Motion Video Instruction Extensions for Alpha

Paul Rubinfeld
Bob Rose
Michael McCallig

Contact:
Paul Rubinfeld
Semiconductor Engineering Group
77 Reed Road, HLO2-3/D11
Hudson, MA 01749

prubinfeld@ds.hlo.dec.com

18-Oct-96

Scope

The Alpha architecture[1] has added several new instructions called Motion Video Instructions (*MVI*) that can be used to accelerate the performance of key algorithms used in emerging motion video technologies. The criteria for selecting these new instructions are based on the fundamental premise that it is important to keep the Alpha architecture "clean" in order to facilitate extremely fast circuit implementations of the architecture. In order to be added, a new instructions must passed two critical tests: the benefits provided must span multiple generations; and, the benefits must be material.

The MVIs for the Alpha architecture were motivated by the desire to perform high quality software motion video encoding using the prevalent ISO/ITU video compression standards. The standards that were addressed were MPEG 1 [2], MPEG 2 (also known as H.262) [3], H.261 [4], and H.263. These are all motion compensated discrete cosine transforms (*DCT*) and inverse discrete cosine transforms (*IDCT*) based compression methods. Thus, a common set of hardware features are able to accelerate the computations for each of these standards.

MPEG 1 is targeted at storing 74 minutes of CD quality audio and VHS quality video on a single speed CD ROM. MPEG 2 is a scaleable compression standard that is applicable to low resolution video through High Definition Television. The most prevalent application is broadcast quality television at approximately 6Mb/sec. H.261 is aimed primarily at ISDN teleconferencing and is scaleable in resolution and quality from a single BRI (128 Kb) up to a full T1 line. H.263 is a new teleconferencing standard that should eventually replace H.261. It is scaleable from sending video and audio over a 28.8 kbaud analog modem to very high bit rates and high resolution.

In all cases, the standard published by ISO or the ITU is an interchange standard. It specifies what the bit-stream should look like. The implementation of the decoder is pretty much determined by the bit-stream specification. The algorithms used to encode the bit-stream, however, provide for a rich set of tradeoffs affecting cost, image quality, and support of extended features. An encoder design provides value added by producing the highest quality pictures for a for a given bit rate within the constraints of the compression engine. The design of the compression engine involves another set of cost verses quality tradeoffs. Thus, each encoder design adds value by optimizing all of these parameters to produce the best video at the lowest cost for a given design point.

Software encoders are available today for H.261 and H.263 at low resolution and low frame rate. Software real time MPEG encoders are not commonly available today. Most of the software encoders today use highly simplified motion estimation algorithms. This is necessary because motion estimation is an extremely large consumer of CPU cycles. In order to meet real time constraints, it is necessary to make a number of serious compromises. This results in video that looks good when there is little motion. However, when there is a lot of complex motion, the quality falls off rapidly. The quality of the motion estimation is, in most cases,
the largest single factor limiting the quality of the video in an implementation.

The goal of the new Alpha MVIs is to enable software encoders that produce quality competitive with dedicated hardware encoders. Thus, the motion estimation techniques that are described below are the same techniques used by many hardware encoders.

The motion estimation operation is far and away the highest consumer of CPU cycles. Typically, the motion estimation workload is an order of magnitude higher than the workload other video operations.

New Instructions

The new instructions are:

1. **PERR Ra.rq, Rb.rq Rc.wq** - Sum of Absolute Differences - The absolute value of the differences between each of the bytes in Ra and Rb is calculated. The sum of these resulting bytes is written to Rc. This instruction provides an order of magnitude performance improvement over code written without the MVI instructions and enables useful software encoding of Motion Compensated Video.
2. **MINUB8, MINSB8, MINUW4, MINSW4, MAXUB8, MAXSB8, MAXUW4, MAXSW4** - Clamping Instructions - These permit minimum and maximum operations on multiple 1-byte (*byte*) or 2-byte data (*word*) stored in an 8-byte data word (*quadword*). Both signed and unsigned versions are supported. At various places in the standard video algorithms, it is necessary to limit the range of results by saturating them to a value. This has been done with a CMOV instruction in the past. These new instructions provide better performance than CMOV because of their 8-way parallelism.
3. **UNPKBL, UNPKBW, PKLB, PKWB** - Packing and unpacking bytes in quadword from words, and 4-byte data words (*longwords*). In motion video, these are useful instructions because pixels are stored as bytes and words, the DCT and IDCT must be done as longwords, and the motion compensation needs 9-bits. The algorithms need to convert the IDCT results from longwords to words, the reference image data from bytes to words, add the two values, then clamp the result and convert the final results back to bytes. Similar examples for using these instructions can be made for generic signal processing examples that use 8-, 16-, or 32-bit data.

Analysis of Motion Estimation Computations

The following analysis assumes that the reader is familiar with the various ISO standards. We will look at several motion estimation algorithms that are frequently used in commercial-quality hardware MPEG encoders (full search estimation, hierarchical search estimation and telescoping search estimation)

Full Search Estimation

The purpose of Motion Estimation is to create a predicted frame from a reference frame or frames already available to the decoder. For each macroblock, a search of the reference frame

is conducted in its vicinity. The predicted frame is created by tiling together the best match found by motion estimation for each macroblock. The encoder then sends the instructions to create the predicted frame and the difference between the predicted frame and the actual frame.

The distance searched for each macroblock is not explicitly set by the ISO standards. Typically, a moving object will cover a similar distance between each frame. Therefore, the practical *search range* needed depends on the number of frames between the reference frame and the current frame. The larger the frame spacing, the larger the search range needs to be. For example, in our MPEG 1 encoder, we search ±22.5 pixels for frames that are three frames apart.

The criterion for best match is also not explicitly set the ISO standards. The most common match algorithm uses the macroblock position that produces the minimum sum of absolute differences. The sum of absolute differences is calculated by subtracting each pixel in the macroblock in the reference frame from each pixel in the macroblock in the new frame. The following equation is evaluated for each search position

$$E(dx,dy) = \sum_{i=0}^{15} \sum_{j=0}^{15} |r(x_0 + dx + i, y_0 + dy + j) - p(x_0 + i, y_0 + j)|$$

where: x_0 *= the X position of the macroblock*
y_0 *= the Y position of the macroblock*
dx *= x offset from macroblock X position*
dy *= y offset from macroblock Y position*
r *is the addressed pixel from the reference frame*
p *is the addressed pixel from the new frame*

The value of dx and dy that produced the lowest error value form x_0 and y_0 coordinates is the motion vector. The number of computations for one search is 256 differences, 256 absolute values, and 255 sum operations.

Hierarchical Motion Estimation

The number of computations can be reduced relative to the full search technique by doing a hierarchical search. In this method, the reference image and the new image are subsampled to half the vertical and half the horizontal resolution. The subsampled images are use to search over the full search area to get the high order bits of the motion vector. Then, the search is done at full resolution over a narrow range around the vector resulting from the subsampled search. Finally, the search is done on the half pixel positions around the full pixel motion vector. For example, if a 2:1 subsample factor is used for the first search, the number of computations to perform one search is 64 differences, 64 absolute value operations, and 63 sums. This is reduction from full pixel search by a factor of 4. In addition, one search covers four pixel positions, so one fourth the number of searches are required. This represents a factor of 16 improvement. However, it is still necessary to conduct the full pixel search over a narrow range and a half pixel search over a ±0.5 pixel area. Taking this into account reduces the improvement to a factor of about 10 instead of 16.

There maybe a small quality loss with the hierarchical search algorithm compared to the full search algorithm of the same search range. However, since the hierarchical algorithm permits a larger area to be search in a fixed amount of time, it may actual improve quality.

Telescoping Search Estimation

The distance required for the search can be reduced by using a telescoping search. In a sequence of frames (I, P, B are specified in the ISO standard, the subscript numbering denote frame sequence):

$I_1\ B_2\ B_3\ P_4$

a typical set of search ranges would be:

Search	Range
$I_1 \rightarrow P_4$	$\pm$ 22.5
$I_1 \rightarrow B_2$	$\pm$ 7.5
$P_1 \rightarrow B_2$	$\pm$ 15.5
$I_1 \rightarrow B_3$	$\pm$ 15.5
$P_4 \rightarrow B_3$	$\pm$ 7.5

The search range is increased as the distance between frames increases to allow for the fact the objects in motion had more time to move. In telescoping search, the searching is done in frame sequence order. The trajectory for a given macroblock is used to predict the new position of the macroblock. That prediction is use to initialize the search for the macroblock in the next frame. This permits searching over a smaller range of pixels. Reverse searches are done in reverse frame order using a similar learning initialization method.

For the above case, $I_1 \rightarrow B_2$ is conducted first using the $\pm$ 7.5 pixel search range. Next, the $I_1 \rightarrow B_3$ search is performed centered around a prediction from the position found for the $I_1 \rightarrow B_2$ search. Assuming the motion is smooth from one frame to the next, the search range for $I_1 \rightarrow B_3$ can be reduced to $\pm$7.5 pixels. Next the $I_1 \rightarrow P_4$ search is conducted with a $\pm$7.5 pixel range starting from the position found from the $I_1 \rightarrow B_3$ search. Next, the $P_4 \rightarrow B_3$ search is conducted followed by the $P_4 \rightarrow B_2$ search. The bottom line here is that all of the search ranges are reduced to the range used for an adjacent frame.

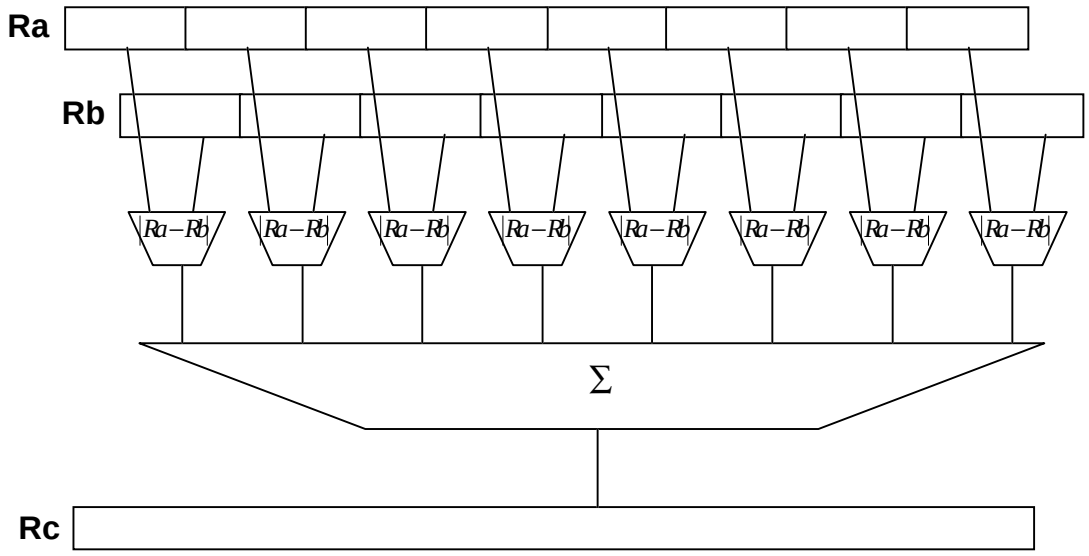
The telescoping search works well for smooth motion. It can fail to find the optimum motion vector for erratic motion in some cases. It works well in typical video sequences.

Selected Method

The above methods are used today in high-quality hardware MPEG encoders. The high-performance Alpha architecture with the new MVI instruction will allow these algorithms to be done in software in real-time.

PERR Instruction

Each search method uses the PERR instruction. It takes eight pixels packed into quadword and computes the absolute differences between two registers. This is shown in the figure on the top of the next page:



PERR retires eight pixels of motion compensation computation. The net result is that eight pixels of motion estimation work can be accomplished in one clock tick of Alpha.

Analysis

The cycle counts will be computed for several interesting ways of doing motion estimation. These number will then be used to compute the cycle counts for doing motion estimation for different frame sizes and frame rates.

A given macroblock search is done in three steps: first, a 2x2 subsample search is used to cover the full search area. This produces a motion vector good to the best 2 pixel position. Next, the ±1 x ±1 area around this motion vector is searched at full resolution to find the best motion vector to 1 pixel position. Finally, the half pixel positions around this motion vector are searched to find the best half pixel motion vector. The new Alpha processors with the MVI instructions are fast enough to support this search strategy for a real time encoder.

2x2 Subsampled Search

The operands require eight registers each, so both macroblocks fit in registers in the Alpha architecture. The eight pixel wide operations cover the horizontal span of the subsampled macroblock. The overhead of loading the registers is minimized by conducting the search in a vertical direction so that for the search inner loop, the top scan is discarded and the bottom scan is added. This permits the *X* alignment work to be saved from one macroblock to the next.

As noted in the previous section, Alpha can do the absolute differences, tree summation, accumulation for one scan to the next, and loads for the next search all in one clock tick. The loop control can be done in two more clock ticks. The total cycle count allocated for one search position is 10 clock ticks.

Full Search

Full search technically means searching every half pixel position. The cycle count for this is prohibitive. A good approximation is to search every full pixel position then find the best half pixel position in the neighborhood of the full pixel match. That is the case that will be analyzed here.

A 16X16 pixel macroblock does not fit in the Alpha integer registers. The impact is minor for a hierarchical motion estimator because the full pixel search range is small.

Each search position requires summing the absolute difference of each pixel over 16X16 pixels. The new instructions can search eight pixels in one clock tick. The search is conducted vertically so the *X* alignment work can be saved. However, it must be stored and loaded since it does not fit in registers. Each eight pixels of work require two loads, a sum of absolute differences, a tree add and an accumulate. The remainder of the overhead can be buried in the remaining cycles available to the Alpha execution units. So, a full pixel search requires 64 CPU cycles.

Half Pixel Search

The half pixel search requires the half pixel data to be interpolated by averaging the macroblock with a macroblock offset by one pixel. Once the interpolated reference data is created, the match operation is conducted in the same matter as the full pixel search. The interpolation consists of eight cases. Four cases; up, down, left and right, require averaging of two positions. The four diagonal cases require averaging of four positions.

Each scan requires 3 *load* instructions and two *align instructions* to create the reference data. The offset data can be created with two more align instructions for X offsets and with one *mov reg* instruction for Y offsets. It should be possible to do all the averaging work and pixel differencing work in 3 ticks for each eight pixels. That results in 96 cycles for each position searched.

Cycle Counts

The following chart shows the important parameters of several frame formats of interest. The cycle count is computed for the various resolutions assuming that a hierarchical search is conducted starting with a 2X2 subsampled search to cover the bulk of the search range. This is followed by a ± 1 pixel search at full resolution and a $\pm 1/2$ pixel search. The computation is done with and without using telescoping search. The data shows that the telescoping search cuts the work load about in half.

For CCIR601 resolution, the computation was done for an MPEG 1 type of motion estimation using the same search range as SIF. This should produce results close to what you would get for MPEG 2 field coding.

	I Frames	P Frames	B Frames
# per second	2	8	20

# of Searches	Dist 1	Dist 2	Dist 3	Cycles/Search
Subsampled	49	225	529	10
Full Pixel	9	9	9	64
Half Pixel	9	9	9	96

Cycles/Mblock			
Subsampled	490	2,250	5,290
Full Pixel	576	576	576
Half Pixel	864	864	864
Total	**1,930**	**3,690**	**6,730**

	QSIF	SIF	CIF	CCIR601
Width	176	352	352	720
Height	120	240	288	480
# Pixels	21,120	84,480	101,376	345,600
Mblocks/Frame	88	330	396	1,350
Frame Rate	30	30	30	30

Hierarchical Srch				
Dist 1 cycles/sec	3,396,800	12,738,000	15,285,600	52,110,000
Dist 2 cycles/sec	6,494,400	24,354,000	29,224,800	99,630,000
Dist 3 cycles/sec	4,737,920	17,767,200	21,320,640	72,684,000
Total	**14,629,120**	**54,859,200**	**65,831,040**	**224,424,000**

Telescoping Srch				
Dist 1 cycles/sec	3,396,800	12,738,000	15,285,600	52,110,000
Dist 2 cycles/sec	3,396,800	12,738,000	15,285,600	52,110,000
Dist 3 cycles/sec	1,358,720	5,095,200	6,114,240	20,844,000
Total	**8,152,320**	**30,571,200**	**36,685,440**	**125,064,000**

Conclusions

The added instructions provide the ability to do motion estimation for QSIF, SIF and CIF resolutions using 10% - 15% on Alpha. CCIR601 resolution uses about half the Alpha. These usage figures can be cut approximately in half if a telescoping search is employed. A telescoping search should produce high quality results for well behaved motion sequences. More random sequences may show some quality loss. The cycle count analysis needs to be extended to cover all of the motion estimation modes in MPEG 2.

Marginal Benefit of Adding Intel-like Multimedia Instruction to Alpha

Several vendors have recently added multimedia instructions (MMX instruction set) to the X86 Architecture [5]. These instructions feature 16-bit integers packed into a 64-bit operand so that one instruction can do four 16-bit operations. The cornerstone on the new instructions is the MUL-ADD instruction, which does four 16-bit multiplies and accumulates the 32-bit products into two 32-bit sums. Interestingly, Intel did not include any instructions that accelerate the pixel error

calculation similar to PERR. This precludes Intel from ever providing a meaningful (near-real time) software motion video estimation solution that meets the ISO/IEC standards.

This section examines why Alpha did not further extend the architecture to include other MMX-like instruction.

The basic reasons the Alpha instruction set was not further expanded to include the new Intel-like extensions (most notably the MUL-ADD instruction) are that 16-bit data size is insufficient for a lot of important applications including the commercial motion video market; memory bandwidth limitations minimizes the overall performance benefit provided by these new instructions; and, any real performance benefit that Alpha would get from such extensions would be small since MMX essentially tries to fix some of the x86 legacy-architecture performance deficiencies which are not present in the Alpha architecture (e.g., limited number of registers, poor floating point organization, etc.)

Limitations to 16-bit processing

There are extreme limitations to processing that can be done with 16-bit data. Even in the motion video markets which only utilizes a highly constrained set algorithms, more than 16-bits are required to be compliant with the ISO/IEC standard algorithms that are based on motion compensation and DCT transform coding. These include MPEG 1 and MPEG2, H.261 teleconferencing and H.263. Adding 16-bit functions is inconsistent with the goal to create MVI instructions that are intended to support these standards and enable growth in algorithm complexity for these markets.

In addition, scientific uses of signal and image processing always require significantly more than 16-bits of accuracy, especially on intermediate results. A considerable amount of pre-programming analysis and scaling within the code is avoided by using the high dynamic range of floating point. The 16-bit limit of MMX comes into play in two ways. First, to prevent overflow, the dynamic range of the data will be less than 16 bits even with the fact that the MMX multiplier product carries the full 32 bits. A typical value for a maximum 16 bits is to restrict the data to 8-10 bits. Since round off noise averages 6 dB/bit, the noise threshold on processing with MMX-like instructions is 50 - 60 dB. It is not enough for scientific applications. Scientific processing also uses very large data records. This is the second way 16-bit processing is a limiting factor. A 32-point fast Fourier transform (FFT) can be calculated with 16-bit processing. A 128 point FFT can often be calculated with 16-bit processing (it is dependent on the exact values in the data record). A million point FFT can never be calculated with 16-bit processing.

Limitations Due to Memory Bandwidth

The marginal benefit of further extending the Alpha instruction set can be seen to be limited for most commercial applications due to memory bandwidth issues. The examples will be drawn from three classes of codes

1. Small codes, where the data set always hits in the cache and the total cycles is a very small percentage of the available cycles.
2. Medium codes, where the cache performance is very good but there are misses. Total cycles is a noticeable percentage of the available cycles.
3. Large code, where the cache performance is poor and total cycles is a large percentage of the available cycles.

The results will show that only medium size codes might benefit from further adding MMX-like instructions to Alpha. Small codes execute so fast that any speedup is unimportant. For large codes, the most important performance issue is memory access, not CPU cycles. MMX-like instructions can do nothing to help. Medium codes might benefit from MMX like instructions. This is a fairly narrow class of problems since it is really a class that sits on the border between codes that fit in cache and codes that do not. The fact that the problem must be computable with 16-bits makes it smaller still. It is unlikely that the performance improvement for a restricted set of codes is worth the addition of MMX-like instructions. It is also true that many of these codes, such as the integer DCT used as an example, can be attacked in other ways. In the case of the discrete cosine transform, sparse matrix techniques are used.

Small Codes

Smaller problems can be handled with fewer bits, and the non-scientific markets referred to above do have relatively small signal and image processing needs. Small problems fit in on-chip caches. It is unlikely that such code is the performance bottleneck in a larger application.

An example is the 4x4 matrix-vector multiply, commonly used to manipulate 3D objects [6]. One matrix is multiplied with many different 4 element vectors. Reference [6] gives the total instructions, but not the schedule or cycle count.

Intel without MMX	[6]	72 instructions
Intel with MMX	[6]	28 instructions

The Alpha processor will do the calculation in floating point. The loop is unrolled 2 times. The compiler generated code keeps 10 of the 16 elements of the 4x4 matrix in registers. The remaining 6 are loaded within the loop as are the two 4 element vectors. The floating point operation count for Alpha for the twice unrolled loop is

loads	14
multiplies	32
adds	24
stores	8

There are also loop and address instructions that multiple issue with the floating point operations. The multiple issue breakdown is

single issues	29
dual issues	20
triple issues	2
quad issues	2
stall cycles	0

total cycles 53 for 2 matrix - vector multiplies

We assume perfect pairing for the MMX code and a 200MHz Intel processor. The Alpha clock is 500 MHz.

Intel with MMX	14 Intel cycles = 35 Alpha cycles
Alpha	26 Alpha cycles

If you allow for pairing to not be perfect on the Intel processor, the Alpha is faster by a factor approaching 2. MMX-like instructions could speed up this code on Alpha. Would it be worth it? Al-

pha can do 19 million 26 cycle operations per second. If only 1% of the CPU time were spent on this operation, Alpha could do 190,000 matrix-vector multiplies per second. This is far more than needed in a typical multimedia application. Alpha can satisfy the need for matrix-vector multiplies and spend less than 1% of the CPU time doing so. Clearly, for this example, the improvements of adding MMX-like instructions to MVI do not pass the "significant benefits" test outlined at the beginning of this paper.

The alert reader will suspect that even 1% is overstated. The premise of this example, and indeed of all the MMX examples in the Intel application notes, is that the data is in the on-chip cache. The Intel processors have a 16KB on-chip data cache and Alpha has up to 96KB on-chip. To stay in these caches, the number of matrix-vector multiplies must be on the order of 8000 (typically it is much higher). Alpha can do this in less than 5 hundredths of one percent of the total processor time (26 * 8000 cycles out of 500,000,000). If there is a processor bottleneck, this is not it.

Large problems

Larger multimedia problems are memory or even disk bound. Adding MMX-like instructions to Alpha would not lead to a speedup. An example is an image dissolve [6]. One image fades out while another fades in over it. The equation the MMX example is based on is

result_pixel = image_1 * (k/255) + image_2 * (1 - k/255)

where k is a constant within each frame and there are 8 bits in each pixel color. As k is varied from 255 to zero, image_1 fades out and image_2 fades in. The best MMX performance was obtained when the image was organized as separate color planes. The MMX instructions could then be used to advantage by processing 4 pixels from a single plane at once. The example from reference [6] is, unlike the matrix-vector multiply example, a real performance issue. The question is, what is limiting the performance and will MMX-like instructions help Alpha? The example images are 640 x 480 pixels (full screen standard TV), 8 bits per pixel and 3 colors. The total data for the two images is

2*640*480*1*3 = 1.8 Megabytes

The example does the fade over 255 frames, or about 8.5 seconds. The total image data processed is 470 Megabytes. The performance bottleneck here is the memory system, not the CPU. The addition of MMX-like instructions will not help.

Medium problems

"Using MMX Instructions in a Fast IDCT Algorithm for MPEG Decoding" [7] describes and gives a hand crafted code listing for an integer discrete cosine transform using MMX instructions. The IDCT is intended for use in an MPEG decoder for displaying compressed video. The MPEG compression/decompression algorithm uses the IDCT on an 8x8 array of 16 bit data. The code provided in [7] does the IDCT in 240 Intel cycles. The skillfully hand-crafted code pairs instructions in 200 of the 240 cycles.

It should be noted in passing that the IDCT algorithm used is not compliant with the IEEE specification for IDCT accuracy and would lead to unacceptable artifacts in a video conferencing application using the H.261, H.262, and H.263 specifications. It is tolerable for home applications of MPEG decoding.

To evaluate the need for MMX-like instructions on Alpha for MPEG decoding, the MPEG decoder from the MPEG Software Simulation Group [8] was compiled, run and profiled. A 150

frame sequence was decompressed into 352x240 color frames. The file of compressed video is 725 Kbytes. The IDCT in this code is IEEE compliant which is required for commercial applications.

The functions that used more than half a percent of the CPU time are shown in figure 1. To focus on processor performance, time reading compressed data or writing decompressed data are not included.

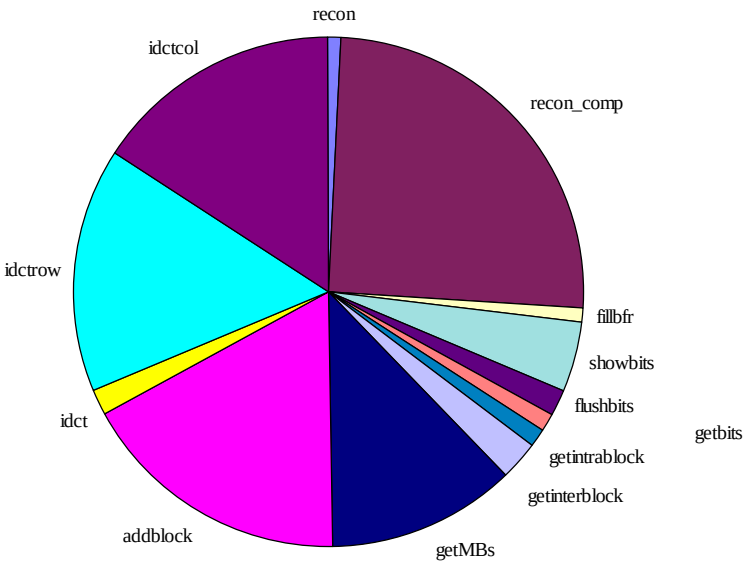


Figure 1. MPEG Decoder Execution Profile

The first point to notice is that there are several functions besides the IDCT that take a sizable amount of the processor.

IDCT (idct idctrow idctcol)	33 %
recon_comp	25 %
addblock	17 %
get_MBs	12 %

87 % of the time is spent in these four areas. Only the 33 % in the IDCT can be helped by MMX instructions. If we get a 2 times speedup, that would translate into 15% on the MPEG decoder.

The IDCT code was written to take advantage of all zero valued coefficients in any row or column of the 8x8 matrix being operated on. The 8x8 IDCT is done by doing eight 8x1 IDCT's on the rows followed by eight 8x1 IDCT's on the columns. The 8 coefficients are loaded and tested for all zero. If they are zero, then a quick exit can be done without the need for the full calculation. The full calculation takes 3 times longer than the quick exit. On Alpha, the cycle count for a complete 8x8 IDCT is 656 cycles.

There is no way to do the quick exit with the MMX code [7] without significantly increasing the schedule. The MMX code has had to unroll the IDCT to such an extent that branch decisions such as this can no longer be made.

Intel with MMX	240 Intel cycles = 600 Alpha cycles
Alpha	656 Alpha cycles

IDCT Scheduled Cycles

Scheduled cycles is only part of the story. Since the data being processed is larger than the on-chip cache, the actual number of cycles will be greater than the scheduled cycles, the difference being the time to access memory and/or the board cache. Reference [7] does not provide any execution times. On Alpha, the off-chip accesses take an additional 325 cycles. If it is assumed that the Intel off-chip accesses are in the same proportion, then an estimate of IDCT time for both processors can be calculated. In the table below, both processors are assumed to have the same absolute time to off-chip cache or memory. The Alpha clock is 2.5 times the Intel clock. All values are in Alpha cycles.

	Alpha	Intel
total scheduled cycles	650	600
off-chip cycles	325	325
total	975	925

IDCT Actual Cycles

An obvious next step is to try to eliminate the off-chip cycles by prefetching the data during arithmetic operations. This requires that the processor continue to execute instructions that do not depend on the data being fetched. Both Alpha and the Intel Pentium Pro can do this. This will get the IDCT down to 20% of the MPEG decoder. MMX-like instructions could then cut this in half, saving 10% on the decoder. It is not clear that this savings is worth the addition of a new class of instructions.

Conclusion

MMX-like instructions have been examined on Alpha in the context of three examples from Intel application notes on MMX. The other examples from these notes have been studied and all fall into one of the three categories described here:

1. The code and data are small enough that it is not a performance bottleneck
2. The code and data are so large that memory, rather than CPU performance, is the bottleneck.
3. The code and data are in an intermediate range where MMX-like instructions could boost performance on the order of 10%.

As with most performance issues, there are several related factors that are difficult if not impossible to break out individually. The most widely discussed advantage of MMX for X86 processors is the four fold parallelism. Another equally important advantage is a four fold increase in the apparent size of the register file. This in turn allows loop unrolling, which allows instruction scheduling to avoid unused issue slots. It is equally valid to consider this expansion of the register set as the first addition to the architecture, with most of the performance advantage deriving from it. The MMX instructions are then just a very clever way of implementing the expanded register set without the hardware difficulties of actually doing so. Alpha has the large register file, has the unrolling, and has a compiler to automatically schedule the code. When looked at from this perspective, it is understandable that MMX would not be nearly as significant for Alpha processors as it is for Intel processors.

It is interesting that Intel did not include a PERR instruction in the MMX extensions. This instruction provides the most computational benefit of all architectural extensions. PERR enables real time or near-real time software video encoding solutions, something that is not possible with Intel's MMX solution.

Further extending Alpha to include MMX-like instructions would not pass the critical tests outlined at the beginning of this paper. The benefits that would be provided would not span multiple generations (the application which can utilize 16-bit accuracy are already very limited) and the actual benefits added by such extensions are very minimal.

References

[1] Alpha Architecture Reference Manual, R. Sites, ed., Digital Press, Burlington, Ma, 1992

[2] ISO/IEC 11172 MPEG-1 Standard, Coding of Moving Pictures and Associated Audio for Digital Storage Media up to about 1.5 Mbits/sec

[3] ICO/IEC MPEG-2 Standard

[4] ITU Standard H.261, Video Codex for Audiovisual Services at P x 64 Kbits/sec

[5] Press release MMX technology, Intel, March 5, 1996.

[6] MMX Technology Overview, Intel.

[7] Using MMX Instructions in a Fast iDCT Algorithm for MPEG Decoding, Intel.

[8] MPEG-2 Encoder / Decoder, Version 1.1, June 1994, MPEG Software Simulation Group