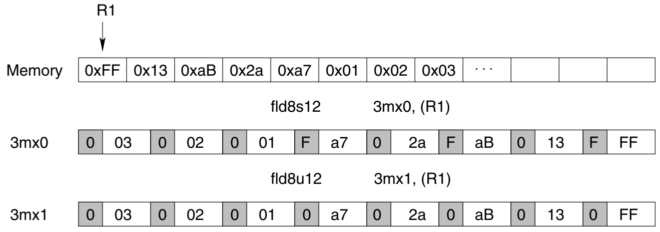


Fig. 2. The `fld8s12` instruction loads eight signed bytes and sign-extends them to 12-bit values, while the `fld8u12` instruction loads eight unsigned bytes and zero-extends them to 12-bit values

format, and writes in a row register. This instruction is useful for those kernels that their input data can be represented by the signed 12-bit, while they use the signed 16-bit storage format. For example, in the DCT kernel, the input data is the signed 9-bit format. It uses the signed 16-bit storage format, while it uses the signed 12-bit for computational format. The instruction `fldc8u12` ("load-column 8-bit to 12-bit unsigned") is used to load a column of the MRF.

Load instructions automatically unpack and store instructions automatically pack and saturate, as illustrated for the load instructions in Figure 2. Store instructions automatically saturate (clip) and pack the subwords. For example, the instruction `fst12s8` saturates the 12-bit signed subwords to 8-bit unsigned subwords before storing them to memory.

Most MMMX ALU instructions are direct counterparts of MMX/SSE instructions. For example, the MMMX instructions `fadd{12,24,48}` (packed addition of 12-, 24-, 48-bit subwords) and `fsub{12,24,48}` (packed subtraction of 12-, 24-, 48-bit subwords) correspond to the MMX instructions `padd{b,w,d}` mm,mm/mem64 and psub{b,w,d} mm,mm/mem64, respectively. MMMX, however, does not support variants of these instructions that automatically saturate the results of the additions to the maximum value representable by the subword data type. They are not needed because as was mentioned the load instructions automatically unpack the subwords and the store instructions automatically pack and saturate. In other words, the MMMX architecture does not support saturation arithmetic.

In several media kernels all elements packed in a register need to be summed, while in other kernels adjacent elements need to be added. Rather than providing different instructions for summing all elements and adding adjacent elements, it has been decided to support adding adjacent elements only but for every packed data type. Whereas summing all elements would probably translate to a multicycle operation, adding adjacent elements is a very simple operation that can most likely be implemented in a single cycle.

Another operation that has been found useful in implementing of several multimedia kernels such as the (I)DCT kernels is the possibility to negate some or all elements in a packed register. The instructions fneg{12,24,48} 3mx0, 3mx1, imm8 negate the 12-, 24-, or 48-bit subwords of the source operand if the corresponding bit in the 8-bit immediate imm8 is set. If subwords are 24- or 48-bit, the four or six higher order bits in the 8-bit immediate are ignored.

The MMMX architecture supports three kinds of multiplication instructions. The first are full multiplication instructions fmulf{12,24}. For example, the fmulf12 instruction multiplies each 12-bit subword in 3mx0 with the corresponding subwords in 3mx1 and produces eight 24-bit results. This means that each result is larger than a subword. Therefore, the produced results are kept in both registers. The second kind of multiplication instructions are the partitioned multiply-accumulate instructions fmadd{12,24}. These instructions perform the operation on subwords that are either 12- or 24-bit, while the MMX instruction pmaddwd performs the MAC operation on subwords that are 16-bit. The MAC operation is an important operation in digital signal processing. This instruction multiplies the eight signed 12-bit values of the destination operand by the eight 12-bit values of the source operand. The corresponding odd-numbered and even-numbered subwords are summed and stored in the 24-bit subwords of the destination operand.

The third type of multiplication is truncation. Truncation is performed by the fmul{12l,12h,24l,24h} instructions. It means that the high or low bits of the results are discarded. When n-bit fixed point values are multiplied with fractional components, the result should be n-bit of precision. Specifically, the instructions fmul12{l,h} multiply the eight corresponding subwords of the source and destination operands and write the low-order (fmul12l) or high-order (fmul12h) 12 bits of the 24-bit product to the destination operand.

4 Multimedia Kernels

Most of the execution time of multimedia applications is spent in multimedia kernels. Therefore, in order to evaluate the proposed techniques, some time

Table 2. Summary of multimedia kernels

Multimedia Kernels	**Description**
Matrix transpose	Matrix transposition is an important kernel for many 2D media kernels.
Vector/Matrix Multiply	Vector/matrix multiply kernel is used in some multimedia standards.
Repetitive Padding	In this kernel, the pixel values at the boundary of the video object is replicated horizontally as well as vertically.
RGB-to-YCbCr	Color space conversion, which is usually used in the encoder stage.
Horizontal DCT	Horizontal DCT in used in most media standards to process the rows of images in order to remove spatial redundancy.
Horizontal IDCT	Horizontal Inverse DCT is used in the multimedia standards in order to reconstruct the rows of the transformed images.
Vertical DCT	Vertical DCT in used in most media standards to process the columns of images in order to remove spatial redundancy.
Vertical IDCT	Vertical IDCT is used in the multimedia standards in order to reconstruct the columns of the transformed images.
Add block	The add block is used in the decoder, during the block reconstruction stage of motion compensation.
2×2 Haar transform	The 2×2 haar transform is used to decompose an image into four different bands.
Inverse 2×2 Haar transform	The inverse 2×2 haar transform is used to reconstruct the original image from different bands.
Paeth prediction	Paeth prediction is used in the PNG standard.
YCbCr-to-RGB	Color space conversion, which is usually used in the decoder stage.
SAD function	The SAD function, which is used in motion estimation kernel to remove temporal redundancies between video frames.
SAD function with interpolation	The SAD function with horizontal and vertical interpolation is used in motion estimation algorithm.
SAD function for image histograms	The SAD function is used for similarity measurements of image histograms.
SSD function	The SSD function, which is used in motion estimation kernel to remove temporal redundancies between video frames.
SSD function with interpolation	The SSD function with horizontal and vertical interpolation is used in motion estimation algorithm.

consuming kernels of multimedia standards have been considered. Table 2 lists the media kernels along with a small description. In order to clarify which proposed techniques have been used in SIMD implementations of media kernels, the presented kernels are divided into two groups. First, kernels that use both extended subwords and the MRF techniques, for instance, the first six kernels. Second, kernels that just use extended subwords technique, for example, the rest of the kernels (twelve kernels).

As was mentioned, the 2D transforms such as (I)DCT are decomposed into two 1D transforms called horizontal and vertical transforms. In order to increase DLP in SIMD implementation of vertical transform, the extended subwords technique is used, while in SIMD implementation of horizontal transform both proposed techniques are needed in order to increase DLP and also to avoid data rearrangement instructions.

5 Performance Evaluation

In this section we evaluate the proposed techniques by comparing the performance of the SIMD implementations that employ the SIMD architectural enhancements to the performance of the MMX and SSE implementations on a single issue processor.

5.1 Evaluation Environment

In order to evaluate the SIMD architectural enhancements, we have used the `sim-outorder` simulator of the SimpleScalar toolset [2]. We have synthesized MMX/SSE and MMMX instructions using the 16-bit annotate field, which is available in the instruction format of the PISA ISA. More detail about our extension to the SimpleScalar toolset can be found in [9].

The main objective is to compare the performance of the MMX and SSE extensions without the proposed techniques to the those extensions with the SIMD architectural enhancements. The main parameters of the modeled processors are depicted in Table 3. The latency and throughput of SIMD instructions are set equal to the latency and throughput of the corresponding scalar instructions. This is a conservative assumption given that the SIMD instructions perform the same operation but on narrower data types. The latency and throughput of SIMD multiplier units are set to 3 and 1 respectively, the same as in the Pentium 3 processor. The latency of SIMD multiplier units in the Pentium 4 processor is 8 cycles.

Three programs have been implemented by C and assembly languages and simulated using the SimpleScalar simulator for each kernel. Each program consists of three parts. One part is for reading the image, the second part is the computational kernel, and the last part is for storing the transformed image. One program was completely written in C. It was compiled using the gcc compiler targeted to the SimpleScalar PISA with optimization level *-O2*. The reading and storing parts of the other two programs were also written in C, but the second part was implemented by hand using MMX/SSE and MMMX. These programs will be referred to as C, MMX, and MMMX for each kernel. All C, MMX, and MMMX codes use the same algorithms. In addition, the correctness of the MMX and MMMX codes were validated by comparing their output to the output of C programs.

The speedup was measured by the ratio of the total number of cycles for the computational part of each kernel for the MMX implementation to the MMMX implementation. In order to explain the speedup, the ratio of dynamic number of instructions has also been obtained. These metrics formed the basis of the comparative study. Ratio of dynamic number of instructions means the ratio of the number of committed instructions for the MMX implementation to the number of committed instructions for the MMMX implementation.

5.2 Performance Evaluation Results

Figure 3 and Figure 4 depict the speedup of MMMX over MMX for media kernels that either use extended subwords technique or use both proposed techniques, respectively. The results have been obtained for one execution of media kernels on a single block on the single issue processor. In addition, these figures show

Table 3. Processor configuration

Parameter	Value
Issue width	1
Integer ALU, SIMD ALU	1
Integer MULT, SIMD MULT	1
L1 Instruction cache	512 set, direct-mapped 64-byte line LRU, 1-cycle hit, total of 32 KB
L1 Data cache	128 set, 4-way, 64-byte line, 1-cycle hit, total of 32 KB
L2 Unified cache	1024 set, 4-way, 64-byte line, 6-cycle hit, total of 256 KB
Main memory latency	18 cycles for first chunk, 2 thereafter
Memory bus width	16 bytes
RUU (register update unit) entries	64
Load-store queue size	8
Execution	out-of-order

the ratio of committed instructions (MMX implementation over MMMX). Both figures show that MMMX performs better than MMX for all kernels except SAD. The speedup in Figure 3 ranges from 0.74 for the SAD kernel to 2.66 for Paeth kernel. MMMX yields a speedup ranging from 1.10 for the 2D IDCT kernel to 4.47 for the Transp.(12) kernel in Figure 4. The most important reason why MMMX improves performance is that it needs to execute fewer instructions than MMX. In the SAD kernel, on the other hand, MMMX needs to execute more instructions than MMX. As Figure 3 shows, the ratio of committed instructions for the SAD kernel is 0.72.

An Special-Purpose `psadbw` Instruction (SPI) [13] has been used in the MMX implementation of the SAD function and the SAD function with interpolation, while in the MMMX implementation this SPI has been synthesized by a few general-purpose SIMD instructions. Both MMX and MMMX employ 8-way parallelism in the SAD function, while MMMX uses more instructions than MMX. MMX employs both 4- and 8-way parallelism in the SAD function with interpolation, which means that it uses many data type conversion instructions. On the contrary, MMMX always employs 8-way parallelism in the SAD function with interpolation kernel. This is the reason that the speedup is almost two for this kernel.

The speedup obtained for the Paeth kernel in Figure 3 is 2.66. The reason is that intermediate data is at most 10 bits wide and MMMX can, therefore, calculate the prediction for eight pixels in each loop iteration while MMX computes the prediction for four pixels. The speedups of MMMX over MMX for the vertical IDCT and 2D IDCT kernels in those figures is less than the speedups for other kernels. This is because the input data of these kernels is 12-bit and some intermediate results are larger than 12-bit. Therefore, the MMMX implementation cannot employ 12-bit functionality (8-way parallel SIMD instructions) all

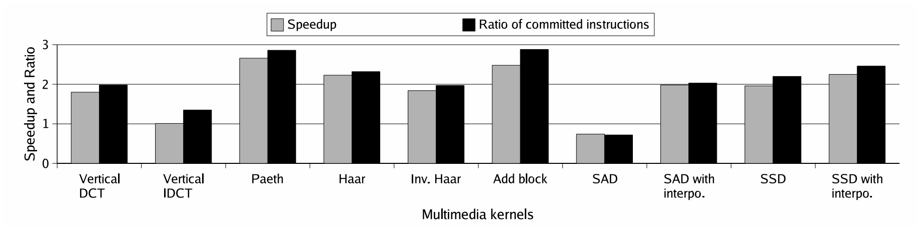


Fig. 3. Speedup of MMMX over MMX as well as the ratio of committed instructions (MMX over MMMX) for multimedia kernels, which use extended subwords technique on a single block on the single issue processor

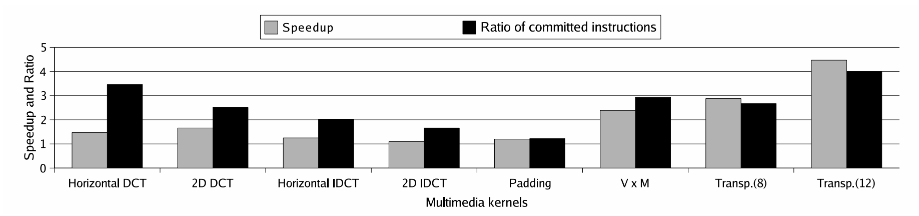


Fig. 4. Speedup of MMMX over MMX as well as the ratio of committed instructions (MMX over MMMX) for multimedia kernels, which use both proposed techniques on a single block on the single issue processor

the time but sometimes has to convert to 4 × 24-bit packed data types. The MMX implementation, on the other hand, is able to use 16-bit functionality all the time.

The reason why MMMX improves performance by just 20% for the Padding kernel in Figure 4 is that the MMX implementation employs the special-purpose `pavgb` instruction which computes the arithmetic average of eight pairs of bytes. More precisely, the `pavgb` instruction is supported in the SSE integer extension to MMX. MMMX does not support this instruction because with extended subwords it offers little extra functionality since it can be synthesized using the more general-purpose instructions `fadd12` and `fsar12` (shift arithmetic right on extended subwords). Nevertheless, because the matrix needs to be transposed between horizontal and vertical padding MMMX provides a speedup.

The two kernels for which the highest speedups are obtained are the 8 × 8 matrix transpose on 8-bit (Transp.(8)) and 12-bit data (Transp.(12)). If the matrix elements are 8-bit, MMMX can use the MRF to transpose the matrix, while MMX requires many pack and unpack instructions to realize a matrix transposition. Furthermore, if the elements are 12-bit (but stored as 16-bit data types), MMMX is able to employ 8-way parallel SIMD instructions, while MMX can only employ 4-way parallel instructions. As a result, MMMX improves performance by more than a factor of 4.47.

The average speedup and ratio of committed instructions for kernels that only use the extended subwords technique are 1.90 and 2.08, respectively, while for the kernels that use both proposed techniques are 2.05 and 2.56. The reduction of the dynamic instruction count in Figure 3 is due to extended subwords and in Figure 4 it is due to extended subwords and the MRF techniques. As a result, the performance benefits obtained by employing both techniques is higher than just using the extended subwords technique. Consequently, a part of the performance benefits is due to extended subwords, which increases DLP and the other part of the performance improvement is due to the MRF that eliminates the data rearrangement instructions. In order to clarify how much of the performance gain is a result of the additional parallelism provided by extended subwords and how much of it is due to the MRF, we disccuss some examples, horizontal transform of DCT and 2D DCTin the following section.

6 Analysis of Each Proposed Technique Separately

As already indicated in Table 2, in the SIMD implementations of some kernels such as the horizontal DCT both proposed techniques have been employed. Consequently, a part of the performance benefits is due to extended subwords, which increases DLP and the other part of the performance improvement is due to the MRF that eliminates the data rearrangement instructions. This section discusses an example, horizontal DCT in order to clarify how much of the performance gain is a result of the additional parallelism provided by extended subwords and how much of it is due to the MRF.

6.1 LLM Algorithm to Implement Discrete Cosine Transform

The discrete cosine transform and its inverse are widely used in several image and video compression applications. JPEG and MPEG partition the input image into 8×8 blocks and perform a 2D DCT on each block. The input elements are often either 8- or 9-bit, and the output is an 8×8 block of 12-bit 2's complement data. In this section, we discuss the LLM [11] technique to implement the DCT.

One of the fastest algorithm to compute the 2D DCT is LLM [11] technique. This algorithm performs a 1D DCT on each row of the 8×8 block followed by a 1D DCT on each column of the transformed 8×8 block. The algorithm has four stages, the output of each stage is the input of next stage. Figure 5 depicts the data flow graph of this algorithm for 8 pixels using fixed-point arithmetic.

Four SIMD implementations of the DCT namely, MMX, MMMX, MMX enhanced with extended subwords, and MMX enhanced with the MRF using LLM algorithm are explained and then their performance evaluation are presented.

6.2 Four Different SIMD Implementations for Horizontal DCT

In this section different SIMD implementations are discussed.

MMX Implementation: In the MMX implementation of this algorithm, however, 16-bit functionality (4-way parallelism) has been used because the input data is either 8- or 9-bit. This means that this kernel needs 16-bit storage format, while the intermediate results are smaller than 16-bit. Data type conversion instructions are not needed because four 16-bit can be loaded from memory to the four subwords of a media register. Although many rearrangement instructions are used in this implementation, this implementation exploits 4-way parallelism in all stages. Figure 6 depicts the MMX/SSE implementation of the first stage of the LLM algorithm for horizontal DCT. As this figure shows some rearrangement instructions are required in this implementation.

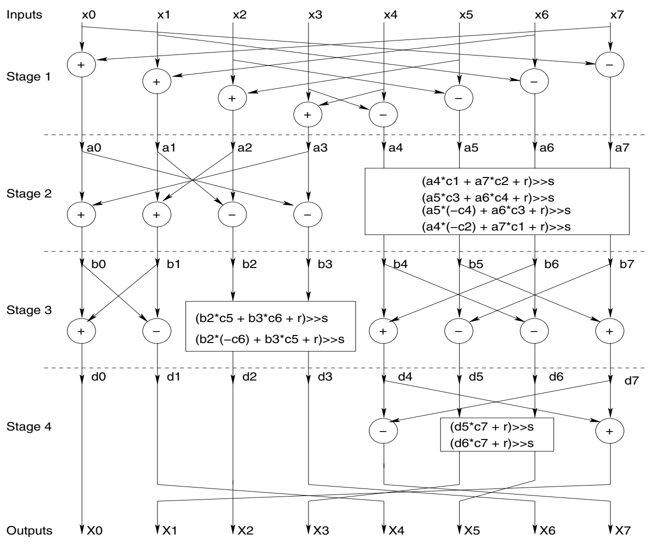


Fig. 5. Data flow graph of 8 pixels DCT using LLM [11] algorithm. The constant coefficients of c, r, and s are provided for fixed-point implementation.

`movq   mm0, (dct)`	;mm0 =	x03	x02	x01	x00
`movq   mm3, 8(dct)`	;mm3 =	x07	x06	x05	x04
`pshufw mm1, mm0,27`	;mm1 =	x00	x01	x02	x03
`pshufw mm2, mm3,27`	;mm2 =	x04	x05	x06	x07
`paddsw mm0, mm2`	;mm0 =	x03+x04	x02+x05	x01+x06	x00+x07
`psubsw mm1, mm3`	;mm1 =	x00-x07	x01-x06	x02-x05	x03-x04

Fig. 6. The MMX/SSE code of the first stage of the LLM algorithm for horizontal DCT

`fldc16s12 3mxc0, (dct)`	; 3mxc0 =	x07	x06	x05	x04	x03	x02	x01	x00
`fldc16s12 3mxc1, 16(dct)`	; 3mxc1 =	x17	x16	x15	x14	x13	x12	x11	x10
`fldc16s12 3mxc2, 32(dct)`	; 3mxc2 =	x27	x26	x25	x24	x23	x22	x21	x20
`fldc16s12 3mxc3, 48(dct)`	; 3mxc3 =	x37	x36	x35	x34	x33	x32	x31	x30
`fldc16s12 3mxc4, 64(dct)`	; 3mxc4 =	x47	x46	x45	x44	x43	x42	x41	x40
`fldc16s12 3mxc5, 80(dct)`	; 3mxc5 =	x57	x56	x55	x54	x53	x52	x51	x50
`fldc16s12 3mxc6, 96(dct)`	; 3mxc6 =	x67	x66	x65	x64	x63	x62	x61	x60
`fldc16s12 3mxc7, 112(dct)`	; 3mxc7 =	x77	x76	x75	x74	x73	x72	x71	x70
`fst12s16s 112(dct), 3mx7`	; (mem) =	x77	x67	x57	x47	x37	x27	x17	x07
`fmov 3mx7 , 3mx0`	; 3mx7 =	x70	x60	x50	x40	x30	x20	x10	x00
`fadd12 3mx0 , 112(dct)`	; 3mx0 =	X	X	X	X	X	X	X	x00+x07
`fsub12 3mx7 , 112(dct)`	; 3mx7 =	X	X	X	X	X	X	X	x00-x07

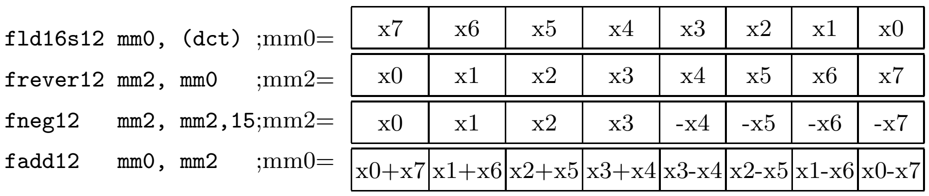
Fig. 7. A part of the MMMX implementation for the horizontal DCT algorithm. “X” denotes to $xi0 \pm xi7$, where $0 \leq i \leq 7$.

MMMX Implementation: MMMX processes eight rows in one iteration. A complete 8×8 block is loaded into eight column registers. After that row registers which have eight subwords are processed. Figure 7 depicts a part of the MMMX implementation of the LLM algorithm. In this figure, “X” refers to $xi0 \pm xi7$, where $0 \leq i \leq 7$. First, eight load column instructions are used to load a complete 8×8 block into column registers. After that two `fadd12` and `fsub12` instructions are needed to process 16 pixels simultaneously. In MMX, on the other hand, four instructions (two `pshufw` instructions, a `paddsw`, and a `psubsw` instructions) are required to process eight pixels.

MMX Enhanced with Extended Subwords: In MMX enhanced with extended subwords (MMX + ES), there are eight 12-bit subwords in each media register. In order to bring these subwords in a form amenable to SIMD processing, new data permutation instructions such as `fshuflh12`, `fshufhl12`, `fshufhh12`, `fshufll12`, and `frever12` are needed. This is because of the following reasons. First, there is no shuffle instructions in MMX. MMX performs data permutation using pack and unpack instructions, while these instructions are not useful for MMX + ES. Second, there is a `pshufw` (packed shuffle word) instruction in SSE that is used for rearrangement of four subwords within a media register, while MMX + ES has eight subwords.

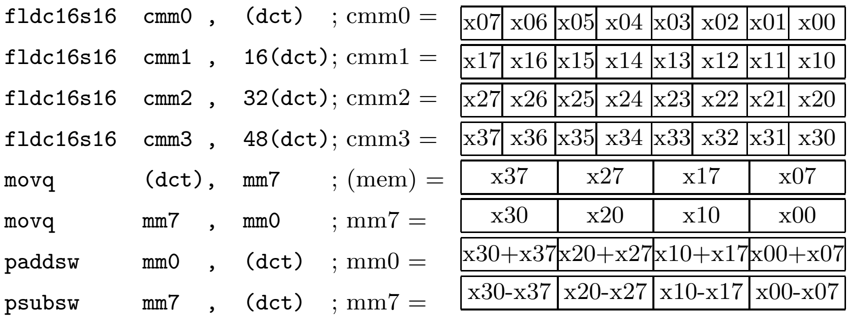
Figure 8 depicts a part of the horizontal DCT code that has been implemented by the MMX + ES. In each loop iteration of this implementation eight pixels are processed, the same as the MMX implementation that was already discussed.

MMX Enhanced with an MRF: In MMX enhanced with an MRF, there are four 128-bit column registers and eight 64-bit registers the same as MMX. Each



```
fld16s12 mm0, (dct) ;mm0=  | x7    | x6    | x5    | x4    | x3    | x2    | x1    | x0    |
frever12 mm2, mm0   ;mm2=  | x0    | x1    | x2    | x3    | x4    | x5    | x6    | x7    |
fneg12   mm2, mm2,15;mm2=  | x0    | x1    | x2    | x3    | -x4   | -x5   | -x6   | -x7   |
fadd12   mm0, mm2   ;mm0=  | x0+x7 | x1+x6 | x2+x5 | x3+x4 | x3-x4 | x2-x5 | x1-x6 | x0-x7 |
```

Fig. 8. A part of the code for horizontal DCT that has been implemented by MMX enhanced by extended subwords



```
fldc16s16  cmm0 ,  (dct)   ; cmm0 =  |x07|x06|x05|x04|x03|x02|x01|x00|
fldc16s16  cmm1 ,  16(dct); cmm1 =   |x17|x16|x15|x14|x13|x12|x11|x10|
fldc16s16  cmm2 ,  32(dct); cmm2 =   |x27|x26|x25|x24|x23|x22|x21|x20|
fldc16s16  cmm3 ,  48(dct); cmm3 =   |x37|x36|x35|x34|x33|x32|x31|x30|
movq       (dct),  mm7     ; (mem) = |   x37   |   x27   |   x17   |   x07   |
movq       mm7  ,  mm0     ; mm7 =   |   x30   |   x20   |   x10   |   x00   |
paddsw     mm0  ,  (dct)   ; mm0 =   | x30+x37 | x20+x27 | x10+x17 | x00+x07 |
psubsw     mm7  ,  (dct)   ; mm7 =   | x30-x37 | x20-x27 | x10-x17 | x00-x07 |
```

Fig. 9. A part of the MMX + MRF implementation of the horizontal DCT algorithm

column register has eight 16-bit subword. Each subword in a column register corresponds to a subword in a row register. Each load column instruction can load eight 16-bit pixels into a column register. In addition, Figure 9 depicts a part of the MMX + MRF implementation of the horizontal DCT algorithm. There are two loop iterations to process an 8×8 block. This means that in each loop iteration, four rows (32 pixels) are processed.

6.3 Experimental Results

Figure 6.3 depicts the speedup of MMX + ES, MMX + MRF, and MMMX over MMX for one execution of an 8×8 horizontal DCT on a single issue processor. In addition, this figure shows the ratio of committed instructions (MMX over the other architectures). The speedup of MMX + ES is 1.15, while the speedup of MMX + MRF is less than 1. These results indicate that using either extended subwords or the MRF techniques is insufficient to eliminate most pack/unpack and rearrangement overhead instructions. In addition, using the MRF is both unuseful and causes performance loss. The MMMX architecture that employs both proposed techniques, on the other hand, yields much more performance benefits. Its speedup is 1.52.

In order to explain the behavior of Figure 6.3, Figure 6.3 shows the number of SIMD computation, SIMD overhead, SIMD ld/st, and scalar instructions for the four different architectures: MMX, MMX + MRF, MMX + ES, and MMMX

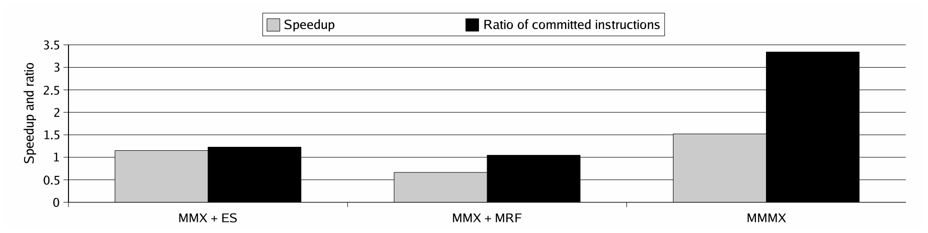


Fig. 10. Speedup of the MMX + ES, MMX + MRF, and MMMX over MMX as well as ratio of committed instructions for an 8 × 8 horizontal DCT on a single issue processor

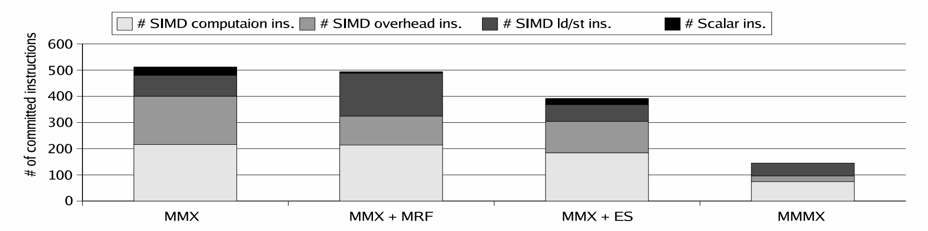


Fig. 11. The number of SIMD computation, SIMD overhead, SIMD ld/st, and scalar instructions in four different architectures, MMX, MMX + MRF, MMX + ES, and MMMX for an 8 × 8 horizontal DCT kernel

for an 8 × 8 horizontal DCT kernel. As this figure shows, the total number of instructions in the MMX + MRF is almost the same as MMX. This means that the former architecture cannot reduce the total number of instructions. The MMX + MRF reduces the number of SIMD overhead instructions, but it increases the number of SIMD ld/st instructions. This is because the MMX + MRF transposes four rows in each iteration and this causes that all eight 64-bit registers are filled. In order to use some of the filled registers for intermediate computations, they are stored and loaded in memory hierarchy and this increases the number of SIMD ld/st instructions. The latency of SIMD ld/st instructions is almost more than the latency of the SIMD overhead instructions. This is the main reason why the MMX + MRF has a performance penalty. MMX + ES, on the other hand, reduces the total number of instructions. The ratio of committed instructions is 1.23 as shown in Figure 6.3.

The extended subwords technique reduces the number of SIMD computation and SIMD ld/st instructions more than the MRF technique, while the latter technique reduces the number of SIMD overhead and scalar instructions more than the former technique. Consequently, these experimental results indicate that using either of these techniques is insufficient to mitigate SIMD computation, SIMD overhead, SIMD ld/st, and scalar instructions. The MMMX architecture that employs both proposed techniques reduces the total number of instructions much more than MMX + MRF and MMX + ES.

7 Conclusions

SIMD architectures suffer from the mismatch between the storage and the computational formats of multimedia data and using data permutation instructions during vectorization. We already proposed two architectural enhancement, the extended subwords and the Matrix Register File (MRF) to alleviate these limitations. The extended subwords provide additional parallelism by avoiding data tyep conversion instructions. The MRF eliminates data permutation instructions. The MMX architecture has been modified by the proposed techniques that was called the Modified MMX (MMMX) architecture. In this paper, we validated the MMMX architecture on a wide range of multimedia kernels. In addition, in order to determine the performance benefits of each proposed technique, we analysised each technique separately. The results showed that employing the MRF (MMX + MRF) and extended subwords (MMX + ES) separately obtain the speedup less than 1 and 1.15, respectively. This is because the total number of instructions in the MMX + MRF is almost the same as MMX. This means that the former architecture cannot reduce the total number of instructions. The MMX + MRF reduces the number of SIMD overhead instructions, but it increases the number of SIMD ld/st instructions. MMX + ES, on the other hand, reduces the total number of instructions. The results indicate that using either extended subwords or the MRF techniques is insufficient to eliminate most pack/unpack and rearrangement overhead instructions. The combination of both techniques should be employed in SIMD implementation.

References

1. Advanced Micro Devices Inc. 3DNow Technology Manual (2000)
2. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An Infrastructure for Computer System Modeling. IEEE Computer 35(2), 59–67 (2002)
3. Bannon, P., Saito, Y.: The Alpha 21164PC Microprocessor. In: IEEE Proc. Compcon 1997, February 1997, pp. 20–27 (1997)
4. Diefendorff, K., Dubey, P.K., Hochsprung, R., Scales, H.: AltiVec Extension to PowerPC Accelerates Media Processing. IEEE Micro 20(2), 85–95 (2000)
5. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edn. Morgan Kaufmann, San Francisco (2002)
6. Huang, L., Lai, M., Dai, K., Yue, H., Shen, L.: Hardware Support for Arithmetic Units of Processor with Multimedia Extension. In: Proc. IEEE Int. Conf. on Multimedia and Ubiquitous Engineering, April 2007, pp. 633–637 (2007)
7. IBM. Synergistic Processor Unit Instruction Set Architecture (January 2007)
8. Jennings, M.D., Conte, T.M.: Subword Extensions for Video Processing on Mobile Systems. IEEE Concurrency 6(3), 13–16 (1998)
9. Juurlink, B., Borodin, D., Meeuws, R.J., Aalbers, G.T., Leisink, H.: The SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT), `http://ce.et.tudelft.nl/~shahbahrami/`
10. Lee, R.B.: Subword Parallelism with MAX-2. IEEE Micro 16(4), 51–59 (1996)
11. Loeffler, C., Ligtenberg, A., Moschytz, G.S.: Practical Fast 1-D DCT Algorithms With 11 Multiplications. In: Proc. Int. Conf. on Acoustical and Speech and Signal Processing, May 1989, pp. 988–991 (1989)

12. Peleg, A., Weiser, U.: MMX Technology Extension to the Intel Architecture. IEEE Micro 16(4), 42–50 (1996)
13. Raman, S.K., Pentkovski, V., Keshava, J.: Implementing Streaming SIMD Extensions on the Pentium 3 Processor. IEEE Micro 20(4), 47–57 (2000)
14. Ranganathan, P., Adve, S., Jouppi, N.P.: Performance of Image and Video Processing with General Purpose Processors and Media ISA Extensions. In: Proc. Int. Symp. on Computer Architecture, pp. 124–135 (1999)
15. Shahbahrami, A.: Avoiding Conversion and Rearrangement Overhead in SIMD Architectures. PhD thesis, Delft University of Technology (September 2008)
16. Shahbahrami, A., Juurlink, B., Vassiliadis, S.: Versatility of Extended Subwords and the Matrix Register File. ACM Transactions on Architecture and Code Optimization (TACO) 5(1) (May 2008)
17. Tremblay, M., Michael 0'Connor, J., Narayanan, V., He, L.: VIS Speeds New Media Processing. IEEE Micro 16(4), 10–20 (1996)