# Instruction Set Architecture Enhancements for Video Processing

Jan-Willem van de Waerdt*[+]
*Philips Semiconductors
San Jose, CA, USA

Stamatis Vassiliadis [+]
[+] Delft University of Technology,
Delft, The Netherlands
Stamatis@dutepp0.et.tudelft.nl

## Abstract

*This paper presents some of the enhancements to the TriMedia instruction set architecture (ISA), as supported by the TM3270 media-processor. We show how the new operations are used to optimize the individual MPEG2 encoder kernels. Furthermore, we quantify the contribution of these kernels to overall MPEG2 encoder performance. We introduce two-slot operations, collapsed load operations wit interpolations, and new multiplication operations. The encoder's texture pipeline for a bi-directionally predicted 8x8 block is performed in 358 VLIW instructions. MPEG2 encoding at CIF resolution at 25 frames per second is achieved within 33.5 MHz. of processor performance.*

## 1. Introduction

Media-processors can be used in many video-processing applications. Their programmability provides flexibility, which can be exploited in different ways. It enables algorithmic changes after design, multiple algorithms can be mapped to the same architecture, faster time-to-market, etc.. When enough performance is available, they provide an interesting alternative to fixed dedicated hardware solutions. Furthermore, when a single programmable platform can address multiple markets, its development costs can be shared.

Progress in video codec research has led to an increasing number of standards, such as MPEG2, MPEG4, and H.264/AVC [1]. More recent standards introduce new encoding tools to offer improved performance in terms of picture quality at a certain bitrate. This, however, does not mean that older standards become obsolete. The introduction of new standards to the market is a gradual process, which means that the latest product needs to support multiple standards, to ensure backward compatibility. The multi-standard requirement makes a programmable platform an interesting implementation platform. The ISAs of modern processor families are continuously updated, to address the compute requirements of new video standards.

This paper presents new operations to the TriMedia architecture, as supported by the TM3270 media-processor. We quantify their performance improvement on MPEG2 encoding. We also show how the new operations can be used to optimize kernels of other video standards, such as MPEG4 and H.264/AVC. The most notable new operations are: two-slot operations [2], collapsed load operations, and multiplication operations with scaling, rounding, and clipping support.

The remainder of this paper is organized as follows. Section 2 gives an overview of the TM3270. Section 3 presents the selection criteria for new operations. Section 4 presents a MPEG2 encoder implementation, and the use of new operations to optimize performance. Section 5 presents the contribution of the optimized kernels to overall MPEG2 encoder performance. Finally, in Section 6, we present our conclusions.

## 2. TM3270 overview

The TM3270 is backward source code compatible with the TriMedia architecture [4],[5]. The processor has a synthesizable design, allowing for fast process technology mapping. The processor achieves a frequency of 450 MHz. in a 90 nm process technology, and measures around 8.1 $mm^2$. The TM3270 has a VLIW architecture. A VLIW instruction may contain up to five operations. Each operation may be *guarded*; i.e. its execution is conditional on a guard register. SIMD arithmetic and shuffle operations allow for efficient manipulation and re-organization of 8-, and 16-bit data types. Operations are grouped into functional units, and most functional units have multiple instantiations. The simple arithmetic functional unit has five instantiations, so up to five simple arithmetic operations can be issued every cycle.

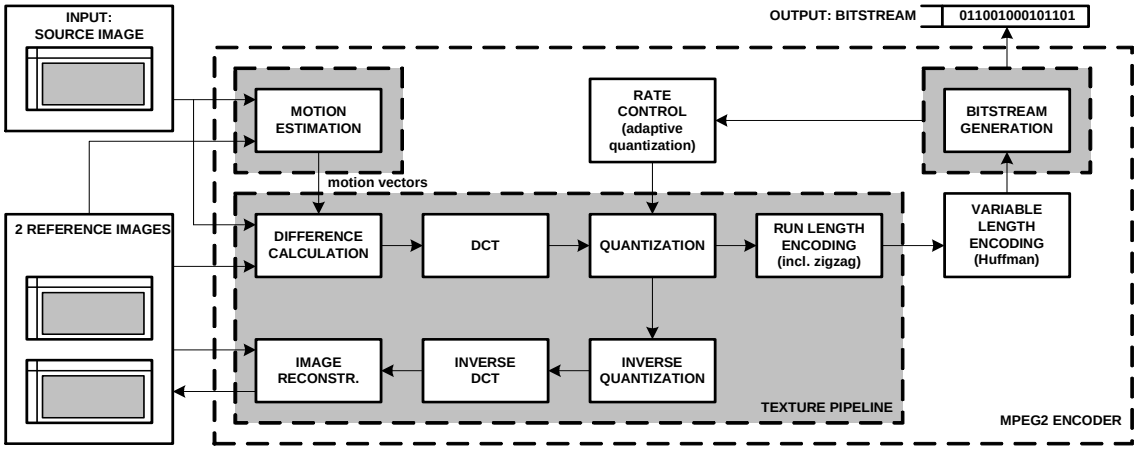


**Figure 1. MPEG2 encoder overview, with a breakdown in kernels.**

The TM3270 supports *two-slot* functional units, which are located in two neighboring issue slots (providing twice the normal register-file bandwidth). As a result, two-slot operations may have up to four 32-bit sources, and may produce two 32-bit results. The TM3270 has a 64 Kbyte instruction, and a 128 Kbyte data cache.

## 3. Selection of new operations

Video and audio processing are the processor's main application domains. Given the higher computational requirements of video processing, most new operations find application in this domain. This explains the MPEG2 encoder as an example application. While identifying potentially interesting operations, we applied certain selection rules:

*a). Fits the processor architecture*.

- No operations with architectural state.
- Operations are limited to up to two issue slots.
- The sub-operand fields of SIMD operations should have the same semantics.

*b). Reuse of available processor resources*. New operations typically add functionality to the existing datapath. The additional silicon area should be kept to a minimum, to allow for a low-cost implementation.

*c). Applicability in multiple domains*.

*d). Significant performance enhancement*. Performance improvement should be measured at the application level, rather than the kernel level.

An expert in the areas of processor architecture and video processing should preferably judge the ISA enhancement as obvious. Table 3 (found at the end of this paper) presents the new operations that are discussed in this paper.

## 4. MPEG2 encoder

Figure 1 gives an overview of a MPEG2 encoder. We started with a plain-vanilla C-implementation of a MPEG2 encoder, and invested 6 man weeks to optimize the implementation for the TM3270. We have not undertaken any optimizations that would compromise MPEG2 compliancy. Most of the optimizations involve the selection of custom operations to reduce computational complexity.

The MPEG2 encoder flow is summarized as follows. Macroblocks of 16x16 image pixels are processed in a left-to-right, top-to-bottom image order. First, motion estimation is performed at macroblock granularity. This kernel decides the encoding mode (intra-code, or predicted), and produces up to two motion vectors (two vectors for bi-directionally predicted macroblocks). Next, the texture pipeline is executed for every 8x8 block within the macroblock. The texture pipeline performs the image reconstruction, and produces a sequence of (run, length) pairs for every block. Finally, the (run, length) sequences are variable length encoded, and a bitstream is generated. This last step is only performed after the texture pipeline has produced the (run, length) sequences of all the blocks within a macroblock.

In the following sections we use the TM3270 ISA enhancements to optimize the grey-shaded parts in Figure 1. All kernel implementations are self-contained functions: function-call and –return overhead are included, function inputs are read from memory, and function outputs are written to memory. Separate intermezzo sections discuss related kernel functionality in other video codec standards such as H.264/AVC.

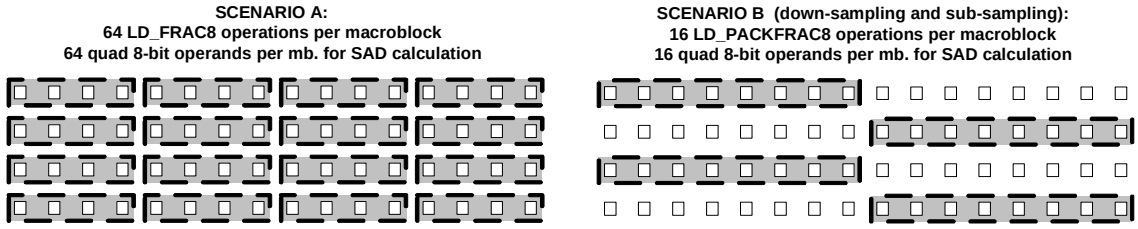


**Figure 2. Two implementations of the block-matching kernel (first four lines).**

## 4.1. Motion estimation

A significant part of the computational complexity of a video encoder is found in motion estimation. Motion estimation searches for temporal correlation between video images, which can be exploited to achieve a high compression factor. We implemented a version of the 3DRS motion estimation algorithm [6]. Most of the computational complexity of this algorithm is found in the "macroblock matching" kernel.

**4.1.1. Macroblock matching.** This kernel determines the similarity of a macroblock in the current image with a motion-displaced macroblock in a reference image. Typically, the match criterion is the Sum-of-Absolute-Differences (SAD) cost function. An optimized implementation uses the LD_FRAC8 collapsed load operation with interpolation to retrieve reference pixels from memory, and perform horizontal interpolation on-the-fly (Figure 2, scenario A). Vertical interpolation can use the QUADAVG operation (present in the original TriMedia ISA). The SUPER_LD32R two-slot operation is used to retrieve eight horizontally neighboring pixels from the current image. For an integer horizontal motion vector component, the SUPER_LD32R operation can also be used to retrieve reference pixels.

**Table 1. Block-matching and estimation, optimized performance.**

| Kernel/ function | New Operations* | | | Total operations | VLIW instructions | Operations/VLIW instr. |
|---|---|---|---|---|---|---|
| | SUPER_LD32R | LD_FRAC8 | LD_PACKFRAC8 | | | |
| Block matching Scenario A | 2*32 | 64 | | 333 | 108 | 3.08 |
| Block matching Scenario B | | | 32 | 198 | 73 | 2.71 |
| Estimator (P-frame) | 32 | 768 | 96 | 2327 | 568 | 4.10 |

* two-slot operations are counted twice: they occupy two issue slots.

Further optimizations to the kernel are possible at a degradation in quality. For example, the LD_PACKFRAC8 operation can be used to perform on-the-fly down-sampling of image pixels. The operation retrieves eight pixels from memory, performs a pair wise weighted average, and returns a quad 8-bit SIMD result that is input to the SAD calculation (the amount SAD inputs is reduced by a factor two). A further performance improvement is achieved by performing sub-sampling, which reduces the amount of pixels involved in the SAD calculation by another factor two (Figure 2, scenario B). Table 1 gives a performance comparison of scenarios A and B. Both scenarios assume a fractional horizontal motion vector component and an integer vertical component.

**4.1.2 The estimator.** The 3DRS algorithm performs multiple matches to find the best reference macroblock, based on the SAD cost function. The complexity analysis of motion estimation is very dependent on the specific algorithm and its implementation. Motion estimation is a much-discussed topic, and it is possible that more efficient algorithms exist, in terms of performance complexity and image quality.

Our implementation of the 3DRS algorithm evaluates a total of 17 motion candidates for every macroblock in a P-frame. The estimator function includes all control overhead related to the 3DRS algorithm. In an initial step 12 motion candidates are evaluated, using the matching of scenario B. Next, a refinement step evaluates 5 candidates, using the matching of scenario A. The matching functions are inlined in the estimator function (Table 2). The estimator function comprises 568 VLIW instructions.

The motion estimation algorithm uses the new SUPER_DUALIMEDIAN two-slot operation to clip two-dimensional positional information to a certain range.

*Intermezzo.* Whereas the MPEG2 standard allows for fractional motion vectors at half image pixel granularity, the MPEG4 and H.264/AVC standards support quarter pixel granularity. Furthermore, the calculation of data at fractional positions is more involved than that of MPEG2; multi-taps filters are

used, rather than the MPEG2 (bi-)linear interpolation. However, the initial block matching steps of the motion estimator can use a MPEG2-like (bi-)linear interpolation as an approximation of the multi-taps filter. When the loss in precision is unacceptable, new two-slot operations, like SUPER_USCALEFIR8UI and SUPER_USCALEMIX8UI, can be used to calculate fractional data at full precision.

## 4.2. Texture pipeline

The texture pipeline is executed for every 8x8 block that is to be encoded. For the discussion in this section we assume that the block belongs to a bi-directional predicted macroblock (we are in the process of encoding a B-frame). Note that uni-directional predicted or intra-coded macroblocks have a lower computational complexity. We restrict the discussion to motion vectors with a fractional horizontal component, and an integer vertical component. Figure 2 reflects the pipelined fashion of the texture coding. Most kernels forward their result to the next kernel in the texture pipeline. We first evaluate the performance of the individual kernels of the texture pipeline, and conclude with an evaluation of the complete texture pipeline. Table 2 summarizes the performance results.

**4.2.1. Difference calculation.** This kernel produces a block of prediction values. Furthermore, the difference between these prediction values and the block data in the current image is computed. The prediction values are 8-bit unsigned values; the difference values are 9-bit signed values. The optimized implementation uses the LD_FRAC8 operation. This operation performs horizontal interpolation for horizontal fractional positions as dictated by the motion vector. To retrieve a reference block at a horizontal fractional position requires 16 LD_FRAC8 operations. For two reference blocks, a total of 2*16=32 LD_FRAC8 operations are required. The block in the current image is located at an integer position, and can be retrieved with 8 SUPER_LD32R operations. Note that the non-optimized implementation needs to retrieve a 9x8 reference block, before it can start with the calculation of a reference block at a horizontal fractional location. For two reference blocks, a total of 2*24=48 load operations are required, and more operations are required to perform the interpolation. The block in the current image is retrieved with 16 load operations.

*Intermezzo*. The MPEG4 and H.264/AVC standards prescribe a non-linear multi-taps filter for the calculation of reference data at fractional positions. This prohibits the use of LD_FRAC8 operation. However, the required interpolation is efficiently implemented with the SUPER_USCALEFIR8UI and SUPER_FIR8UI operations. Given eight horizontal neighboring image pixels $r_0$, $r_1$, …, $r_7$, with $r_i$ located at horizontal position $i$, the H.264/AVC standard calculates $p$ at horizontal position 3½ using a 6-taps filter:

$$p = 2r_1 - 10r_2 + 40r_3 + 40r_4 - 10r_5 + 2r_6 + 32$$
$$p = Max(Min(p >> 6, 0), 255)$$

The MPEG4 standard calculates $p$ at horizontal position 3½ using a 8-taps filter:

$$p = -2r_0 + 6r_1 - 12r_2 + 40r_3 + 40r_4 - 12r_5 + 6r_6 - 2r_7 + rounding$$
*(with rounding either 31, or 32)*
$$p = Max(Min(p >> 6, 0), 255)$$

For H.264/AVC the SUPER_USCALEFIR8UI operation performs the required calculation: filtering, rounding, scaling, and clipping to the range of an unsigned 8-bit integer. For MPEG4/AVC with a rounding value of 32, the same operation can be used. For a rounding factor of 31, the SUPER_FIR8UI can be used to calculate the 8-taps filter, additional operations are required to perform rounding, scaling, and clipping.

**4.2.2. DCT.** This kernel produces a block of frequency coefficients. The 8x8 two-dimensional (2D) DCT is row-column separated into 8-points 1D transforms. We use the Chen algorithm [7]. The algorithm makes frequent use of butterfly and rotate operators. Both operators have two inputs and produce two outputs. The butterfly and rotate operators are defined by:

*Butterfly (input0, input1):*
    *output0 = input0 + input1*
    *output1 = input0 – input1*
*Rotate (input0, input1):*
    *output0 = input0*cos(a) – input1*sin(a)*
    *output1 = input0*sin(a) + input1*cos(a)*

The TM3270 ISA includes new operations that allow for calculation of the 2D DCT with signed 16-bit arithmetic. These operations use dual 16-bit SIMD arithmetic. As a result, two independent 1D DCTs can be calculated in parallel. The butterfly operation uses the dual 16-bit addition and subtraction operations: DSPIDUALADD and DSPIDUALSUB (both operations are present in the original TriMedia ISA). Two independent butterfly structures are calculated in parallel using one DSPIDUALADD and one DSPIDUALSUB operation. The rotate operation uses

the new dual 16-bit SUPER_DUALISCALEMIX operation. Two of these two-slot operations calculate two independent rotate structures in parallel. Since, the operation scales its in-between result by a factor $2^{14}$, the partial results of the DCT calculation can be kept within a signed 16-bit representation. Furthermore, the operation's rounding improves the accuracy of the calculation. A similar operation, in terms of rounding and scaling, is used for the multiplication of partial DCT results: DUALISCALEUI_RZ. The compiler kept the working set of this kernel in registers; i.e. no spill code was generated.

*Intermezzo*. To avoid accuracy problems, the H.264/AVC standard prescribes an *integer* transform for the spatial-to-frequency domain translation and vice verse. It is efficiently implemented with signed 16-bit arithmetic operations DSPIDUALADD and DSPIDUALSUB, and the new SUPER_DUALIMIX operation.

**4.2.3. Quantization.** This kernel produces a block of quantized frequency coefficients, based on a single quantizer value (we assume that the quantization matrix values are all the same[1]). The quantizer value input is the product of the quantization matrix value and the macroblock quantizer value. The MPEG2 standard defines the *de*-quantization for a predicted block by:

*dequant_coeff = ((2 * quant_coeff + k) * quantizer) / 32*
*quantizer = macroblock_quantizer * matrix_quantizer*
*k = Sign (quant_coeff);*
*/: division with truncation to '0'.* *(Equation 1)*

For the de-quantization, an exact implementation of the above definition is required to ensure accuracy. However, for the quantization we decided upon a low complexity approximation. The approximation performs a single multiplication of the frequency coefficient with a pre-computed value based on the quantizer value. It uses the new dual 16-bit DUALISCALEUI_RZ operation (multiplication with scaling, and rounding to zero). This operation performs the quantization of two frequency coefficients. For a total of 64 coefficients, 32 DUALISCALEUI_RZ operations are required. The non-optimized implementation provides the same functionality, but uses 32-bit arithmetic.

The quantization kernel also produces a "coded" value. This value identifies whether all the quantized coefficients have a '0' value. In this case, a possibility exists to shorten the execution path through the texture pipeline.

**Table 2. Kernels and functions, optimized and non-optimized performance.**

| Kernel/ function | Optimized/ (yes/no) Path (full/short) | Old operations | | New Operations* | | | | | Other operations | Total operations | VLIW instructions | Operations/ VLIW instruction |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | LD32 | ST32 | SUPER_LD32R | LD_FRAC8 | DUALADDSUB | SUPER_DUAL ISCALEMIX | DUAL_ ISCALEUI_RZ | | | | |
| Difference calculation | no | 68 | 48 | | | | | | 190 | 306 | 78 | 3.92 |
| | yes | 4 | 48 | 2*8 | 32 | | | | 163 | 263 | 62 | 4.24 |
| Dct | no | 32 | 32 | | | | | | 738 | 802 | 179 | 4.48 |
| | yes | | 32 | 2*16 | | | 2*48 | 32 | 188 | 380 | 103 | 3.69 |
| Quantization | no | 32 | 32 | | | | | | 391 | 455 | 98 | 4.64 |
| | yes | | 32 | 2*16 | | | | 32 | 35 | 131 | 32 | 4.09 |
| Run length | no | 32 | 65 | | | | | | 292 | 357 | 76 | 4.70 |
| Dequantization | no | 32 | 32 | | | | | | 269 | 333 | 74 | 4.50 |
| | yes | | 32 | 2*16 | | 32 | | | 140 | 236 | 55 | 4.39 |
| Idct | no | 32 | 32 | | | | | | 655 | 719 | 162 | 4.44 |
| | yes | | 32 | 2*16 | | | 2*48 | 48 | 189 | 397 | 109 | 3.64 |
| Reconstruction | no | 48 | 16 | | | | | | 126 | 190 | 57 | 3.33 |
| | yes | | 16 | 2*24 | | | | | 140 | 204 | 43 | 4.74 |
| Sum of kernels | no | 276 | 257 | | | | | | 2629 | 3162 | 724 | 4.37 |
| | yes | 4 | 257 | 2*96 | 32 | 32 | 2*96 | 112 | 1147 | 1968 | 480 | 4.10 |
| Text. pipeline | full | 12 | 104 | 2*16 | 32 | 32 | 2*96 | 112 | 1063 | 1579 | 358 | 4.41 |
| Text. pipeline (non-coded) | full | 12 | 92 | 2*16 | 32 | 32 | 2*96 | 112 | 1111 | 1615 | 383 | 4.22 |
| | short | 9 | 36 | 2*16 | 32 | | 2*48 | 64 | 409 | 678 | 196 | 3.46 |
| Text. pipeline (non-coded, 48) | full | 12 | 76 | 2*16 | 32 | 25 | 2*91 | 104 | 1006 | 1469 | 339 | 4.33 |
| | short | 9 | 36 | 2*16 | 32 | | 2*48 | 48 | 397 | 650 | 180 | 3.61 |

* two-slot operations are counted twice, since they occupy two issue slots.

**4.2.4. Run-length encoding.** This kernel processes the quantized coefficients in a zigzag order and produces a sequence of (run, length) pairs. It checks the presence of '0' values, and produces ("run", "length") pairs of quantized coefficients "run", with a "length" value indicating the amount of preceding '0' values in zigzag order. The use of new TM3270 operations, like SUPER_LD32R, does not improve performance.

**4.2.5. Dequantization.** This kernel produces a block of dequantized frequency coefficients, based on an inverse quantizer value (Equation 1). The optimized implementation is performed with dual 16-bit arithmetic, and uses the new dual 16-bit DUALADDSUB operation. This operation is used to add the sign bit to the doubled *quant_coeff*:

```
(2 * dual_quant_coeff + k) =
DUALADDSUB (DSPIDUALADD (
                    dual_quant_coeff, dual_quant_coeff),
                0x00010001)
```

**4.2.6. IDCT.** This kernel produces an block of image pixel difference values. The 8x8 2D IDCT is row-column separated into 8-points 1D transforms. We use a version of the Loeffler algorithm [9]. Like the forward DCT, the algorithm makes frequent use of butterfly and rotate operators. The optimized implementation is performed with dual 16-bit arithmetic. The rounding and scaling capabilities of SUPER_DUALISCALEMIX and DUALISCALEUI_RZ allow for a standard compliant accuracy. The compiler kept the working set of this kernel in registers; i.e. no spill code was generated.

**4.2.7. Image reconstruction.** This kernel produces a block of reconstructed image pixels. The prediction values are represented by 8-bit unsigned integers; the difference values are represented by signed 16-bit integers. The initial steps of the reconstruction are done with 16-bit arithmetic, and the final step clips the results to an unsigned 8-bit integer range. The optimized implementation uses the SUPER_LD32R operation.

**4.2.8. Putting it all together.** The previous sections discussed the implementation of the individual kernels as self-contained functions. The "sum of kernels" rows illustrate the performance improvement: 720 VLIW instructions for the non-optimized, and 480 VLIW instructions for the optimized version. Given the pipelined organization, it is possible to combine the kernels into a single "TexturePipeline" function through function-inlining. This gives the compiler the opportunity to remove function-call, and –return overhead, and to pass in-between kernel results through registers, rather than through memory. Furthermore, the compiler is offered more operation level parallelism, which might allow for a higher operations/VLIW instruction ratio. As a result, the schedule length is significantly reduced: Table 2 gives 480 VLIW instructions for the optimized "sum of kernels", and 358 VLIW instructions for "TexturePipeline" function.

Two further optimizations were investigated: 1). The "coded" value as produced by the quantization kernel was used to identify blocks for which no data needs to be encoded (Table 2: "non-coded"). In this case, the run-length encoding, inverse quantization, and IDCT kernels do not need to be performed. The image reconstruction kernel uses the prediction values as the reconstructed pixels. For non-coded blocks a shorter execution path through the texture pipeline is achieved. Note that this optimization does not compromise image quality. 2). We reduced the amount of encoded quantized frequency coefficients to the first 48 coefficients in zigzag order. The non-coded coefficients are assumed '0', which allows for optimizations in some of the kernels (Table 2: "non-coded, 48"). In [8] it was shown that for low bitrate applications, the encoding of only a few frequency coefficients in the low frequency domain resulted in only limited image quality degradation. Applying both optimizations reduces the schedule length to 180 VLIW instructions for the short execution path.

### 4.3. Bitstream generation

The bitstream generator writes bit patterns to a sequential stream located in memory. It makes frequent use of the "PutBits" function, which takes a bit pattern of a certain size, and writes the bits to the stream. The "PutBits" function does not use any new operations. MPEG2 bit patterns are typically 24 bits or less in size. The TM3270's ability to perform non-aligned memory access allows for an efficient implementation, since a single store operation can update up to 25 sequential bits in memory.

## 5. MPEG2 performance evaluation

We use a cycle accurate C-model of the processor, which was automatically generated from the Verilog HDL model, for performance evaluation. The processor operates at 450 MHz., and is attached to a 32-bit 200 MHz. DDR SDRAM off-chip memory.

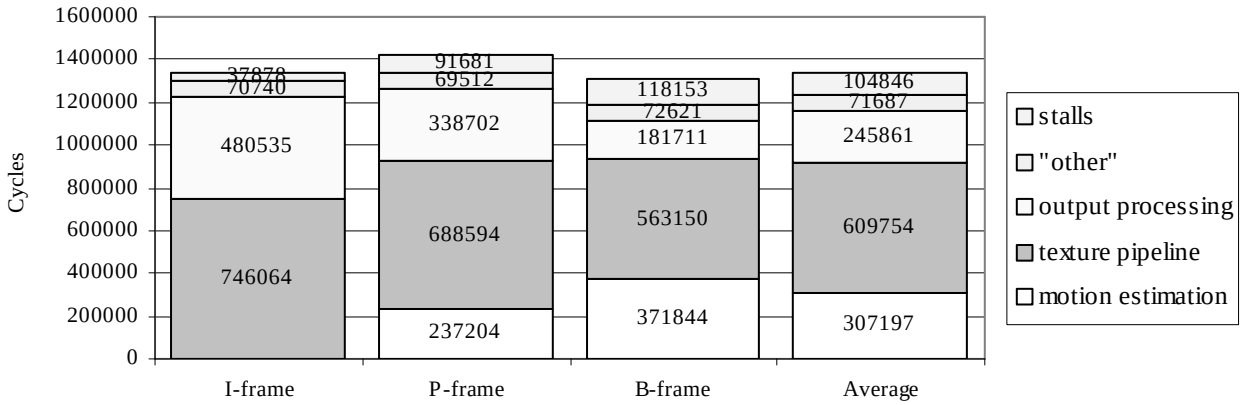


**Figure 3. MPEG2 performance breakdown into major functions.**

We encoded the "Foreman" sequence at CIF resolution at 25 frames per second, with a target bitrate of 500,000 bits per second. The rate control kernel controls the bitrate through the macroblock quantization factor. Frames are represented in a 4:2:0 format, resulting in six block per macroblock. A MPEG2 "group of pictures" (GOP) includes 12 frames, and the frame types in display order are given by the pattern: I-B-B-P-B-B-P-B-B-P-B-B. The motion estimation for P-frames is performed as described in Section 4.1.2 (17 motion vectors are evaluated). For B-frames, motion estimation is performed with two reference images. To balance the complexity of B-frames with that of P-frames, the amount of block matches per reference image is reduced to 10.

Figure 3 gives an overview of the computational complexity. The numbers provide average compute requirements per frame. P- and B-frames have higher complexity, since they require motion estimation. The "average" numbers are calculated based on the frame type frequencies as defined by the GOP pattern. On average, 1,339,223 cycles are required to encode a frame, at 25 frames per second, resulting in a 33.5 MHz. processor load. The cycle budget is divided into a VLIW budget and a stall cycle budget. The instruction budget is divided into: motion estimation, texture pipeline, output processing (which includes the variable length encoding and bitstream generation kernels), and "other". The "other" partition includes MPEG2 control code, including e.g. rate control.

## 6. Conclusion

We have described new operations, and have shown that they can significantly improve the performance of MPEG2 encoder kernels. Collapsed load operations with interpolation allow for a motion estimation function that evaluates 17 macroblock candidates, but only comprises 568 VLIW instructions. This low complexity 3DRS motion estimation algorithm accounts for roughly 25% of the overall budget of the MPEG2 encoder (for B-frames). New multiplication operations allow for a standard compliant implementation of (I)DCT kernels with dual 16-bit arithmetic. The encoder's texture pipeline for a bi-directionally predicted 8x8 block is performed in 358 or less VLIW instructions, through function-inlining of the individual kernels. The optimized kernels account for more than 90% of the overall budget of the MPEG encoder. Using the kernel optimizations, the MPEG2 encoder can be performed in 33.5 MHz. for the evaluated "Foreman" sequence at CIF resolution.

## 7. References

[1] I.E.G. Richardson, H.264 and MPEG-4 video compression, video coding for next-generation multimedia, Wiley, 2003.

[2] J.T.J. van Eijndhoven et al., "TriMedia CPU64 architecture", Proc. of the ICCD, pp. 586-592, October 1999.

[3] J.H. Kuo, C.C. Ho, K.L. Huang, J. Shiu, and J.L. Wu, "A low-cost media-processor based real-time MPEG-4 video decoder", IEEE Trans. on Consumer Electronics, vol. 49, no. 4, pp. 1488-1497, November 2003.

[4] S. Rathnam, and G. Slavenburg, "An architectural overview of the programmable multimedia processor, tm-1", Proc. of the COMPCON, pp. 319-326, 1996.

[5] T. Halfhill, "Philips powers up for video", Microprocessor Report, http://www.mpronline.com/, November 2003.

[6] G. de Haan et al., "True-motion estimation with 3-D recursive search block matching", ICCE Trans. on Circuits and Systems for Video Technology, vol. 3, pp. 368-379, October 1993.

[7] W. Chen, C. Harrison, and S.C. Fralick, "A fast computational algorithm for the discrete cosine transform", IEEE Trans. on Communications, vol. COM-25, no. 9, pp. 1004-1011, September 1977.

[8] B. Girod, and K.W. Stuhlmuller, “A content-dependent fast DCT for low bit-rate video coding”, Proc. of the ICIP, vol. 3, pp. 80-84, October 1998.

[9] C. Loeffler, A. Ligtenberg, and G.S. Moschytz, “Practical fast 1-D DCT algorithm with 11 multiplications”, Proc. of the ICASSP, pp. 988-991, May 1989.

**Table 3. Some of the TM3270 new operations.**

| Operation | DESCRIPTION |
|---|---|
| SUPER_LD32R<br>rsrc3 rsrc4 -> rdest1 rdest2<br><br>Semantics: Load two 32-bit words.<br>Note: description is in big endian mode. | data0 = Mem[rsrc1]; data1 = Mem[rsrc1 + 1]; data2 = Mem[rsrc1 + 2]<br>data3 = Mem[rsrc1 + 3]; data4 = Mem[rsrc1 + 4]; data5 = Mem[rsrc1 + 5]<br>data6 = Mem[rsrc1 + 6]; data7 = Mem[rsrc1 + 7]<br>rdest1 = (data0 << 24) \| (data1 << 16) \| (data2 << 8) \| data3<br>rdest2 = (data4 << 24) \| (data5 << 16) \| (data6 << 8) \| data7 |
| LD_FRAC8<br>rsrc1 rsrc2 -> rdest1<br><br>Semantics: Collapsed load; load combined with linear interpolation.<br>Note: description is in big endian mode. | data0 = Mem[rsrc1]; data1 = Mem[rsrc1 + 1]; data2 = Mem[rsrc1 + 2]<br>data3 = Mem[rsrc1 + 3]; data4 = Mem[rsrc1 + 4]<br>rdest1[31:24] = (data0*(16-rsrc2[3:0]) + data1*rsrc2[3:0] + 8) >> 4<br>rdest1[23:16] = (data1*(16-rsrc2[3:0]) + data2*rsrc2[3:0] + 8) >> 4<br>rdest1[15:8] = (data2*(16-rsrc2[3:0]) + data3*rsrc2[3:0] + 8) >> 4<br>rdest1[7:0] = (data3*(16-rsrc2[3:0]) + data4*rsrc2[3:0] + 8) >> 4 |
| LD_PACKFRAC8<br>rsrc1 rsrc2 -> rdest1<br><br>Semantics: Collapsed load; load combined with linear interpolation.<br>Note: description is in big endian mode. | data0 = Mem[rsrc1]; data1 = Mem[rsrc1 + 1]; data2 = Mem[rsrc1 + 2]<br>data3 = Mem[rsrc1 + 3]; data4 = Mem[rsrc1 + 4]; data5 = Mem[rsrc1 + 5]<br>data6 = Mem[rsrc1 + 6]; data7 = Mem[rsrc1 + 7]<br>rdest1[31:24] = (data0*(16-rsrc2[3:0]) + data1*rsrc2[3:0] + 8) >> 4<br>rdest1[23:16] = (data2*(16-rsrc2[3:0]) + data3*rsrc2[3:0] + 8) >> 4<br>rdest1[15:8] = (data4*(16-rsrc2[3:0]) + data5*rsrc2[3:0] + 8) >> 4<br>rdest1[7:0] = (data6*(16-rsrc2[3:0]) + data7*rsrc2[3:0] + 8) >> 4 |
| SUPER_QUADUSCALEMIXUI<br>rsrc1 rsrc2 rsrc3 rsrc4 -> rdest1 | temp = rsrc1[31:24] * rsrc2[31:24] + rsrc3[31:24] * rsrc4[31:24]<br>rdest1[31:24] = Max (Min ((temp + 0x20) >> 6, oxff), 0)<br>temp = rsrc1[23:16] * rsrc2[23:16] + rsrc3[23:16] * rsrc4[23:16]<br>rdest1[23:16] = Max (Min ((temp + 0x20) >> 6, oxff), 0)<br>temp = rsrc1[15:8] * rsrc2[15:8] + rsrc3[15:8] * rsrc4[15:8]<br>rdest1[15:8] = Max (Min ((temp + 0x20) >> 6, 0xff), 0)<br>temp = rsrc1[7:0] * rsrc2[7:0] + rsrc3[7:0] * rsrc4[7:0]<br>rdest1[7:0] = Max (Min ((temp + 0x20) >> 6, 0xff), 0) |
| SUPER_USCALEFIR8UI<br>rsrc1 rsrc2 rsrc3 rsrc4 -> rdest1 | temp = rsrc1[31:24] * rsrc2[31:24] + rsrc1[23:16] * rsrc2[23:16]<br>+ rsrc1[15:8] * rsrc2[15:8] + rsrc1[7:0] * rsrc2[7:0]<br>+ rsrc3[31:24] * rsrc4[31:24] + rsrc3[23:16] * rsrc4[23:16]<br>+ rsrc3[15:8] * rsrc4[15:8] + rsrc3[7:0] * rsrc4[7:0]<br>rdest1 = Max (Min ((temp + 0x20) >> 6, 0xff), 0) |
| SUPER_IFIR8UI<br>rsrc1 rsrc2 rsrc3 rsrc4 -> rdest1 | rdest1 = rsrc1[31:24] * rsrc2[31:24] + rsrc1[23:16] * rscr2[23:16]<br>+ rsrc1[15:8] * rsrc2[15:8] + rsrc1[7:0] * rscr2[7:0]<br>+ rsrc3[31:24] * rsrc4[31:24] + rsrc3[23:16] * rscr4[23:16]<br>+ rsrc3[15:8] * rsrc4[15:8] + rsrc3[7:0] * rscr4[7:0] |
| SUPER_DUALISCALEMIX<br>rsrc1 rsrc2 rsrc3 rsrc4 -> rdest1 | temp = rsrc1[31:16] * rsrc2[31:16] + rsrc3[31:16] * rsrc4[31:16]<br>rdest1[31:16] = Max (Min ((temp + 0x2000) >> 14, 0x7fff), -0x8000)<br>temp = rsrc1[15:0] * rsrc2[15:0] + rsrc3[15:0] * rsrc4[15:0]<br>rdest1[15:0] = Max (Min ((temp + 0x2000) >> 14, 0x7fff), -0x8000) |
| SUPER_DUALIMIX<br>rsrc1 rsrc2 rsrc3 rsrc4 -> rdest1 rdest 2 | rdest1 = rsrc1[31:16] * rsrc2[31:16] + rsrc3[31:16] * rsrc4[31:16]<br>rdest2 = rsrc1[15:0] * rsrc2[15:0] + rsrc3[15:0] * rsrc4[15:0] |
| DUALISCALEUI_RZ<br>rsrc1 rsrc2 -> rdest1<br><br><br><br>Note: “RZ”, rounding to zero. | temp = rsrc1[31:16] * rsrc2[31:16]<br>rounding = (rsrc2[31:16] < 0) ? 0x3fff : 0x0000; /* "towards zero" */<br>rdest1[31:16] = Max (Min ((temp+rounding) >> 14, 0x7fff), - 0x8000)<br>temp = rsrc1[15:0] * rsrc2[15:0]<br>rounding = (rsrc2[15:0] < 0) ? 0x3fff : 0x0000; /* "towards zero" */<br>rdest1[15:0] = Max (Min ((temp+rounding) >> 14, 0x7fff), - 0x8000) |
| DUALADDSUB<br>rsrc1 rsrc2 -> rdest1 | rdest1[31:16] = (rsrc1[31:16] == 0) ? rsrc1[31:16]<br>: ((rsrc1[31:16] > 0) ? rsrc1[31:16] + rsrc2[31:16]<br>: rsrc1[31:16] - rsrc2[31:16])<br>rdest1[15:0] = (rsrc1[15:0] == 0) ? rsrc1[15:0]<br>: ((rsrc1[15:0] > 0) ? rsrc1[15:0] + rsrc2[15:0]<br>: rsrc1[15:0] - rsrc2[15:0]) |
| SUPER_DUALIMEDIAN<br>rsrc1 rsrc2 rsrc3 -> rdest1; | rdest1[31:16] = Min (Max (Min (rsrc1[31:16], rsrc2[31:16]), rsrc3[31:16]),<br>Max (rsrc1[31:16], rsrc2[31:16]))<br>rdest1[15:0] = Min (Max (Min (rsrc1[15:0], rsrc2[15:0]), rsrc3[15:0]),<br>Max (rsrc1[15:0], rsrc2[15:0])) |