7 Discrete Cosine Transform

7.1 OVERVIEW

The Discrete Cosine Transform, or DCT, transforms data into a format that can be easily compressed. The characteristics of the DCT make it ideally suited for image compression algorithms. These algorithms let you minimize the amount of data needed to recreate a digitized image.

Reducing digitized images into the least amount of data possible has the following advantages:

- Less memory required to store images
- Channel bandwidth efficiency increased when you transmit images
- Less time may be needed to analyze images

Performing the DCT on a digitized image creates a data array that can be compressed by data compaction algorithms. Then, data can be stored or transmitted in its compacted form. The image quality depends on the amount of quantization used in the compaction algorithm. To reproduce the original image, the data is retrieved from memory, uncompacted, and an inverse DCT is performed.

Some of today's most popular image data compression applications include:

- Teleconferencing using motion-compensated video codecs
- ISDN multimedia communications including voice, video, text, and images
- Video channel transmission using commercial geosynchronous telecommunications satellites
- Digital facsimile transmission using dedicated equipment and personal computers

This chapter describes a basic implementation of the DCT.

7 Discrete Cosine Transform

7.2 BACKGROUND

Several image data compression algorithms use the DCT to remove spatial data redundancies in two-dimensional (2D) data. Images are subdivided into smaller, two-dimensional blocks. These blocks are then processed independently of the neighboring blocks.

Figure 7.1 illustrates how a two-dimensional discrete cosine transform is performed on a block of data. In general, the two dimensional, discrete cosine transform (2D DCT) transforms an (n x n) data array into an (n x n) result array. First the DCT transforms the columns, then it transforms the rows. The resulting data elements are called the *transform coefficients*, or *DCT coefficients*. For example, if you use 8 x 8 blocks of 8-bit input data, an 8-point DCT is performed on each row in the block. This creates a new 8 x 8 block of data. Next, an 8-point DCT is performed on each column of the new block. This generates an 8 x 8 block of 12-bit output values. These 64 output values are the DCT coefficients.

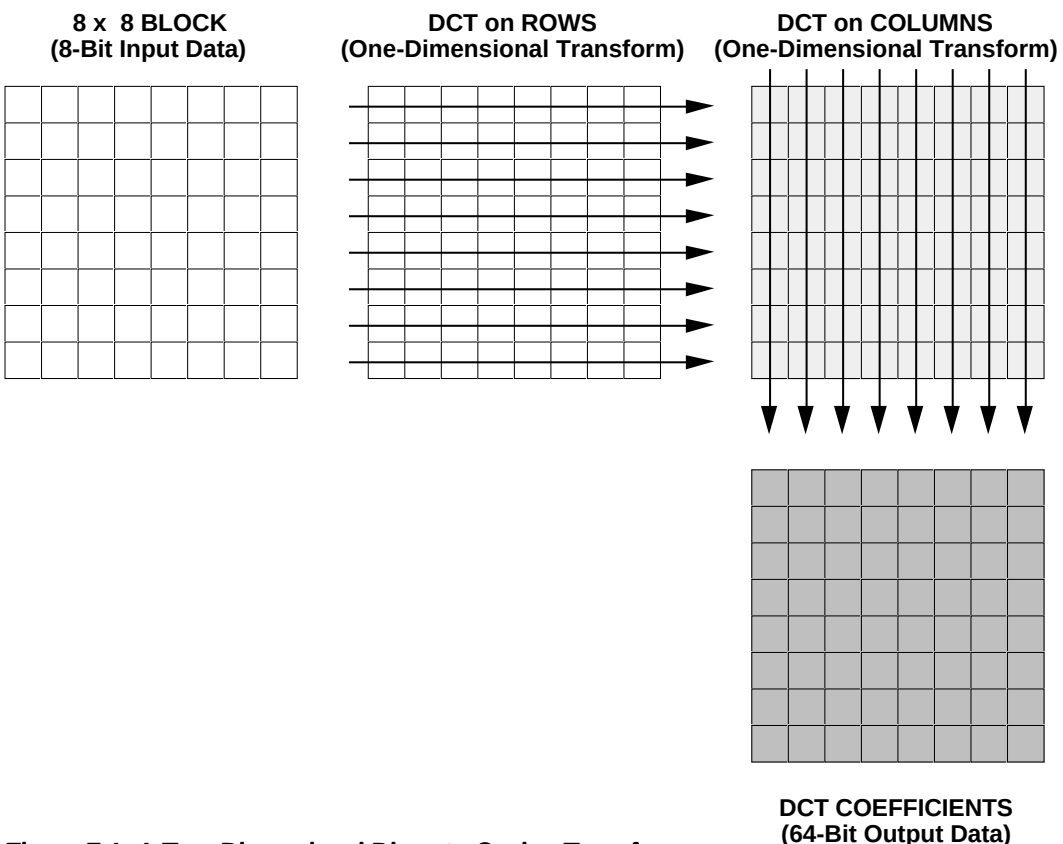


Figure 7.1 A Two-Dimensional Discrete Cosine Transform

After the DCT is calculated, the data can be reduced to concentrate the important information into a few of the coefficients, leaving the remaining coefficients equal to zero, or otherwise "insignificant." Typically, the (n x n) result array is quite sparse; this is the desired energy compaction effect.

When you transmit only the coefficients with large values, the total volume of data is reduced. You can use several methods to choose which coefficients to transmit. Once the coefficients are chosen and quantized, you can use additional algorithms, such as Huffman coding or run-length coding algorithms, to achieve a higher data compression ratio.

Either the DCT or the discrete Fourier transform (DFT) could be used in image compression algorithms, however, the characteristics of the DCT make it better suited for execution on ADSP-2100 family processors. Table 7.1 highlights the important differences between these two transforms.

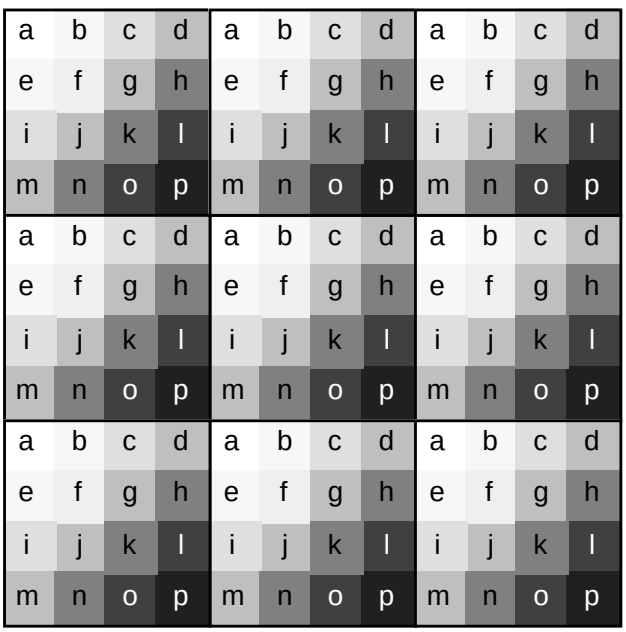
Cosine Transform	*Fourier Transform*
Real arithmetic only, real data and coefficients	Requires complex arithmetic, complex data, complex coefficients
Excellent image energy compaction	Very good image energy compaction
Phase information not available	Phase, magnitude available
Blocking artifact not as apparent	Blocking artifact noticeable
Assumes data outside window is mirror-image of data inside window	Assumes data outside window is a duplicate of data window, shifted

Table 7.1 Cosine Transform vs. Fourier Transform Characteristics

Often, a receiver can reconstruct a complex image with relatively few retained transform coefficients. Depending on the number of Fourier coefficients retained in DFT compression, the reconstructed image may exhibit visible block boundaries because of the Gibbs phenomenon. This is called the blocking artifact. Figure 7.2 demonstrates how images reconstructed from DCT coefficients exhibit less blocking artifact than those reconstructed from Fourier coefficients.

7 Discrete Cosine Transform

Discrete Fourier Transform



Discrete Cosine Transform

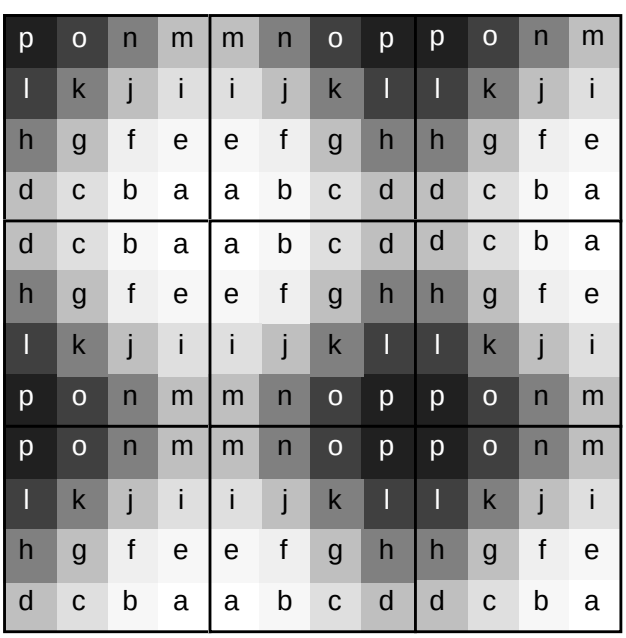


Figure 7.2 The DCT Reduces The Blocking Artifact

The letters in Figure 7.2 correspond to pixel intensity. The top half of Figure 7.2 shows four 4 x 4 blocks after the DFT is performed. Because of the way the DFT transforms a block, it expects the neighboring blocks to be exact copies. Notice that the differing intensity of adjacent pixels in neighboring blocks gives the appearance of borders between the blocks, thus increasing blocking artifacts.

The bottom half of Figure 7.2 shows four blocks after the DCT is performed. The DCT expects neighboring blocks to be mirror images. In this example, adjacent pixels across the borders of the 4 x 4 blocks appear to have the same intensity, thereby reducing blocking artifact.

Computationally, the DCT is more efficient than the DFT because it is a completely real transform and does not require complex variables or arithmetic. You can increase the computational speed of the DCT by using a fast cosine transform (FCT or FDCT) algorithm. This chapter describes implementation of the DCT using a fast algorithm first presented by H. S. Hou (see references).

Table 7.2 lists benchmark times for executing the DCT using ADSP-2100 family microcomputers.

DSP	*Processor Speed*	*8x8*	*16x16*
		2492 cycles	10046 cycles
ADSP-2101/2111	20 MHz	0.1246 ms	0.5023 ms
ADSP-2171	33 MHz	0.0755 ms	0.3044 ms

Table 7.2 Benchmark Times For Executing The DCT

This chapter includes programming examples for the one-dimensional DCT and the two-dimensional DCT. One-dimensional DCTs are used more often for speech compression applications, while two-dimensional transforms are commonly used for image data compression. The two-dimensional transform is implemented by performing one-dimensional transforms on each row of an image, then performing one-dimensional transforms on each column (refer to Figure 7.1). The routines printed in this chapter are in-place routines. In other words, the results of the DCT computation are written over the input data buffer values.

7 Discrete Cosine Transform

The formal, mathematical definition for the one-dimensional cosine transform is:

$$F(u) = \frac{2c(u)}{N} \sum_{m=0}^{N-1} f(m) \cos\left(\frac{(2m+1)u\pi}{2N} \right), \text{ where } u = 0, 1, 2, \ldots, N-1$$

where

$$c(u) = \frac{\sqrt{2}}{2}, \text{ for } u = 0$$

$$c(u) = 1, \text{ for } u = 1, 2, 3, \ldots, N-1$$

Often, textbooks define c(u) at u=0 as one divided by the square-root of two. The square-root of two divided by two yields the same result and is used to simplify calculations. The output coefficient F(0) is often referred to as the DC component, or the DC term.

The two-dimensional DCT is mathematically defined as:

$$F(u, v) = \frac{4c(u,v)}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} f(m, n) \cos\left(\frac{(2m+1)u\pi}{2N} \right) \cos\left(\frac{(2n+1)v\pi}{2N} \right)$$

for u,v = 0,1,2,...,N-1

where

$$c(u, v) = \frac{1}{2}, \text{ for } u = v = 0$$

$$c(u, v) = 1, \text{ for } u, v = 1, 2, \ldots, N-1$$

Two-dimensional transforms are equivalent to 2N-pt one-dimensional transforms.

7.3 COMPUTATIONAL METHODS

There are many fast algorithms for accelerating the computation of the DCT. Most algorithms can be classified into one of three categories:

- indirect computation
- direct matrix factorization
- recursive computation

Indirect computation involves doubling the length of an N-pt sequence to a 2N-pt sequence with its mirror image. The result is geometrically even waveform about its centerpoint. A 2N-pt FFT is then performed on that sequence and its results are multiplied by a complex exponential phase-shift vector. When correctly executed, the result is the DCT. However, this process requires lengthy FFT calculations that involve complex numbers.

Implementing the one-dimensional DCT described in the above equation, reordering the input and output sequences, and ignoring the DC scale coefficient c(u) yields a simple matrix-multiply operation. Using this as a starting point, the *matrix factorization* techniques yielded several fast algorithms that required significantly less arithmetic operations than the full-matrix multiply. Most importantly, these techniques do not require complex operations or many data moves, unlike the FFT method.

The drawback to matrix factorization methods is that a new value of N requires factoring the matrix again. This yields a different solution and compromises flexibility.

Probably the best (numerically and arithmetically efficient) method is a *recursive method* proposed by H. S. Hou in 1987.

7.4 HOU'S FAST DISCRETE COSINE ALGORITHM

Hou's Fast Discrete Cosine Transform Algorithm (FDCT) is numerically stable, fast and recursive. Similar to the Cooley-Tukey FFT algorithm, It generates the next larger DCT from two identical, smaller DCTs. This deviates from direct factorization algorithms that factor the desired N-pt DCT matrix. In that, the higher order matrices are generated from lower order DCT matrices instead. Refer to Hou's paper (see the references) for a tutorial on the DCT in general and his algorithmic implementation.

Hou's algorithm can be efficiently implemented on ADSP-2100 family processors because of the DSP's internal architecture. This architecture provides an ALU, MAC, and barrel shifter connected in parallel through

the internal result bus. The results of any ALU, MAC or shifter operation is available to any of these computational units on the next processor cycle. Also, during any ALU, MAC, or shifter operation, two new operands (one from data memory, one from program memory) can be fetched. This means that you can perform a multiply accumulate instruction and fetch a new cosine value (from PM) and new data value (from DM) during the same instruction cycle.

Zero-overhead looping and the recursive nature of Hou's algorithm, gives two-dimensional DCTs an extraordinarily fast execution time. Because you can mix index and modify registers in both data address generators, you do not need to sequence data in memory. You choose the correct modify register after a data access so the index register points to the next desired data value. The resulting assembly code is easy to read, self-documenting, and can be separated into small, manageable modules. (For more information, see Section 7.5, "Zig-Zag Scanning of DCT Coefficients", and Section 7.6, "Zig-Zag Scanning and ADSP-21xx Processors")

The programming examples presented in this chapter perform the cosine transform in-place. This means the input data values are read from a buffer (arranged in normal, sequential order) and the results are written back to the same buffer in the same order. Hou's method was chosen because it simplifies two-dimensional transforms and it uses relatively little data memory space. If you do not need an in-place DCT, change the pointer to point to a different output buffer before the bit-reversing routine is called.

Figure 7.3 illustrates that an N=16 DCT can be recursively separated into smaller DCTs (N=8, N=4, and N=2.) This diagram was drawn with the same notation that Hou used to demonstrate the recursive nature of this method, and to extend the diagram to N=16.

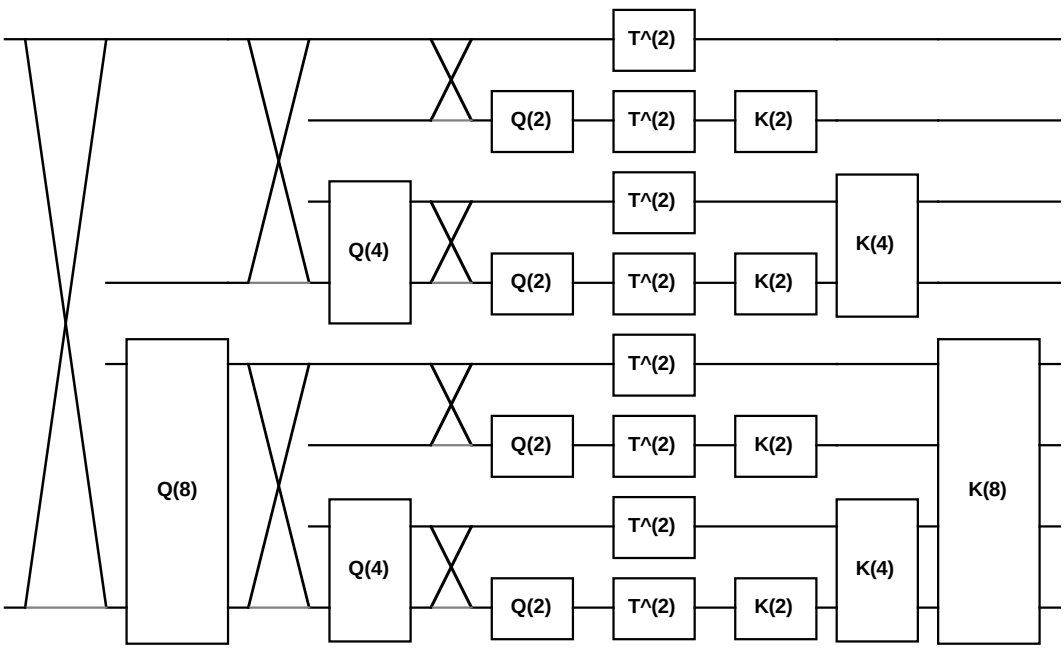


Figure 7.3. Implementation Of An N=16 DCT Using Hou's Matrix Notation

Figure 7.4 shows the equivalent signal flow graph of a fast 16-pt DCT. The code in Listings 7.1–7.10 implements this structure. The first routine, called DIF16, reads the input data and performs eight, decimation in frequency, radix-2, real butterflies with real cosine coefficients. The results are stored in a 16-word buffer called TMP. Next, the DIF8 routine computes two sets of four DIF real butterflies on the TMP buffer, in-place. Then DIF4 computes four sets of two DIF butterflies on TMP, in place. This is followed by DIF2, which performs eight sets of a single DIF butterfly on TMP, in-place. Referring to the diagram shown in Figure 7.3, the section just described implements the Q(8), Q(4), Q(2), and T^(2) blocks. The next subroutines that are called in Listing 7.1, implement the K(2), K(4), K(8) blocks, and the last subroutine, called DCBREV (in Listings 7.1 and 7.10), scales the DC term by the reciprocal of the square root of two and performs the bit-reversing required to write the final TMP buffer values to the original input buffer locations in normal order. The TMP buffer is used to hold the in-place results of each subroutine between successive subroutine calls.

Very little memory is required to compute the DCT. In addition to the input data buffer, only 16 data memory locations are used for a 16-pt DCT (8 locations for an 8-pt DCT). These locations comprise the TMP buffer. Listings 7.10 and 7.18 use two additional data memory locations for the

two-dimensional implementations of the 16-pt and 8-pt transforms, respectively. These two locations, called XADR and XADR2, store pointers to the correct row and columns of a two-dimensional array during the two-dimensional transform.

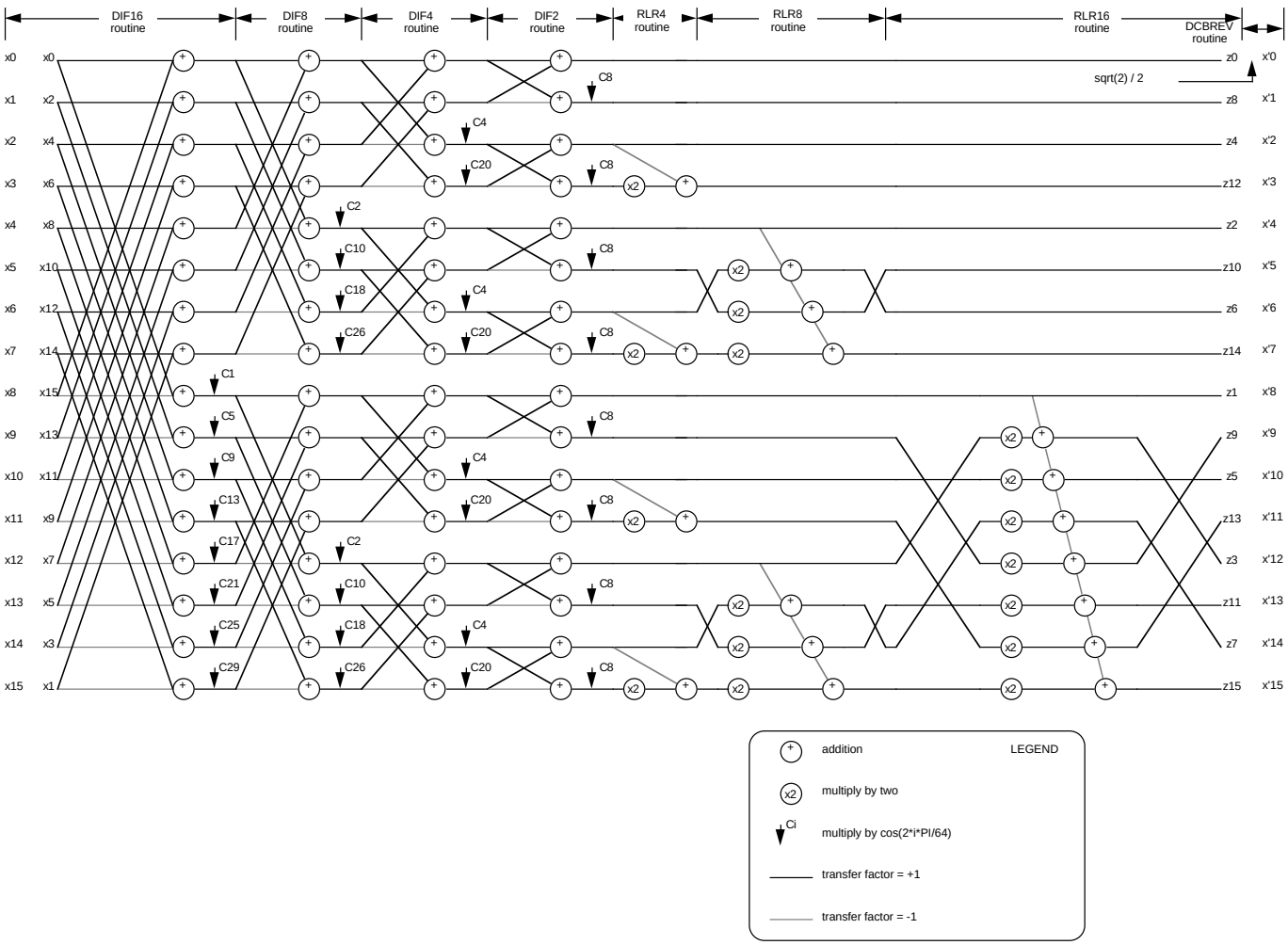


Figure 7.4. Signal Flow Graph For A Fast DCT

These listings also use program memory efficiently. Only 15 real cosine coefficients are stored in program memory for the 16-pt transforms, and seven real cosine coefficients for the 8-pt transforms. The instructions require few program locations because they are stored in short, recursively called subroutines. Notice that the same subroutines are used when the two-dimensional transforms are performed on a row or a column, even though the data elements are spaced differently in data memory during a row DCT and a column DCT.

Listings 7.1–7.10 use the 16-pt transforms and Listings 7.11–7.18 use the 8-pt transforms. Most subroutines are modified versions of the 16-pt subroutines. For example, the 8-pt DCT does not use an equivalent of DIF16 (Listing 7.2), so input data must be read from memory in the DIF8_8 routine (Listing 7.12). Bit-reversing the TMP buffer is performed over an 8-location buffer in DCBREV_8 (Listing 7.17) instead of the 16-location buffer used in DCBREV (Listing 7.9). Also, the routines that perform in-place operations on TMP (Listing 7.12–7.17) are shortened since the TMP buffer is only eight locations long in the 8-pt case.

7.5 ZIG-ZAG SCANNING OF DCT COEFFICIENTS

Processing 8 x 8 data blocks and getting 8 x 8 result blocks does not reduce data. The reduction occurs after the DCT is computed. All 64 DCT coefficients are passed through a quantization stage. The compression ratio dictates the amount of quantization. After quantization, you still have 64 DCT coefficients, but more values are equal to zero or to the same value as “neighboring” coefficients. The next step is to perform run-length coding on the quantized DCT coefficients. This process reduces the data. To efficiently run-length code the quantized DCT coefficients, you must transform the 8 x 8 block with a two-dimensional transform so similar values appear together frequently. Transforming the 8 x 8 block by rows and columns, for example, is not spatially efficient for compacted information. Various scanning standards specify and use two-dimensional Zig-Zag scanning.

Figure 7.5 is an example of zig-zag scanning of quantized coefficient addresses.

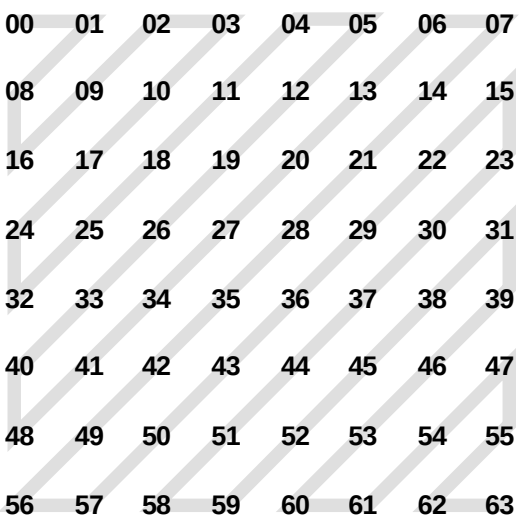


Figure 7.5. Zig-Zag Scanning of Quantized Coefficient Addresses

In this example, scanning starts at address 00 and continues through address 63.

7.6 ZIG-ZAG SCANNING & ADSP-21XX PROCESSORS

The data address generators (DAGs) in the ADSP-21xx Family core architecture are used for indirect data addressing. Each DAG uses four sets of registers consisting of three types of dedicated address registers for address computation. The three types of address registers are *index* (I) registers, *modify* (M) registers, and *length* (L) registers. Address computations are independent of arithmetic computations by the processor's ALU, MAC, and SHIFTER. During the same instruction cycles, the processor can perform arithmetic computations and data addressing in parallel.

Index registers hold absolute addresses that point to memory locations. Modify register contents are automatically added to index register contents during an indirect address operation. As a result, the index register points to the next desired data element for the next instruction cycle. Length registers are used with dedicated modulus arithmetic to make sure the index pointers stay in the circular buffers when circular addressing is used. Each of the four index registers can hold the complete,

absolute address for any addressable location in memory. When an instruction specifies an index register, it also specifies which modify register to use. The length register is not specified by the instruction since length and index registers are associated.

In an instruction, you can specify one of four possible modify registers with an index register, and you can change the address modify value without wasting cycles to change the modify register contents. You can do this by selecting a different M register to be used for a given I register. For additional information about using DAG resisters, refer to the *ADSP-2100 Family User's Manual.*

While following the zig-zag scanning trace in Figure 7.5, you may have noticed that between data elements there are four different address offset amounts. You can effectively zig-zag scan an 8 x 8 block of data stored in two dimensional format without adding addressing overhead and you eliminate the need to copy data to other memory locations for straightforward addressing. Also, you can combine indirect addressing operations with arithmetic computations. This means you can perform zig-zag scanning of quantized DCT coefficients in parallel with other required operations. You can add zig-zag scanning to your program without adding execution time and code space.

7.7 LISTINGS

This section contains the listing for this chapter.

7 Discrete Cosine Transform

```
{ ONE DIMENSIONAL, FAST, DISCRETE COSINE TRANSFORM, 16 POINTS
      Implementation:
            as described by Hsieh S. Hou in IEEE Transactions on Acoustics,
            Speech, and Signal Processing, Vol. ASSP-35, No. 10, October 1987

      Target Processor:
            ADSP-2100 family of DSP processors from Analog Devices, Inc.

      Execution Benchmark:
            318 instruction cycles — ADSP-2101 — 15.90 us at CLKIN=20.00Mhz

      Memory Storage Requirement:
            272 PM = 257 program memory code, 15 program memory data (coefficients)
            32 DM = 16 data memory scratch pad, 16 data memory (16-pt vector)
            Note: resulting transform coefficients written over original input data
            Assumes: unsigned 8-bit input data, signed 16-bit output coefficients

      Release History: 27-March-1989, extensively revised: 17-July-1989
            Revised: 23-July-1989 Revised for ADSP-2101: 28-July-1993

      Analog Devices, Inc., DSP Division, P.O.Box 9106, Norwood, MA 02062, USA }

.module/ram/abs=0       fast_16pt_dct;
.var/pm/ram             cosvals[15];                { cosine coefficients }
.var/circ/abs=0x3800    tmp[16];                    { temporary scratch memory }
.var                    x[16];                      { 16pt vector to transform }
.global     tmp;
.external   DIF16, DIF8, DIF4, DIF2, RLR4, RLR8, RLR16, DC_AND_BREV;
.init       x: <fhex.dat>;
.init cosvals[00]:  h#7F6200, h#70E200, h#513300, h#252800,
                    h#F37500, h#C3AA00, h#9D0E00, h#858300;
.init  cosvals[08]: h#7D8A00, h#471C00, h#E70800, h#959300;
.init  cosvals[12]: h#764100, h#CF0500;
.init  cosvals[14]: h#5A8200;

{ interupt vector table }
rest: jump setup; rti; rti; rti;                    { jump here on reset }
            rti; rti; rti; rti;                     { irq2 interrupt vector }
            rti; rti; rti; rti;                     { sport0 tx interrupt vector }
            rti; rti; rti; rti;                     { sport0 rx interrupt vector }
            rti; rti; rti; rti;                     { sport1 tx interrupt vector }
            rti; rti; rti; rti;                     { sport1 rx interrupt vector }
            rti; rti; rti; rti;                     { timer interrupt vector }
```

```
setup:  l0=0; l1=0; l2=0; l3=0; l5=0; l6=0;
dct16:  i2=^x; i3=^x+15; i6=^cosvals;
            m2=2; m3=-2; m5=1; m6=1; m7=-3; se=1;
            call DIF16;
            call DIF8;
            call DIF4;
            call DIF2;
            call RLR4;
            call RLR8;
            call RLR16;
            i5=^x;
            call DC_AND_BREV;
idwait: idle;
            jump idwait;
.endmod;
```

Listing 7.1 One-Dimensional Fast Discrete Cosine Transform (16 Points) Routine

7 Discrete Cosine Transform

```
.module/ram do_DIF16;                { 1 16-way DIF }
.external tmp;
.entry DIF16;

DIF16:
   i0=^tmp;
   i1=^tmp+8;
   m1=1;

   ax1=dm(i3,m3);

   af=pass ax1, ax0=dm(i2,m2);
   ar=ax0+af, ax1=dm(i3,m3), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss);

   cntr=6;
   do x16 until ce;
      af=pass ax1, ax0=dm(i2,m2);
      ar=ax0+af, ax1=dm(i3,m3), my0=pm(i6,m6);
      ar=ax0-af, dm(i0,m1)=ar;
x16:     mr=ar*my0(ss), dm(i1,m1)=mr1;

   af=pass ax1, ax0=dm(i2,m2);
   ar=ax0+af, my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss), dm(i1,m1)=mr1;
   dm(i1,m1)=mr1;
   rts;                              { end 1 16-way DIF }
.endmod;
```

Listing 7.2 DIF16 Subroutine

```
.module/ram do_DIF8;                {2 8-way DIFs }
.external tmp;
.entry DIF8;

DIF8:
   i0=^tmp;
   i1=^tmp+4;
   i2=^tmp;
   i3=^tmp+4;
   m0=5;

   ax1=dm(i3,m1);
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m1), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss);
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m1), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss), dm(i1,m1)=mr1;
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m0), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss), dm(i1,m1)=mr1;
   af=pass ax1, ax0=dm(i2,m0);
   ar=ax0+af,                    my0=pm(i6,m7);
   ar=ax0-af, dm(i0,m0)=ar;
   mr=ar*my0(ss), dm(i1,m1)=mr1;
   dm(i1,m0)=mr1;

   ax1=dm(i3,m1);
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m1), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss);
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m1), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss), dm(i1,m1)=mr1;
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m1), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss), dm(i1,m1)=mr1;
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af,                    my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss), dm(i1,m1)=mr1;
   dm(i1,m1)=mr1;

   rts;                          { end 2 8-way DIFs }
.endmod;
```

Listing 7.3 DIF8 Subroutine

```
.module/ram do_DIF4;          { 4 4-way DIFs }
.external tmp;
.entry DIF4;

DIF4:
   i0=^tmp;
   i1=^tmp+2;
   i2=^tmp;
   i3=^tmp+2;
   m2=3;

   ax1=dm(i3,m1);
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m2), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss);
   af=pass ax1, ax0=dm(i2,m2);
   ar=ax0+af,                 my1=pm(i6,m6);
   ar=ax0-af, dm(i0,m2)=ar;
   mr=ar*my1(ss), dm(i1,m1)=mr1;
   dm(i1,m2)=mr1;

   ax1=dm(i3,m1);
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m2);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss);
   af=pass ax1, ax0=dm(i2,m2);
   ar=ax0+af;
   ar=ax0-af, dm(i0,m2)=ar;
   mr=ar*my1(ss), dm(i1,m1)=mr1;
   dm(i1,m2)=mr1;

   ax1=dm(i3,m1);
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m1);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss);
   af=pass ax1, ax0=dm(i2,m0);
   ar=ax0+af;
   ar=ax0-af, dm(i0,m0)=ar;
   mr=ar*my1(ss), dm(i1,m1)=mr1;
   dm(i1,m1)=mr1;
```

```
   ax1=dm(i2,m1);
   af=pass ax1, ax0=dm(i3,m1);
   ar=ax0+af, ax1=dm(i2,m1);
   ar=ax0-af, dm(i1,m1)=ar;
   mr=ar*my0(ss);
   af=pass ax1, ax0=dm(i3,m1);
   ar=ax0+af;
   ar=ax0-af, dm(i1,m1)=ar;
   mr=ar*my1(ss), dm(i0,m1)=mr1;
   dm(i0,m1)=mr1;
   rts;                        { end 4 4-way DIFs }
.endmod;
```

Listing 7.4 DIF4 Subroutine

```
.module/ram do_DIF2;          { 8 2-way DIFs }
.external tmp;
.entry DIF2;

DIF2:
   i0=^tmp;
   i1=^tmp+1;
   i2=^tmp;
   i3=^tmp+1;
   m0=2;

   ax1=dm(i3,m0);
   af=pass ax1, ax0=dm(i2,m0);
   ar=ax0+af, ax1=dm(i3,m0), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m0)=ar;
   mr=ar*my0(ss);

   cntr=6;
   do difx2 until ce;
      af=pass ax1, ax0=dm(i2,m0);
      ar=ax0+af, ax1=dm(i3,m0);
      ar=ax0-af, dm(i0,m0)=ar;
   difx2:  mr=ar*my0(ss), dm(i1,m0)=mr1;


   af=pass ax1, ax0=dm(i2,m0);
   ar=ax0+af;
   ar=ax0-af, dm(i0,m0)=ar;
   mr=ar*my0(ss), dm(i1,m0)=mr1;

   dm(i1,m1)=mr1;
   rts;                       { end 8 2-way DIFs }
.endmod;
```

Listing 7.5 DIF2 Subroutine

```
.module do_RLR4;
.external tmp;
.entry RLR4;

RLR4:
   i0=^tmp+3;
   i1=^tmp+2;
   i2=i0;
   m0=4;

   si=dm(i0,m0);
   ay0=dm(i1,m0);

   cntr=3;
   do rlrx4 until ce;
      sr=ashift si (hi),      si=dm(i0,m0);
      ar=sr1-ay0,             ay0=dm(i1,m0);

   rlrx4: dm(i2,m0)=ar;
   sr=ashift si (hi);
   ar=sr1-ay0;
            dm(i2,m0)=ar; rts;
.endmod;
```

Listing 7.6 RLR4 Subroutine

```
.module do_RLR8;
.external tmp;
.entry RLR8;

RLR8:
   i0=^tmp+4;
   i1=^tmp+12;
   m0=1;
   m1=2;
   m2=-1;
   m3=-2;
                  ay0=dm(i0,m1);
                  si=dm(i0,m2);
   sr=ashift si (hi);
   ar=sr1-ay0,          si=dm(i0,m0);
   af=-ar,              dm(i0,m0)=ar;
   sr=ashift si (hi);
   ar=sr1+af,           si=dm(i0,m3);
   af=-ar,              dm(i0,m1)=ar;
   sr=ashift si (hi),   ay0=dm(i1,m1);
   ar=sr1+af,           si=dm(i1,m2);
   sr=ashift si (hi),   dm(i0,m0)=ar;
   ar=sr1-ay0,          si=dm(i1,m0);
   af=-ar,              dm(i1,m0)=ar;
   sr=ashift si (hi);
   ar=sr1+af,           si=dm(i1,m3);
   af=-ar,              dm(i1,m1)=ar;
   sr=ashift si (hi);
   ar=sr1+af;
                        dm(i1,m0)=ar;
   rts;
.endmod;
```

Listing 7.7 RLR8 Subroutine

```
.module do_RLR16;
.entry RLR16;

RLR16:
   i0=h#0407;     { h#0407 = bitrev(^tmp+8) when ^tmp=h#3800 }
   i1=i0;
   m0=2048;       { bitrev modifier = 16384/N = 2048 }
   ena bit_rev;
            ay0=dm(i0,m0);
            si=dm(i0,m0);
   sr=ashift si (hi),   dm(i1,m0)=ay0;
   ar=sr1-ay0,          si=dm(i0,m0);
   af=-ar,              dm(i1,m0)=ar;

   cntr=5;
   do rlrx16 until ce;
      sr=ashift si (hi);
      ar=sr1+af,  si=dm(i0,m0);
   rlrx16: af=-ar,      dm(i1,m0)=ar;

   sr=ashift si (hi);
   ar=sr1+af;
                        dm(i1,m0)=ar;

   dis bit_rev;
   rts;
.endmod;
```

Listing 7.8 RLR16 Subroutine

7 Discrete Cosine Transform

```
.module do_DC_and_brev;
.const sqrt2div2=h#5A82;
.external tmp;
.entry DC_AND_BREV;

DC_AND_BREV:      mx0=dm(tmp);
DCterm:     my0=sqrt2div2;
         mr=mx0*my0(rnd);     { calculate DC term using sqrt(2)/2 }
         dm(tmp)=mr1;
descramble:       i0=h#0007;  { h#0007 = bitrev(^tmp) when ^tmp=h#3800 }
         m0=1024;             { bitrev modifier = 16384/N = 1024 }
         cntr=16;
         ena bit_rev;
         do unbrev until ce;
            ax0=dm(i0,m0);          { read from bit-reversed tmp buffer }
unbrev:           dm(i5,m5)=ax0;    { write to normal ordered x buffer }
         dis bit_rev;
         rts;
.endmod;
```

Listing 7.9 DC_AND_BREV Subroutine

Discrete Cosine Transform 7

```
{ TWO DIMENSIONAL, FAST, DISCRETE COSINE TRANSFORM, 16 x 16 POINTS

   Implementation:
      as described by Hsieh S. Hou in IEEE Transactions on Acoustics,
      Speech, and Signal Processing, Vol. ASSP-35, No.10, October 1987

   Target Processor:
      ADSP-2100 family of DSP processors from Analog Devices, Inc.

   Execution Benchmark:
      10046 instruction cycles - ADSP-2101 - 0.5023 ms at CLKOUT=20.00 MHz

   Memory Storage Requirement:
      304 PM = 291 program memory code, 15 program memory data (coefficients)
      274 DM = 18 data memory scratch pad, 256 data memory (16x16 image)
      Note: resulting transform coefficients written over original input data
Assumes: unsigned 8-bit input data, signed 16-bit output coefficients

   Release History: 27-March-1989, extensively revised: 17-July-1989
      Revised: 23-July-1989 Revised for ADSP-2101: 28-July-1993

   Analog Devices, Inc., DSP Division, P.O.Box 9106, Norwood, MA 02062, USA }

.module/ram/abs=0       fast_16x16_dct;
.var/pm/ram             cosvals[15];            { cosine coefficients }
.var/circ/abs=0x3800    tmp[16];                { temporary scratch memory }
.var              xadr, xadr2;
.var              x[256];                       { 16x16 block to transform }
.global  tmp;
.external   DIF16, DIF8, DIF4, DIF2, RLR4, RLR8, RLR16, DC_AND_BREV;
.init    x: <xx.dat>;
.init  cosvals[00]: h#7F6200, h#70E200, h#513300, h#252800,
           h#F37500, h#C3AA00, h#9D0E00, h#858300; .init  cosvals[08]: h#7D8A00,
h#471C00, h#E70800, h#959300; .init  cosvals[12]: h#764100, h#CF0500;
.init  cosvals[14]: h#5A8200;

      jump setup; rti; rti; rti;                { jump here at reset }
      rti; rti; rti; rti;                       { irq2 interrupt vector }
      rti; rti; rti; rti;                       { sport0 tx interrupt vector }
      rti; rti; rti; rti;                       { sport0 rx interrupt vector }
      rti; rti; rti; rti;                       { sport1 tx interrupt vector }
      rti; rti; rti; rti;                       { sport1 rx interrupt vector }
      rti; rti; rti; rti;                       { timer interrupt vector }
setup:  l0=0; l1=0; l2=0; l3=0; l5=0; l6=0; m6=1; m7=-3; se=1;
```

(listing continues on next page)

```
{ calculate the DCT values for the row addresses }
rows: si=^x;                          { cols: ^x }
   dm(xadr)=si;
   i2=si;
   si=^x+15;                          { cols: ^x+240 }
   dm(xadr2)=si;
   i3=si;
   m5=1;                              { cols: 16 }
   cntr=16;
   do rowdcts until ce;
      i6=^cosvals;
      m2=2;                           { cols: 32 }
      m3=-2;                          { cols: -32 }
      call DIF16;
         call DIF8;
         call DIF4;
      call DIF2;
      call RLR4;
      call RLR8;
      call RLR16;
      si=dm(xadr);
      i5=si;
      call DC_AND_BREV;

nextrow: ay0=16;                      { cols: 1 }
      ax0=dm(xadr);
      ar=ax0+ay0;
      dm(xadr)=ar;
      i2=ar;
      ax0=dm(xadr2);
      ar=ax0+ay0;
      dm(xadr2)=ar;

rowdcts: i3=ar;
```

```
{ calculate  DCT values for column addresses }
cols: si=^x;                        { cols: ^x }
   dm(xadr)=si;
   i2=si;
   si=^x+240;                       { cols: ^x+240 }
   dm(xadr2)=si;
   i3=si;
   m5=16;                           { cols: 16 }
   cntr=16;
   do coldcts until ce;
      i6=^cosvals;
      m2=32;                        { cols: 32 }
      m3=-32;                       { cols: -32 }
      call DIF16;
      call DIF8;
      call DIF4;
      call DIF2;
      call RLR4;
      call RLR8;
      call RLR16;
      si=dm(xadr);
      i5=si;
      call DC_AND_BREV;
nextcol: ay0=1;                     { cols: 1 }
      ax0=dm(xadr);
      ar=ax0+ay0;
      dm(xadr)=ar;
      i2=ar;
      ax0=dm(xadr2);
      ar=ax0+ay0;
      dm(xadr2)=ar;
coldcts: i3=ar;
wait1:  idle;
     jump wait1;
.endmod;
```

Listing 7.10 Two-Dimensional Fast Discrete Cosine Transform (16 X 16 Points) Routine

7 Discrete Cosine Transform

```
{ ONE DIMENSIONAL, FAST, DISCRETE COSINE TRANSFORM, 8 POINTS
   Implementation:
      as described by Hsieh S. Hou in IEEE Transactions on Acoustics,
      Speech, and Signal Processing, Vol. ASSP-35, No. 10, October 1987

   Target Processor:
      ADSP-2100 family of DSP processors from Analog Devices, Inc.

   Execution Benchmark:
      158 instruction cycles — ADSP-2101 — 7.90 us at CLKIN=20.00MHz

   Memory Storage Requirement:
      163 PM = 156 program memory code, 7 program memory data (coefficients)
      16 DM = 8 data memory scratch pad, 8 data memory (8-pt vector)
      Note: resulting transform coefficients written over original input data
      Assumes: unsigned 8-bit input data, signed 16-bit output coefficients

   Release History:
      27-March-1989, Revised: 23-July-1989, Revised for ADSP-2101 28-july-1993

   Analog Devices, Inc., DSP Division, P.O.Box 9106, Norwood, MA 02062, USA }

.module/ram/abs=0       fast_8pt_dct;
.var/pm/ram             cosvals[7];             { cosine coefficients }
.var/circ/abs=0x3800    tmp[8];                 { temporary scratch memory }
.var              x[8];                         { 8-pt vector to transform }
.global           tmp;
.external   DIF8_8, DIF4_8, DIF2_8, RLR4_8, RLR8_8, DC_AND_BREV_8;
.init    x: <x.dat>;
.init  cosvals[0]: h#7D8A00, h#471C00, h#E70800, h#959300;
.init  cosvals[4]: h#764100, h#CF0500;
.init  cosvals[6]: h#5A8200;

      jump setup; rti; rti; rti;                { jump here at reset }
      rti; rti; rti; rti;                       { irq2 interrupt vector }
      rti; rti; rti; rti;                       { sport0 tx interrupt vector }
      rti; rti; rti; rti;                       { sport0 rx interrupt vector }
      rti; rti; rti; rti;                       { sport1 tx interrupt vector }
      rti; rti; rti; rti;                       { sport1 rx interrupt vector }
      rti; rti; rti; rti;                       { timer interrupt vector }
```

```
setup:   l0=0; l1=0; l2=0; l3=0; l5=0; l6=0; m6=1; se=1;
dct8:    i2=^x;
      i3=^x+7;
      m5=1;
      i6=^cosvals;
      m2=2;
      m3=-2;
      call DIF8_8;
      call DIF4_8;
      call DIF2_8;
      call RLR4_8;
      call RLR8_8;
      i5=^x;
      call DC_AND_BREV_8;
wait2:   idle;
      jump wait2;
.endmod;
```

Listing 7.11 One-Dimensional Fast Discrete Cosine Transform (8 Points) Routine

```
.module/ram do_DIF8_8;        { 1 8-way DIFs }
.external tmp
.entry DIF8_8;

DIF8_8:
   i0=^tmp;
   i1=^tmp+4;
   m1=1;

   ax1=dm(i3,m3);
   af=pass ax1, ax0=dm(i2,m2);
   ar=ax0+af, ax1=dm(i3,m3), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss);

   cntr=2;
   do d8x8 until ce;
      af=pass ax1, ax0=dm(i2,m2);
      ar=ax0+af, ax1=dm(i3,m3), my0=pm(i6,m6);
      ar=ax0-af, dm(i0,m1)=ar;
   d8x8: mr=ar*my0(ss), dm(i1,m1)=mr1;

   af=pass ax1, ax0=dm(i2,m2);
   ar=ax0+af,     my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss), dm(i1,m1)=mr1;
   dm(i1,m1)=mr1;

   rts;                       { end 1 8-way DIFs }
.endmod;
```

Listing 7.12 DIF8_8 Subroutine

```
.module/ram do_DIF4_8;        { 2 4-way DIFs }
.external tmp;
.entry DIF4_8;

DIF4_8:
   i0=^tmp;
   i1=^tmp+2;
   i2=^tmp;
   i3=^tmp+2;
   m2=3;

   ax1=dm(i3,m1);
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m2),  my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss);
   af=pass ax1, ax0=dm(i2,m2);
   ar=ax0+af,                      my1=pm(i6,m6);
   ar=ax0-af, dm(i0,m2)=ar;
   mr=ar*my1(ss), dm(i1,m1)=mr1;
   dm(i1,m2)=mr1;

   ax1=dm(i3,m1);
   af=pass ax1, ax0=dm(i2,m1);
   ar=ax0+af, ax1=dm(i3,m2);
   ar=ax0-af, dm(i0,m1)=ar;
   mr=ar*my0(ss);
   af=pass ax1, ax0=dm(i2,m2);
   ar=ax0+af;
   ar=ax0-af, dm(i0,m2)=ar;
   mr=ar*my1(ss), dm(i1,m1)=mr1;
   dm(i1,m2)=mr1;

   rts;                       { end 2 4-way DIFs }
.endmod;
```

Listing 7.13 DIF4_8 Subroutine

```
.module/ram do_DIF2_8;        { 4 2-way DIFs }
.external tmp;
.entry DIF2_8;

DIF2_8:
   i0=^tmp;
   i1=^tmp+1;
   i2=^tmp;
   i3=^tmp+1;
   m0=2;

   ax1=dm(i3,m0);

   af=pass ax1, ax0=dm(i2,m0);
   ar=ax0+af, ax1=dm(i3,m0), my0=pm(i6,m6);
   ar=ax0-af, dm(i0,m0)=ar;
   mr=ar*my0(ss);

   af=pass ax1, ax0=dm(i2,m0);
   ar=ax0+af, ax1=dm(i3,m0);
   ar=ax0-af, dm(i0,m0)=ar;
   mr=ar*my0(ss), dm(i1,m0)=mr1;

   af=pass ax1, ax0=dm(i2,m0);
   ar=ax0+af, ax1=dm(i3,m0);
   ar=ax0-af, dm(i0,m0)=ar;
   mr=ar*my0(ss), dm(i1,m0)=mr1;

   af=pass ax1, ax0=dm(i2,m0);
   ar=ax0+af;
   ar=ax0-af, dm(i0,m0)=ar;
   mr=ar*my0(ss), dm(i1,m0)=mr1;

   dm(i1,m1)=mr1;
   rts;                       { end 4 2-way DIFs }
.endmod;
```

Listing 7.14 DIF2_8 Subroutine

```
.module do_RLR4_8;
.external tmp;
.entry RLR4_8;

RLR4_8:
   i0=^tmp+3;
   i1=^tmp+2;
   i2=i0;
   m0=4;
                  si=dm(i0,m0);
                  ay0=dm(i1,m0);
   sr=ashift si (hi),   si=dm(i0,m0);
   ar=sr1-ay0,                ay0=dm(i1,m0);
            dm(i2,m0)=ar;
   sr=ashift si (hi);
   ar=sr1-ay0;
            dm(i2,m0)=ar;
   rts;
.endmod;
```

Listing 7.15 RLR4_8 Subroutine

```
.module do_RLR8_8;
.external tmp;
.entry RLR8_8;

RLR8_8:
   i0=^tmp+4;
   i1=^tmp+12;
   m0=1;
   m1=2;
   m2=-1;
   m3=-2;
            ay0=dm(i0,m1);
            si=dm(i0,m2);
   sr=ashift si (hi);
   ar=sr1-ay0,          si=dm(i0,m0);
   af=-ar,        dm(i0,m0)=ar;
   sr=ashift si (hi);
   ar=sr1+af,           si=dm(i0,m3);
   af=-ar,        dm(i0,m1)=ar;
   sr=ashift si (hi);
   ar=sr1+af;
            dm(i0,m0)=ar;

   rts;
.endmod;
```

Listing 7.16 RLR8_8 Subroutine

```
.module do_DC_and_brev_8;
.const sqrt2div2=h#5A82;
.external tmp;
.entry DC_AND_BREV_8;

DC_AND_BREV_8:    mx0=dm(tmp);
DCterm:     my0=sqrt2div2;
         mr=mx0*my0(rnd);          { calculate DC term using sqrt(2)/2 }
         dm(tmp)=mr1;
descramble:       i0=h#0007;       { h#0007 = bitrev(^tmp) when ^tmp=h#3800 }
         m0=2048;                  { bitrev modifier = 16384/N = 2048
         cntr=8;
         ena bit_rev;
         do unbrev until ce;
            ax0=dm(i0,m0);         { read from bit-reversed tmp buffer }
unbrev:           dm(i5,m5)=ax0;   { write to normal ordered x buffer }
         dis bit_rev;
         rts;
.endmod;
```

Listing 7.17 DC_AND_BREV_8 Subroutine

7 Discrete Cosine Transform

```
{ TWO DIMENSIONAL, FAST, DISCRETE COSINE TRANSFORM, 8 x 8 POINTS
   Implementation:
        as described by Hsieh S. Hou in IEEE Transactions on Acoustics,
        Speech, and Signal Processing, Vol. ASSP-35, No. 10, October 1987

   Target Processor:
        ADSP-2100 family of DSP processors from Analog Devices, Inc.

   Execution Benchmark:
        2492 instruction cycles — ADSP-2101 — 0.12460 ms at CLKOUT=20.00MHz

   Memory Storage Requirement:
        208 PM = 201 program memory code, 7 program memory data (coefficients)
        16 DM = 8 data memory scratch pad, 8 data memory (8-pt vector)
        Note: resulting transform coefficients written over original input data
        Assumes: unsigned 8-bit input data, signed 16-bit output coefficients

   Release History: 27-March-1989, Revised: 23-July-1989, Revised for ADSP-2101
            28-July-1993

   Analog Devices, Inc., DSP Division, P.O.Box 9106, Norwood, MA 02062, USA }

.module/ram/abs=0       fast_8x8_dct;
.var/pm/ram             cosvals[15];            { cosine coefficients }
.var/circ/abs=0x3800    tmp[8];                 { temporary scratch memory }
.var              xadr, xadr2;
.var              x[64];                        { 8x8 block to transform }
.global     tmp;
.external   DIF8_8, DIF4_8, DIF2_8, RLR4_8, RLR8_8, DC_AND_BREV_8;
.init    x: <xx.dat>;
.init   cosvals[0]: h#7D8A00, h#471C00, h#E70800, h#959300;
.init   cosvals[4]: h#764100, h#CF0500;
.init   cosvals[6]: h#5A8200;

   jump setup; rti; rti; rti;                   { start here on reset }
   rti; rti; rti; rti;                          { irq2 interrupt vector }
   rti; rti; rti; rti;                          { sport0 tx interrupt vector }
   rti; rti; rti; rti;                          { sport0 rx interrupt vector }
   rti; rti; rti; rti;                          { sport1 tx interrupt vector }
   rti; rti; rti; rti;                          { sport1 rx interrupt vector }
   rti; rti; rti; rti;                          { timer interrupt vector }
```

```
setup:  l0=0; l1=0; l2=0; l3=0; l5=0; l6=0; m6=1; se=1;
rows:   si=^x;                                    { cols: ^x }
   dm(xadr)=si
   i2=si;
   si=^x+7;                                       { cols: ^x+56 }
   dm(xadr2)=si;
   i3=si;
   m5=1;                                          { cols: 8 }
   cntr=8;
   do rowdcts until ce;
        i6=^cosvals;
        m2=2;                                     { cols: 16 }
        m3=-2;                                    { cols: -16 }
        call DIF8_8;
        call DIF4_8;
        call DIF2_8;
        call RLR4_8;
        call RLR8_8;
        si=dm(xadr);
        i5=si;
        call DC_AND_BREV_8;
nextrow: ay0=8;                                   { cols: 1 }
        ax0=dm(xadr);
        ar=ax0+ay0;
        dm(xadr)=ar;
        i2=ar;
        ax0=dm(xadr2);
        ar=ax0+ay0;
        dm(xadr2)=ar;
rowdcts: i3=ar;
cols:   si=^x;                                    { cols: ^x }
   dm(xadr)=si;
   i2=si;
   si=^x+56;                                      { cols: ^x+56 }
   dm(xadr2)=si;
   i3=si;
   m5=8;                                          { cols: 8 }
   cntr=8;
   do coldcts until ce;
        i6=^cosvals;
        m2=16;                                    { cols: 16 }
        m3=-16;                                   { cols: -16 }
        call DIF8_8;
        call DIF4_8;
        call DIF2_8;
        call RLR4_8;
        call RLR8_8;
        si=dm(xadr);
        i5=si;
        call DC_AND_BREV_8;
```

(listing continues on next page)

```
nextcol: ay0=1;                                    { cols: 1 }
        ax0=dm(xadr);
        ar=ax0+ay0;
        dm(xadr)=ar;
        i2=ar;
        ax0=dm(xadr2);
        ar=ax0+ay0;
        dm(xadr2)=ar;
coldcts: i3=ar;
wait3:  idle;
   jump wait3;
.endmod;
```

Listing 7.18 Two-Dimensional Fast Discrete Cosine Transform (8 X 8 Points) Routine

7.8 REFERENCES

Hou. H. 1986. "The Fast Recursive Algorithm for Computing the Discrete Cosine Transform," *SPIE Conference Proceedings*, vol. 697, pp. 18-25.

Kamanjar & Rao. 1982. "Fast Algorithms for the 2D DCT," *IEEE Trans on Computers*, pp. 899-906.

Lee, Byeong. 1984. "A New Algorithm to Compute the DCT," *IEEE ASSP*, vol. 32, No. 6, pp. 1243-1245.

Magal & Heiman. 1985. "Image Coding System–A Single Processor Implementation," *IEEE MilCom*, vol. 3, pp. 628-634.

Unknown. 1988. "A 1 Chip VLSI for Real Time Two Dimensional Discrete Cosine Transform," *ISACS Conference Proceedings.*