An 8x8 DCT Implementation on the Motorola DSP56800E

Application Note

by
Brad Zwernemann

AN2123/D
Rev. 0, 08/2001

MOTOROLA

How to reach us:

USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado, 80217.
1–303–675–2140 or 1–800–441–2447

JAPAN: Motorola Japan Ltd.; SPS, Technical Information Center, 3–20–1, Minami–Azabu, Minato–ku,
Tokyo 106–8573 Japan. 81–3–3440–3569

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd., Silicon Harbour Centre, 2 Dai King Street,
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852–26668334

Technical Information Center: 1–800–521–6274

HOME PAGE: http://www.motorola.com/semiconductors/

Abstract and Contents

The 8x8 discrete cosine transform (DCT) is an efficient, real-valued transform often used in image compression. Special, fast algorithms for the DCT have been developed to accommodate the many arithmetic operations involved in implementing the DCT directly. In this application an implementation of a fast DCT algorithm is presented for the Motorola DSP56800E processor. The details of the implementation are discussed and the results are presented.

1 Introduction

This document describes the implementation of a two-dimensional (2-D) discrete cosine transform (DCT) on the Motorola DSP56800E. The DCT is a member of the family of sinusoidal transforms and is often used in image compression. The DCT is a real-valued transform with mirror-periodicity and as a result is more efficient than the discrete Fourier transform (DFT) for compression. This application implements an 8x8 2-D DCT for compression algorithms such as JPEG.

The one-dimensional (1-D) DCT is defined as:

$$(1)\quad V(k) = C_N(k)\sum_{x=0}^{N-1} s(x)\cos\left(\frac{(2x+1)k\pi}{2N}\right) \quad k = 0,1,\ldots,N-1 \qquad \textbf{Eqn. 1}$$

The corresponding 2-D DCT is defined as:

$$Y(k_1,k_2) = C_N(k_1)C_N(k_2)\sum_{y=0}^{N-1}\sum_{x=0}^{N-1} s(x,y)\cos\left(\frac{(2x+1)k_1\pi}{2N}\right)\cos\left(\frac{(2y+1)k_2\pi}{2N}\right) \qquad \textbf{Eqn. 2}$$

where

$$(3)\quad C_N(k) = \begin{cases}\sqrt{1/N} & k = 0\\ \sqrt{2/N} & k = 1,2,\ldots,N-1\end{cases} \qquad \textbf{Eqn. 3}$$

2 Implementing the DCT

Implementing the DCT as in Equation 1 and Equation 2 is inefficient due to the large number of adds and multiplies, so a fast algorithm is used. There are several fast algorithms for implementing the 2-D DCT. These algorithms exploit symmetries to greatly reduce the number of operations required in Equation 2. Algorithms that optimize the entire 2-D structure are efficient but complex. Another option is to optimize the 1-D algorithm. For this application a method using an optimized 1-D DCT as a kernel for the 2-D DCT was chosen in order to keep the implementation simple. This method utilizes the flow graph in Figure 1 to calculate the 1-D DCTs of each row of the 8x8 input, the result of which is an 8x8 intermediate value *V*(i,j).

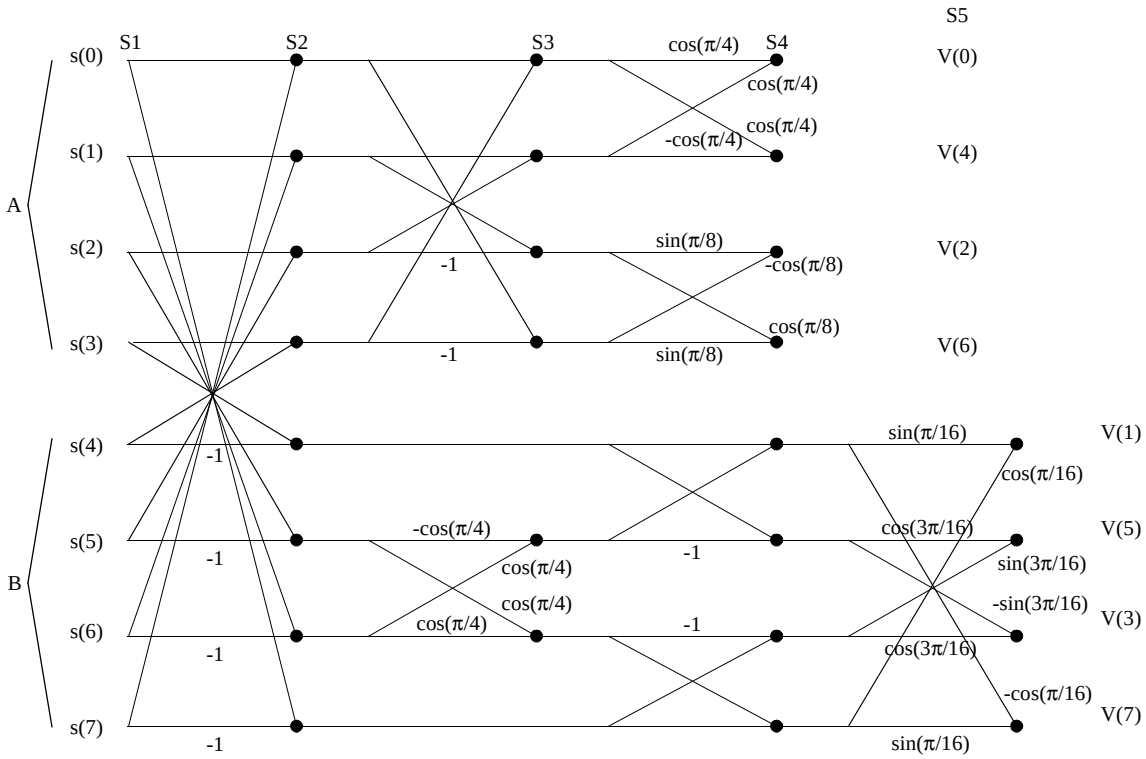

Figure 1. 1-D DCT Flow Graph [1]

The intermediate result is created with the first pass of a loop that calculates eight 8-point 1-D DCTs. This loop is then used again in a second pass with *V*(i,j) as the input and *Y*(n,m), the final result of the 2-D DCT, as the output. In the second pass, the DCTs of the columns of *V*(i,j) are calculated. To simplify the use of memory pointers and prevent the need to parse through both rows and columns, the output of each pass is stored transposed. Therefore, upon entry to the second pass the input *V*(i,j) is transposed, so we calculate for rows as in the first pass rather than columns. Working with rows for both passes simplifies pointer usage. The result of the second pass is stored transposed giving a correctly oriented final result *Y*(n,m).

Each pass requires a scale factor of $\sqrt{(2/N)}$. After both passes the overall scale factor is equal to the square of the individual pass factors or 2/N. There are several ways to implement the scaling. A direct implementation scales the intermediate result *V*(i,j) by the $\sqrt{(2/N)}$ factor. This method requires extra multiplies for the scaling and limits precision due to round-off errors. The most precise way to implement the scaling is to scale the final result, but this method still uses extra multiplies and must also use memory moves making the method inefficient. Therefore, to reduce the number of multiplies needed while maintaining precision, the scaling factor $\sqrt{(2/N)}$ is not included directly in the DCT calculations.

In this implementation, we utilize the overall scaling 2/N which for N = 8 is equivalent to 1/4. The scaling is accomplished in the final result calculation by dividing the second pass coefficients (e.g., cos(π/4) in Figure 1) by 4. Note that only the coefficients in the final multiplies of the flow graph (S4 and S5 in Figure 1) are scaled. The division is done outside the program to create a second scaled set of coefficients. This method allows for near-optimal precision without having to add any multiplies or moves.

One benefit of the particular algorithm used here is that after the calculations for S2 are done the algorithm can be split into two independent parts. The first part, part A in Figure 1, includes the top four outputs of the flow graph and the second part includes the bottom four, part B. Each part is only dependant on four intermediate results between stages. These results can be kept in registers to limit memory traffic. Working with all eight outputs simultaneously would require either eight data registers or more extensive memory usage.

Another helpful property of this particular flow graph is the ‘a+b, a–b’ structure that allows data to be held in registers for multiple calculations. This structure is found throughout the graph and helps cut down on memory traffic. For example, in Code Example 1, the results s2(7) and s2(0) are kept in the registers y0 and b respectively for use in S3.

Code Example 1. ‘a+b, a–b’ Structure

```
in C:
            s2[0] = input[i][0] + input[i][7];
            s2[7] = input[i][0] - input[i][7];

in assembly:

            move.w      x:(r0)+,y0   x:(r3)-,x0   ;// s1(0) -> y0, s1(7) -> x0
            tfr         y0,b                      ;// s1(0) -> b
            sub         x0,y0                     ;// y0 = s1(0) - s1(7) = s2(7)
            add         x0,b                      ;// b = s1(0) + s1(7) = s2(0)
```

3 Implementation on the DSP56800E

The DSP56800E software that implements the 2-D DCT uses two loops to accomplish the two passes of the DCT described in Section 2. The inner loop is the kernel 1-D DCT of the flow graph. The outer loop calls the inner loop twice for the two passes. The first pass of the inner loop calculates DCTs for the rows of the input. The output of the first pass is then used as the input for the second pass. The output of the second pass is the final result of the function.

3.1 Initialization

When the function is called, a pointer to the 8x8 word input signal should be in register R2 and a pointer to the final 8x8 word output should be in register R3. The function can be called from C code with the command `DCT_2D(input,output)` where `input` and `output` are the appropriate pointers.

Address registers R0, R1, R2, R3, R4, R5, and N as well as data registers X0, Y, A, B, C, and D are used in this implementation. Constants for the coefficients are defined at the beginning of the code. Although the flow diagram uses a total of six different coefficients, there are twenty coefficients stored as constants. The additional coefficients are repeated coefficients for more efficient pointer use and scaled coefficients for output scaling. Notice that *C41* in the second set of coefficients is not scaled because it is not used in the final stage of calculations. Space is defined for the output of the first pass with the label *V*. Space for storage of flow graph stage results is defined with the label *S*.

Code Listing 1 shows the memory allocation for the variables and the *S* and *V* spaces.

Code Listing 1. Memory Allocation

```
            ORG     X:

cos_tab     DC      $5a82,$30fb,$7641,$5a82  ;// c41, s81, c81, c41
cos_tab2    DC      $18f8,$7d8a,$6a6d,$471c  ;// s161, c161, c163, s163
cos_tab3    DC      $7d8a,$18f8,$16a1,$0c3f  ;// c161, s161, c41/4, s81/4
cos_tab4    DC      $1d90,$5a82,$063e,$1f63  ;// c81/4, c41, s161/4, c161/4
cos_tab5    DC      $1a9b,$11c7,$1f63,$063e  ;// c163/4, s163/4, c161/4, s161/4
                                             ;// c41 = cos(pi/4)
                                             ;// c81 = cos(pi/8)      s81 = sin(pi/8)
                                             ;// c161 = cos(pi/16)    s161 = sin(pi/16)
                                             ;// c163 = cos(3pi/16)   s163 = sin(3pi/16)

S           DS      8                        ;// holds results of flow graph stages
V           DS      64                       ;// holds intermediate DCT of rows of input
```

The initial instructions of the function are outside of the loop structure and serve to initialize pointer values in registers. The input pointer is transferred to R0, the output pointer is transferred to R4, and an offset input pointer is transferred to R3. The pointer to the coefficient table is put in R5, the pointer to the space *S* is put into R1, the pointer to the space *V* is put into R2, and the value 16 is loaded into the N register for indexing. The inner loop consists of seven sections that correspond to the flow graph. The top portion of the graph (part A) has three stages of calculations: S2, S3, and S4. These stages are completed first followed by the bottom portion of the flow graph (part B) which includes four stages of calculations: S2, S3, S4, and S5. Refer to the corresponding assembly code below.

Code Listing 2. Pointer Initialization and Loop Calls

```
FDCT_2D:

tfra    r3,r4                           ;// output -> r4
tfra    r2,r0                           ;// input -> r0
moveu.w #cos_tab,r1                     ;// cos_table -> r1
tfra    r1,r5                           ;// cos_table -> r5
moveu.w #S,r1                           ;// S -> r1
moveu.w #V,r2                           ;// V -> r2
move.w  #16,N                           ;// index for transposed output storage
adda    #7,r0,r3                        ;// input+7 -> r3
do      #2,EndOuterLoop                 ;// rows then cols
do      #8,EndInnerLoop                 ;// 1-D DCT
```

3.2 Part A of Flow Graph

Beginning with S2 (A), the input values are read from memory and stored in registers using the input pointers R0 and R3. This stage calculates the input values for S3(A) which are kept in registers B, D, Y0, and Y1. The input values for S3(B) are calculated and stored in memory using R1 as a pointer to the memory space S. Near the end of S2(A) the pointer to the coefficient table is transferred from R5 to R3 where it will be used for the remainder of the iteration of the loop. Refer to the corresponding assembly code below. Stage 2A is shown in Code Listing 3.

Code Listing 3. Stage 2A

```
;// start stage 2A (stage 1 is the collection of data into input array)

move.w  x:(r0)+,y0      x:(r3)-,x0      ;// s1(0) -> y0, s1(7) -> x0
tfr     y0,b                            ;// s1(0) -> b
sub     x0,y0                           ;// y0 = s1(0) - s1(7)
add     x0,b    x:(r0)+,y1 x:(r3)-,x0   ;// b = s1(0) + s1(7) = s2(0), s1(1) -> y1, s1(6) -> x0
tfr     b,d                             ;// d = s2(0)
tfr     y1,b    y0,x:(r1)+              ;// s1(1) -> b, y0 -> s2(7)
sub     x0,y1                           ;// y1 = s1(1) - s1(6) = s2(6)
add     x0,b    x:(r0)+,y0 x:(r3)-,x0   ;// b = s1(1) + s1(6) = s2(1), s1(2) -> y0, s1(5) -> x0
tfr     y0,a    y1,x:(r1)+              ;// s1(2) -> a, y1 -> s2(6)
sub     x0,a    x:(r0)+,y1 x:(r3)+,c    ;// a = s1(2) - s1(5) = s2(5), s1(3) -> y1, s1(4) -> c
add     x0,y0                           ;// y0 = s1(2) + s1(5) = s2(2)
tfr     y1,a    a,x:(r1)+               ;// s1(3) -> a, a1 -> s2(5)
tfra    r5,r3                           ;// cos_table -> r3
sub     c,a                             ;// c = s1(3) - s1(4) = s2(4)
add     c,y1                            ;// y1 = s1(3) + s1(4) = s2(3)
move.w  a,x:(r1)                        ;// a -> s2(4)
```

For stage S3(A) all of the necessary inputs are in registers from S2(A) so there is no need for memory moves. The results are calculated and kept in registers A, B, C, and D as inputs for stage S4(A), the final stage for section A. This stage uses the inputs created in S3(A) and two coefficients from the table to calculate the values V(0), V(2), V(4), and V(6) which are stored to memory using the pointer to V held in R2. The register R2 is incremented using the N register with a value of 16. This causes the values to be stored transposed in the 8x8 space V. Stages 3A and 4A are shown in Code Listing 4.

Code Listing 4. Stages 3A and 4A

```
;// stage 3A

tfr     y1,a                            ;// s2(3) -> a
add     d,a                             ;// a = s2(0) + s2(3) = s3(0)
sub     y1,d                            ;// d = s2(0) - s2(3) = s3(3)
tfr     b,c                             ;// s2(1) -> c
tfr     y0,b                            ;// s2(2) -> b
add     c,b                             ;// b = s2(1) + s2(2) = s3(1)
sub     y0,c                            ;// c = s2(1) - s2(2) = s3(2)
;// stage 4A

tfr     b1,y1                           ;// s3(1) -> y1
add     a,b     x:(r3)+,y0              ;// b = s3(1) + s3(0), c41 -> y0
sub     y1,a    x:(r3)+,x0              ;// a = s3(0) - s3(1), s81 -> x0
mpyr    y0,b1,b                         ;// b = c41*(s3(1) + s3(0)) = s4(0)
mpyr    y0,a1,a x:(r3)+,y1              ;// a = c41*((0) - s3(1)) = s4(1), c81 -> y1
move.w  b1,x:(r2)+N                     ;// s4(0) -> v(0)
mpyr    y1,d1,b                         ;// b = s3(3)*c81
macr    x0,c1,b                         ;// b = s3(2)*s81 + s3(3)*c81 = s4(2)
mpyr    -y1,c1,y1                       ;// y = -c81*s3(2)
macr    d1,x0,y1                        ;// y = s81*s3(3) - c81*s3(2) = s4(3)
move.w  b1,x:(r2)+N                     ;// s4(2) -> v(2)
move.w  a1,x:(r2)+N                     ;// s4(1) -> v(4)
```

3.3 Part B of Flow Graph

Part B starts with S3 in the flow graph. The values stored to S in S2(A) are now retrieved using the R1 register. The register is decremented and reset to the beginning of S in this process. These values are put into the A, B, C, and D registers. In this stage two of the values are unchanged and passed directly to the next stage in registers C and D. The other two values are calculated using one coefficient and left in registers A and B for the next stage. The next stage S4(B) does not require coefficients and is therefore very simple. The four input values for S5(B) are calculated and left in registers Y0, B, C, and D. Stages B3 and B4 are shown in Code Listing 5.

Code Listing 5. Stages 3B and 4B

```
;// end of top side, start stage 3B

move.w  x:(r1)-,c                       ;// c = s2,3(4)
move.w  x:(r1)-,a                       ;// a = s2(5)
move.w  x:(r1)-,b                       ;// b = s2(6)
move.w  x:(r1),d                        ;// d = s2,3(7)
move.w  x:(r3)+,y0                      ;// c41 -> y0
mpyr    b1,y0,b                         ;// c41*s2(6) -> b
mpyr    a1,y0,a y1,x:(r2)+              ;// c41*s2(5) -> a, s4(3) -> v(6)
adda    #-41,r2                         ;// reset r2 to v(1)
tfr     a,y1                            ;// c41*s2(5) -> y1
add     b,a                             ;// a = c41*s2(5) + c41*s2(6) = s3(6)
sub     y1,b    x:(r3)+,y1              ;// b = c41*s2(6) - c41*s2(5) = s3(5), s161 -> y1

;// start stage 4B

tfr     d1,y0                           ;// s3(6) -> y0
add     a,y0                            ;// a = s3(7) + s3(6) = s4(7)
sub     a,d                             ;// d = s3(7) - s3(6) = s4(6)
tfr     b,x0                            ;// y0 = s3(5)
add     c,b     x:(r3)+,a1              ;// b = s3(5) + s3(4) = s4(4), s161 -> y1, c161 -> a1
sub     x0,c                            ;// c = s3(4) - s3(5) = s4(5)
```

Stage S5(B) is the most complex of the seven segments due to the use of four coefficients and the storage to memory of the final results. The R3 register continues to be used as the coefficient pointer and R2 is the pointer to the output of the inner loop. As in stage S4(A), R2 is incremented with the N register for transposed storage to memory. This stage calculates V(1), V(3), V(5), and V(7). This stage also resets the pointers for the next iteration of the inner loop. R0 is incremented by four and points to the next row of the input. R3 is given an offset value of R0, and R2 is reset to the next column of the intermediate result V. Stage B5 is shown in Code Listing 6.

Code Listing 6. Stage 5B

```
;// start stage 5B

mpyr    a1,y0,a x:(r3)+,x0          ;// a = c161*s4(7), c163 ->x0
macr    b1,y1,a x:(r3)+,y1          ;// a = s161*s4(4) + c161*s4(7) = v(1), s163 -> y1
nop                                 ;// pipeline stall
move.w  a1,x:(r2)+N                 ;// a1 -> v(1)
mpyr    -c1,y1,a                    ;// a = -s163*s4(5)
macr    d1,x0,a                     ;// a = c163*s4(6) - s163*s4(5) = v(3)
adda    #4,r0                       ;// r0 = input+8
move.w  a1,x:(r2)+N                 ;// a1 -> v(3)
mpyr    y1,d1,a                     ;// a = s163*s4(6)
macr    c1,x0,a                     ;// a = c163*s4(5) + c163*s4(6) = v(5)
move.w  x:(r3)+,y1                  ;// c161 -> y1
move.w  x:(r3),x0                   ;// s161 -> x0
mpyr    -b1,y1,b                    ;// b = -c161*s4(4)
macr    y0,x0,b a1,x:(r2)+N         ;// b = s4(7)*s161 - c161*s4(4) = v(7), a1 -> v(5)
adda    #7,r0,r3                    ;// input+7 -> r3
move.w  b1,x:(r2)                   ;// b1 -> v(7)
adda    #-55,r2                     ;// reset r2 to v+1
```

3.4 Reinitialization for Second Iteration of Outer Loop

After eight iterations of the inner loop several adjustments are made before beginning the second iteration of the outer loop. The pointer to the coefficient table, R5, is set to point to the second, scaled set of coefficients. The register R0 is loaded with an offset value of R2 and is now the pointer to *V*, which is the input for the second iteration of the outer loop. The R2 register is loaded with the value from R4 which is the pointer to the final result. Finally R3 is loaded with an offset value of R0 as an offset input pointer. Teh reinitializatioin code shown in Code Listing 7.

Code Listing 7. Setup for Second Iteration of Outer Loop

```
adda    #10,r5                      ;// set cos_tab to second set of coef's
adda    #-8,r2,r0                   ;// get rid of +8 to V by inner loop, put in r0 as input
tfra    r4,r2                       ;// output -> r2
adda    #7,r0,r3                    ;// input+7 -> r3
```

The second iteration of the outer loop is run. At the end of the second iteration the final result is in the 8x8 memory space pointed to by the output pointer passed into the function.

4 The Inverse DCT

When the DCT is used in an application such as image compression, it is necessary to use the inverse of the transform for the image decompression. The 2-D Inverse DCT (IDCT) is defined as:

$$s(x,y) = C_N(k_1)C_N(k_2)\sum_{k_2=0}^{N-1}\sum_{k_1=0}^{N-1}Y(k_1,k_2)\cos\left(\frac{(2x+1)k_1\pi}{2N}\right)\cos\left(\frac{(2y+1)k_2\pi}{2N}\right) \qquad \textit{Eqn. 4}$$

The equation for the IDCT is the same as that of the DCT with the indices k1 and x reversed and the indices k2 and y reversed in the summations. The reversal of these indices affects the values of the cosine factors in such a way that the algorithm developed for the DCT cannot be reused for the IDCT with simple cosine factor changes. However, the IDCT can be implemented with the 1-D signal flow using the same flow graph as the DCT with one major change. The signal flow from the flow graph in Figure 1 must be reversed. The resulting graph can then be used in the calculation of the IDCT.

The assembly language implementation of the IDCT differs from that of the DCT in its order of operations but it is very similar in technique and form. Performance of the IDCT is also similar to the DCT in both code size and cycle count. Due to the similarities with the implementation of the DCT, the IDCT implementation is not presented here.

5 Results

The methods used in this implementation maintain precision while optimizing the algorithm for use with the 56800E processor. All rounding operations were performed with the 56800E rounding capabilities through the `MPYR` and `MACR` instructions. The precision and accuracy were confirmed by testing the output of the function against the output of a floating point C implementation and deriving the RMS error over the 8x8 data. The program size and execution speed are shown in Table 4-1.

Table 5-1. Program Cycle Count and Memory

Code Segment	Cycle Count	Program Words
Init	8	3
Data		92
Kernel	1193	95

6 Acknowledgements

The author would like to thank Kim-Chyan Gan for assistance with algorithm development, and Joseph Gergen for assistance with algorithm development and assembly code optimization.

7 References

[1] W. H. Chen, C. H. Smith, and S. C. Fralick, “A Fast Computational Algorithm for the Discrete Cosine Transform,” *IEEE Trans. Communications*, vol. 25, pp. 1004–1009, 1977.