On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x

Eduardo Asbun and Chiouguey Chen

Texas Instruments, Inc.

Abstract

This application report describes the implementation of MPEG-4 motion compensation on the Texas Instruments (TI™) TMS320C62x digital signal processor (DSP). MPEG-4 is a standard for coding of audiovisual information being developed by the Motion Picture Experts Group (MPEG). MPEG-4 became an International Standard in December 1998. Motion compensation is a basic component of MPEG-4 and other video compression standards such as MPEG-1, MPEG-2, H.261, H.263, and H.263+. Code development flow to increase performance is discussed. Implementation issues, such as memory access pattern and code size versus performance, are also examined.
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x
Introduction

This application report describes the implementation of MPEG-4 motion compensation on the TMS320C62x. MPEG-4 [1] is an ISO/IEC standard for coding of audiovisual information being developed by the Motion Picture Experts Group (MPEG). Tools are being specified to support functionalities such as interactivity between you and the application, universal accessibility, and high degree of compression. Motion compensation is a basic component MPEG-4 video and other video compression standards such as MPEG-1, MPEG-2, H.261, H.263, and H.263+. Therefore, the work presented in this application report is relevant to implementations of any of these standards on the 'C62x.

TMS320C62x Fixed-Point DSP

The TMS320C62x devices are fixed-point DSPs that feature the VelociTI™ architecture [2]. The VelociTI architecture is a high-performance, advanced, very-long-instruction-word (VLIW) architecture developed by Texas Instruments. VelociTI, together with the development tool set and evaluation tools, provides faster development time and higher performance for embedded DSP applications through increased instruction-level parallelism.

DSPs such as the 'C62x are well suited for the implementation of motion compensation. This application presents a high degree of parallelism that can be exploited to meet real-time requirements typical of video conferencing applications. The operations required by motion compensation, such as bilinear interpolation, are readily implemented on a fixed-point DSP.

Block-Based Video Compression

Contiguous frames in a video sequence have a high degree of temporal correlation, as they are taken at small intervals of time (typically 10-30 frames per second). In order to achieve a significant ratio of compression, temporal redundancy between frames is exploited using a compression technique known as interframe coding. As described in the Motion Estimation and Motion Compensation section, a frame is selected as a reference, and subsequent frames are predicted from it.

Interframe coding, however, does not work well for frames that exhibit low temporal correlation, such as scene changes. In this case a different model is used, where spatial correlation between adjacent pixels is exploited. This technique is known as intraframe coding. Pixels are transformed into another domain in order to reduce redundancy and to compact the energy of the signal. The Discrete Cosine Transform and the Wavelet Transform, among others, have been used for this purpose.

The color space used in this implementation of MPEG-4 Motion Compensation [3] is YUV with 4:2:0 chrominance subsampling. The chrominance components are subsampled by a factor of 2 in both the horizontal and vertical directions, as shown in Figure 1. A pixel uses one byte, and can take a value between 0 and 255.
Figure 1. YUV 4:2:0 Chrominance Subsampling

For the purpose of motion estimation and compensation, a frame is partitioned into macroblocks, 16 × 16 arrays of nonoverlapping luminance (Y) pixels together with one 8 × 8 block of spatially corresponding pixels for each of the chrominance components (U and V). A macroblock of luminance pixels is composed of four 8 × 8 blocks of pixels.

MPEG-4 makes provisions to handle a variety of frame input formats. Our implementation targets the Common Intermediate Format (CIF), 352 × 288 pixels in the luminance (Y) component. Other formats (QCIF, 4CIF, ITU-R 601, etc) can be supported as the frame size is an input parameter to our program.

A CIF-sized YUV 4:2:0 frame occupies 101 376 bytes for the Y component, and 25 344 bytes for each of the chrominance (U and V) components, for a total of 152 064 bytes. Pixels in a frame are stored row-wise in memory; that is, pixel (i,j) is adjacent in memory to pixels (i,j−1) and (i,j+1). The luminance component (Y) is stored first, followed by the U and V (chrominance) components.

Motion Estimation and Motion Compensation

The movement of a macroblock from the reference frame to the current frame is determined using a technique known as motion estimation [4, 5]. A search of a macroblock in the current frame is conducted over a portion (or all) of the reference frame. The best matching macroblock (under certain criteria) is selected, and a motion vector is obtained, as shown in Figure 2. A motion vector consists of horizontal and vertical components. This motion vector can be expressed in integer or half-pixel accuracy. Half-pixel accuracy corresponds to bilinear interpolation.
For regions where one motion vector per macroblock does not provide with an accurate description of the translation of the pixels in a macroblock, four motion vectors per macroblock (one motion vector per block) can be used. This mode may be used on a macroblock-by-macroblock basis.

In MPEG-4, a mode known as *unrestricted motion vectors* is used. In this mode, motion vectors can point to positions outside the reference frame. This mode is particularly useful for scenes with higher degree of motion, when objects move around the edges entering and exiting the frame.

A *predictive frame* is constructed from the motion vectors obtained for all macroblocks in the frame. Macroblocks from the reference frame are replicated at the new locations indicated by the motion vectors. This technique is known as *motion compensation*. A *predictive error frame* (PEF) is calculated by taking the difference between the current and predicted frames, and intraframe encoded. Since the energy of the PEF is likely to be low, the amount of bits necessary for its encoding is small.

Motion compensation is used both at the encoder and the decoder to produce a motion compensated version of the current frame. This motion compensated frame is reconstructed by using the predicted frame and the PEF. Since motion compensation is used at both ends of the video codec and it is a time-consuming task, an efficient implementation is of the essence.

**Implementing Motion Compensation on the 'C62x**

The motion compensation module receives the frame size, a set of motion vectors, and the reference frame as input. For each macroblock, there is one or four motion vectors. In our implementation, motion compensation is done on a block-by-block basis. Therefore, when one motion vector per macroblock is used, the motion compensation routine is called four times with same motion vector for each block in the macroblock. This is done to reduce code size. However, our implementation can be extended to operate on macroblocks.
A C language version of motion compensation (referred to as “C code”) was implemented first. The goal was to validate the code and verify correctness by using actual sequences and motion vector data. The sequences were reconstructed using the C code, and compared to the reconstruction done by an implementation of H.263 from which the motion vector data was obtained.

The C code was profiled to identify the portions of the code where performance could be increased. Intrinsics were introduced to produce a “Natural C” version of the code. Intrinsics are special functions that map directly to inlined ‘C62x/C67x instructions. “_nassert” was used to indicate the optimizer that loops were executed a certain number of times.

“Optimized C” was obtained after refining the Natural C code, using code transformations and unrolling certain loops. For example, when a block of pixels is copied from one location in memory to another, four pixels can be copied at a time by casting a pointer to byte into a pointer to integer. Other optimizations are described in the following sections.

The critical performance areas of the Optimized C code were isolated. Linear assembly code (assembly code that has not been register-allocated and is unscheduled) was written for these loops. See the TMS320C62x/C67x Programmer’s Guide for more information about code development flow.

The decision of using linear assembly was based on the following considerations:

- Computationally intensive routines of the code can be written as C callable routines
- It gives the programmer greater control over the use of the resources
- Can pass memory bank information to the tools
- The code can be recompiled to make use of improved versions of the Code Generation Tools as they become available
- The performance obtained with linear assembly is very close to the performance that would be obtained using hand-optimized assembly
- It is easier to write and debug than hand-optimized assembly
- It allows the code to be reused in other ’C6x platforms

Our implementation of motion compensation supports integer and half-pixel accuracy. Therefore, four cases are considered:

- Case A: Integer accuracy
- Case B: Half-pixel accuracy in the horizontal direction
- Case C: Half-pixel accuracy in the vertical direction
- Case D: Half-pixel accuracy in both the horizontal and vertical directions

These four cases are illustrated in Figure 3.
The bilinear interpolation scheme is:

\begin{align*}
a &= A \\
b &= \frac{(A + B + 1 - \text{rounding\_type})}{2} \\
c &= \frac{(A + C + 1 - \text{rounding\_type})}{2} \\
d &= \frac{(A + B + C + D + 2 - \text{rounding\_type})}{4}
\end{align*}

where "/" denotes division by truncation and \text{rounding\_type} takes the value of 0 or 1.

These four cases represent the core of the computation in motion compensation and are the kernels that have been optimized for the ‘C62x. Code for cases a, b, c, and d (C code, Natural C, Optimized C, Linear Assembly, and Assembly Code generated by the Code Generation Tools) is shown in Appendices A, B, C, and D, respectively. The complete C code for motion compensation, including driver programs, is shown in Appendix E.

Memory bank conflicts is an issue that can impact the performance of the implementation of motion compensation. The ‘C62x has four memory banks, 2-bytes-wide each. Because each bank is single-ported memory, only one access to each bank is allowed per cycle. Two accesses to a single bank in a given cycle result in a memory stall that halts all pipeline operation for one cycle, while the second value is read from memory. (See the TMS320C62x/C67x Programmer’s Guide for more information about internal memory banks of the C6000 family.) In our implementation, care has been taken to minimize memory bank conflicts by staggering accesses to data in internal memory. We assume that both the current and reference blocks are in internal memory.

The motion vectors are represented in Q1 notation; that is, the decimal point is placed between bits 0 and 1. This notation allows handling of half-pixel accuracy motion vector information. For example, “0000 0011b” represents the number 1.5.
Case A: Integer Accuracy

Motion compensation case A involves copying a block of pixels from the reference frame into the current frame. The corresponding C language implementation is shown in Example 1.

Example 1. C Language Implementation of Case A

```c
for (m=0; m<8; m++) {
    for (n=0; n<8; n++) {
        current_block[m][n] = reference_block[m][n];
    }
}
```

An improvement over the original C code would be to transfer four pixels at a time (one word) by casting a pointer to byte into a pointer to word (4 bytes). Also, loop unrolling (used on the inner loop) would improve performance. However, writing linear assembly gives better control over how memory accesses are performed.

Since a motion vector can point to any pixel in the reference frame, the alignment of the reference block is not known. When a word is loaded, if the word address is not aligned on a word boundary, an incorrect value is loaded. Therefore, the worst case (byte alignment) needs to be assumed, and four cases had to be considered as shown in Figure 4.

Figure 4. Four Possible Memory Alignments for a Reference Block

One possible solution to the memory alignment problem in the implementation of case A would be to transfer one byte at a time as in the original C code. A more efficient strategy is to copy a row of eight pixels at a time. To assure correct memory alignment, three words (which always contain all eight pixels) are read. Since the memory address of the reference block is known, the corresponding case in Figure 4 can be determined. By manipulating the words accordingly, it is possible to pack all eight pixels in a row into two words that can be stored in memory. An example of an access to a reference block, which is byte aligned, is shown in Figure 5. The linear assembly code for case A is shown in Appendix A.
In Figure 5, the bytes in memory are shown in Big Endian order, while the bytes in registers are shown in Little Endian order.

Case B: Half-Pixel Accuracy in the Horizontal Direction

Motion compensation case B involves bilinear interpolation according to the formulas previously shown in the Implementing Motion Compensation on the C62x section. The corresponding C language implementation is shown in Example 2.

Example 2. C Language Implementation of Case B

```c
for (m=0; m<8; m++) {
    for (n=0; n<8; n++) {
        current_block[m][n] = (reference_block[m][n] + reference_block[m][n+1] + 1 - rounding_type) / 2;
    }
}
```

The first improvement on the C code is to perform division by 2 using right shift by one position. The standard calls for division by truncation; therefore, right shift is a fast, correct implementation. Another improvement is to compute a constant that replaces the term 1 - rounding_type, saving one substraction per iteration.

Due to memory alignment issues similar to case A, the strategy chosen was to interpolate one pixel at a time. To produce efficient code, the inner loop was unrolled in order to convert the nested loop into a single loop. The reason is that the code generation tools produce more efficient code for single loops than for nested loops.
A row of pixels resides in contiguous memory locations. Therefore, the access pattern to internal memory can be predetermined in order to avoid memory bank hits. The linear assembly code for case B is shown in Appendix B.

**Case C: Half-Pixel Accuracy in the Vertical Direction**

Motion compensation case C, as in case B, involves bilinear interpolation according to the formulas previously shown in the Implementing Motion Compensation on the 'C62x section. The corresponding C language implementation is shown in Example 3.

**Example 3. C Language Implementation of Case C**

```c
for (m=0; m<8; m++) {
    for (n=0; n<8; n++) {
        current_block[m][n] = (reference_block[m][n]
            + reference_block[m+1][n]
            + 1 - rounding_type) / 2;
    }
}
```

Pixels located in rows above and below a current pixel do not reside in adjacent memory locations, they are located at a distance equal to the frame width. Since the width a CIF-size frame is a multiple of 16 (and, thus, a multiple of 8), these pixels reside in the same memory bank. Therefore, if read in the same cycle, a pipeline stall would occur.

To solve this problem, a strategy similar to case B was used: pixels are processed column-by-column, creating a single loop. An advantage of this strategy is that it facilitates the job of the code generation tools. The linear assembly code for case C is shown in Appendix C.

**Case D: Half-Pixel Accuracy in Both the Horizontal and Vertical Directions**

In motion compensation case D, four pixels from the reference frame are used to produce one pixel in the current frame. The corresponding C language implementation is shown in Example 4.

**Example 4. C Language Implementation of Case D**

```c
for (m=0; m<8; m++) {
    for (n=0; n<8; n++) {
        current_block[m][n] = (reference_block[m][n]
            + reference_block[m][n+1]
            + reference_block[m+1][n]
            + reference_block[m+1][n+1]
            + 2 - rounding_type) / 4;
    }
}
```
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x

The strategy used is similar to case B, that is, pixels are processed row-wise. The inner loop is unrolled to create a single loop, and two pointers are used to access pixels in contiguous rows. Due to software pipelining, accesses to contiguous rows do not cause memory bank conflicts, because the instructions get reordered. Even though a single pointer would have been sufficient, the use of two pointers for loading pixels from memory indicates to the code generation tools the independence between load instructions. The linear assembly code for case D is shown in Appendix D.

Code Benchmarks

Benchmarks for the code described in the Implementing Motion Compensation on the ‘C62x section are summarized in Table 1. The code benchmarked is shown in Appendices A-D. These benchmarks were obtained using Code Generation Tools version 2.10. The options used for the compiler were: -o3 –mt –pm. See the TMS320C6000 Optimizing C Compiler User’s Guide (SPRU187) for more information about the compiler tools.

Table 1. Code Benchmarks for Motion Compensation of an 8 x 8 Block

<table>
<thead>
<tr>
<th>Motion Compensation</th>
<th>C code</th>
<th>Natural C</th>
<th>Optimized C</th>
<th>Linear Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clock</td>
<td>Code size (FPs)</td>
<td>Clock</td>
<td>Code size (FPs)</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td></td>
<td>count</td>
<td></td>
</tr>
<tr>
<td>Case A</td>
<td>574</td>
<td>4</td>
<td>571</td>
<td>4</td>
</tr>
<tr>
<td>Case B</td>
<td>1023</td>
<td>6</td>
<td>1020</td>
<td>6</td>
</tr>
<tr>
<td>Case C</td>
<td>1023</td>
<td>8</td>
<td>1020</td>
<td>7</td>
</tr>
<tr>
<td>Case D</td>
<td>1346</td>
<td>9</td>
<td>1341</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† fetch packets

For case A, there is a large difference in code size between Natural C and Optimized C because of loop unrolling. The same effect is present in cases B, C, and D, between Optimized C and Linear Assembly. For case D, the difference in code size is larger because two rows are being processed at a time.

Conclusions

In this application report we have presented an implementation of MPEG-4 motion compensation on the ‘C62x. C code was written and validated, and the most computationally intensive portions of the code were optimized for the ‘C62x. Linear assembly code was written for certain routines to maximize performance.

Linear assembly has several advantages over C and hand-optimized assembly:

- It gives the programmer greater flexibility than C to access the resources available in the DSP
- It reduces the overhead of calling a C function
- It is easier to write and debug than assembly
- The code generation tools are able to generate very efficient code, with performance almost as good as hand-optimized assembly
Imaging applications exhibit a high degree of parallelism that can be exploited by VLIW architectures such as the 'C6x. Although the code generation tools are able to produce efficient code, it is important to identify the portions of the code where the program spends most of the time. Improving the performance of this code can boost the overall performance of the application.

The advantage of using software pipelining in this application is limited by the number of times a loop is performed. Since a block is processed at a time, the trip count for both inner and outer loops is 8. Consider a nested loop, where the inner loop is software pipelined. The kernel of the loop is executed only a small number of times, because the inner loop has to be unrolled several times to software pipeline the loop. Therefore, the DSP is not being used at full capacity. To solve this problem, the outer loop could be folded into the inner loop to maximize the number of cycles at which the 'C62x runs at top performance. This solution would require to write hand-optimized code.

Another way to further improve performance would be to add instructions to the 'C6x that can operate on pixels. For example, an instruction that can add four pairs of pixels would accelerate the interpolation step in motion compensation.

The size of internal memory available to the CPU is an issue that can impact the performance of applications that operate on large amounts of data, such as video coding. Although this issue was not addressed in this application report (it was assumed that data is already in internal memory), it is important to devise a strategy to manage data so that it is available to the CPU when needed. The efficiency of this strategy will determine the overall performance of the implementation of MPEG-4 motion compensation.
References


2) TMS320C62x/C67x Technical Brief, [SPRU197], April 1998.


Appendix A  Motion Compensation Code: Case A

C Code

/* Integer accuracy: copy block */
void MC_case_a(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size)
{
    int m, n;
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = ref[r_x+m][r_y+n];
        }
    }
}

Natural C Code

/* Integer accuracy: copy block */
void MC_case_a(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size)
{
    int m, n;
    _nassert(size==8);
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = ref[r_x+m][r_y+n];
        }
    }
}

Optimized C Code

/* Integer accuracy: copy block */
void MC_case_a(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size)
{
    int m;
    _nassert(size==8);
    for(m=0; m<size; m++) {
        curr[c_x+m][c_y    ] = ref[r_x+m][r_y    ];
        curr[c_x+m][c_y+1] = ref[r_x+m][r_y+1];
        curr[c_x+m][c_y+2] = ref[r_x+m][r_y+2];
        curr[c_x+m][c_y+3] = ref[r_x+m][r_y+3];
        curr[c_x+m][c_y+4] = ref[r_x+m][r_y+4];
        curr[c_x+m][c_y+5] = ref[r_x+m][r_y+5];
        curr[c_x+m][c_y+6] = ref[r_x+m][r_y+6];
        curr[c_x+m][c_y+7] = ref[r_x+m][r_y+7];
    }
}
Linear Assembly

; Linear Assembly version of "MC_case_a"

_def        _MC_case_a
.sect        ".text"
_MC_case_a:   .cproc       ref, curr, r_x, c_x, r_y, c_y, num_cols
.reg    r_temp1, r_temp2, c_temp1, c_temp2
.reg    p_r, p_c, np_r
.reg    lshift, rshift, count
.reg    r_w1, r_w2, r_w3, r_w4
.reg    temp

; Calculate pointers "p_c" and "p_r"

SHL   r_x, 0x05, r_temp1   ; r_temp1 = r_x * NUM_COLS
SHL   c_x, 0x05, c_temp1   ; c_temp1 = c_x * NUM_COLS
ADD   r_y, ref,  r_temp2     ; r_temp2 = ref + r_y
ADD   c_y, curr, c_temp2     ; c_temp2 = curr + c_y
ADD   r_temp1, r_temp2, p_r  ; p_r = r_temp1 + r_temp2
ADD   c_temp1, c_temp2, p_c  ; p_c = c_temp1 + c_temp2
SUB   num_cols, 2, num_cols  ; To update np_r and p_c

; Initialize loop counter
MVK   8, count               ; Loop performed 8 times

; Obtain distance for shifting
MVK   0xFFFC, temp           ; To obtain two LSBits
AND   p_r,    temp, np_r     ; Word-aligned access only
AND   p_r,  0x0003, rshift   ; Two LSB define alignment
SUB.L 0x04, rshift, lshift   ; Obtain "lshift"
SHL   rshift, 0x03, rshift   ; # bits for left  shift
SHL   lshift, 0x03, lshift   ; # bits for right shift

; Loop
loop:   .trip 8
LDW   *np_r++[1], r_w1        ; Load first  word
LDW   *np_r++[1], r_w2        ; Load second word
LDW   *np_r++[num_cols], r_w3 ; Load third  word
SHRU  r_w1, rshift, r_w1     ; Make room for LSByte
SHL   r_w3, lshift, r_w3     ; Make room for MSByte
SHL   r_w2, lshift, r_w4     ; Get LSByte
SHRU  r_w2, rshift, r_w2     ; Get MSByte
OR    r_w1, r_w4, r_w1       ; Obtain actual 1st word
OR    r_w2, r_w3, r_w2       ; Obtain actual 2nd word
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x

```
STW r_w1, *p_c++[1] ; Store 1st word
STW r_w2, *p_c++[num_cols] ; Store 2nd word
ADD p_c, 4, p_c ; Since num_cols is short by a word
[count] SUB count, 1, count ; Loop back
[count] B loop
.endproc
```

Output from Assembly Optimizer

```
;**********************************************************************
;* TMS320C6x ANSI C Codegen                         Version 2.10.beta *
;* Date/Time created: Fri Aug 21 11:57:22 1998                        *
;**********************************************************************

;**********************************************************************
;* GLOBAL FILE PARAMETERS                                             *
;*                                                                    *
;*   Architecture        : TMS320C6200                                *
;*   Endian              : Little                                     *
;*   Interrupt Threshold : Disabled                                   *
;*   Memory Model        : Small                                      *
;*   Speculative Load    : Threshold = 0                              *
;*   Redundant Loops     : Enabled                                    *
;*   Pipelining          : Enabled                                    *
;*   Debug Info          : No Debug Info                              *
;*                                                                    *
;**********************************************************************

FP .set A15
DP .set B14
SP .set B15
.global $bss

; Linear Assembly version of "MC_case_a"
.def _MC_case_a
.sect ".text"
.sect ".text"

;**********************************************************************
;* FUNCTION NAME: _MC_case_a                                          *
;*                                                                    *
;*   Regs Modified     : A0,A3,A4,A5,A6,A7,A8,B0,B1,B4,B5,B6,B7,B8,B9 *
;*   Regs Used  : A0,A3,A4,A5,A6,A7,A8,A10,B0,B1,B3,B4,B5,B6,B7,B8,B9 *
;**********************************************************************

_MC_case_a:
** ------------------------------------------------------------------*

;**-- ---------------------------------------------------------------*
;_MC_case_a: .cproc ref, curr, r_x, c_x, r_y, c_y, num_cols
;_reg r_temp1, r_temp2, c_temp1, c_temp2
;_reg p_r, p_c, np_r
;_reg lshift, rshift, count
;_reg r_w1, r_w2, r_w3, r_w4
;_reg temp
```
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x
Searching for software pipeline schedule at ...  

\( ii = 4 \)  Schedule found with 4 iterations in parallel  

Done 

Epilog not removed : Speculative load is beyond user threshold  

Speculative Load Threshold : Unknown  

L2:  ; PIPED LOOP PROLOG

; loop:  .trip 8

\[
\begin{align*}
\text{ADD} & .L2 \ 4,B4,B4 \quad ^9 \\
\text{LDW} & .D1T1 \ *A0++,A3 \quad ^{44} \quad \text{Load first word} \\
\text{ADD} & .L2 \ 4,B4,B4 \quad ^9 \\
\text{LDW} & .D2T2 \ *B4++[B7],B6 \quad ^{46} \quad \text{Load third word} \\
\text{LDW} & .D1T1 \ *A0++,A5 \quad ^{45} \quad \text{Load second word} \\
\text{MV} & .L1X \ B4,A0 \quad ^9 \\
\text{ADD} & .L2 \ 4,B4,B4 \quad @ ^9 \\
\text{LDW} & .D1T1 \ *A0++,A3 \quad @ ^{44} \quad \text{Load first word} \\
\text{ADD} & .L2 \ 4,B4,B4 \quad @ ^9 \\
\text{SHRU} & .S1 \ A3,A6,A3 \quad ^{48} \quad \text{Make room for LSByte} \\
\text{LDW} & .D2T2 \ *B4++[B7],B6 \quad @ ^{46} \quad \text{Load third word} \\
\text{LDW} & .D1T1 \ *A0++,A5 \quad @ ^{45} \quad \text{Load second word} \\
\text{SHRU} & .S2X \ A5,B9,B8 \quad ^{51} \quad \text{Get MSByte} \\
\text{SHL} & .S1 \ A5,A7,A4 \quad ^{50} \quad \text{Get LSByte} \\
\text{MV} & .L1X \ B4,A0 \quad @ ^9 \\
\text{OR} & .L1 \ A3,A4,A4 \quad ^{53} \quad \text{Obtain actual 1st word} \\
\text{B} & .S2 \ loop \quad ^{62} \\
\text{ADD} & .L2 \ 4,B4,B4 \quad @ @ ^9 \\
\text{SHL} & .S1X \ B6,A7,A5 \quad ^{49} \quad \text{Make room for MSByte} \\
\text{LDW} & .D1T1 \ *A0++,A3 \quad @ @ ^{44} \quad \text{Load first word} \\
\text{ADD} & .L2 \ 4,B4,B4 \quad @ @ ^9 \\
\text{STW} & .D1T1 \ A4,*A8++ \quad ^{56} \quad \text{Store 1st word} \\
\text{STW} & .D2T2 \ B8,*B5++[B7] \quad ^{57} \quad \text{Store 2nd word} \\
\text{SUB} & .L2 \ B0,0x1,B0 \quad @ ^{61} \quad \text{Loop back} \\
\text{SHRU} & .S2X \ A5,B9,B8 \quad @ ^{51} \quad \text{Get MSByte} \\
\text{SHL} & .S1 \ A5,A7,A4 \quad @ ^{50} \quad \text{Get LSByte} \\
\text{MV} & .L1X \ B4,A0 \quad @ @ ^9
\end{align*}
\]

On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x
ADD  .D2  0x4,B5,B5    ; ^|59| Since num_cols
  ; is short by a word
MV   .S1X  B5,A8     ; ^|9|
OR   .L1   A3,A4,A4   ; @|53| Obtain actual 1st word
[ B0] B .S2   loop     ; @ ^|62|
ADD  .L2  4,B4,B4     ; @ @ @ ^|9|
ADD  .L1  4,A8,A8     ; ^|9|
SHL  .S1X  B6,A7,A5   ; @ |49| Make room for MSByte
LDW  .D1T1  *A0++,A3  ; @ @ @ ^|44| Load first word
ADD  .L2  4,B4,B4     ; @ @ @ ^|9|
SHL  .S1X  B6,A7,A5   ; @ ^|50|  Make room for MSByte
LDW  .D1T1  *A0++,A3  ; @ @ @ ^|44| Load first word
STW  .D1T1  A4,*A8++  ; @ ^|56|  Store 1st word
STW  .D2T2  *B4++[B7],B6  ; @ @ @ ^|46| Load third word
SHR   .S2X  A5,B9,B8  ; @ @ |51| Get MSByte
SHL   .S1  A5,A7,A4   ; @ @ |50| Get LSBYTE
MV    .L1X  B4,A0     ; @ @ |51| Get LSBYTE
ADD  .D2  0x4,B5,B5    ; @ ^|59| Since num_cols is
  ; short by a word
MV   .S1X  B5,A8     ; @ ^|9|
OR   .L1   A3,A4,A4   ; @ @ |53| Obtain actual 1st word
ADD  .L1  4,A8,A8     ; @ ^|9|
SHL   .S1X  B6,A7,A5   ; @ |49| Make room for MSByte
ADD  .S2  4,B5,B5     ; @ @ |51| Get LSBYTE
SHR   .S1X  B6,A7,A5   ; @ @ |49| Make room for MSByte
STW  .D1T1  A4,*A8++  ; @ @ @ ^|56|  Store 1st word
STW  .D2T2  *B4++[B7],B6  ; @ @ @ ^|46| Load third word
SHR   .S2X  A5,B9,B8  ; @ @ |51| Get MSByte
SHL   .S1  A5,A7,A4   ; @ @ |50| Get LSBYTE
ADD  .D2  0x4,B5,B5    ; @ ^|59| Since num_cols is
  ; short by a word
MV   .S1X  B5,A8     ; @ ^|9|
OR   .L1   A3,A4,A4   ; @ @ |53| Obtain actual 1st word
ADD  .L1  4,A8,A8     ; @ ^|9|
SHL   .S1X  B6,A7,A5   ; @ @ |49| Make room for MSByte
ADD  .S2  4,B5,B5     ; @ @ @ |51| Get LSBYTE
SHR   .S1X  B6,A7,A5   ; @ @ |49| Make room for MSByte
STW  .D1T1  A4,*A8++  ; @ @ @ ^|56|  Store 1st word
STW  .D2T2  *B4++[B7],B6  ; @ @ @ ^|46| Load third word

** ----------- PIPED LOOP EPILOG -----------------------------------------------**

ADD  .S2  4,B5,B5    ; @ ^|9|
OR   .L2X  B8,A5,B8   ; @ |54| Obtain actual 2nd word
SHRU  .S1  A3,A6,A3   ; @ |48| Make room for LSBYTE
LDW  .D2T2  *B4++[B7],B6  ; @ @ @ ^|46| Load third word
LDW  .D1T1  *A0++,A3  ; @ @ @ ^|44| Load second word
STW  .D1T1  A4,*A8++  ; @ ^|56|  Store 1st word
STW  .D2T2  *B4++[B7],B6  ; @ @ @ ^|46| Load third word
SHRU  .S2X  A5,B9,B8  ; @ |51| Get MSByte
SHL   .S1  A5,A7,A4   ; @ |50| Get LSBYTE
MV    .L1X  B4,A0     ; @ @ |51| Get LSBYTE
ADD .D2 0x4,B5,B5 ; @@@ ^59| Since num_cols is short by a word

MV .S1X B5,A8 ; @@@ ^9|

ADD .L1 4,A8,A8 ; @@@ ^9|

;;;;;;;;;;------------------------------------------------------------------*

MVC .S2 B1,CSR ; |40|
B .S2 B3
NOP 5
; BRANCH OCCURS

; .endproc
Appendix B  Motion Compensation Code: Case B

C Code

/* Interpolate rows   */
void MC_case_b(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size,
const int rounding_type)
{
    int m, n;
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = (ref[r_x+m][r_y+n ]
                         + ref[r_x+m][r_y+n+1]
                         + 1 - rounding_type)/2;
        }
    }
}

Natural C Code

/* Interpolate rows   */
void MC_case_b(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size,
const int rounding_type)
{
    int m, n;
    _nassert(size>=8);
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = (ref[r_x+m][r_y+n ]
                         + ref[r_x+m][r_y+n+1]
                         + 1 - rounding_type)/2;
        }
    }
}

Optimized C Code

/* Interpolate rows   */
void MC_case_b(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size,
const int rounding_type)
{
    int m, n;
    _nassert(size>=8);
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = (ref[r_x+m][r_y+n ]
                         + ref[r_x+m][r_y+n+1]
                         + 1 - rounding_type)>>1;
        }
    }
}
Linear Assembly

; Linear Assembly version of "MC_case_b"

.LDEF _MC_case_b
.sect "text"

MC_case_b: .cproc ref, curr, r_x, c_x, r_y, c_y, num_cols, rounding
.reg p_r, p_c
.reg r_temp1, r_temp2, c_temp1, c_temp2
.reg r_a, r_b, temp
.reg count, const

; Calculate pointers "p_c" and "p_r"

SHL r_x, 0x05, r_temp1 ; r_temp1 = r_x * NUM_COLS
SHL c_x, 0x05, c_temp1 ; c_temp1 = c_x * NUM_COLS
ADD r_y, ref, r_temp2 ; r_temp2 = ref + r_y
ADD c_y, curr, c_temp2 ; c_temp2 = curr + c_y
ADD r_temp1, r_temp2, p_r ; p_r = r_temp1 + r_temp2
ADD c_temp1, c_temp2, p_c ; p_c = c_temp1 + c_temp2
SUB 1, rounding, const ; const = 1 - rounding

; Initialize loop counter
MVK 8, count ; Loop performed 8 times

; Loop
loop: .trip 8

LDBU *+p_r[0], r_a ; Load 1st byte
LDBU *+p_r[1], r_b ; Load 2nd byte
ADD r_a, const, temp ; 1.1 First part of op
ADD r_b, temp, temp ; 1.2 Second part of op
SHRU temp, 1, temp ; 1.3 Divide by 2 (w/truncation)
STB temp, *+p_c[0] ; 1.4 Store result
LDBU *+p_r[2], r_a ; Load 3rd byte
ADD r_b, const, temp ; 2.1 First part of op
ADD r_a, temp, temp ; 2.2 Second part of op
SHRU temp, 1, temp ; 2.3 Divide by 2 (w/truncation)
STB temp, *+p_c[1] ; 2.4 Store result
LDBU *+p_r[3], r_b  ; Load 4th byte
ADD r_a, const, temp  ; 3.1 First part of op
ADD r_b, temp, temp  ; 3.2 Second part of op
SHRU temp, 1, temp  ; 3.3 Divide by 2 (w/truncation)
STB temp, *+p_c[2]  ; 3.4 Store result

LDBU *+p_r[4], r_a  ; Load 5th byte
ADD r_b, const, temp  ; 4.1 First part of op
ADD r_a, temp, temp  ; 4.2 Second part of op
SHRU temp, 1, temp  ; 4.3 Divide by 2 (w/truncation)
STB temp, *+p_c[3]  ; 4.4 Store result

LDBU *+p_r[5], r_b  ; Load 6th byte
ADD r_a, const, temp  ; 5.1 First part of op
ADD r_b, temp, temp  ; 5.2 Second part of op
SHRU temp, 1, temp  ; 5.3 Divide by 2 (w/truncation)
STB temp, *+p_c[4]  ; 5.4 Store result

LDBU *+p_r[6], r_a  ; Load 7th byte
ADD r_b, const, temp  ; 6.1 First part of op
ADD r_a, temp, temp  ; 6.2 Second part of op
SHRU temp, 1, temp  ; 6.3 Divide by 2 (w/truncation)
STB temp, *+p_c[5]  ; 6.4 Store result

LDBU *+p_r[7], r_b  ; Load 8th byte
ADD r_a, const, temp  ; 7.1 First part of op
ADD r_b, temp, temp  ; 7.2 Second part of op
SHRU temp, 1, temp  ; 7.3 Divide by 2 (w/truncation)
STB temp, *+p_c[6]  ; 7.4 Store result

LDBU *+p_r[8], r_a  ; Load 9th byte
ADD r_b, const, temp  ; 8.1 First part of op
ADD r_a, temp, temp  ; 8.2 Second part of op
SHRU temp, 1, temp  ; 8.3 Divide by 2 (w/truncation)
STB temp, *+p_c[7]  ; 8.4 Store result

ADD p_c, num_cols, p_c  ; Move p_c to next row
ADD p_r, num_cols, p_r  ; Move p_r to next row

[count] SUB count, 1, count  ; Loop back
[count] B loop
.endproc
Output from Assembly Optimizer

;**********************************************************************
;* GLOBAL FILE PARAMETERS                                              *
;**********************************************************************
;* Architecture        : TMS320C6200                                        *
;* Endian              : Little                                              *
;* Interrupt Threshold : Disabled                                          *
;* Memory Model        : Small                                               *
;* Speculative Load    : Threshold = 0                                     *
;* Redundant Loops     : Enabled                                             *
;* Pipelining          : Enabled                                             *
;* Debug Info          : No Debug Info                                       *
;**********************************************************************

FP .set A15
DP .set B14
SP .set B15
.global $bss

; Linear Assembly version of "MC_case_b"
.def _MC_case_b
.sect .".text"
.sect .".text"

;* FUNCTION NAME: _MC_case_b                                             *
;**********************************************************************
;* Regs Modified: A0,A1,A3,A4,A5,A6,A7,A9,B0,B1,B2,B3,B4,B5,B6,B7, B8,B9 *
;* Regs Used    : A0,A1,A3,A4,A5,A6,A7,A8,A9,A10,B0,B1,B2,B3,B4,B5, B6,B7,B8,B9,B10 *
;**********************************************************************

_MC_case_b:
** ------------------------------------------------------------------*

SUB .L1X 0x1,B10,A3 ; 9 |
MV    .L1X B3,A9        ; 9 |
SHL   .S2 B6,0x5,B4     ; 19 | c_temp1 = c_x*NUM_COLS
SHL   .S1 A6,0x5,A5     ; 18 | r_temp1 = r_x*NUM_COLS
ADD   .D1 A8,A4,A4      ; 9 |
ADD   .L2 B8,B4,B5      ; 9 |
ADD     .L1     A5,A4,A6     ; |24|  p_r = r_temp1 + r_temp2
||         MV      .S1X    B4,A0        ; |19|
||         MV      .L2X    A10,B9       ; |9|
ADD     .L1X    A0,B5,A5     ; |25|  p_c = c_temp1 + c_temp2
||         MV      .L2X    A6,B2        ; |30|
||         MVC     .S2     0x8,B4       ; |30|  Loop performed 8 times
MVC     .S2     CSR,B3       ; |30|
||         MV      .L2X    A5,B7        ; |30|
||         MV      .L1X    B4,A1        ; |30|
AND     .L2     -2,B3,B4     ; |30|
MVC     .S2     B4,CSR       ; |30|
||         SUB     .L1     A1,1,A1      ; |30|

;*--------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : loop
;*      Known Minimum Trip Count         : 8
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 1
;*      Unpartitioned Resource Bound     : 9
;*      Partitioned Resource Bound(*)    : 9
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     4        5
;*      .D units                     9*       8
;*      .M units                     0        0
;*      .X cross paths               3        6
;*      .T address paths             8        9*
;*      Long read paths              4        4
;*      Long write paths             0        0
;*      Logical ops (.LS)           3        6  (.L or .S unit)
;*      Addition ops (.LSD)          7        6  (.L or .S or .D unit)
;*      Bound(.L .S .LS)            4        6
;*      Bound(.L .S .D .LS .LSD)     8        9*
;*      Searching for software pipeline schedule at ...
;*         ii = 9  Did not find schedule
;*         ii = 10 Schedule found with 2 iterations in parallel
;*      Done
;*      Epilog not removed : Speculative load is beyond user threshold
;*      Speculative Load Threshold : Unknown
;*
;*--------------------------------------------------------------------*
L2:        ; PIPED LOOP PROLOG
; loop:              .trip 8
LDBU    .D2T2   *+B2(3),B4   ;  ^|49|  Load 4nd byte
LDBU    .D1T2   *+A6(4),B8   ;  ^|55|  Load 5th byte
NOP             1
LDBU    .D1T1   *+A6(5),A0   ;  ^|61|  Load 6th byte
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x

LDBU .D1T1 *+A6(6),A4 ;  ^67| Load 7th byte
LDBU .D2T2 *+B2(1),B1 ;  ^37| Load 2nd byte
LDBU .D2T2 *+B2(2),B8 ;  ^43| Load 3rd byte
ADD .L2X B4,A3,B6 ;  |56| 4.1 First part of op
ADD .L2X B8,A3,B6 ;  |62| 5.1 First part of op
ADD .S2 B8,B6,B8 ;  |57| 4.2 Second part of op
SHRU .S2 B8,0x1,B5 ;  |58| 4.3 Divide by 2 (w/tru)
STB .D2T2 B5,**B7(3) ;  ^59| 4.4 Store result
ADD .L2X A0,B6,B5 ;  |63| 5.2 Second part of op
ADD .S1 A0,A3,A0 ;  |68| 6.1 First part of op

;** ------------------------------------------------------------------*
loop:        ; PIPED LOOP KERNEL
SHRU .S2 B5,0x1,B5 ;  |64| 5.3 Divide by 2 (w/tru)
ADD .L1 A4,A0,A4 ;  |69| 6.2 Second part of op
ADD .S1 A4,A3,A0 ;  |74| 7.1 First part of op
LDBU .D1T1 *+A6(7),A4 ;  ^73| Load 8th byte
ADD .L2X B1,A3,B6 ;  |44| 2.1 First part of op
ADD .D2 B2,B9,B2 ;  ^46| Move p_r to next row
LDBU .D1T1 *+A6(8),A4 ;  ^79| Load 9th byte
SHRU .S1 A4,0x1,A4 ;  |70| 6.3 Divide by 2 (w/tru)
ADD .S2 B8,B6,B8 ;  |45| 2.2 Second part of op
ADD .L2X B8,A3,B0 ;  |50| 3.1 First part of op
MV .L1X B2,A6 ;  ^9| 6.3 Divide by 2 (w/tru)
LDBU .D1T2 *+B2(3),B4 ;  @ ^49| Load 4th byte
SHRU .S2 B8,0x1,B8 ;  |46| 2.3 Divide by 2 (w/tru)
ADD .L2 B4,B0,B4 ;  |51| 3.2 Second part of op
LDBU .D1T2 *+A6(4),B8 ;  @ ^55| Load 5th byte
[ A1] SUB .L1 A1,0x1,A1 ;  |88| Loop back
STB .D1T1 A4,**A5(5) ;  ^71| 6.4 Store result
STB .D2T2 B8,**B7(1) ;  ^47| 2.4 Store result
SHRU .S2 B4,0x1,B4 ;  |52| 3.3 Divide by 2 (w/tru)
ADD .L2X B6,A3,B6 ;  |38| 1.1 First part of op
STB .D2T2 B4,**B7(2) ;  ^53| 3.4 Store result
[ A1] B .S1 loop ;  ^89
ADD .L2 B1,B6,B4 ;  |39| 1.2 Second part of op
LDBU .D1T1 *+A6(5),A0 ;  @ ^61| Load 6th byte
ADD .L1 A4,A0,A0 ;  |75| 7.2 Second part of op
ADD .S1 A4,A3,A7 ;  |80| 8.1 First part of op
SHRU .S2 B4,0x1,B6 ;  |40| 1.3 Divide by 2 (w/tru)
LDBU .D1T1 *+A6(6),A4 ;  @ ^67| Load 7th byte
LDBU .D2T2 *+B2(1),B1 ;  @ ^37| Load 2nd byte
SHRU .S1 A0,0x1,A7 ;  |76| 7.3 Divide by 2 (w/tru)
ADD .D1 A4,A7,A4 ;  |81| 8.2 Second part of op
STB .D1T1 A7,+A5(6) ; @ \textsuperscript{77} 7.4 Store result
SHRU .S1 A4,0x1,A4 ; @ \textsuperscript{82} 8.3 Divide by 2 (w/tr)

STB .D1T2 B5,+A5(4) ; @ \textsuperscript{65} 5.4 Store result
ADD .L2 B7,B9,B7 ; @ \textsuperscript{85} Move p_c to next row
STB .D2T1 A0,*B7 ; @ \textsuperscript{41} 1.4 Store result

STB .D1T1 A4,+A5(7) ; @ \textsuperscript{83} 8.4 Store result
MV .L1X B7,A5 ; @ \textsuperscript{9}

MVC .S2 B3,CSR ; \texttt{30}
B .S2X A9
NOP 5
; BRANCH OCCURS

; .endproc
Appendix C  Motion Compensation Code: Case C

C Code

/* Interpolate rows */
void MC_case_c(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size,
const int rounding_type)
{
    int m, n;
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = (ref[r_x+m][r_y+n]
            + ref[r_x+m+1][r_y+n]
            + 1 - rounding_type)/2;
        }
    }
}

Natural C Code

/* Interpolate rows */
void MC_case_c(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size,
const int rounding_type)
{
    int m, n;
    _nassert(size>=8);
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = (ref[r_x+m][r_y+n]
            + ref[r_x+m+1][r_y+n]
            + 1 - rounding_type)/2;
        }
    }
}

Optimized C Code

/* Interpolate rows */
void MC_case_c(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size,
const int rounding_type)
{
    int m, n;
    _nassert(size>=8);
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = (ref[r_x+m][r_y+n]
            + ref[r_x+m+1][r_y+n]
            + 1 - rounding_type)>>1;
        }
    }
}
Linear Assembly

; Linear Assembly version of "MC_case_c"

.def  _MC_case_c
.sect  "text"

_MC_case_c:
.cproc  ref, curr, r_x, c_x, r_y, c_y, num_cols, rounding

.reg    p_r, p_c, ptr_temp
.reg    r_temp1, r_temp2, c_temp1, c_temp2
.reg    r_a, r_b, temp
.reg    count, const

; Calculate pointers "p_c" and "p_r"

SHL   r_x, 0x05, r_temp1     ; r_temp1 = r_x * NUM_COLS
SHL   c_x, 0x05, c_temp1     ; c_temp1 = c_x * NUM_COLS
ADD   r_y, ref,  r_temp2     ; r_temp2 = ref  + r_y
ADD   c_y, curr, c_temp2     ; c_temp2 = curr + c_y
ADD   r_temp1, r_temp2, p_r  ; p_r = r_temp1 + r_temp2
ADD   c_temp1, c_temp2, p_c  ; p_c = c_temp1 + c_temp2
SUB   1, rounding, const     ; const = 1 - rounding

; Initialize loop counter
MVK   8, count               ; Loop performed 8 times

; Loop
loop:     .trip 8

LDBU  *p_r++[num_cols], r_a  ; Load 1st byte
LDBU  *p_r++[num_cols], r_b  ; Load 2nd byte
ADD   r_a, const, temp       ; 1.1 First part of op
ADD   r_b, temp, temp        ; 1.2 Second part of op
SHRU  temp, 1, temp          ; 1.3 Divide by 2 (w/truncation)
STB   temp, *p_c++[num_cols] ; 1.4 Store result

LDBU  *p_r++[num_cols], r_a  ; Load 3rd byte
ADD   r_b, const, temp       ; 2.1 First part of op
ADD   r_a, temp, temp        ; 2.2 Second part of op
SHRU  temp,1, temp           ; 2.3 Divide by 2 (w/truncation)
STB   temp, *p_c++[num_cols] ; 2.4 Store result

LDBU  *p_r++[num_cols], r_b  ; Load 4nd byte
ADD   r_a, const, temp       ; 3.1 First part of op
ADD   r_b, temp, temp        ; 3.2 Second part of op
SHRU  temp, 1, temp          ; 3.3 Divide by 2 (w/truncation)
STB temp, *p_c++[num_cols]; 3.4 Store result

LDBU *p_r++[num_cols], r_a; Load 5th byte
ADD r_b, const, temp; 4.1 First part of op
ADD r_a, temp, temp; 4.2 Second part of op
SHRU temp, 1, temp; 4.3 Divide by 2 (w/truncation)
STB temp, *p_c++[num_cols]; 4.4 Store result

LDBU *p_r++[num_cols], r_b; Load 6th byte
ADD r_a, const, temp; 5.1 First part of op
ADD r_b, temp, temp; 5.2 Second part of op
SHRU temp, 1, temp; 5.3 Divide by 2 (w/truncation)
STB temp, *p_c++[num_cols]; 5.4 Store result

LDBU *p_r++[num_cols], r_a; Load 7th byte
ADD r_b, const, temp; 6.1 First part of op
ADD r_a, temp, temp; 6.2 Second part of op
SHRU temp, 1, temp; 6.3 Divide by 2 (w/truncation)
STB temp, *p_c++[num_cols]; 6.4 Store result

LDBU *p_r++[num_cols], r_b; Load 8th byte
ADD r_a, const, temp; 7.1 First part of op
ADD r_b, temp, temp; 7.2 Second part of op
SHRU temp, 1, temp; 7.3 Divide by 2 (w/truncation)
STB temp, *p_c++[num_cols]; 7.4 Store result

LDBU *p_r, r_a; Load 9th byte
ADD r_b, const, temp; 8.1 First part of op
ADD r_a, temp, temp; 8.2 Second part of op
SHRU temp, 1, temp; 8.3 Divide by 2 (w/truncation)
STB temp, *p_c++[num_cols]; 8.4 Store result

SHL num_cols, 0x03, ptr_temp; Multiply by 8
SUB p_r, ptr_temp, p_r; Adjust ptr to ref frame
ADD p_r, 0x01, p_r; Move p_r to next col
SUB p_c, ptr_temp, p_c; Adjust ptr to cur frame
ADD p_c, 0x01, p_c; Move p_c to next col

[count] SUB count, 1, count; Loop back
[count] B loop
.endproc

Output from Assembly Optimizer

;******************************************************************************
;* TMS320C6x ANSI C Codegen Version 2.10.beta *
;* Date/Time created: Fri Aug 21 11:57:25 1998 *
;******************************************************************************

;******************************************************************************
;* GLOBAL FILE PARAMETERS *
;*
;* Architecture : TMS320C6200 *
;* Endian : Little *
;* Interrupt Threshold : Disabled *

On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x
SOFTWARE PIPELINE INFORMATION

Loop label : loop
Known Minimum Trip Count : 8
Known Max Trip Count Factor : 1
Loop Carried Dependency Bound(\^) : 12
Unpartitioned Resource Bound : 9
Partitioned Resource Bound(*) : 9
Resource Partition:

<table>
<thead>
<tr>
<th>A-side</th>
<th>B-side</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L units</td>
<td>0</td>
</tr>
<tr>
<td>.S units</td>
<td>5</td>
</tr>
<tr>
<td>.D units</td>
<td>9*</td>
</tr>
<tr>
<td>.M units</td>
<td>0</td>
</tr>
<tr>
<td>.X cross paths</td>
<td>4</td>
</tr>
<tr>
<td>.T address paths</td>
<td>9*</td>
</tr>
<tr>
<td>Long read paths</td>
<td>4</td>
</tr>
<tr>
<td>Long write paths</td>
<td>0</td>
</tr>
<tr>
<td>Logical ops (.LS)</td>
<td>4</td>
</tr>
<tr>
<td>Addition ops (.LSD)</td>
<td>9</td>
</tr>
<tr>
<td>Bound(.L .S .LS)</td>
<td>5</td>
</tr>
<tr>
<td>Bound(.L .S .D .LS .LSD)</td>
<td>9*</td>
</tr>
</tbody>
</table>

Searching for software pipeline schedule at ...
ii = 12 Did not find schedule
ii = 13 Schedule found with 2 iterations in parallel
Done

Epilog not removed : Speculative load is beyond user threshold
Speculative Load Threshold : Unknown

L2: ; PIPED LOOP PROLOG
; loop: .trip 8
LDBU .D1T1 *A3+[A7],A0 ; |35| Load 1st byte
NOP 1
LDBU .D1T1 *A3+[A7],A0 ; |37| Load 2nd byte
LDBU .D1T1 *A3+[A7],A3 ; |43| Load 3rd byte
LDBU .D1T1 *A3+[A7],A3 ; |49| Load 4nd byte
LDBU .D1T1 *A3+[A7],A5 ; |55| Load 5th byte
ADD .L1 A0,A4,A3 ; |38| 1.1 First part of op
MV .L2X A3,B4 ; |9|
ADD .L1 A0,A3,A6 ; |39| 1.2 Second part of op
ADD .S1 A0,A4,A0 ; |44| 2.1 First part of op
LDBU .D2T2 *B4+[B5],B6 ; |61| Load 6th byte
ADD .L1 A3,A0,A0 ; |45| 2.2 Second part of op
ADD .S1 A3,A4,A5 ; |50| 3.1 First part of op
LDBU .D2T2 *B4+[B5],B7 ; |67| Load 7th byte
SHL .S2 B5,0x3,B9 ; |85| Multiply by 8
ADD .D1 A3,A5,A6 ; |51| 3.2 Second part of op
SHRU .S1 A6,0x1,A8 ; |40| 1.3 Divide by 2 (w/tru)
LDBU .D2T2 *B4+[B5],B8 ; |73| Load 8th byte
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x

/** **) ściagnij Piped Loop Kernel

loop:

STB     .D1T1   A8,*A9+[A7] ;  41  1.4 Store result
SHRU    .S1     A0,0x1,A8 ;  46  2.3 Divide by 2 (w/tru)
ADD      .L1     A5,A4,A0 ;  62  5.1 First part of op
SUB      .L2     B4,B9,B4 ;  87  Adjust ptr to ref fra
LDBU     .D2T2   *B4,B7 ;  79  Load 9th byte
SHRU    .S1     A6,0x1,A6 ;  52  3.3 Divide by 2 (w/tru)
STB     .D1T1   A8,*A9+[A7] ;  47  2.4 Store result
ADD      .L1     A3,A4,A3 ;  56  4.1 First part of op
ADD      .L2     0x1,B4,B4 ;  88  Move p_r to next col
STB     .D1T1   A6,*A9+[A7] ;  53  3.4 Store result
ADD      .S1     A5,A3,A0 ;  57  4.2 Second part of op
ADD      .L2X    B6,A0,B4 ;  63  5.2 Second part of op
MV       .L1X    B4,A3 ;  9  Load 1st byte

ADD      .L2X    B7,A4,B6 ;  74  7.1 First part of op
SHRU     .S2     B4,0x1,B4 ;  64  5.3 Divide by 2 (w/tru)
SHRU     .S1     A0,0x1,A5 ;  58  4.3 Divide by 2 (w/tru)
ADD      .L1X    B6,A4,A0 ;  68  6.1 First part of op
LDBU     .D1T1   *A3+[A7],A0 ;  35  Load 1st byte
ADD      .S2     B8,B6,B6 ;  75  7.2 Second part of op
ADD      .L1X    B8,A4,A0 ;  80  8.1 First part of op
STB     .D1T1   A5,*A9+[A7] ;  59  4.4 Store result
ADD      .L2X    B7,A0,B8 ;  69  6.2 Second part of op

[ B0] SUB     .D2     B0,0x1,B0 ;  93  Loop back
MV       .L2X    A9,B1 ;  9  Load 2nd byte
SHRU     .S2     B8,0x1,B8 ;  70  6.3 Divide by 2 (w/tru)
LDBU     .D1T1   *A3+[A7],A0 ;  37  Load 3rd byte
ADD      .L2X    B7,A0,B6 ;  81  8.2 Second part of op
SHRU     .S2     B6,0x1,B4 ;  76  7.3 Divide by 2 (w/tru)
STB     .D2T2   B4,*B1+[B5] ;  65  5.4 Store result
LDBU     .D1T1   *A3+[A7],A3 ;  43  Load 4th byte

[ B0] SUB     .D2     B0,0x1,B6 ;  82  8.3 Divide by 2 (w/tru)
B .S1     loop ;  94  6.4 Store result
STB     .D2T2   B8,*B1+[B5] ;  71
LDBU     .D1T1   *A3+[A7],A3 ;  49  Load 4th byte
STB     .D2T2   B4,*B1+[B5] ;  77  7.4 Store result
LDBU     .D1T1   *A3+[A7],A5 ;  55  Load 5th byte
STB     .D2T2   B6,*B1+[B5] ;  83  8.4 Store result
ADD      .L1     A0,A4,A3 ;  38  1.1 First part of op
MV       .L2X    A3,B4 ;  9  Load 6th byte
SUB      .L2     B1,B9,B6 ;  90  Adjust ptr to cur fra
ADD      .L1     A0,A3,A6 ;  39  1.2 Second part of op
ADD      .S1     A0,A4,A0 ;  44  2.1 First part of op
LDBU     .D2T2   *B4+[B5],B6 ;  61  Load 7th byte
ADD .L2  0x1,B6,B1 ; ^91 Move p_c to next col
ADD .L1  A3,A0,A0 ; @ 45 2.2 Second part of op
ADD .S1  A3,A4,A5 ; @ 50 3.1 First part of op
LDBU .D2T2 *B4+[B5],B7 ; @ ^67 Load 7th byte
MV   .L1X  B1,A9 ; ^9
SHL  .S2  B5,0x3,B9 ; @ 85 Multiply by 8
ADD  .D1  A3,A5,A6 ; @ 51 3.2 Second part of op
SHRU .S1  A6,0x1,A8 ; @ 40 1.3 Divide by 2 (w/tr)
LDBU .D2T2 *B4+[B5],B8 ; @ ^73 Load 8th byte

;** ------------------------------------------------------------------*

L4: ; PIPED LOOP EPILOG

STB .D1T1 A8,*A9+[A7] ; @ ^41 1.4 Store result
SHRU .S1  A0,0x1,A8 ; @ 46 2.3 Divide by 2 (w/tr)
ADD  .L1  A5,A4,A0 ; @ 62 5.1 First part of op
SUB  .L2  B4,B9,B4 ; @ ^87 Adjust ptr to ref fr
LDBU .D2T2 *B4,B7 ; @ 79 Load 9th byte
SHRU .S1  A6,0x1,A6 ; @ 52 3.3 Divide by 2 (w/tr)
STB .D1T1 A8,*A9+[A7] ; @ ^47 2.4 Store result
ADD  .L1  A3,A4,A3 ; @ 56 4.1 First part of op
ADD  .L2  0x1,B4,B4 ; @ ^88 Move p_r to next col
STB .D1T1 A6,*A9+[A7] ; @ ^53 3.4 Store result
ADD  .S1  A5,A3,A0 ; @ 57 4.2 Second part of op
ADD  .L2X B6,A0,B4 ; @ 63 5.2 Second part of op
MV  .L1X  B4,A3 ; @ ^9
ADD  .L2X B7,A4,B6 ; @ 74 7.1 First part of op
SHRU .S2  B4,0x1,B4 ; @ 64 5.3 Divide by 2 (w/tr)
SHRU .S1  A0,0x1,A5 ; @ 58 4.3 Divide by 2 (w/tr)
ADD  .L1X B6,A4,A0 ; @ 68 6.1 First part of op
ADD  .S2  B8,B6,B6 ; @ 75 7.2 Second part of op
ADD  .L1X B8,A4,A0 ; @ 80 8.1 First part of op
STB .D1T1 A5,*A9+[A7] ; @ ^59 4.4 Store result
ADD  .L2X B7,A0,B8 ; @ 69 6.2 Second part of op
MV  .L2X  A9,B1 ; @ ^9
SHRU  .S2  B2,0x1,B8 ; @ 70 6.3 Divide by 2 (w/tr)
ADD  .L2X B7,A0,B6 ; @ 81 8.2 Second part of op
SHRU  .S2  B6,0x1,B4 ; @ 76 7.3 Divide by 2 (w/tr)
STB .D2T2 B4,*B1+[B5] ; @ ^65 5.4 Store result
SHRU  .S2  B6,0x1,B6 ; @ 82 8.3 Divide by 2 (w/tr)
STB .D2T2 B8,*B1+[B5] ; @ ^71 6.4 Store result
STB .D2T2 B4,*B1+[B5] ; @ ^77 7.4 Store result
STB .D2T2 B6,*B1+[B5] ; @ ^83 8.4 Store result
SUB  .L2  B1,B9,B6 ; @ ^90 Adjust ptr to cur fr
ADD  .L2  0x1,B6,B1 ; @ ^91 Move p_c to next col
MV  .L1X  B1,A9 ; @ ^9

;** ------------------------------------------------------------------*
MVC .S2 B2,CSR ; |30|
B .S2 B3
NOP 5
; BRANCH OCCURS

; .endproc
Appendix D  Motion Compensation Code: Case D

C Code

/* Interpolate rows and columns */
void MC_case_d(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size,
const int rounding_type)
{
    int m, n;
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = (ref[r_x+m ][r_y+n ]
                    + ref[r_x+m ][r_y+n+1]
                    + ref[r_x+m+1][r_y+n ]
                    + ref[r_x+m+1][r_y+n+1]
                    + 2 - rounding_type)>>2;
        }
    }
}

Natural C Code

/* Interpolate rows and columns */
void MC_case_d(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size,
const int rounding_type)
{
    int m, n;
    _nassert(size>=8);
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = (ref[r_x+m ][r_y+n ]
                    + ref[r_x+m ][r_y+n+1]
                    + ref[r_x+m+1][r_y+n ]
                    + ref[r_x+m+1][r_y+n+1]
                    + 2 - rounding_type)/4;
        }
    }
}

Optimized C Code

/* Interpolate rows and columns */
void MC_case_d(uchar ref[NUM_ROWS][NUM_COLS],
uchar curr[NUM_ROWS][NUM_COLS], const int r_x, const int c_x,
const int r_y, const int c_y, const int size,
const int rounding_type)
{
    int m, n;
    _nassert(size>=8);
    for(m=0; m<size; m++) {
        for(n=0; n<size; n++) {
            curr[c_x+m][c_y+n] = (ref[r_x+m ][r_y+n ]
                    + ref[r_x+m ][r_y+n+1]
                    + ref[r_x+m+1][r_y+n ]
                    + ref[r_x+m+1][r_y+n+1]
                    + 2 - rounding_type)*4;
        }
    }
}
for(n=0; n<size; n++) {
    curr[c_x+m][c_y+n] = (ref[r_x+m][r_y+n ]
        + ref[r_x+m][r_y+n+1]
        + ref[r_x+m+1][r_y+n ]
        + ref[r_x+m+1][r_y+n+1]
        + 2 - rounding_type)>>2;
} }

Linear Assembly

; Linear Assembly version of "MC_case_d"

.def _MC_case_d
.sect ".text"

_MC_case_d: cproc ref, curr, r_x, c_x, r_y, c_y, num_cols, rounding
.reg p_r, p_r1, p_r2, p_c
.reg r_temp1, r_temp2, c_temp1, c_temp2
.reg r_a1, r_a2, r_b1, r_b2, temp, temp1, temp2
.reg count, const, ptr_temp

; Calculate pointers "p_c" and "p_r"
SHL r_x, 0x05, r_temp1 ; r_temp1 = r_x * NUM_COLS
SHL c_x, 0x05, c_temp1 ; c_temp1 = c_x * NUM_COLS
ADD r_y, ref, r_temp2 ; r_temp2 = ref + r_y
ADD c_y, curr, c_temp2 ; c_temp2 = curr + c_y
ADD r_temp1, r_temp2, p_r ; p_r = r_temp1 + r_temp2
ADD c_temp1, c_temp2, p_c ; p_c = c_temp1 + c_temp2
SUB 2, rounding, const ; const = 2 - rounding

; Initialize loop counter
MVK 8, count ; Loop performed 8 times
MV p_r, p_r1
ADD p_r, num_cols, p_r2

; Loop
loop: .trip 8
LDBU *+p_r1[0], r_a1 ; Load 1st byte/1st pair
LDBU *+p_r2[0], r_a2 ; Load 2nd byte/1st pair
ADD r_a1, r_a2, temp1
LDBU *+p_r1[1], r_b1 ; Load 1st byte/2nd pair
LDBU *+p_r2[1], r_b2 ; Load 2nd byte/2nd pair
ADD temp1, const, temp1 ; 1.1 First part of op
ADD r_b1, r_b2, temp2 ; 1.2 Second part of op
ADD temp1, temp2, temp ; 1.3 Third part of op
SHRU temp, 2, temp ; 1.4 Divide by 4 (w/truncation)
STB temp, *+p_c[0] ; 1.5 Store result
LDBU  *+p_r1[2], r_a1          ; Load 1st byte/3rd pair
LDBU  *+p_r2[2], r_a2          ; Load 2nd byte/3rd pair
ADD   temp2, const, temp2     ; 2.1 First part of op
ADD   r_a1, r_a2, temp1        ; 2.2 Second part of op
ADD   temp1, temp2, temp       ; 2.3 Third part of op
SHRU  temp, 2, temp            ; 2.4 Divide by 4 (w/truncation)
STB   temp, *+p_c[1]           ; 2.5 Store result

LDBU  *+p_r1[3], r_b1          ; Load 1st byte/4th pair
LDBU  *+p_r2[3], r_b2          ; Load 2nd byte/4th pair
ADD   temp1, const, temp1      ; 3.1 First part of op
ADD   r_b1, r_b2, temp2        ; 3.2 Second part of op
ADD   temp1, temp2, temp       ; 3.3 Third part of op
SHRU  temp, 2, temp            ; 3.4 Divide by 4 (w/truncation)
STB   temp, *+p_c[2]           ; 3.5 Store result

LDBU  *+p_r1[4], r_a1          ; Load 1st byte/5th pair
LDBU  *+p_r2[4], r_a2          ; Load 2nd byte/5th pair
ADD   temp2, const, temp2      ; 4.1 First part of op
ADD   r_a1, r_a2, temp1        ; 4.2 Second part of op
ADD   temp1, temp2, temp       ; 4.3 Third part of op
SHRU  temp, 2, temp            ; 4.4 Divide by 4 (w/truncation)
STB   temp, *+p_c[3]           ; 4.5 Store result

LDBU  *+p_r1[5], r_b1          ; Load 1st byte/6th pair
LDBU  *+p_r2[5], r_b2          ; Load 2nd byte/6th pair
ADD   temp1, const, temp1      ; 5.1 First part of op
ADD   r_b1, r_b2, temp2        ; 5.2 Second part of op
ADD   temp1, temp2, temp       ; 5.3 Third part of op
SHRU  temp, 2, temp            ; 5.4 Divide by 4 (w/truncation)
STB   temp, *+p_c[4]           ; 5.5 Store result

LDBU  *+p_r1[6], r_a1          ; Load 1st byte/7th pair
LDBU  *+p_r2[6], r_a2          ; Load 2nd byte/7th pair
ADD   temp2, const, temp2      ; 6.1 First part of op
ADD   r_a1, r_a2, temp1        ; 6.2 Second part of op
ADD   temp1, temp2, temp       ; 6.3 Third part of op
SHRU  temp, 2, temp            ; 6.4 Divide by 4 (w/truncation)
STB   temp, *+p_c[5]           ; 6.5 Store result

LDBU  *+p_r1[7], r_b1          ; Load 1st byte/8th pair
LDBU  *+p_r2[7], r_b2          ; Load 2nd byte/8th pair
ADD   temp1, const, temp1      ; 7.1 First part of op
ADD   r_b1, r_b2, temp2        ; 7.2 Second part of op
ADD   temp1, temp2, temp       ; 7.3 Third part of op
SHRU  temp, 2, temp            ; 7.4 Divide by 4 (w/truncation)
STB   temp, *+p_c[6]           ; 7.5 Store result

ADD   p_r1, num_cols, p_r1     ; Move p_r1 to next col
ADD   p_r2, num_cols, p_r2     ; Move p_r2 to next col
ADD   p_c, num_cols, p_c ; Move p_c to next col

[count] SUB   count, 1, count ; Loop back
[count] B     loop
.endproc

Output from Assembly Optimizer

;*****************************************************************************
;* TMS320C6x ANSI C Codegen                                               *
;* Date/Time created: Fri Aug 21 11:57:27 1998                           *
;*****************************************************************************

;*****************************************************************************
;* GLOBAL FILE PARAMETERS                                                  *
;*****************************************************************************

* Architecture        : TMS320C6200
* Endian              : Little
* Interrupt Threshold : Disabled
* Memory Model        : Small
* Speculative Load    : Threshold = 0
* Redundant Loops     : Enabled
* Pipelining          : Enabled
* Debug Info          : No Debug Info
*

FP      .set     A15
DP      .set     B14
SP      .set     B15
.global  $bss

; Linear Assembly version of "MC_case_d"

.def     _MC_case_d

.sect    ".text"
.sect    ".text"

;*****************************************************************************
;* FUNCTION NAME: _MC_case_d                                               *
;*****************************************************************************

;*****************************************************************************
;* Regs Modified: A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,*     *
;*                  B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,SP          *
;*****************************************************************************

_MC_case_d:
** ------------------------------------------------------------------*

;*****************************************************************************
;* Regs Used:     A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,*     *
;*                  B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,SP          *
;*****************************************************************************

_MC_case_d:
** ------------------------------------------------------------------*

;*****************************************************************************

;*****************************************************************************
;* Regs Modified: ref, curr, r_x, c_x, r_y, c_y, num_cols, rounding *
;*****************************************************************************

;*****************************************************************************

;*****************************************************************************
;* Regs Used:     const, ptr_temp                                     *
;*****************************************************************************

;*****************************************************************************

STW     .D2T2   B11,*SP--(32); |9|
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x

STW .D2T1 A10, SP(4) ; 9
STW .D2T2 B3, SP(24) ; 9

STW .D2T2 B10, SP(28) ; 9
SHL .S1 A6, 0x5, A3 ; 18  r_temp1 = r_x*NUM_COLS
ADD .L1 A8, A4, A0 ; 9

STW .D2T1 A11, SP(8) ; 9
ADD .L1 A3, A0, A3 ; 24  p_r = r_temp1 + r_temp2
SHL .S2 B6, 0x5, B4 ; 19  c_temp1 = c_x*NUM_COLS
ADD .L2 B8, B4, B5 ; 9

MV .L2X A3, B1 ; 32
MV .L1X B4, A0 ; 24
ADD .S1 A3, A10, A6 ; 24
STW .D2T1 A14, SP(20) ; 9

ADD .L1X A0, B5, A10 ; 25  p_c = c_temp1 + c_temp2
MV .L2X A6, B3 ; 32
STW .D2T1 A13, SP(16) ; 9
MVK .S2 0x8, B0 ; 30  Loop performed 8 times
MV .S1 A10, A2 ; 24

MVC .S2 CSR, B11 ; 32
MV .L2X A10, B10 ; 32
SUB .L1X 0x2, B10, A11 ; 9
STW .D2T1 A12, SP(12) ; 9

AND .L2 -2, B11, B4 ; 32
MVC .S2 B4, CSR ; 32
SUB .L2 B0, 2, B0 ; 32

;*-----------------------------------------------*
;* SOFTWARE PIPELINE INFORMATION
;* Loop label : loop
;* Known Minimum Trip Count : 8
;* Known Max Trip Count Factor : 1
;* Loop Carried Dependency Bound(*) : 1
;* Unpartitioned Resource Bound : 13
;* Partitioned Resource Bound(*) : 13
;* Resource Partition:
;* A-side   B-side
;* .L units  0  0
;* .S units  5  4
;* .D units  13* 13*
;* .M units  0  0
;* .X cross paths  8  8
;* .T address paths  13* 13*
;* Long read paths  6  2
;* Long write paths  0  0
;* Logical ops (.LS)  8  8 (.L or .S unit)
;* Addition ops (.LSD)  12  4 (.L or .S or .D unit)
;* Bound(.L .S .LS)  7  6
;* Bound(.L .S .D .LS .LSD)  13* 10
;*
Searching for software pipeline schedule at ...   
ii = 13 Schedule found with 3 iterations in parallel   
Done   
Epilog not removed : Speculative load is beyond user threshold   
Speculative Load Threshold : Unknown   

L2: ; PIPED LOOP PROLOG
loop: .trip 8

LDBU .D2T2  +B1(8),B7        ;  ^97  Load 1st byte/9th pai
LDBU .D1T1  +A6(3),A3        ;  ^58  Load 2nd byte/4th pai
LDBU .D2T2  +B3(3),B4        ;  ^42  Load 2nd byte/2nd pai
LDBU .D2T2  +B1(7),B5        ;  ^89  Load 1st byte/8th pai
LDBU .D1T1  +A6(7),A4        ;  ^90  Load 2nd byte/8th pai
LDBU .D2T2  +B1(6),B2        ;  ^81  Load 1st byte/7th pai
LDBU .D1T1  +A6(4),A4        ;  ^66  Load 2nd byte/5th pai
LDBU .D2T2  +B1(5),B5        ;  ^73  Load 1st byte/6th pai
LDBU .D1T1  +A6(5),A7        ;  ^74  Load 2nd byte/6th pai
LDBU .D2T2  +B1(4),B5        ;  ^65  Load 1st byte/5th pai
LDBU .D1T1  +A6(2),A7        ;  ^50  Load 2nd byte/3rd pai
LDBU .D2T2  +B1(3),B6        ;  ^57  Load 1st byte/4th pai
LDBU .D1T1  +A6(6),A4        ;  ^82  Load 2nd byte/7th pai
LDBU .D2T2  +B1(2),B8        ;  ^49  Load 1st byte/3rd pai
ADD .L1X  B5,A4,A1                ;  ^92  7.2 Second part of op
LDBU .D1T1  +A6(8),A3        ;  ^98  Load 2nd byte/9th pai
LDBU .D2T2  *B1,B8            ;  ^37  Load 1st byte/1st pai
ADD .L1  A1,A11,A9              ;  ^99  8.1 First part of op
ADD .L1X  B5,A7,A8              ;  ^76  5.2 Second part of op
LDBU .D2T2  +B1(1),B7        ;  ^41  Load 1st byte/2nd pai
ADD .L2X  B5,A4,B5              ;  ^68  4.2 Second part of op
ADD .L1  A8,A11,A5              ;  ^83  6.1 First part of op
LDBU .D2T2  *B3,B5            ;  ^38  Load 2nd byte/1st pai
ADD .S1  A6,A2,A6               ;  ^106  Move p_r2 to next co
ADD .L2X  B5,A11,B6             ;  ^75  5.1 First part of op
ADD .L1X  B6,A3,A0               ;  ^60  3.2 Second part of op
ADD .S1  A0,A11,A7               ;  ^67  4.1 First part of op
ADD .L2X  B1,A2,B1               ;  ^105  Move p_r1 to next co
ADD .L1X  B8,A7,A12             ;  ^52  2.2 Second part of op
MV  .L2X  A6,B3                   ;  ^9
ADD .S1  A12,A11,A14             ;  ^59  3.1 First part of op
ADD .L1X  B7,A3,A3               ;  ^100  8.2 Second part of op
LDBU .D2T2  +B1(8),B7        ;  ^97  Load 1st byte/9th pai
LDBU .D1T1  +A6(3),A3        ;  ^58  Load 2nd byte/4th pai
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x

STB .D1T1 A5,*+A10(5) ; ^87| 6.5 Store result

[ B0] B .S2 loop ; ^110| 6.2 Loop
ADD .L2X B5,A4,B5 ; @ 68| 4.2 Second part of op
ADD .L1 A8,A11,A5 ; @ 83| 6.1 First part of op
LDBU .D2T2 *B3,B5 ; @ ^ 38| Load 2nd byte/1st pa
ADD .S1 A6,A2,A6 ; @ ^106| Move p_r2 to next c

STB .D1T2 B8,*+A10(3) ; ^71| 4.5 Store result
STB .D2T1 A0,*B10 ; ^47| 1.5 Store result
ADD .L2X B5,A11,B6 ; @ 75| 5.1 First part of op
ADD .L1X B6,A3,A0 ; @ 60| 3.2 Second part of op

STB .D1T2 B9,*+A10(4) ; ^79| 5.5 Store result
STB .D2T1 A12,*+B10(1) ; ^55| 2.5 Store result
ADD .S1 A0,A11,A7 ; @ 67| 4.1 First part of op
ADD .L2X B1,A2,B1 ; @ ^105| Move p_r1 to next c
ADD .L1X B8,A7,A12 ; @ 52| 2.2 Second part of op

MV .L2X A6,B3 ; @ ^9| 9.1 Move
ADD .L1X A12,A11,A14 ; @ 59| 3.1 First part of op
ADD .L1X B7,A3,A3 ; @ 100| 8.2 Second part of op
LDBU .D2T2 *+B1(8),B7 ; @ ^ 97| Load 1st byte/9th p
LDBU .D1T1 *+A6(3),A3 ; @ ^ 58| Load 2nd byte/4th p

ADD .S1 A10,A2,A10 ; @ 107| Move p_c to next co
STB .D1T1 A13,*+A10(7) ; ^103| 8.5 Store result
ADD .L2X B5,A7,B7 ; @ 69| 4.3 Third part of op
ADD .S2 B7,B4,B4 ; @ 44| 1.3 Third part of op
ADD .L1 A3,A9,A3 ; @ 101| 8.3 Third part of op
LDBU .D2T2 *+B3(1),B4 ; @ ^ 42| Load 2nd byte/2nd p

MV .L2X A10,B10 ; ^9| 9.2 Move
ADD .S2 B8,B5,B5 ; @ 39| 3.2 Move
ADD .L1 A14,A0,A0 ; @ 61| 3.3 Third part of op
SHRU .S1 A3,0x2,A13 ; @ 102| 8.4 Divide by 4 (w/t)
LDBU .D2T2 *+B1(7),B5 ; @ ^ 89| Load 1st byte/8th p
LDBU .D1T1 *+A6(7),A4 ; @ ^ 90| Load 2nd byte/8th p

;** ------------------------------------------------------------------*
L4: ; PIPED LOOP EPILOG
SHRU .S2 B7,0x2,B8 ; @ 70| 4.4 Divide by 4 (w/tr)
ADD .L1X B4,A11,A0 ; @ 51| 2.1 First part of op
ADD .L2X B5,A11,B5 ; @ 43| 1.1 First part of op
SHRU .S1 A0,0x2,A7 ; @ 62| 3.4 Divide by 4 (w/tr)
LDBU .D2T2 *+B1(6),B2 ; @ ^ 81| Load 1st byte/7th p
LDBU .D1T1 *+A6(4),A4 ; @ ^ 66| Load 2nd byte/5th p

ADD .L2X B6,A8,B5 ; @ 77| 5.3 Third part of op
ADD .L1X B2,A4,A9 ; @ 84| 6.2 Second part of op
ADD .S1 A12,A0,A0 ; @ 53| 2.3 Third part of op
ADD .S2 B5,B4,B4 ; @ 45| 1.3 Third part of op
LDBU .D2T2 *+B1(5),B5 ; @ ^ 73| Load 1st byte/6th p
LDBU .D1T1 *+A6(5),A7 ; @ ^ 74| Load 2nd byte/6th p

ADD .L1 A9,A11,A0 ; @ 91| 7.1 First part of op
SHRU .S1 A0,0x2,A12 ; @ 54| 2.4 Divide by 4 (w/tr)
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x

SHRU .S2   B4,0x2,B6 ; @ |46| 1.4 Divide by 4 (w/tr)
LDBU .D2T2 *+B1(4),B5 ; @ @ |65| Load 1st byte/5th p
LDBU .D1T1 *+A6(2),A7 ; @ @ |50| Load 2nd byte/3rd p

SHRU .S2   B5,0x2,B9 ; @ |78| 5.4 Divide by 4 (w/tr)
ADD .S1   A0,A1,A8 ; @ |93| 7.3 Third part of op
MV .L1X   B6,A0 ; @ |9|
LDBU .D2T2 *+B1(3),B6 ; @ @ |57| Load 1st byte/4th p
LDBU .D1T1 *+A6(6),A4 ; @ @ |82| Load 2nd byte/7th p

SHRU .S1   A8,0x2,A8 ; @ |94| 7.4 Divide by 4 (w/tr)
LDBU .D2T2 *+B1(2),B8 ; @ @ |49| Load 1st byte/3rd p
ADD .L1X   B5,A4,A1 ; @ @ |92| 7.2 Second part of op
LDBU .D1T1 *+A6(8),A3 ; @ @ |98| Load 2nd byte/9th p

ADD .S1   A9,A5,A5 ; @ |85| 6.3 Third part of op
STB .D1T1 A7,*+A10(2) ; @ @ |63| 3.5 Store result
LDBU .D2T2 *B1,B8 ; @ @ |37| Load 1st byte/1st p
ADD .L1    A1,A11,A9 ; @ @ |99| 8.1 First part of op

SHRU .S1   A5,0x2,A5 ; @ |86| 6.4 Divide by 4 (w/tr)
STB .D1T1 A8,*+A10(6) ; @ @ |95| 7.5 Store result
ADD .L1X   B5,A7,A8 ; @ @ |76| 5.2 Second part of op
LDBU .D2T2 *B3,B5 ; @ @ |41| Load 1st byte/2nd p
ADD .S1    A6,A2,A6 ; @ @ |106| Move p_r2 to next

STB .D1T2 B8,*+A10(3) ; @ @ |71| 4.5 Store result
STB .D2T1 A0,*B10 ; @ @ |47| 1.5 Store result
ADD .L2X   B5,A11,B6 ; @ @ |75| 5.1 First part of op
ADD .L1X   B6,A3,A0 ; @ @ |60| 3.2 Second part of op

STB .D1T2 B9,*+A10(4) ; @ @ |79| 5.5 Store result
STB .D2T2 A12,*B10(1) ; @ @ |55| 2.5 Store result
ADD .S1    A0,A11,A7 ; @ @ |67| 4.1 First part of op
ADD .L2X   B1,A2,B1 ; @ @ |105| Move p_r1 to next
ADD .L1X   B8,A7,A12 ; @ @ |52| 2.2 Second part of op

MV .L2X    A6,B3 ; @ @ |9|
ADD .S1    A12,A11,A14 ; @ @ |59| 3.1 First part of op
ADD .L1X   B7,A3,A3 ; @ @ |100| 8.2 Second part of op

ADD .S1    A10,A2,A10 ; @ @ |107| Move p_c to next c
STB .D1T1 A13,*+A10(7) ; @ @ |103| 8.5 Store result
ADD .L2X   B5,A7,B7 ; @ @ |69| 4.3 Third part of op
ADD .S2    B7,B4,B4 ; @ @ |44| 1.2 Second part of op
ADD .L1    A3,A9,A3 ; @ @ |101| 8.3 Third part of op

MV .L2X    A10,B10 ; @ @ |9|
ADD .S2    B8,B5,B5 ; @ @ |39|
ADD .L1    A14,A0,A0 ; @ @ |61| 3.3 Third part of op
SHRU .S1   A3,0x2,A13 ; @ @ |102| 8.4 Divide by 4 (w/
On the Implementation of MPEG-4 Motion Compensation Using the TMS320C62x

SHRU .S2  B7,0x2,B8 ; @0 70  4.4 Divide by 4 (w/t)
ADD .L1X  B4,A11,A0 ; @0 51  2.1 First part of op
ADD .L2X  B5,A11,B5 ; @0 43  1.1 First part of op
SHRU .S1  A0,0x2,A7 ; @0 62  3.4 Divide by 4 (w/t)
ADD .L2X  B6,A8,B5 ; @0 77  5.3 Third part of op
ADD .L1X  B2,A4,A9 ; @0 84  6.2 Second part of op
ADD .S1  A12,A0,A0 ; @0 53  2.3 Third part of op
ADD .S2  B5,B4,B4 ; @0 45  1.3 Third part of op
ADD .L1  A9,A11,A0 ; @0 91  7.1 First part of op
SHRU .S1  A0,0x2,A12 ; @0 54  2.3 Third part of op
SHRU .S2  B5,0x2,B9 ; @0 78  5.4 Divide by 4 (w/t)
ADD .S1  A0,A1,A8 ; @0 93  7.3 Third part of op
MV .L1X  B6,A0 ; @0 9|
SHRU .S1  A8,0x2,A8 ; @0 94  7.4 Divide by 4 (w/t)
ADD .S1  A9,A5,A5 ; @0 85  6.3 Third part of op
STB .D1T1  A7,**A10(2) ; @0 ^63  3.5 Store result
SHRU .S1  A5,0x2,A5 ; @0 86  6.4 Divide by 4 (w/t)
STB .D1T1  A8,**A10(6) ; @0 ^95  7.5 Store result
STB .D1T1  A5,**A10(5) ; @0 ^87  6.5 Store result
STB .D1T2  B8,**A10(3) ; @0 ^71  4.5 Store result
STB .D2T1  A0,**B10 ; @0 ^47  1.5 Store result
STB .D2T1  B9,**A10(4) ; @0 ^79  5.5 Store result
STB .D2T1  A12,**B10(1) ; @0 ^55  2.5 Store result
NOP 1
ADD .S1  A10,A2,A10 ; @0 ^107 Move p_c to next c
STB .D1T1  A13,**A10(7) ; @0 ^103  8.5 Store result
MV .L2X  A10,B10 ; @0 ^9|

;** ------------------------------------------------------------------*
LDW .D2T2  *+SP(24),B3
MVC .S2  B11,CSR ; 32|
LDW .D2T1  *+SP(20),A14
LDW .D2T1  *+SP(16),A13
LDW .D2T1  *+SP(12),A12
LDW .D2T1  *+SP(8),A11
LDW .D2T1  *+SP(4),A10
B .S2  B3
LDW .D2T2  *+SP(28),B10
LDW .D2T2  *+SP(32),B11
NOP 4
; BRANCH OCCURS
; .endproc
Appendix E  Complete C Code for Motion Compensation

motion_comp.c

 /*************************************************************************
 /*                                                              */
 /* Motion Compensation for MPEG-4                               */
 /*                                                              */
 /* Program: Motion_comp.c                                       */
 /*                                                              */
 /*************************************************************************/

#include <stdio.h>
#include "globals.h"

/* Perform motion compensation */

void perform_MC(uchar ref[NUM_ROWS][NUM_COLS],
 uchar curr[NUM_ROWS][NUM_COLS], const Pixel_Pos position,
 const short size, const MV motion_vector)
{
  int m, n, c_x, c_y, r_x, r_y, mv_x, mv_y;
  int mv_case; /* Type of interpolation */

  /* First, determine the type of interpolation that will be performed. Half-pixel accuracy is supported. If a MV is in quarter-pixel format, which currently is not supported, results are the following:
   - If MV is X.25 or -X.75, no interpolation is done.
   - If MV is -X.25 or X.75, half-pixel interpolation is done.
 "X" is any positive integer. The above results from checking bit 1 of a MV. */

  mv_case = CASE_a; /* Initially assume integer accuracy*/

  /* Notice that CASE_d occurs when both CASE_b and CASE_c occur. */
  /* Half-pixel accuracy is done when bit 1 of a MV is "1". (Q2) */
  if (motion_vector.x & 0x02) /* Half-pixel MC is Top-Down */
    mv_case |= CASE_c;
  if (motion_vector.y & 0x02) /* Half-pixel MC is Left-Right*/
    mv_case |= CASE_b;

  /* TL corner coord of MB in appropriate component of curr image */
  c_x = (int)position.x;
  c_y = (int)position.y;

  /* Coord of TL corner of MB in appropriate component of ref image*/
  /* These coord are relative to the TL corner of current block */
  mv_x = (int)(motion_vector.x>>2); mv_y = int(motion_vector.y>>2);

  /* These coord are relative to the TL corner of reference image */
r_x = c_x + mv_x;                  r_y = c_y + mv_y;

switch (mv_case) {

  case CASE_a:                          /* Integer accuracy   */
    MC_case_a(ref, curr, r_x, c_x, r_y, c_y, 8, 0); /* Copy block */
    break;

  case CASE_b:                          /* Interpolate rows   */
    MC_case_b(ref, curr, r_x, c_x, r_y, c_y, NUM_COLS, 0);
    break;

  case CASE_c:                          /* Interpolate columns */
    MC_case_c(ref, curr, r_x, c_x, r_y, c_y, NUM_COLS, 0);
    break;

  case CASE_d:                          /* Interpolate rows and columns */
    MC_case_d(ref, curr, r_x, c_x, r_y, c_y, NUM_COLS, 0);
    break;

  default:
    puts("Invalid interpolation scheme!");
    puts("Error in motion_comp");
    exit(1);
} /* switch */

/* Motion compensation using four motion vectors per macroblock */
/* performed on the Y, U, and V signal components. */
void motion_comp_4mv(Image *image_ref, Image *image_curr,
const Image_Size size,   const short mb_size,
const Four_MV *motion_vector) {

  short a, b, i, j, k, m, n;                      /* Indices */
  short x, y, mv_x, mv_y;
  short num_mb_tb, num_mb_lr;
  short blk_size;
  short mv_case;                    /* Indicates type of interpolation */
  Pixel_Pos tl_pixel;             /* Indicates top-left pixel of a subblock */
  MV chrom_mv;

  num_mb_tb = size.rows >> MB_SIZE_EXP;       /* # MB top-bottom */
  num_mb_lr = size.cols >> MB_SIZE_EXP;       /* # MB left-right */

  blk_size = mb_size>>1;                /* Block size is half MB size */

  _nassert(num_mb_tb < 2);
  _nassert(num_mb_lr < 2);

  for(i=0; i<num_mb_tb; i++) {
    for(j=0; j<num_mb_lr; j++) {

k = (i*num_mb_lr) + j;        /* Seq #: mv's are in a list */
/* MC for the Y component */
/* FOUR_MV: For all four blocks in MB */

/* Block 0 (TL) */
tl_pixel.x = i*mb_size;      tl_pixel.y = j*mb_size;
perform_MC(image_ref->y, image_cur->y, tl_pixel,
blk_size, motion_vector[k].mv[0]);

/* Block 1 (TR) */
    tl_pixel.y += blk_size;
perform_MC(image_ref->y, image_cur->y, tl_pixel,
blk_size, motion_vector[k].mv[1]);

/* Block 2 (BL) */
tl_pixel.x += blk_size;      tl_pixel.y = j*mb_size;
perform_MC(image_ref->y, image_cur->y, tl_pixel,
blk_size, motion_vector[k].mv[2]);

/* Block 3 (BR) */
    tl_pixel.y += blk_size;
perform_MC(image_ref->y, image_cur->y, tl_pixel,
blk_size, motion_vector[k].mv[3]);

/* MC for the U and V components */
/* There is only one block of each chrom components per MB */
tl_pixel.x = i*blk_size;     tl_pixel.y = j*blk_size;

if (motion_vector[k].mode == ONE_MV) {
    chrom_mv.x = motion_vector[k].mv[0].x;
    chrom_mv.y = motion_vector[k].mv[0].y;
}
else {
    /* FOUR_MV */
    /* The motion vector for the chrominance components is */
    /* the sum of all four MVs in a MB divided by 8. */
    chrom_mv.x = (motion_vector[k].mv[0].x
                + motion_vector[k].mv[1].x
                + motion_vector[k].mv[2].x
                + motion_vector[k].mv[3].x)>>3;
    chrom_mv.y = (motion_vector[k].mv[0].y
                + motion_vector[k].mv[1].y
                + motion_vector[k].mv[2].y
                + motion_vector[k].mv[3].y)>>3;
}

/* MC for the U component */
perform_MC(image_ref->u, image_cur->u, tl_pixel,
blk_size, chrom_mv);
/* MC for the V component */
perform_MC(image_ref->v, image_cur->v, tl_pixel,
blk_size, chrom_mv);
} } /* for i,j */
main.c

#include <stdio.h>
#include "globals.h"

int motion_comp_4mv(Image *image_curr, Image *image_ref,
                     const Image_Size, const short mb_size, const Four_MV *);
Image *allocate_image(Image_Size);
Four_MV *allocate_mv(Image_Size, short);
void create_image(Image *, Image_Size);
void get_motion_vectors(Four_MV *, Image_Size, short);
void print_image(Image *, Image_Size);

int main(void)
{
  Image *image_curr; /* Current image (Y) */
  Image *image_ref; /* Reference image (Y) */
  Image_Size image_size; /* Image size */
  short mb_size; /* Macroblock size */
  Four_MV *motion_vectors; /* Motion vectors (hor & vert) */

  /* Setting up parameters */
  image_size.rows = NUM_ROWS;
  image_size.cols = NUM_COLS;
  mb_size = MB_SIZE;

  /* Allocate reference and current images */
  image_ref  = allocate_image( image_size );
  image_curr = allocate_image( image_size );

  /* Allocate motion vectors */
  motion_vectors = allocate_mv(image_size, mb_size);

  /* Create reference image */
  create_image(image_ref, image_size);

  /* Read motion vector information */
  get_motion_vectors(motion_vectors, image_size, mb_size);

  /* Do motion compensation */
  puts("Performing Motion Compensation...");
  motion_comp_4mv(image_ref,image_curr,image_size,mb_size,motion_vectors);"
/* Done. Exit... */
puts("Motion Compensation: done.");
print_image( image_curr, image_size);
}
TI Contact Numbers

INTERNET

TI Semiconductor Home Page
www.ti.com/sc

TI Distributors
www.ti.com/sc/docs/distmenu.htm

PRODUCT INFORMATION CENTERS

Americas
Phone +1(972) 644-5580
Fax +1(972) 480-7800
Email sc-infomaster@ti.com

Europe, Middle East, and Africa
Phone
Deutsch +49-(0) 8161 80 3311
English +44-(0) 1604 66 3399
Español +34-(0) 90 23 54 0 28
Français +33-(0) 1-30 70 11 64
Italiano +33-(0) 1-30 70 11 67
Fax +44-(0) 1604 66 33 34
Email epic@ti.com

Japan
Phone
International +81-3-3344-5311
Domestic 0120-81-0026
Fax
International +81-3-3344-5317
Domestic 0120-81-0036
Email pic-japan@ti.com

Asia
Phone
International +886-2-23786800
Domestic
Australia 1-800-881-011
TI Number -800-800-1450
China 10810
TI Number -800-800-1450
Hong Kong 800-96-1111
TI Number -800-800-1450
India 000-117
TI Number -800-800-1450
Indonesia 001-801-10
TI Number -800-800-1450
Korea 080-555-2804
Malaysia 1-800-800-011
TI Number -800-800-1450
New Zealand 000-911
TI Number -800-800-1450
Philippines 105-11
TI Number -800-800-1450
Singapore 800-0111-111
TI Number -800-800-1450
Taiwan 0800-006800
Thailand 0019-991-1111
TI Number -800-800-1450
Fax 886-2-2378-6808
Email tiasia@ti.com

TI and VelocITI are trademarks of Texas Instruments Incorporated.
IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE (“CRITICAL APPLICATIONS”). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER’S RISK.

In order to minimize risks associated with the customer’s applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty, or endorsement thereof.

Copyright © 1999 Texas Instruments Incorporated