Bit-Reverse and Digit-Reverse: Linear-Time Small Lookup Table Implementation for the TMS320C6000

APPLICATION REPORT: SPRA440

Chad Courtney
C6000 DSP Applications

Digital Signal Processing Solutions
May 1998

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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 application 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, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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.

TRADEMARKS

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Figures

Tables

Bit-Reverse and Digit-Reverse: Linear-Time Small Lookup Table Implementation for the TMS320C6000

Abstract

This application report describes a fast method of implementing bit-reverse and digit-reverse routines using a small lookup table. The author provides background on the bit-reverse and digit-reverse routines including theory behind the implementation and how the linear-time small lookup table method is implemented.

Product Support

World Wide Web

Our World Wide Web site at **www.ti.com** contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Email

For technical issues or clarification on switching products, please send a detailed email to **dsph@ti.com**. Questions receive prompt attention and are usually answered within one business day.

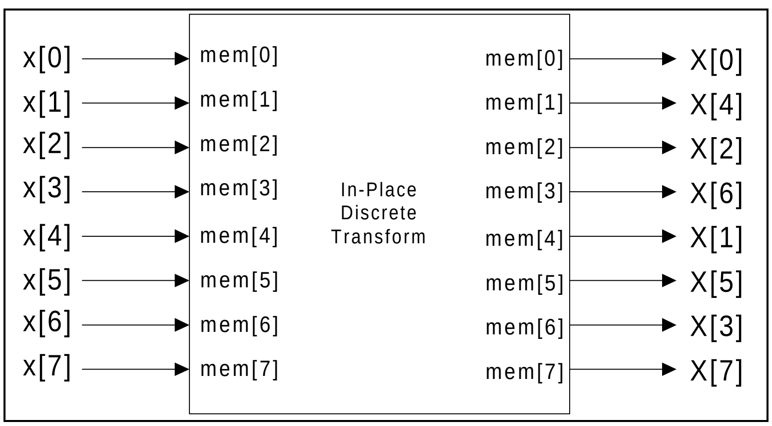
Introduction

Bit-reverse and digit-reverse routines are routines in which the data is reordered based on its index value from 0 to N-1, where N is the number of points to be bit/digit-reversed.

Discrete transforms are the main users of bit-reverse and digit-reverse routines. Discrete transforms take discrete inputs in one domain and convert them to discrete inputs in another. For example, a fast fourier transform (FFT) takes a discrete time domain input and transforms it into the discrete frequency domain output (i.e. x(t) -> X(jwt).)

Many discrete transforms (FFT, DCT, IDCT, DST, etc.) are executed in place using the same memory locations for both the input and output. This reduces both data size and algorithmic complexity. Bit/digit-reversing routines are needed to take full advantage of in-place execution. For example, if the in-place routine uses decimation-in-frequency (DIF) decomposition, the input is in normal order but the output is in bit/digit-reverse order, as shown in Figure 1.

Figure 1. In-Place Discrete Transform Using Decimation-in-Frequency



To view the resulting output in normal order, the results must be bit-reversed. Also note that, if the in-place discrete transform uses a decimation-in-time (DIT) decomposition, the inputs will require bit/digit-reversing and the outputs will be in normal order.

There is a direct correlation between the normal order and bit-reversed order shown using the in-place discrete transform example in Figure 1. The in-place discrete transforms input is a normal order 8-point array. The output is a bit-reversed ordered 8-point array. The storage of the input array is normal order so x[0]-x[7] line up with their respective memory locations 0-7, as shown in Figure 1.

The order of storage of the output array is in bit-reversed order compared to their respective memory locations. This is illustrated in Table 1, where the memory locations and the bit-reversed order output are shown in hex format and bit format, respectively.

Table 1. Memory vs. Output Hex and Bit Output

Hex Format		**Bit Format**	
Memory Location	**Bit Reverse Order Output**	**Memory Location**	**Bit Reverse Order Output**
mem[0]	X[0]	mem[000]	X[000]
mem[1]	X[4]	mem[001]	X[100]
mem[2]	X[2]	mem[010]	X[010]
mem[3]	X[6]	mem[011]	X[110]
mem[4]	X[1]	mem[100]	X[001]
mem[5]	X[5]	mem[101]	X[101]
mem[6]	X[3]	mem[110]	X[011]
mem[7]	X[7]	mem[111]	X[111]

As shown on the Bit Format side of Table 1, the bit notation of the memory locations and the bit notation of the bit-reversed ordered output are swapped.

This can be seen more clearly when viewing normal order sorting and bit-reverse order sorting in a tree diagram (see Figure 2).

Figure 2. Order Sorting Tree

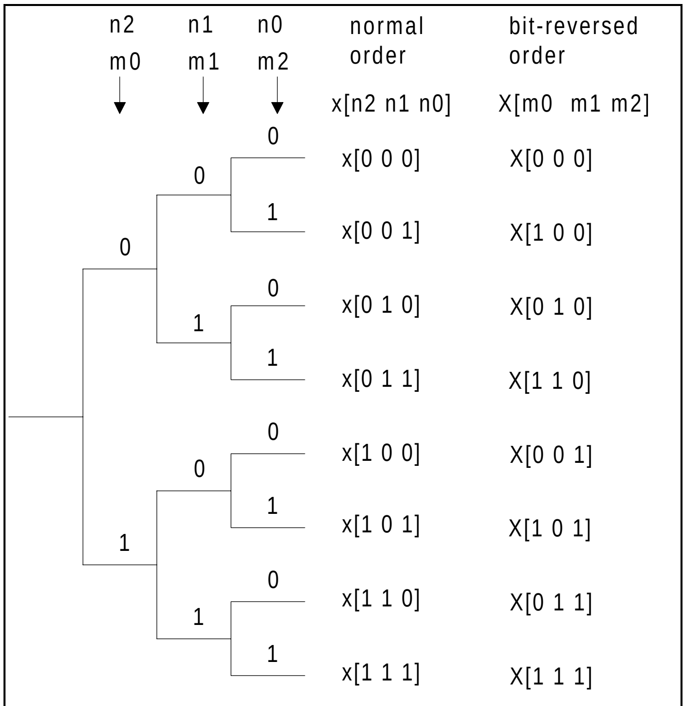


Figure 2
Order Sorting Tree

Normal order sorting sorts by looking at the most significant bit (n2 in Figure 2). If the most significant bit is a zero, it is placed in the upper half of the tree; if it is a one, it is placed in the lower half of the tree. The top half and bottom half subtrees are then sorted using the same criteria on the second most significant bit (n1 in Figure 2). This process is repeated until the array is completely sorted.

Bit-reversed order sorting, as shown in the tree diagram, is sorted by looking at the least significant bit (n0 in Figure 2). If the least significant bit is zero, it is placed in the upper half of the tree; if it is a one, it is placed in the lower half of the tree. The top half and bottom half subtrees are then sorted using the same criteria on the second least significant bit, n1.

This process is repeated until the array is completely sorted. Thus, to go from bit-reversed order to normal order or visa-versa, you simply “reverse the bits” of the desired value to produce the appropriate offset from a base memory location (that is, for desired value X[n2 n1 n0] of a bit reversed array, use the offset of [n0 n1 n2] from the beginning of the array). In our case, since the base memory location is zero, the offset is the memory location.

To perform the bit/digit-reversal of an array of data in place requires the swapping of the values having indices that are the bit/digit-reversal of one another. Note that when traversing an array during a bit/digit-reversal routine, you do not swap values twice (that is, if you swap memory location [001] with [100], do not swap [100] with [001]); otherwise, you will place them back in original order. One way to avoid this is to set i to the bit-reverse of j, then only swap x[i] with x[j] if i < j. This will ensure that a double swapping error does not occur.

Digit-reversal is similar to bit-reversal – actually, bit-reversal is the single digit case of digit-reversal. Digit reversal reverses digits instead of bits. For example, a radix-4 FFT produces an output resulting in 2-digit digit reverse order. To perform the 2-digit digit-reverse ordering, swap the two least significant bits with the two most significant bits, then the second pair of least significant bits with the second pair of most significant bits, and so on. Figure 3 shows a tree diagram of 2-digit digit-reverse order sorting for a 16-point, 2-digit digit-reversed order output.

Figure 3. 2-Digit Digit Order Sorting Tree

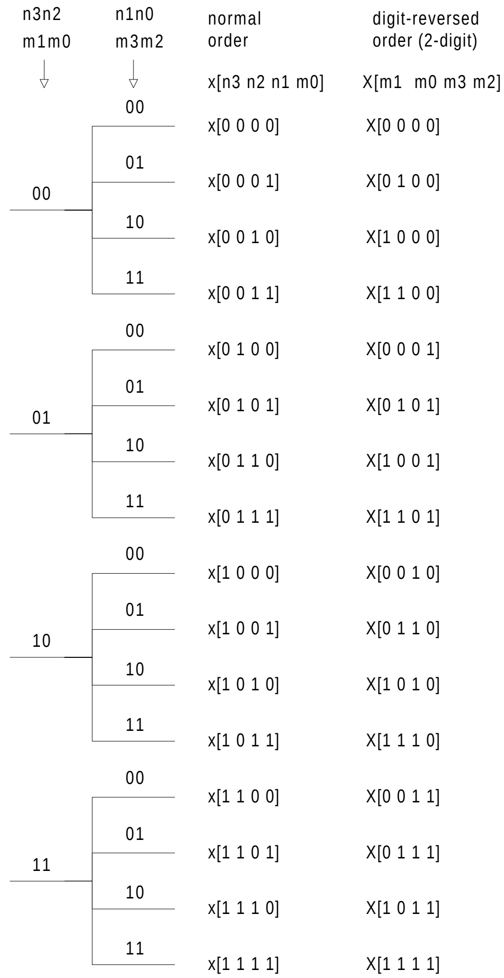


Figure 3
2-Digit Digit Order Sorting Tree

It thus appears easy to write a quick routine to perform an in-place bit or digit-reverse routine. Simply swap some bits or digits to produce a bit/digit-reversed order to be used as offsets from a base address and ensure nothing is double swapped. This method is okay for small number of points but let’s say that we want to do a bit-reverse on 16k points which is 2^14, giving us a total of 14 bits to manipulate thus requiring 7 bit pairs to be swapped.

This routine would require a cycle count on the order of $N\log_2 N$ cycles to complete, which is relatively slow. One could also use a lookup table method in which all of the bit-reverse order values are in a table ready to be used for swapping. This would produce a cycle count on the order of N cycles to complete (assuming the lookup table already existed) but would also require the extra 16K Halfwords of data space to store the lookup table, which can be significant.

Instead our new routine uses a small lookup table, usually the size is sqrt(N) but no more than x*sqrt(N) (where x is the square root of the radix value: sqrt(2) for bit-reverse, sqrt(4) for 2-digit digit-reverse, etc.) In this case it is 128 values and the routine still only requires a cycle count on the order of N cycles. In addition, since the lookup table requires only the values 0-127, it fits in 128 Bytes instead of 16K Halfwords. That is 1/256 in size of the old lookup table.

Linear Time Small Lookup Table Bit-Reverse Routine:

The idea behind a linear time small lookup table bit-reverse routine is to take the tree used in the bit-reversed order sorting, such as the one shown in Figure 2, and break it into smaller identical trees. Then we use the bit-reverse order sorted values from the smaller table as our lookup table. This can be seen using Figure 4, which shows a tree diagram with normal and bit-reverse order sorting.

Figure 4. Normal and Bit-Reverse Order Sorting

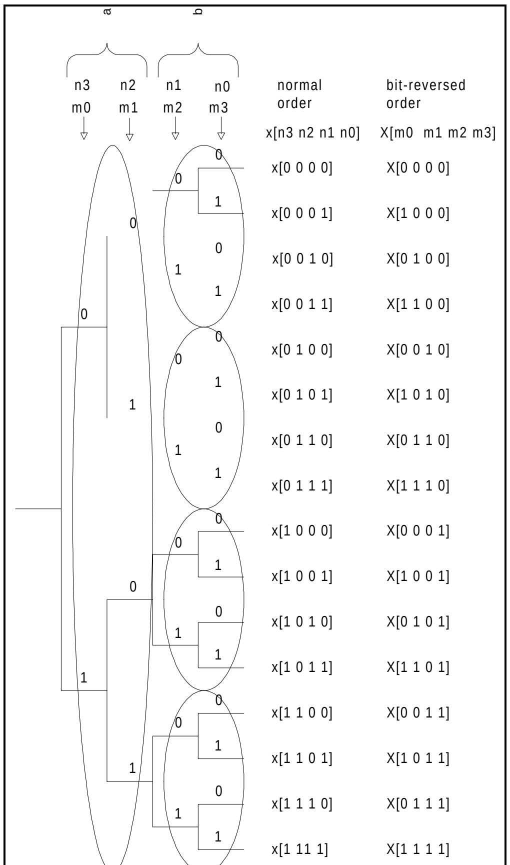


Figure 4
Order Sorting Tree

The circles in Figure 4 show how it can be broken into equal subtrees containing half of the number of bits (levels) of the whole tree. Since these subtrees are identical, we only need to create a bit-reversed index of one of the subtrees. By combining the bit-reversed order index values of the upper half of the bits (the a bits) and the lower half of the bits (the b bits), a bit-reversed routine for the array of N points can be achieved. This is done in linear time, producing a cycle count of order N cycles and using a relatively small lookup table.

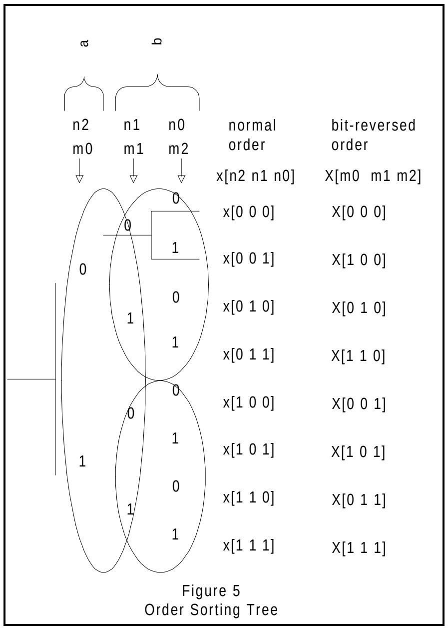
C code used to perform the in-place bit-reversal of an array can be found in *Appendix A, Program 1*. In this routine, the a bits are the upper half bits of the tree shown in Figure 4. Thus, it is the outer loop of the program, the bit-reversed index values based on the a bits are the lower bits of the offset pointer j, and the offset pointer i goes through normal order (0 - N-1).

Note that the portion of j produced by the b bits and the bit-reversed index values based on the b bits are shifted right by `nbot.` This is the number of bits the bottom index produces. Thus by combining the index[a] and the index[b], the appropriate index[i] is yielded with a table of the size sqrt[N] instead of N. In addition, the linear speed of a lookup table is still obtained. Note that the C program for producing a digit-reversed index of any radix (for bit-reverse radix-2 is used) can be found in *Appendix C, Program 3*.

This works well for a 16-point in-place bit-reversal since there is an even number of bits. Nevertheless, we have to do a little more work to accommodate one with an odd number of bits such as 8 points, which has 3 bits. A tree for an 8-point bit-reverse order sort is shown in Figure 5.

In Figure 5 we see that the identical subtrees cross over between levels (that is, sharing the n1 bit.) This is accommodated by using an “`astep`” to get only the a bits on the outer loop. In the case of bit-reverse, `astep` is set to 1 when `nbits` is even and set to 2 when `nbits` is odd. By setting `astep` to 2, we only look at the upper half of the tree where n1 is zero, the a half; the lower half of the tree, the b half, takes care of the general case of n1. This is shown in Figure 5 and in the C code.

Figure 5. 8-point Bit-Reverse Order Sort



Linear Time Small Lookup Table Routine:

The digit-reverse routine is simply an extension of the bit-reverse routine in which the digit is a set of bits requiring swapping. The tree used in the digit-reverse order sorting is similar to the one shown in Figure 3, broken into smaller identical trees. Then we use the digit-reverse order sorted values from the smaller table as our lookup table. This is shown using Figure 6 for a radix-4 (2-digit) tree.

Figure 6. Two-Digit Order Sorting Tree with Small Lookup Table

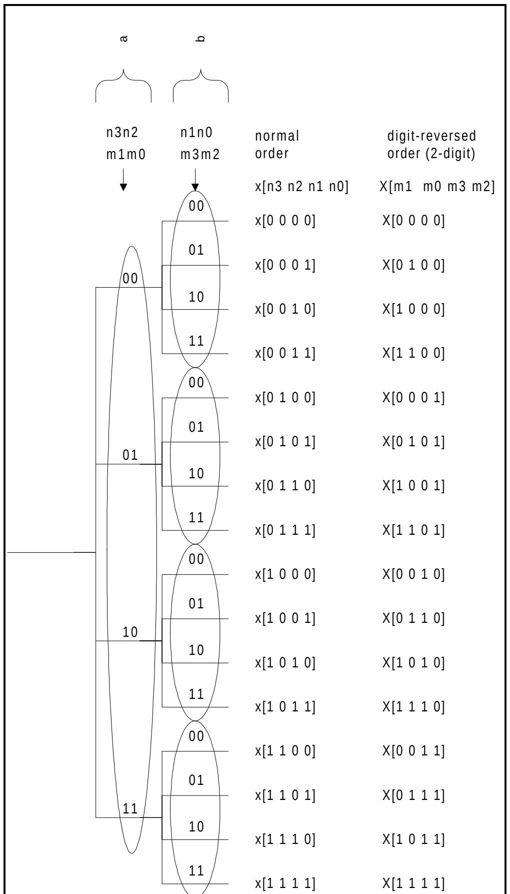


Figure 6
2-Digit Digit Order Sorting
Tree with small lookup table

The tree diagram in Figure 6 shows normal and digit-reverse order sorting. The circles show how it can be broken into equal subtrees containing half of the number of digits (levels) of the whole tree. Since these subtrees are identical, we only need to create a digit-reversed index of one of the subtrees. By combining the digit-reversed order index values of the upper half of the digits (the a digits) and the lower half of the digits (the b digits), a digit-reversed routine for the array of N points can be achieved. This is done in linear time producing a cycle count of order of N cycles and using a relatively small lookup table.

C code used to perform the in-place digit-reversal of an array can be found in the Appendix B, *Program 2*. In this routine the a digits are the upper half bits of the tree shown in Figure 6; thus, it is the outer loop of the program. The digit-reversed index values based on the a digits are the lower digits of the offset pointer j; the offset pointer i goes through normal order (0 - N-1).

Note that the portion of j produced by the digit-reversed index values based on the b digits is shifted right by `nbot.` This is the number of digits times digit size (in this case 2) the bottom index produces. Thus by combining the index[a] and the index[b], the appropriate index[i] is produced with a table of the size sqrt[N] instead of N and the linear speed of a full lookup table is still obtained. Note that the C program for producing a digit-reversed index of any radix (for bit-reverse radix-2 is used) can be found in the Appendix C, *Program 3*.

This works well for a 16-point in-place digit-reversal since there is an even number of digits. Nevertheless, there is a little more work required to accommodate one with an odd number of digits, such as 64-points, which has 3 digits. A tree for a 64-point digit-reverse order sort would have identical subtrees crossing over between levels (that is, sharing the n3n2 digit pair) similar to the bit-reverse order sorting shown in Figure 5. This is accommodated by using an "`astep`" to get only the a digits on the outer loop.

In the case of digit-reverse, `astep` is set to 1 when `nbits/radN` is even, where `radN` is the number of bits in a digit (1 for bit or radix-2, 2 for 2-digit or radix-4, 3 for 3-digit or radix 8, etc.) and set to `radix` when `nbits/radN` is odd. By setting `astep` to `radix` we look only at the upper half of the tree where n3n2 is zero, the a half; the lower half of the tree, the b half, takes care of the general cases of n3n2. This can be seen in the C code.

IMPROVED Bit/Digit-Reverse Routines:

There are two ways to reduce cycle counts further.

- Reduce the total number of times the bit/digit-reversed lookup table is accessed
- Eliminate some of the data loads that we know will not be swapped from the code altogether

Set up the code so that it only performs lookups with “a” when the least significant bit/digit is zero and lookups with “b” when the most significant bit/digit is zero. Using bit-reverse gives us a starting point of `0X0` and its bit-reverse value `0Y0`. In bit-reversed order are four combinations of the middle bits represented by `X` (see Table 2).

Table 2. Combinations of the Middle Bits Represented by X

i<j cond.	i0	0X0	j0	0Y0
i<j always	i1	0X1	j1	1Y0
i>j never	~~i2~~	~~1X0~~	~~j2~~	~~0Y1~~
i<j cond.	i3	1X1	j3	1Y1

Table 2

These are generated by adding offsets of `halfn` (n/2) or 1 to the starting points of `0X0` and `0Y0` thus four pairs of data indices are created by only loading one pair of lookup table values. Thus reducing the number of loads from the lookup tables by a factor of four. The second reduction is by removing the loading and storing of the values indexed by `1X0` and `0Y1` since in this case `i` is always greater than `j` and thus the swap will never be completed. Note that if this was placed in the program conditionally, even though it would never be executed, it would still take up the same amount of cycle time as if it had been executed. Thus one fourth of the total data loads and stores are removed from the bit-reverse program by removing this segment of code. The C and assembly code for improved bit-reversing routine can be found in the Appendix as programs 4 and 5.

This can be extended to digit-reverse by unrolling the code to accept digits. Table 3 shows 2-bit digit reverse, from which can be seen that we only load one pair of digit-reverse lookup table values for sixteen potential data loads. This greatly reduces the cycle counts do to loads.

Also from this one can see that *i* is known to be greater than *j* six out of the sixteen potential swaps and thus can be removed from the code all together.

Table 3. 2-Bit Digit Reverse

i<j cond.	i0	00X00	j0	00Y00
i<j always	i1	00X01	j1	01Y00
i<j always	i2	00X10	j2	10Y00
i<j always	i3	00X11	j3	11Y00
i>j never	~~i4~~	~~01X00~~	~~j4~~	~~00Y01~~
i<j cond.	i5	01X01	j5	01Y01
i<j always	i6	01X10	j6	10Y01
i<j always	i7	01X11	j7	11Y01
i>j never	~~i8~~	~~10X00~~	~~j8~~	~~00Y10~~
i>j never	~~i9~~	~~10X01~~	~~j9~~	~~01Y10~~
i<j cond.	iA	10X10	jA	10Y10
i>j always	iB	10X11	jB	11Y10
i>j never	~~iC~~	~~11X00~~	~~jC~~	~~00Y11~~
i>j never	~~iD~~	~~11X01~~	~~jD~~	~~01Y11~~
i>j never	~~iE~~	~~11X10~~	~~jE~~	~~10Y11~~
i<j cond.	iF	11X11	jF	11Y11

Table 3

Appendix A. Program 1


```
void bitrev(int *x, unsigned char *index, int n){
      short i,j,a,b;
      int         xi, xj;
      short nbits, nbot, ntop, ndiff, n2, astep;

      /* short leftzeros; */

      short *xs = (short *) x;

      /* *************************************
      * To calculate nbits on the C62XX it is easier
      * to use the left most bit detect directive as follows
      *     leftzeros = 31 - _lmbd(1,n);
      *     nbits = 31 - leftzeros;
      * ************************************* */

      nbits = 0;
      i = n;
      while (i > 1){
            i = i >> 1;
            nbits++;
      }

      nbot  = nbits >> 1;
      ndiff = nbits & 1;
      ntop  = nbot + ndiff;
      n2          = 1 << ntop;
      astep = 1 << ndiff;

      for (a = 0, i = 0; a < n2; a += astep){
            for (b = 0; b < n2; b++, i++) {
                  j = (index[b] << nbot) + index[a];
                  if (i < j) {
                        xi = x[i];
                        xj = x[j];
                        x[i] = xj;
                        x[j] = xi;
                  }
            }
      }
}
```

Appendix B. Program 2


```
void digitrev(int *x, unsigned char *index, int n, int radix){
      short i,j,a,b;
      int         xi, xj;
      short nbits, nbot, ntop, ndiff, n2, astep, radN;
      /* short leftzeros; */
      short *xs = (short *) x;
      /* ************************************
      * To calculate nbits and radN on the C62XX it is easier
      * to use the left most bit detect directive as follows
      *     leftzeros = 31 - _lmbd(1,n);
      *     nbits = 31 - leftzeros;
      * &   leftzeros = 31 - _lmbd(1,radix);
      *     radN = 31 - leftzeros;
      * ************************************ */

      nbits = 0;
      i = n;
      while (i > 1){
            i = i >> 1;
            nbits++;
      }

      radN = 0;
      i = radix;
      while (i > 1){
            i = i >> 1;
            radN++;
      }

      nbot  = nbits / (2*radN);
      nbot  = nbot * radN;
      ndiff = nbits % (2*radN);
      ntop  = nbot + ndiff;
      n2          = 1 << ntop;
      astep = 1 << ndiff;

      for (a = 0, i = 0; a < n2; a += astep){
            for (b = 0; b < n2; b++, i++) {
                  j = (index[b] << nbot) + index[a];
                  if (i < j) {
                        xi = x[i];
                        xj = x[j];
                        x[i] = xj;
                        x[j] = xi;
                  }
            }
      }
}
```

Appendix C. Program 3


```
void digitrev_index(unsigned char *index, int n2, int radix){

      int         i,j,k;
      index[0] = 0;
      for ( i = 1, j = n2/radix + 1; i < n2 - 1; i++){
            index[i] = j - 1;
            for (k = n2/radix; k*(radix-1) < j; k /= radix)
                        j -= k*(radix-1);
            j += k;
      }
      index[n2 - 1] = n2 - 1;
}
```

Appendix D. Program 4


```
void bitrev_improved(int *x, unsigned char *index, int n){

      int         I, a, b, ia, ib, ibs;
      short i0, i1, i2, i3;
      short j0, j1, j2, j3;
      int         xi0, xi1, xi2, xi3;
      int         xj0, xj1, xj2, xj3;
      short t;
      int         mask, nbits, nbot, ntop, ndiff, n2, halfn;
      short *xs   = (short *) x;

      nbits = 0;
      i = n;
      while (i > 1){
            i = i >> 1;
            nbits++;}

      nbot  = nbits >> 1;
      ndiff = nbits & 1;
      ntop  = nbot + ndiff;
      n2          = 1 << ntop;
      mask  = n2 - 1;
      halfn = n >> 1;

      for   (i0 = 0; i0 < halfn; i0 += 2) {
            b     = i0 & mask;
            a     = i0 >> nbot;
            if (!b) ia  = index[a];
            ib    = index[b];
            ibs   = ib << nbot;

            j0    = ibs + ia;
            t     = i0 < j0;
            xi0   = x[i0];
            xj0   = x[j0];
            if (t){x[i0] = xj0;
                  x[j0] = xi0;}

            i1    = i0 + 1;
            j1    = j0 + halfn;
            xi1   = x[i1];
            xj1   = x[j1];
            x[i1] = xj1;
            x[j1] = xi1;

            i3    = i1 + halfn;
            j3    = j1 + 1;
            xi3   = x[i3];
```

```
            xj3   = x[j3];
            if (t){x[i3] = xj3;
                  x[j3] = xi3;}
      }
}
```

Appendix E. Program 5


```
************************************************************************
*
*
*      TI Proprietary Information
*            Internal Data
*
*      BITREV
*
*      SWAPS VALUES IN AN ARRAY IN A BIT REVERSED FASHION
*                  - assumes complex imaginery pairs
*                  - assumes n is a power of 2
*
*      AUTHOR: NAT SESHAN
*
************************************************************************
      .global _bitrev
      .text

_bitrev:
START_TIME:
            LMBD  .L1   1,    A6,   A1    ; leftzeros = lmbd(1, n)
||          MV    .L2X  A4,   B8          ; copy x
||          MVK   .S2   31,   B0          ; constant 31
||          STW   .D2   A15,  *B15--      ; push A15
||          SUB   .S1X  B15,  8,    A15   ; copy stack pointer

            SUB   .L1X  B0,   A1,   A8    ; nbits = 31 - leftzeros
||          SHR   .S2X  A6,   1,    B6    ; halfn = n >> 1
||          ZERO  .S1   A3                ; i0 = 0
||          STW   .D1   A10,  *A15--[2]   ; push A10
||          STW   .D2   B10,  *B15--[2]   ; push B10

            SHR   .S1   A8,   1,    A0    ; nbot = nbits >> 1
||          AND   .L1   A8,   1,    A11   ; ndiff = nbits & 1
||          SHR   .S2   B6,   1,    B5    ; loop n/4 +2 times
||          STW   .D1   A11,  *A15--[2]   ; push A11
||          STW   .D2   B11,  *B15--[2]   ; push B11

            ADD   .D1   A0,   A11,  A11   ; ntop = nbot + ndiff
||          MVK   .S1   1,    A2,         ; constant 1
||          ADD   .L2   2,    B5,   B2    ; loop n/4 +2
||          MVK   .S2   1,    B1          ; setup priming count
||          MV    .L1X  B4,   A5          ; copy index

            SHL   .S1   A2,   A11,  A1    ; n2 = 1 << ntop
||          ZERO  .L1   A10               ; zero A10
||          STW   .D1   A12,  *A15        ; push A12
||          STW   .D2   B12,  *B15--[2]   ; push B12
```

```
            SUB   .L2X  A1,   1,    B13   ; mask = n2 - 1
||          ZERO  .L1   A1                ; prevent stores on first
iteration
||          STW   .D2   B13,  *B15--      ; push B13

            SHR   .S1   A3,   A0,   A11   ;** a = i0 >> nbot
||          AND   .L2X  A3,   B13,  B0    ;** b = i0 & mask

            LDB   .D2   *B4[B0],    B0    ;** ib = index[b]
||          ADD   .L2X  A3,   1,    B5    ;** i1 = i0 + 1

            ADD         B5,   B6,   B7    ;** i3 = i1 + halfn

            LDW   .D2   *B8[B7],    B9    ;** xi3 = x[i3]
||          ZERO  .D1   A12               ; zero A12

LOOP:
      [A1]  STW   .D2   B9,   *B8[B0]     ; if (t) x[j3] = xi3
||    [B2]  SUB         B2,   1,    B2    ; decrement loop counter
||          MPY   .M1   A1,   1,    A2    ; copy t
||          LDW   .D1   *A4[A3],    A11   ;* xi0 = x[i0]

      [A1]  STW   .D1   A11,  *A4[A10]    ; if (t) x[j0] = xi0
||    [B2]  B     .S2   LOOP              ; for loop
||          SHL   .S1X  B0,   A0,   A10   ;* ibs = ib << nbot
||          ADD         A3,   2,    A3    ;* ai0 += 2
||          MPY   .M2   B5,   1,    B10   ;* copy ai1
||          LDW   .D2   *B8[B5],    B11   ;* xi1 = x[i1]
||          MPY   .M1   A3,   1,    A9    ;* copy ai0

      [!B1] STW   .D2   A11,  *B8[B10]    ; x[i1] = xj1
||    [!B1] STW   .D1   B11,  *A4[A6]           ; x[j1] = xi1
||          ADD         A10,  A12,  A10   ;* j0 = ibs + ia
||          SHR   .S1   A3,   A0,   A11   ;** a = i0 >> nbot
||          AND   .L2X  A3,   B13,  B0    ;** b = i0 & mask

            ADD   .L1X  A10,  B6,   A6    ;* j1 = j0 + halfn
||          MPY   .M2   B7,   1,    B12   ;* copy ai3
||    [B1]  SUB         B1,   1,    B1    ; decrement priming counter
||          LDB   .D2   *B4[B0],    B0    ;** ib = index[b]
||          ADD   .L2X  A3,   1,    B5    ;** i1 = i1 + 1
||    [!B0] LDB   .D1   *A5[A11],   A12   ;** if (!b) ia = index[a]

      [A1]  STW   .D2   B0,   *B8[B12]    ; if (t) x[i3] = xj3
||          ADD   .L2X  A6,   1,    B0    ;* j3 = j0 + 1
||    [!B1] CMPLT .L1   A9,   A10,  A1    ;* t = i0 < j0
||          LDW   .D1   *A4[A6],    A11   ;* xj1 = x[j1]
||    [B1]  MPY   .M1   A4,   0,    A1    ; prime conditional store
||          ADD         B5,   B6,   B7    ;** i3 = i1 + halfn

            LDW   .D1   *A4[A10],   A7    ;* xj0 = x[j0]
||          LDW   .D2   *B8[B7],    B9    ;** xi3 = x[i3]

      [A2]  STW   .D1   A7,   *A4[A8]     ; if (t) x[i0] = xj0
```

```
||          LDW   .D2   *B8[B0],     B0    ;* xj3 = x[j3]
||          MPY   .M1   A9,   1,     A8    ;* copy ai0 again
END_LOOP:
            LDW   .D1   *A15, A12          ; pop A12
||          LDW   .D2   *++B15,      B13   ; pop B13

            LDW   .D1   *++A15[2],  A11   ; pop A11
||          LDW   .D2   *++B15[2],  B12   ; pop B12

            LDW   .D1   *++A15[2],  A10   ; pop A10
||          LDW   .D2   *++B15[2],  B11   ; pop B11
||          B     .S2   B3                 ; return

            LDW   .D1   *++A15,      B10   ; pop A15
||          LDW   .D2   *++B15[3],  A15   ; pop B10

            NOP   4
END_TIME
STOP:       NOP
```