Matrix Functions ■ 3

Matrices are useful in image processing and graphics algorithms because they provide a natural two-dimensional format to store x and y coordinates. Many signal processing algorithms perform mathematical operations on matrices. These operations range from scaling the elements in the matrix to performing an autocorrelation between two signals described as matrices.

The three basic matrix operations discussed in this chapter are

- multiplying one matrix by a vector
- multiplying one matrix by another matrix
- finding the inverse of a matrix

This chapter describes three ADSP-21000 family assembly language subroutines that implement these operations. The matrix buffers, interrupt vector tables, and DAG registers must be set up by the calling routine before the routines can access the matrices. Optionally, each subroutine can include setup code so it can run independently of a calling routine. This setup code is conditionally assembled by using the assembler's `-Didentifier` command line switch.

The implementations are based on the matrix algorithms described in [EMBREE91].

3 Matrix Functions

3.1 STORING A MATRIX

To minimize index register usage, the two-dimensional array matrix is stored in a single-dimensional array buffer and element positions are kept track of by the processor's DAG registers. The program can access any matrix element by using the index, modify, and length registers. The elements are stored in *row major order;* all the elements of the first row are first in the buffer, then all of the elements in the second row, and so forth.

For example, a 3×3 matrix with these elements

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

is placed in a nine element buffer in this order

$\{A_{11}, A_{12}, A_{13}, A_{21}, A_{22}, A_{23}, A_{31}, A_{32}, A_{33}\}$

The elements can be read continuously and consecutively if you

- use circular buffering
- use the correct modify values for index pointers
- keep track of the pointer in the matrix

To read the elements in a row consecutively, set the index pointer to the location of the first element in the row and set the modify value to 1.

To read the elements in a column consecutively, set the index register to the location of the first element in the column and set the modify value equal to the length of the row.

For example, to read the first column in the 3 ×3 matrix example, set the index pointer to point to A_{11} and the set modify value to three (3). After an indirect memory read of the first element, the index pointer points to A_{21}, the second column element.

This method of storing and accessing a matrix keeps available more DAG registers than using a different index pointer for each row of the matrix.

3.2 MULTIPLICATION OF A M×N MATRIX BY AN N×1 VECTOR

This section discusses how to multiply a two-dimensional matrix of arbitrary size, *M×N*, by a vector (one-dimensional matrix), *N×1*.

3.2.1 Implementation

The matrix elements are stored in the buffer `mat_a`, with length of *M×N* (where *M* is the number of rows in a matrix and *N* is the number of columns).

The vector is stored in a separate buffer, `mat_b`, with length of *N*. (The number of rows in the vector must equal the number of columns of the matrix.)

The result of the multiplication is another vector stored in a buffer, `mat_c`, with length of *M* (the number of rows of the first matrix).

The *M×N* matrix and *M*×1 result vector are stored in a data memory segment, while the *N*×1 multiply vector is stored in a program memory segment. This lets the algorithm take advantage of the dual fetch of the multifunction instructions to access the next two elements of the matrix and vector while multiplying the current two elements.

The multiplication of the matrix and vector is performed in a two-level deep loop. The inner loop, `row`, is a single instruction that performs the multiplication of the current two elements and accumulates the previous multiplication while fetching the next two elements to be multiplied. This `row` loop is performed *N* times (the number of columns) to account for the number of multiply/accumulate steps done for each row.

The outer loop, `column`, takes the final accumulated result and stores it in the `mat_c` result buffer. The loop also clears the accumulator result, R8, so that it can be used again for the next `row` loop. The `column` loop is performed *M* times (the number of rows) to account for the number of times it has to perform the `row` loop operations.

For efficient loops, the operation of multiplication, accumulation and the next two data fetches are all performed in the same multifunction instruction. Each multiply is done on operands which were fetched on the pervious iteration of the inner loop. Similarly, each addition accumulates the product from the multiply in the previous loop iterations. Before the loops are entered the first time, the operands must be preloaded from memory.

This technique of preloading registers so multifunction instructions can be used in the loop body is called “rolling the loop.” To roll the loop, the first instructions outside of the `row` and `column` loops clear the accumulator, fetch the first two elements to be multiplied and multiplies them while fetching the next two elements.

One cycle can be saved when clearing the accumulator by performing an exclusive or operation (`XOR`) between the accumulator register (R8) and itself. This lets the processor fetch the next two elements while performing a computation, which is faster than using one cycle to clear the accumulator by loading R8 with a zero and then a second cycle to perform the fetches. The processor cannot perform a register load and two data fetches in the same cycle.

This trick of combining the computation with data fetches in a single instruction is also used for the last instruction of the `column` loop when clearing the accumulator for the next loop and storing the final accumulation result to the `mat_c` buffer. Since this loop may be performed many times (depending on the number of rows in the matrix), it can greatly reduce the time spent executing the algorithm.

The accumulation operation of the multifunction instruction used in the `row` loop is performed last. When the accumulation of the last element is performed for the current row, the multiplier has already multiplied the first two elements of the next row and the second two elements have been fetched. Provided the current row is not the last one, the extra multiplication and data fetches roll over into the next iteration of the loop.

When performing the accumulation on the last elements of the last row, the index pointers of the input buffers wrap around to the start of the buffer; the multiplication and data fetches for the first row are repeated. Since those operations are redundant, their destination registers can be written over after the routine completes. Note that the index pointers are also modified and point to the third elements in the matrices when the routine is finished. Therefore, the pointers must be restored if the same matrices must be used in a subsequent routine.

3.2.2 Code Listing–M×N By N×1 Multiplication

```
/*********************************************************************************

File Name
   MxNxNx1.ASM

Version
   25-APR-91

Purpose
   Matrix times a Vector.

   Matrix dimensions are arbitrary. Matrix A accessed as a circular buffer so
that the         last iteration of the inner loop will do a dummy read from a
known location.

   Use the -Dexample Assembler Preprocessor Switch to include assembly of an
   example calling routine

Equations Implemented
   [Mx1]=A[MxN]*B[Nx1]

Calling Parameters
   Constants: m, n
   pm(mat_b[n]) row major, dm(mat_a[m*n]) row major,
   M1=1;
   M9=1;
   B0=mat_a;    L0=@mat_a;
   B1=mat_c;    L1=0;
   B8=mat_b;    L8=@mat_b;

Return Values
   dm(mat_c[m]) row major

Registers Affected
   F0,F4,F8,F12, I0,I1,I8

Cycle Count
   cycles=6+M(3+N)+5   (entrance + core + 5 cache)

# PM Locations
   pm code=8 words, pm data=n words

# DM Locations
   dm data=m*n+m words
*********************************************************************************/
```

(listing continues on next page)

```
/* dimension constants */
#define          M 4
#define          N 4

#ifndef example
.GLOBAL mxnxnx1;
.EXTERN mat_a, mat_b,mat_c;
#endif

#ifdef example
.SEGMENT/DM dm_data;
.VAR             mat_a[M*N]=”mat_a.dat”;
.VAR             mat_c[M];
.ENDSEG;

.SEGMENT/PM pm_data;
.VAR             mat_b[N]=”mat_bb.dat”;
.ENDSEG;

.SEGMENT/PM      rst_svc;
          dmwait=0x21; /* set dm waitstates to zero */
          pmwait=0x21; /* set pm waitstates to zero */
          jump setup;
.ENDSEG;

          /* example calling code */
.SEGMENT/PM pm_code;
setup:           m1=1;
          m9=1;
          b0=mat_a;     l0=@mat_a;
          b1=mat_c;     l1=0;
          b8=mat_b;     l8=@mat_b;
          call mxnxnx1;
          idle;
.ENDSEG;
#endif

          /* matrix multiply starts here */
.SEGMENT/PM pm_code;
mxnxnx1:  r8=r8 xor r8, f0=dm(i0,m1), f4=pm(i8,m9); /* clear f8 */
          f12=f0*f4,  f0=dm(i0,m1), f4=pm(i8,m9);
          lcntr=M, do column until lce;
             lcntr=N, do row until lce;
row:            f12=f0*f4, f8=f8+f12, f0=dm(i0,m1), f4=pm(i8,m9);
column:       r8=r8 xor r8, dm(i1,m1)=f8;
          rts;
.ENDSEG;
```

Listing 3.1 MxNxNx1.asm

3.3 MULTIPLICATION OF A M×N MATRIX BY A N×O MATRIX

This section discusses how to multiply two matrices of arbitrary size.

3.3.1 Implementation

The two input matrices are stored in separate data memory and program memory segments so multifunction instructions can be used to produce efficient code.

- The first input matrix ($M \times N$), `mat_a`, is stored in data memory.

- The second input matrix ($N \times O$), `mat_b`, is stored in program memory.

- The result matrix ($M \times O$), `mat_c`, is stored in data memory.

You can think of matrix-matrix multiplication as several matrix-vector multiplications. The matrix is always the first input matrix and each "vector" is a column in the second matrix. As discussed in the previous section, the matrix-vector multiplication code consisted of two loops: `row` and `col`. To obtain the matrix-matrix multiplication code a third loop is added, `colrow`, that simply repeats the entire block of matrix-vector code for the number of columns in the second matrix.

Repeating this code, however, requires that the modifications of the index pointers for each matrix be handled differently. The modify value for `mat_a (m1)` is still set to 1 to increment through the matrix row by row. However, to consecutively increment through the columns of `mat_b`, the modify value (m10) needs to be set to the number of columns in that matrix. For this code example, the second matrix is a 4×4 matrix, so the modify value is 4. The result matrix is written to column by column and has the same modify value.

After a single loop through the matrix-vector code, a column in the result matrix is complete. Because the code is looped to reduce the number of cycles inside the loop, the position of the index pointers are incorrect to perform the next matrix-vector multiplication. Modify instructions are included at the end of the `rowcol` loop that modify the index pointers so they begin at the correct position. The index register for `mat_a` is modified to point to the beginning of the first row and first column. The index registers (pointers) for both `mat_b` and the result matrix `mat_c` are modified to point to the beginning of the next column.

The pointer to the matrix `mat_a` is modified to point to the first element of the first row by performing a dummy fetch with a modify value of –2. (Because the loop is rolled, the DAG fetches the first two row elements again.)

A dummy fetch is also used to modify the matrix pointer for `mat_b` to point to the first element of the next column. The modify value for this dummy fetch is *–(O * 2 – 1)*, where *O* is the number of columns in the matrix. Because of loop rolling, the index pointer points to the third column element. Therefore, the index pointer needs to be adjusted backwards by two rows minus one element. For this code example, the index pointer points to the third column position. To make the pointer point to the first element of the next column, modify the pointer by *–(4 * 2 – 1) = –7.*

Finally, the pointer to the result matrix `mat_c` is modified to point to the first element of the next column by modifying the index pointer by one. Since the instruction that writes the result is the last one in the loop, loop rolling does not affect this index pointer. Dummy reads modify the index registers, instead of modify instructions, so that a multifunction instruction can be performed. This reduces the number of cycles in the loop.

The loop `colrow` is then executed again, which calculates the result for the next column in the result matrix. The loop `colrow` repeats until all the columns in the result matrix are filled.

The final pointer positions are as follows

- position (1,1) for `mat_a` (beginning of the buffer and matrix),
- position (2,1) for `mat_b` (row 2, column 1)
- position (2,1) for the result `mat_c` (row 2, column 1).

To use the same matrices in a subsequent routine, first reset the index pointers to the beginning of each buffer.

3.3.2 Code Listing–M×N By N×O Multiplication

```
/
*********************************************************************************

File Name
   MxNxNxO.ASM

Version
   25-APR-91

Purpose
   Matrix times a Matrix.

   The three matrices have arbitrary dimensions. Matrix A accessed as a
   circular buffer so that the last iteration of the inner loop will do a
   dummy read from a known location.

   Use the -Dexample Assembler Preprocessor Switch to include assembly of an
   example calling routine

Equations Implemented
   C[MxO]=A[MxN]*B[NxO]

Calling Parameters
   Constants: m, n, o
   pm(mat_b[N*O]) row major, dm(mat_a[M*N]) row major
   dm(mat_c[M*O]) row major
   M1=1;
   M2=-2;
   M3=o;
   M9=-(o*2-1);
   M10=o;
   B0=mat_a;    L0=@mat_a;
   B1=mat_c;    L1=@mat_c;
   B8=mat_b;    L8=@mat_b;


Return Values
   F0,F4,F8,F12, I0,I9, B8

Registers Affected
   F0,F4,F8,F12, I0,I1,I8

Cycle Count
   cycles=4+o(m(n+2)+5)+7     (entrance + core + 7 cache)

# PM Locations
   pm code=11 words, pm data=NxO words,

# DM Locations
   dm data=MxN+MxO words

********************************************************************************/
```

(listing continues on next page)

```
/* dimension constants */
#define         M 4
#define         N 4
#define         O 4

#ifndef example
.GLOBAL mxnxnxo;
.EXTERN mat_a, mat_b, mat_c;
#endif

#ifdef example
.SEGMENT/DM dm_data;
.VAR            mat_a[M*N]=”mat_a.dat”;
.VAR            mat_c[M*O];
.ENDSEG;

.SEGMENT/PM pm_data;
.VAR            mat_b[N*O]=”mat_b.dat”;
.ENDSEG;

.SEGMENT/PM     rst_svc;      /* reset vector */
          dmwait=0X21; /* set dm waitstates to zero */
          pmwait=0X21; /* set pm waitstates to zero */
          jump setup;
.ENDSEG;

          /* example calling code */
.SEGMENT/PM pm_code;
setup:          m1=1;
          m2=-2;
          m3=O;
          m9=-(O*2-1);
          m10=O;
          b0=mat_a;    l0=@mat_a;
          b1=mat_c;    l1=@mat_c;
          b8=mat_b;    l8=@mat_b;
          call mxnxnxo;
          idle;
.ENDSEG;
#endif

          /* matrix multiply starts here */
.SEGMENT/PM pm_code;
mxnxnxo:  lcntr=O, do colrow until lce;
             r8=r8 xor r8, f0=dm(i0,m1), f4=pm(i8,m10); /* clear f8 */
             f12=f0*f4,  f0=dm(i0,m1), f4=pm(i8,m10);
             lcntr=M, do column until lce;
                 lcntr=N, do row until lce;
row:                f12=f0*f4, f8=f8+f12, f0=dm(i0,m1), f4=pm(i8,m10);
column:           r8=xor r8, dm(i1,m3)=f8;
                 f0=dm(i0,m2), f4=pm(i8,m9); /* modify with dummy fetches */
colrow:      modify(i1,1);
          rts;
.ENDSEG;
```

Listing 3.2 MxNxNxO.asm

3.4 MATRIX INVERSION

The inversion of a matrix is used to solve a set of linear equations. The format for the linear equations is

$$Ax = b$$

The vector x contains the unknowns, the matrix A contains the set of coefficients, and the vector b contains the solutions of the linear equations. The matrix A must be a non-singular square matrix. To get the solution for x, multiply the inverse of matrix A by the constant vector, b. The inverse matrix is useful if a different constant vector b is used with the same equations. The same inverse can be used to solve for the new solutions.

Because of round-off error that occurs during the elimination process, it is hard to get accurate results when inverting large matrices. The Gauss-Jordan method elimination with full pivoting, however, provides a highly accurate matrix inverse.

Gauss-Jordan elimination can become numerically unstable unless pivoting is used. Full pivoting is the interchanging of rows and columns in a matrix to have the largest magnitude element on the diagonal of the matrix. This diagonal element, called the pivot, is then used to divide the other elements of the row. The row is used to eliminate other column elements to obtain the identity matrix. The same elimination procedures are performed on an original identity matrix. Once the matrix is reduced to the identity matrix, the original identity matrix will contain the inverse matrix. This resulting matrix must be adjusted for any interchanging of rows and columns.

The Gauss-Jordan algorithm is an in-place algorithm: the input matrix and output result are stored in the same buffer of data.

The algorithm is subdivided into five sections:

- The first section searches for the largest element of the matrix.
- The second section places that element on the diagonal of the matrix making it the pivot element. The row that contained the pivot element is marked so that it won't be used again.
- The third section of code divides the rest of the row by that pivot element.

- The fourth section performs the in-place elimination.
- The first four sections are repeated until all of the rows of the matrix have been checked for a pivot point.
- The fifth section performs the corrections for swapping rows and columns.

3.4.1 Implementation

The matrix is in data memory and uses *N*×*N* locations, where *N* is the size of the matrix (*N*=3 for a 3×3 matrix). The pivot flag (`pf`) and swap column (`swc`) arrays are also stored in data memory. The swap row array (`swr`) is in program memory.

This routine uses all of the universal registers (F0-F15) and eight of the DAG registers (I0-I8) as either data storage or temporary registers. Therefore, if this routine is called from a routine that uses these registers, switch to the secondary registers before starting the routine. Make sure to switch back to the primary registers when done.

The first four sections of the algorithm are enclosed in the `full_pivot` loop. Each pass through the loop searches for the largest element in the matrix, places that element on the diagonal, and performs the in-place elimination. The loop repeats for every row of the matrix.

The first section of the algorithm searches the entire matrix for the largest magnitude pivot value in the nested loops `row_big` and `column_big`. At the beginning of each loop, it checks if the pivot flag is set for that row or column. If the pivot flag is set, that row or column contains a pivot point that has already been used. Since any element in a row or column that previously contained a pivot point cannot be reused, the loop is skipped and the index pointer is modified to point to the next row or column.

The loop performs a comparison of all the elements in the matrix and the largest value is stored in register F12—the pivot element. The row that contained the pivot point is stored in the `pf` buffer so that any elements in that row will not be used again. A test is performed to see if the pivot element is a zero. If the pivot point is zero, the matrix is singular and does not have a realizable inverse. The routine returns from the subroutine and the error flag is register F12 containing a zero.

The second section of the algorithm checks if the pivot element is on the diagonal of the matrix. If the pivot element is not on the diagonal, corresponding rows and columns are swapped to place the element on the diagonal. The position of the pivot element is stored in the counters R2 and R10. If these two numbers are equal, then the element is already on a diagonal and the algorithm skips to the next section of the algorithm. If the numbers are not equal, the loop `swap_row` is performed. This loop swaps the corresponding row and column to place the pivot element on the diagonal of the matrix. The row and column numbers that were swapped are stored in separate arrays called `swr (row)` and `swc (column)`. These values will be used in the fifth section to correct for any swapping that has occurred.

The third section of the algorithm divides all the elements of the row containing the pivot point by the pivot point. The inverse of the pivot point is found with the macro `DIVIDE`. The result of the macro is stored in the f1 register. The other elements in the row are then multiplied by the result in the loop `divide_row`.

The fourth section of the algorithm performs the in place elimination. The elimination process occurs within the two loops `fix_row` and `fix_column`. The results of the elimination replace the original elements of the matrix.

These four sections described are repeated *N* times, where *N* is the number of rows in the matrix.

The fifth section of the algorithm is executed after the entire matrix is reduced. This section fixes the matrix if any row and column swapping was done. The algorithm reads the values stored in the arrays `swr` and `swc` and swaps the appropriate columns if the values are not zero.

3.4.2 Code Listing—Matrix Inversion

```
/
*********************************************************************************

File Name
   MATINV.ASM

Version
   May 6 1991

Purpose
   Inverts a square matrix using the Gauss-Jordan elimination.
   algorithm with full pivoting.

   See  P.M. Embree and B. Kimble. C Language Algorithms For Digital
   Signal Processing. Chap. 6, Sect. 6.2.3, pp. 326-329.Prentice-Hall, 1991

Equations Implemented
   C[MxO]=A[MxN]*B[NxO]

Calling Parameters
           dm(mat_a[n*n]) row major, dm(pf[n+1]), dm(swc[n]);
           pm(swr[n]);
           r14=n;      (n= number of rows (columns))
           m0=1; m1=-1;
           m8=1; m9=-1;
           b0=mat_a;
           b1=pf;
           b7=swc; l7=0;
           b8=swr; l8=0;

Return Values
   dm(mat_a[n*n]) row major;
   f12=0.0 -> matrix is singular


Registers Affected
   f0 - f15,
   i0 - i7, i8, m2

Cycle Count
    maximum number= worst case= 7.5n**3+25n**2+25.5n+23 (approximated)

# PM Locations
   pm code= 93 words, pm data= n words

# DM Locations
   dm data= n*n+2n+1 words
```

```
********************************************************************************/

/* To assemble the example below type the following command

                   asm21k -Dexample matinv  */

#include  “macros.h”

#define   n      3

#ifndef          example
.GLOBAL          mat_inv;
.EXTERN          mat_a;
#endif

#ifdef           example
.SEGMENT/DM      dm_data;
.VAR      mat_a[n*n]= “mat_a1.dat”;
.VAR      pf[n+1];
.VAR      swc[n];
.ENDSEG;

.SEGMENT/PM rst_svc;
     dmwait=0x21;      /*set dm waitstates to zero*/
           pmwait=0x21;/*set pm waitstates to zero*/
     jump setup;
.ENDSEG;

.SEGMENT/PM      pm_data;
.VAR      swr[n];
.ENDSEG;
          /* example calling code */

.SEGMENT/PM      pm_code;
setup:           b0=mat_a; /*i0 -> a(row,col)*/
          b1=pf;   /*i1 -> pf= pivot_flag*/
          b7=swc;  /*i7 -> swc= swap_col*/
          b8=swr;  /*i8 -> swr= swap_row*/
          l7=0;
          l8=0;
          m0=1;
          m1=-1;
          m8=1;
          m9=-1;
          r14=n;
          call mat_inv;
```

(listing continues on next page)

```
          idle;
.ENDSEG;
#endif

                 /* Matrix inversion starts here */
.SEGMENT/PM      pm_code;
mat_inv:  r13=r14*r14(ssi), b3=b0;
          l0=r13;  /*matrix in a circular data buffer*/
          b4=b0;
          b5=b0;
          b6=b0;
          l3=l0;
          l4=l0;
          l5=l0;
          l6=l0;
          r13=r14+1, b2=b1;
          l1=r13;  /*pf in a circular data buffer*/
          l2=l1;
          f9=0.0;
          f8=2.0;  /*2.0 is required for DIVIDE_macro*/
          f7=1.0;  /*1.0 is a numerator for DIVIDE_macro*/
          r13=fix f9, m2=r14;
          lcntr=r14, do zero_index until lce;
           dm(i7,m0)=r13, pm(i8,m8)=r13;
zero_index: dm(i1,m0)=r13;
          f0=pass f9, dm(i1,m0)=r13; /*f0= big*/

          lcntr=r14, do full_pivot until lce;
                 /*find the biggest pivot element*/
           r1=pass r13, r11=dm(i1,1); /*r1= row no., r11= pf(row)*/
           lcntr=r14, do row_big until lce;
              r11=pass r11, i4=i3; /*check if pf(row) is zero*/
              if ne jump (PC,12), f4=dm(i0,m2); /*i0 -> next row*/
              r5=pass r13, r15=dm(i2,1); /*r5= col no., r15= pf(col)*/
              lcntr=r14, do column_big until lce;
                 r15=pass r15; /*check if pf(col) is zero*/
                 if ne jump column_big (db);
                 f4=dm(i0,1); /*f4= a(row,col)*/
                 f6=abs f4;
                 comp(f6,f0); /*compare abs_element to big*/
                 if lt jump column_big;
                 f0=pass f6, f12=f4; /*f0= abs_element, f12= pivot_element*/
                 r2=pass r1, r10=r5; /*r2= irow, r10= icol*/
column_big:       r5=r5+1, r15=dm(i2,1);
row_big:    r1=r1+1, r11=dm(i1,1);

          /*swap rows to make this diagonal the biggest absolute pivot*/
           f12=pass f12, m5=r10; /*check if pivot is zero, m5= icol*/
           if eq rts; /*if pivot is zero, matrix is singular*/
           r1=r2*r14 (ssi), dm(m5,i1)=r5; /*pf(col) not zero*/
           r5=r10*r14 (ssi), m6=r1;
           comp(r2,r10), r1=dm(i3,m6); /*i3 -> a(irow,col)*/
           dm(i7,m0)=r10, pm(i8,m8)=r2; /*store icol in swc and irow in swr*/
                   if eq jump row_divide (db);
           r2=pass r13, m7=r5;
           modify(i4,m7); /*i4 -> a(icol,col)*/
           i5=i4;
           lcntr=r14, do swap_row until lce;
            f4=dm(i3,0);             /*f4= temp= a(irow,col)*/
            f0=dm(i5,0);             /*f0= a(icol,col)*/
            dm(i3,1)=f0;             /*a(irow,col)= a(icol,col)*/
swap_row:         dm(i5,1)=f4;       /*a(icol,col)= temp*/

                 /*divide the row by the pivot*/
row_divide:       f6=pass f7, i5=i4;
           DIVIDE(f1,f6,f12,f8,f3); /*f1= pivot_inverse*/
           i6=i5;
           f4=dm(i4,1);
           lcntr=r14, do divide_row until lce;
              f5=f1*f4, f4=dm(i4,1);
divide_row:    dm(i6,1)=f5;
           dm(m5,i5)=f1;

                 /*fix the other rows by subtracting*/
           lcntr=r14, do fix_row until lce;
               comp(r2,r10), i6=i5; /*check if row= icol*/
               if eq jump (PC,8), f4=dm(i0,m2); /*i0 -> next row*/
               f4=dm(m5,i0); /*temp= a(row,icol)*/
               dm(m5,i0)=f9;
               f3=dm(i6,1);
               lcntr=r14, do fix_column until lce;
                  f3=f3*f4, f0=dm(i0,0);
                  f0=f0-f3, f3=dm(i6,1);
fix_column:        dm(i0,1)=f0;
fix_row:       r2=r2+1;
full_pivot:  f0=pass f9, i3=i0;

                 /*fix the affect of all the swaps for final answer*/
          r0=dm(i7,m1), r1=pm(i8,m9); /*i7 -> swc(N-1), i8 -> swr(N-1)*/
          r0=dm(i7,m1), r1=pm(i8,m9); /*r0= swc(N-1), r1= swr(N-1)*/
          lcntr=r14, do fix_swap until lce;
             comp(r0,r1), m5=r0; /*m5= swc(swap)*/
             if eq jump fix_swap;
             m4=r1; /*m4= swr(swap)*/
             lcntr=r14, do swap until lce;
             f4=dm(m4,i0); /*f4= temp= a(row,swr(swap))*/
             f0=dm(m5,i0); /*f0= a(row,swc(swap))*/
             dm(m4,i0)=f0; /*a(row,swr(swap))= a(row,swc(swap))*/
             dm(m5,i0)=f4; /*a(row,swc(swap))= temp*/
swap:        modify(i0,m2);
fix_swap: r0=dm(i7,m1), r1=pm(i8,m9);
          rts;
.ENDSEG;
```

Listing 3.3 matinv.asm

3.5 REFERENCES

[EMBREE91] Embree, P. and B. Kimble. 1991. *C Language Algorithms For Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall.