Library of Macros for Optimization Using eMAC and MAC

Programmer's Manual

Document Number: CFLMOPM
Rev. 1.0
10/2005

Freescale Semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

emac

Contents

About This Book

This programmer's manual provides a detailed description of a set of macros used for optimizations.

The information in this book is subject to change without notice, as described in the disclaimers on the title page. As with any technical documentation, it is the reader's responsibility to be sure he is using the most recent version of the documentation.

To locate any published errata or updates for this document, refer to the world-wide web at http://www.freescale.com/coldfire.

Audience

This manual is intended for system software developers and applications programmers who want to develop products with ColdFire processors. It is assumed that the reader understands microprocessor system design, basic principles of software and hardware, and basic details of the ColdFire® architecture.

Organization

This document is organized into five chapters.

Chapter 1	"Overview" includes a general description of the library of Macros.
Chapter 2	"Macros for 1D Array Operations" describes the macros used for 1D Array operations.
Chapter 3	"Macros for 2D Array Operations" describes the macros used for 2D Array operations.
Chapter 4	"Macros for DSP Algorithms" includes the description of several macros used for DSP algorithms.
Chapter 5	"Macros for Mathematical Functions" includes the description of several macros used for common mathematical operations.
Chapter 6	"QuickStart for CodeWarrior" includes a step-by-step description of how to create a new project in CodeWarrior using the library of Macros.

Conventions

This document uses the following notational conventions:

```
CODE            Courier in box indicates code examples.
```

`Prototypes` Courier is used for code in function prototypes.

formulas Italics is used for formulas.

- All source code examples are in C and Assembly.

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

FRAC32 Data type that represents 32-bit signed fractional value

FIXED64 Data type that represents 64-bit signed value, with 32 bits in integer part and 32 bits in fractional part

References

The following documents were referenced to write this document:

1. *ColdFire Family Programmer's Reference*, Rev. 3
2. *MCF5249 ColdFire User's Manual*, Rev. 0
3. *MCF5282 ColdFire User's Manual*, Rev. 2.3
4. The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/)

Revision History

The following table summarizes revisions to this manual since the previous release (Rev. 1.4).

Revision History

Revision Number	Date of release	Substantive Changes
1.0	10/2005	Initial Public Release

Chapter 1
Overview

The Library of Macros was designed to ensure efficient programming of the ColdFire processor by using MAC and eMAC units where applicable.

This document is the main document describing the Library of Macros and it provides information on each macro in the library:

- "Macros Description" provides general information about a macro, including a description and its purpose.
- "Parameters Description" provides information on the invoking technique of a macro, as well as its parameters and returned value.
- "Description of Optimization" provides information on techniques that were used during macro optimization.

1.1 Project Resources

The following resources were used in the project:

- Targets

  MCF5249 Evaluation board (M5249C3)

  MCF5206 Evaluation board (M5206EC3)

  MCF5282 Evaluation board (M5282EVB)

- Compilation tools

  Metrowers Codewarrior for ColdFire V4.0

  Metrowers Codewarrior for ColdFire V5.0

  WindRiver Diab RTA 4.4b Suite

  gcc 3.3.3 GNU compiler

1.2 Structure of the Project and Installation

The Library of Macros has the following structure:

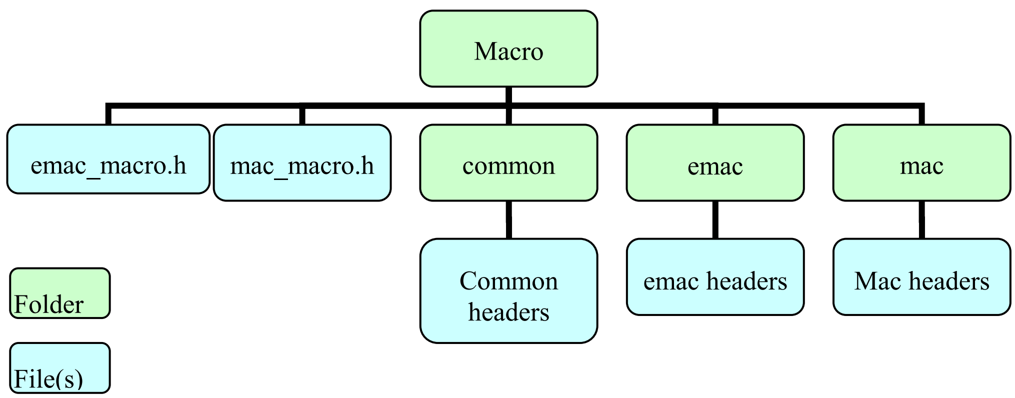

Figure 1-1. Structure of Macro Library

There are two main parts for the library:

The library for the eMAC unit

The library for the MAC unit

Each part has its own header file: “mac_macro.h” and “emac_macro.h,” respectively. Each part also includes some common macros and can be logically divided in four sections:

- 1D array operations
- 2D array operations
- DSP algorithms
- Mathematical functions

To use the library of macros within your project, first of all you have to include the appropriate C header file. Include file mac_macro.h if you use the MAC unit, or file emac_macro.h if you use the eMAC unit in your program. To avoid macroname conflict, you shouldn't include both headers in the same program. Moreover, there is no need to include them both, because macros for the same functions are doubled in these headers.

Chapter 2
Macros for 1D Array Operations

2.1 ARR1D_SUM_UL, ARR1D_SUM_SL

2.1.1 Macros Description

These macros compute the sum of the array elements of unsigned/signed values. This sum is computed by the following formula:

$$res = \sum_{i=0}^{size-1} x_i$$

where x_i, – element of the input vector, *size* – number of elements in the input vector

2.1.2 Parameters Description

Call(s):

```
int ARR1D_SUM_UL(unsigned long *src, int size)
```

```
int ARR1D_SUM_SL(signed long* src, int size)
```

Parameters:

Table 2-1 ARR1D_SUM Parameters

src	in	Pointer to the source vector
size	in	Number of elements in vector

Returns: The ARR1D_SUM macros return the unsigned/signed sum of array elements.

2.1.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE; i++)
        res += arr1[i];
```

Optimization can be done using the following techniques:

1. Loop unrolling by four
2. Postincrement addressing mode to access input array elements
3. Descending loop organization

The following should be noticed:

- The d0 register always holds the sum of array elements.
- The a0 register holds the pointer to input array.
- The d1 register is the counter.

Optimized code:

```
loop1:
    add.l (a0)+,d0
    add.l (a0)+,d0
    add.l (a0)+,d0
    add.l (a0)+,d0
    subq.l #1,d1
bne loop1
```

2.1.4 Differences Between the ARR1D_SUM_UL and the ARR1D_SUM_SL Macros

The type of ARR1D_SUM_UL parameters (*src) is unsigned long.

The type of ARR1D_SUM_SL parameters (*src) is signed long.

2.2 ARR1D_ADD2_UL, ARR1D_ADD2_SL

2.2.1 Macros Description

These macros compute the elementwise sum of two vector arrays with unsigned/signed values. The elementwise sum is computed by the following formula:

$$x_i = x_i + y_i$$
$$x_i \in X, y_i \in Y, i \in [0, size - 1];$$

where *X, Y* – input vectors, x_i, y_i – element of the corresponding vector, *size* – number of elements in the input vectors

2.2.2 Parameters Description

Call(s):

```
int ARR1D_ADD2_UL(unsigned long *dest, unsigned long *src, int size)
```

```
int ARR1D_ADD2_SL(signed long* dest,signed long* src, int size)
```

Parameters:

Table 2-2 ARR1D_ADD2 Parameters

dest	in/out	Pointer to the destinstion vector
src	in	Pointer to the source vector
size	in	Number of elements in vector

Returns: The ARR1D_ADD2 macro generates unsigned/signed output values, which are stored in the array pointed to by the parameter *dest*.

2.2.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE; i++)
    arr_c[i] += arr1[i];
```

Optimization can be done using the following techniques:

1. Loop unrolling by four.
2. Every four values of array dest used in each iteration are loaded with only one movem instruction.
3. Every four values of array src used in each iteration are loaded using postincrement addressing mode while performing additons.
4. After perfoming additions, the resulting four values in each iteration are stored with only one movem instruction.
5. If the number of elements is not divisible by 4, the tail elements are processed in regular order.

Optimized code:

```
    move.l size,d1
    move.l d1,d2
    asr.l  #2,d1
    beq l1
l2:
    movem.l (a0),d3-d6
    add.l (a1)+,d3
    add.l (a1)+,d4
    add.l (a1)+,d5
    add.l (a1)+,d6
    movem.l d3-d6,(a0)
    add.l #16,a0
    subq.l #1,d1
    bne l2
l1:
    and.l #3,d2
    beq l4
l3:
    move.l (a0),d3
    add.l (a1)+,d3
    move.l d3,(a0)+
    subq.l #1,d2
    bne l3
l4:
```

2.2.4 Differences Between the ARR1D_ADD2_UL and the ARR1D_ADD2_SL Macros

The type of ARR1D_ADD2_UL parameters (*dest, *src) is *unsigned long.*

The type of ARR1D_ADD2_SL parameters (*dest, *src) is *signed long*.

2.3 ARR1D_ADD3_UL, ARR1D_ADD3_SL

2.3.1 Macros Description

These macros compute the elementwise sum of two vector arrays with unsigned/signed values, and store the results to a third vector with unsigned/signed values. The elementwise sum is computed by the formula:

$$z_i = x_i + y_i$$
$$x_i \in X, y_i \in Y, z_i \in Z, i \in [0, size - 1];$$

where *X*, *Y* – input vectors, x_i, y_i – elements of the corresponding vectors, *Z* – resultant vector, z_i – element of vector Z, *size* – number of elements in the input vectors

2.3.2 Parameters Description

Call(s):

```
int ARR1D_ADD3_UL(unsigned long* dest, unsigned long* src1, unsigned long*
src2, int size)
```

```
int ARR1D_ADD3_SL(signed long* dest, signed long* src1, signed long* src2, int
size)
```

Parameters:

Table 2-3. ARR1D_ADD3 Parameters

dest	in/out	Pointer to the destinstion vector
src1	in	Pointer to the source vector1
src2	in	Pointer to the source vector2
size	in	Number of elements in vector

Returns: The ARR1D_ADD3 macro generates unsigned/signed output values, which are stored in the array pointed to by the parameter *dest*.

2.3.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE; i++)
    arr_c[i] += arr1[i];
```

Optimization can be done using the following techniques:

1. Loop unrolling by four.
2. Every four values of array dest used in each iteration are loaded with only one movem instruction.
3. Every four values of array src used in each iteration are loaded using postincrement addressing mode while performing additons.
4. After perfoming additions, the resulting four values in each iteration are stored with only one movem instruction.
5. If the number of elements is not divisible by 4, the tail elements are processed in regular order.

Optimized code:

```
        move.l size,d1
        move.l d1,d2
        asr.l  #2,d1
        beq l1
    l2:
        movem.l (a0),d3-d6
        add.l (a1)+,d3
        add.l (a1)+,d4
        add.l (a1)+,d5
        add.l (a1)+,d6
        movem.l d3-d6,(a0)
        add.l #16,a0
        subq.l #1,d1
        bne l2
```

```
l1:
    and.l #3,d2
    beq l4
l3:
    move.l (a0),d3
    add.l (a1)+,d3
    move.l d3,(a0)+
    subq.l #1,d2
    bne l3
l4:
```

2.3.4 Differences Between the ARR1D_ADD3_UL and the ARR1D_ADD3_SL Macros

The type of ARR1D_ADD3_UL parameters (*dest, *src1, *src2) is *unsigned long.*

The type of ARR1D_ADD3_SL parameters (*dest, *src1, *src2) is *signed long*.

2.4 ARR1D_ADDSC_UL, ARR1D_ADDSC_SL

2.4.1 Macros Description

This macro computes the elementwise sum of a vector array of unsigned/signed values with a scalar unsigned/signed value. The elementwise sum is computed by the formula:

$$x_i = x_i + scalar$$
$$x_i \in X, i \in [0, size - 1];$$

where X – input vector, x_i – element of vector X, *scalar* – variable with an unsigned/signed value, *size* – number of elements in the input vectors

2.4.2 Parameters Description

Call(s):

```
int ARR1D_ADDSC_UL(unsigned long* arr, int size, unsigned long scal)
```

```
int ARR1D_ADDSC_SL(signed long* arr, int size, signed long scal)
```

Parameters:

Table 2-4. ARR1D_ADDSC Parameters

arr	in/out	Pointer to the vector
size	in	Number of elements in vector
scal	in	Scalar value

Returns: The ARR1D_ADDSC macro generates unsigned/signed output values, which are stored in the array pointed to by the parameter *arr*.

2.4.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE; i++)
    arr_c[i] += scalar;
```

Optimization can be done using the following techniques:

1. Loop unrolling by four.
2. Every four values of array arr used in each iteration are stored using postincrement addressing mode while performing additons.
3. If the number of elements is not divisible by 4, the tail elements are processed in regular order.

Optimized code:

```
        move.l d1,d2
        asr.l #2,d1
        beq l1
    l2:
        add.l d0,(a0)+
        add.l d0,(a0)+
        add.l d0,(a0)+
        add.l d0,(a0)+
        subq.l #1,d1
        bne l2
    l1:
        and.l #3,d2
        beq l4
    l3:
        add.l d0,(a0)+
```

```
        subq.l #1,d2
        bne l3
    l4:
```

2.4.4 Differences Between the ARR1D_ADDSC_UL and the ARR1D_ADDSC_SL Macros

The type of ARR1D_ADDSC_UL parameters (*arr, scale) is *unsigned long*.

The type of ARR1D_ADDSC_SL parameters (*arr, scale) is *signed long*.

2.5 ARR1D_PROD_UL, ARR1D_PROD_SL

2.5.1 Macros Description

These macros compute the product of the vector array of unsigned/signed values. The product is computed by the formula:

$$res = \bigcap_{i=0}^{i=size-1} x_i$$
$$x_i \in X;$$

where *res* – result value, *X* – input vector, x_i – element of the X vector, *size* – number of elements in the input vectors

2.5.2 Parameters Description

Call(s):

```
int ARR1D_PROD_UL(unsigned long *arr, int size)

int ARR1D_PROD_SL(signed long *arr, int size)
```

Parameters:

Table 2-5. ARR1D_PROD Parameters

arr	in/out	Pointer to the vector
size	in	Number of elements in vector

Returns: The ARR1D_PROD macro generates an unsigned/signed output value, which is returned by the macro.

2.5.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE; i++)
    res_c *= arr1[i];
```

Optimization can be done using the following techniques:

1. Loop unrolling by four.
2. Every four values of array *arr* used in each iteration are loaded using postincrement addressing mode while performing multiplications.
3. If the number of elements is not divisible by 4, the tail elements are processed in regular order.

Optimized code:

```
        move.l size,d1
        move.l d1,d2
        moveq.l #1,d0
        asr.l #2,d1
        beq out1
    loop1:
        mulu.l (a0)+,d0
        mulu.l (a0)+,d0
        mulu.l (a0)+,d0
        mulu.l (a0)+,d0
        subq.l #1,d1
        bne loop1
    out1:
        and.l #3,d2
        beq out2
    loop2:
```

```
    mulu.l (a0)+,d0
    subq.l #1,d2
    bne loop2
out2:
```

2.5.4 Differences Between the ARR1D_PROD_UL and the ARR1D_PROD_SL Macros

The type of ARR1D_PROD_UL parameters (*arr) is *unsigned long.*

The type of ARR1D_PROD_SL parameters (*arr) is *signed long*.

ARR1D_PROD_UL uses the *mulu* instruction for multiplication.

ARR1D_PROD_SL uses the *muls* instruction for multiplication to keep the signs of operands.

2.6 ARR1D_MUL2_SL, ARR1D_MUL2_UL

2.6.1 Macros Description

These macros perform multiplication of two vector arrays of unsigned/signed values.

2.6.2 Parameters Description

Call(s):

```
int ARR1D_MUL2_UL(unsigned long* dest,unsigned long* src,long size)
int ARR1D_MUL2_SL(long* dest,long* src,long size)
```

Parameters:

Table 2-6. ARR1D_MUL2 Parameters

dest	in	Pointer to the destination vector
src	in	Pointer to the source vector
size	in	Number of elements in vectors

Returns: The ARR1D_MUL2 macro generates an unsigned/signed output vector, which is the result of dest and src multiplication, and is pointed to by dest.

2.6.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE; i++)
    arr_c[i] *= arr1[i];
```

Optimization for MAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
3. The first four values are loaded using one movem instruction.

Optimized code (uses MAC unit):

```
        lea -60(a7),a7
        movem.l d2-d7/a2-a5,(a7)
        move.l #0,d0
        move.l d0,MACSR
        moveq.l #16,d0
        move.l dest,a0
        move.l src,a1
        move.l size,d1
        move.l d1,d2
        asr.l #2,d1
        beq out1
        move.l #0,ACC0
        movem.l (a1),d7/a3-a5
        add.l d0,a1
    loop1:
        movem.l (a0),d3-d6
        macl.l d7,d3,(a1)+,d7,ACC0
        move.l ACC0,d3
        move.l #0,ACC0
        macl.l a3,d4,(a1)+,a3,ACC0
        move.l ACC0,d4
```

```
        move.l #0,ACC0
        macl.l a4,d5,(a1)+,a4,ACC0
        move.l ACC0,d5
        move.l #0,ACC0
        macl.l a5,d6,(a1)+,a5,ACC0
        move.l ACC0,d6
        move.l #0,ACC0
        movem.l d3-d6,(a0)
        add.l d0,a0
        subq.l #1,d1
        bne loop1
    out1:
        and.l #3,d2
        beq out2
        sub.l d0,a1
    loop2:
        move.l (a0),d3
        muls.l (a1)+,d3
        move.l d3,(a0)+
        subq.l #1,d2
        bne loop2
    out2:
        movem.l (a7),d2-d7/a2-a5
        lea 60(a7),a7
```

Optimization for eMAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using four accumulators for pipelining.
3. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
4. The first four values are loaded using one movem instruction.

Optimized code (uses eMAC unit):

```
    lea -60(a7),a7
        movem.l d2-d7/a2-a5,(a7)
        moveq.l #16,d0
        move.l dest,a0
```

```
        move.l src,a1
        move.l size,d1
        move.l d1,d2
        asr.l #2,d1
    beq out1
        move.l #0,ACC0
        move.l #0,ACC1
        move.l #0,ACC2
        move.l #0,ACC3
        movem.l (a1),d7/a3-a5
        add.l d0,a1
    loop1:
        movem.l (a0),d3-d6
        macl.l d7,d3,(a1)+,d7,ACC0
        macl.l a3,d4,(a1)+,a3,ACC1
        macl.l a4,d5,(a1)+,a4,ACC2
        macl.l a5,d6,(a1)+,a5,ACC3
        movclr.l ACC0,d3
        movclr.l ACC1,d4
        movclr.l ACC2,d5
        movclr.l ACC3,d6
        movem.l d3-d6,(a0)
        add.l d0,a0
        subq.l #1,d1
        bne loop1
    out1:
        and.l #3,d2
        beq out2
        sub.l d0,a1
    loop2:
        move.l (a0),d3
        muls.l (a1)+,d3
        move.l d3,(a0)+
        subq.l #1,d2
        bne loop2
    out2:
        movem.l (a7),d2-d7/a2-a5
        lea 60(a7),a7
```

2.6.4 Differences Between ARR1D_MUL2_UL and ARR1D_MUL2_SL

The ARR1D_MUL2_UL macro uses the unsigned mode of the MAC unit, while ARR1D_MUL2_SL macro uses signed mode.

2.7 ARR1D_MUL3_SL, ARR1D_MUL3_UL

2.7.1 Macros Description

The ARR1D_MUL2_UL macro uses the unsigned mode of the MAC unit, while ARR1D_MUL2_SL macro uses signed mode.

2.7.2 Parameters Description

Call(s):

```
int ARR1D_MUL3_UL(unsigned long *dest, unsigned long *src, unsigned long
*src2, int size)

int ARR1D_MUL3_SL(long *dest, long *src1, long *src2, int size)
```

Parameters:

Table 2-7. ARR1D_MUL3 Parameters

dest	in	Pointer to the destination vector
src1	in	Pointer to the source1 vector
src2	in	Pointer to the source2 vector
size	in	Number of elements in vectors

Returns: The ARR1D_MUL3 macro generates an unsigned/signed output vector, which is the result of the src1 and src2 multiplication, and is pointed to by dest.

2.7.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE; i++)
```

```
arr_c[i] = arr1[i] * arr2[i];
```

Optimization for MAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
3. First four values are loaded using one movem instruction.

Optimized code (uses MAC unit):

```
            lea -60(a7),a7
    movem.l d2-d7/a2-a5,(a7)
    move.l #0x40,d0
    move.l d0,MACSR
    moveq.l #16,d0
    move.l dest,a0
    move.l src1,a1
    move.l src2,a2
    move.l size,d1
    move.l d1,d2
    asr.l #2,d1
    beq out1
    move.l #0,ACC0
    movem.l (a1),d7/a3-a5
    add.l d0,a1
loop1:
    movem.l (a2),d3-d6
    macl.l d7,d3,(a1)+,d7,ACC0
    move.l ACC0,d3
    move.l #0,ACC0
    macl.l a3,d4,(a1)+,a3,ACC0
    move.l ACC0,d4
    move.l #0,ACC0
    macl.l a4,d5,(a1)+,a4,ACC0
    move.l ACC0,d5
    move.l #0,ACC0
    macl.l a5,d6,(a1)+,a5,ACC0
    move.l ACC0,d6
    move.l #0,ACC0
    movem.l d3-d6,(a0)
```

```
    add.l d0,a2
    add.l d0,a0
    subq.l #1,d1
    bne loop1
out1:
    and.l #3,d2
    beq out2
    sub.l d0,a1
loop2:
    move.l (a2)+,d3
    mulu.l (a1)+,d3
    move.l d3,(a0)+
    subq.l #1,d2
    bne loop2
out2:
    movem.l (a7),d2-d7/a2-a5
    lea 60(a7),a7
```

Optimization for eMAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using 4 accumulators for pipelining.
3. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
4. The first four values are loaded using one movem instruction.

Optimized code (uses eMAC unit):

```
    lea -60(a7),a7
    movem.l d2-d7/a2-a5,(a7)
    moveq.l #16,d0
    move.l dest,a0
    move.l src1,a1
    move.l src2,a2
    move.l size,d1
    move.l d1,d2
    asr.l #2,d1
    beq out1
    move.l #0,ACC0
    move.l #0,ACC1
```

```
    move.l #0,ACC2
    move.l #0,ACC3
    movem.l (a1),d7/a3-a5
    add.l d0,a1
loop1:
    movem.l (a2),d3-d6
    macl.l d7,d3,(a1)+,d7,ACC0
    macl.l a3,d4,(a1)+,a3,ACC1
    macl.l a4,d5,(a1)+,a4,ACC2
    macl.l a5,d6,(a1)+,a5,ACC3
    movclr.l ACC0,d3
    movclr.l ACC1,d4
    movclr.l ACC2,d5
    movclr.l ACC3,d6
    movem.l d3-d6,(a0)
    add.l d0,a2
    add.l d0,a0
    subq.l #1,d1
    bne loop1
out1:
    and.l #3,d2
    beq out2
    sub.l d0,a1
loop2:
    move.l (a2)+,d3
    mulu.l (a1)+,d3
    move.l d3,(a0)+
    subq.l #1,d2
    bne loop2
out2:
    movem.l (a7),d2-d7/a2-a5
    lea 60(a7),a7
```

2.7.4 Differences Between ARR1D_MUL3_UL and ARR1D_MUL3_SL

The ARR1D_MUL3_UL macro uses unsigned mode of the MAC unit, while the ARR1D_MUL3_SL macro uses signed mode.

2.8 ARR1D_MULSC_SL, ARR1D_MULSC_UL

2.8.1 Macros Description

These macros perform multiplication of one vector array by scalar unsigned/signed value.

2.8.2 Parameters Description

Call(s):

```
int ARR1D_MULSC_UL (long* arr,long size,unsigned long scal)
```

```
int ARR1D_MULSC_SL (long* arr,long size, long scal)
```

Parameters:

Table 2-8. ARR1D_MULSC Parameters

arr	in	Pointer to the destination vector
size	in	Number of elements in vectors
scal	in	Scalar value

Returns: The ARR1D_MULSC macro generates an unsigned/signed output vector, which is the result of the arr multiplication by scal, and is pointed to by arr.

2.8.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE; i++)
        arr_c[i] *= scalar;
```

Optimization for MAC unit can be done using the following techniques:

1. Loop unrolling by four.

2. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.

3. The first four values are loaded using one movem instruction.

Optimized code (uses MAC unit):

```
lea -60(a7),a7
    movem.l d2-d6/a2-a5,(a7)
    move.l #0,d0
    move.l d0,MACSR
    move.l arr,a0
    move.l scal,d0
    move.l size,d1
    move.l d1,d2
    asr.l #2,d1
    beq out1
    move.l #0,ACC0
    moveq.l #16,d7
loop1:
    movem.l (a0),d3-d6
    mac.l d0,d3,ACC0
    move.l ACC0,d3
    move.l #0,ACC0
    mac.l d0,d4,ACC0
    move.l ACC0,d4
    move.l #0,ACC0
    mac.l d0,d5,ACC0
    move.l ACC0,d5
    move.l #0,ACC0
    mac.l d0,d6,ACC0
    move.l ACC0,d6
    move.l #0,ACC0
    movem.l d3-d6,(a0)
    add.l d7,a0
    subq.l #1,d1
    bne loop1
out1:
    and.l #3,d2
    beq out2
loop2:
    move.l (a0),d3
    muls.l d0,d3
    move.l d3,(a0)+
```

```
    subq.l #1,d2
    bne  loop2
out2:
    movem.l (a7),d2-d6/a2-a5
    lea 60(a7),a7
```

Optimization for eMAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using 4 accumulators for pipelining.
3. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
4. The first four values are loaded using one movem instruction.
5.

Optimized code (uses eMAC unit):

```
    lea -60(a7),a7
    movem.l d2-d6/a2-a5,(a7)
    move.l arr,a0
    move.l scal,d0
    move.l size,d1
    move.l d1,d2
    asr.l #2,d1
    beq out1
    move.l #0,ACC0
    move.l #0,ACC1
    move.l #0,ACC2
    move.l #0,ACC3
    moveq.l #16,d7
loop1:
    movem.l (a0),d3-d6
    mac.l d0,d3,ACC0
    mac.l d0,d4,ACC1
    mac.l d0,d5,ACC2
    mac.l d0,d6,ACC3
    movclr.l ACC0,d3
    movclr.l ACC1,d4
    movclr.l ACC2,d5
    movclr.l ACC3,d6
```

```
        movem.l d3-d6,(a0)
        add.l d7,a0
        subq.l #1,d1
        bne loop1
    out1:
        and.l #3,d2
        beq out2
    loop2:
        move.l (a0),d3
        muls.l d0,d3
        move.l d3,(a0)+
        subq.l #1,d2
        bne loop2
    out2:
        movem.l (a7),d2-d6/a2-
```

2.8.4 Differences Between ARR1D_MULSC_UL and ARR1D_MULSC_SL

The ARR1D_MULSC_UL macro uses unsigned mode of the MAC unit, while the ARR1D_MULSC_SL macro uses signed mode.

2.9 ARR1D_MAX_S, ARR1D_MAX_U

2.9.1 Macros Description

Macro search for a maximum element in 1D array of signed or unsigned integer values.

2.9.2 Parameters Description

Call(s):

```
ARR1D_MAX_S(signed long *src, int size)

ARR1D_MAX_U(unsigned long *src, int size)
```

The elements are held in array src[]. The src[] array is searched for a maximum from 0 to size-1. Prior to any call of ARR1D_MAX_S and ARR1D_MAX_U macros, the user must allocate memory for src[] array either in static or in dynamic memory. The types of the array and the invoking macro must correspond.

Parameters:

Table 2-9. ARR1D_MAX_S, ARR1D_MAX_U Parameters

src	In	Pointer to the input array.
size	In	Number of elements in the input array.

Returns: The ARR1D_MAX_S and ARR1D_MAX_U macros return the maximum element's index as their result, which is why they can be used in an assignment operation.

2.9.3 Description of Optimization

These macros do not use any multiplication operations. Therefore, it is not suitable to use MAC and eMAC instructions for optimization of these macros. This is why instructions from the Integer Instruction Set were used for optimization. For signed and unsigned values, appropriate comparison insructions were used. All optimization issues are the same for both macros.

The following optimization techniques were used:

1. Multiple load/store operations for accessing array elements
2. Loop unrolling by four
3. Descending loop organization

Particular techniques of optimization are reviewed below.

C code:

```
for(i = 0; i <= SIZE; i++)
           {
            if (arr_c[i]>max)
                {   max = arr_c[i];
                    index = i;
                }
        }
```

Optimized code :

```
l2: ; taken from ARR1D_MAX_S macro
    movem.l (a0),d1-d4   ;multiple load operations to access
    cmp.l   d1,d5        ;source array elements
bge     c1               ;making comparisons beetwen four
    move.l  d1,d5        ;elements because of loop unrolling
```

```
            addq.l  #1,d6
            move.l  d6,a3       ;index is accumulated in d6
        bra c2
        c1:
            addq.l  #1,d6
        c2:
            cmp.l   d2,d5
        bge   c3
            move.l  d2,d5
            addq.l  #1,d6
            move.l  d6,a3
            bra c4
        c3:
            addq.l  #1,d6
        c4:
        cmp.l   d3,d5
        bge   c5
            move.l  d3,d5
            addq.l  #1,d6
            move.l  d6,a3
        bra c6
        c5:
            addq.l  #1,d6
        c6:
        cmp.l   d4,d5
        bge c7
            move.l  d4,d5
            addq.l  #1,d6
            move.l  d6,a3
        bra c8
        c7:
             addq.l  #1,d6
        c8:
        add.l   #16,a0
            subq.l  #1,d0   ;descending loop organization
    bne l2
        l1:
```

2.9.4 Differences Btween ARR1D_MAX_U and ARR1D_MAX_S

For signed and unsigned values, appropriate comparison insructions were used.

2.10 ARR1D_MIN_S, ARR1D_MIN_U

2.10.1 Macros Description

Macros search for a minimum element in 1D array of signed or unsigned integer values.

2.10.2 Parameters Description

Call(s):

```
ARR1D_MIN_S(signed long *src, int size)
```

```
ARR1D_MIN_U(unsigned long *src, int size)
```

The elements are held in array src[]. The src[] array is searched for minimum from 0 to size-1. Prior to any call of ARR1D_MIN_S and ARR1D_MIN_U macros, the user must allocate memory for src[] array either in static or in dynamic memory. The types of the array and the invoking macro must correspond.

Parameters:

Table 2-10. ARR1D_MIN_S, ARR1D_MIN_U Parameters

src	in	Pointer to the input array.
size	in	Number of elements in the input array.

Returns: The ARR1D_MIN_S and ARR1D_MIN_U macros return the minimum element's index as their result, which is why they can be used in an assignment operation.

2.10.3 Description of Optimization

These macros do not use any multiplication operations. Therefore, it is not suitable to use MAC and eMAC instructions to optimize these macros. This is why instructions from the Integer Instruction Set were used for optimization. For signed and unsigned values, appropriate comparison insructions were used. All optimization issues are the same for both macros.

The following optimization techniques were used:

1. Multiple load/store operations to access to array's elements
2. Loop unrolling by four
3. Decsending loop organization

Particular techniques of optimization are reviewed below.

C code:

```
for(i = 0; i <= SIZE; i++)
           {
            if (arr_c[i]<min)
                {   min = arr_c[i];
                    index = i;
                }
        }
```

Optimized code :

```
l2: ;taken from ARR1D_MIN_U macro
    movem.l (a0),d1-d4  ; multiple load operations to access
    cmp.l   d1,d5       ; source array elements
bls     c1
    move.l  d1,d5   ;making comparisons beetwen four
    addq.l  #1,d6   ;elements because of loop unrolling
    move.l  d6,a3
bra c2
c1:
    addq.l  #1,d6   ;index is accumulated in d6
c2:
    cmp.l   d2,d5
bls   c3
    move.l  d2,d5
    addq.l  #1,d6
    move.l  d6,a3
bra c4
c3:
    addq.l  #1,d6
c4:
cmp.l   d3,d5
bls   c5
    move.l  d3,d5
    addq.l  #1,d6
    move.l  d6,a3
bra c6
 c5:
    addq.l  #1,d6
 c6:
cmp.l   d4,d5
```

```
                              bls c7
                                  move.l  d4,d5
                                  addq.l  #1,d6
                                  move.l  d6,a3
                              bra c8
                               c7:
                                  addq.l  #1,d6
                               c8:
                              add.l   #16,a0
                                  subq.l  #1,d0   ;descending loop organization
                              bne l2
```

2.10.4 Differences Between ARR1D_MIN_U and ARR1D_MIN_S

For signed and unsigned values, appropriate comparison insructions were used.

2.11 ARR1D_CAST_SWL, ARR1D_CAST_UWL

2.11.1 Macros Description

These macros convert an array of word data elements to an array of long data elements. ARR1D_CAST_SWL is used for signed values, and ARR1D_CAST_UWL for unsigned values. The Library of Macros only supports long data element arrays, so these macros need to be used when a programmer wants to use the library with word data element arrays. After these macros complete their conversion, any macro from this library can be used for word data.

2.11.2 Parameters Description

Call(s):

```
ARR1D_CAST_SWL(signed short *src, signed long *dest, int size)

ARR1D_CAST_UWL(unsigned short *src, unsigned long *dest, int size)
```

The original elements are held in array src[], and the converted elements are stored in array dest[]. Both arrays run from 0 to size-1. Prior to any call of ARR1D_CAST_SWL or ARR1D_CAST_UWL, the user must allocate memory for both src[] and dest[] arrays, either in static or dynamic memory.

Parameters:

Table 2-11. ARR1D_CAST_SWL, ARR1D_CAST_UWL Parameters

dest	out	Pointer to the output array of *size* of signed or unsigned long data elements, depending on the type of a macro.
src	In	Pointer to the input array of of *size* of signed or unsigned long data elements, depending on the type of a macro.
size	in	Number of elements in input and output arrays

Returns: The ARR1D_CAST_SWL and ARR1D_CAST_UWL macros generate output values, which are stored in the array pointed to by *dest*.

2.11.3 Description of Optimization

These macros do not use any multiplication operations. Therefore, it is not suitable to use MAC and eMAC instructions to optimize these macros. This is why instructions from the Integer Instruction Set were used for optimization.

The following optimization techniques were used:

1. Multiple load/store operations to access array elements
2. Loop unrolling by four
3. Decsending loop organization

Particular techniques of optimization are reviewed below.

C code:

```
for(i = 0; i < SIZE; i++)
    arr_c[i] = (long)arr1[i];
```

Optimized code :

```
l2: ;taken from ARR1D_CAST_SWL
    movem.l (a0),d2/d4  ; multiple load operations to access
    move.l  d2,d3      ; source array elements
    move.l  d4,d5      ;convertion performed by four elements
    swap.w  d2         ;because of loop unrolling
    swap.w  d4
    ext.l   d2
```

```
                ext.l   d3  ; in ARR1D_CAST_UWL andi.l   #0xffff,d2
                ext.l   d4  ; instruction was used
                ext.l   d5
                movem.l d2-d5,(a1)  ;multiple store operation
                addq.l   #8,a0
                add.l    #16,a1
                subq.l   #1,d0  ;descending loop organization
            bne    l2
```

2.11.4 Differences Between ARR1D_CAST_SWL and ARR1D_CAST_UWL

ARR1D_CAST_SWL is used for signed values, and ARR1D_CAST_UWL is used for unsigned values. For ARR1D_CAST_SWL, ext.l instruction is used, and for ARR1D_CAST_UWL, andi.l instruction is used.

Chapter 3 Macros for 2D Array Operations

3.1 ARR2D_SUM_UL, ARR2D_SUM_SL

3.1.1 Macros Description

These macros compute the sum of the array elements of unsigned/signed values. This sum is computed by the formula:

$$res = \sum_{i=0}^{size1-1} \sum_{j=0}^{size2-1} x_{ij}$$

where x_{ij}, – element of the input array, *size1* – number of rows of input array, *size1* – number of columns in the input array

3.1.2 Parameters Description

Call(s):

```
int ARR2D_SUM_UL(unsigned long *src, int size1, int size2)

int ARR2D_SUM_SL(signed long* src, int size1, size2)
```

Parameters:

Table 3-1. ARR2D_SUM Parameters

src	in	Pointer to the source vector
size1	in	Number of raws in array
size2	In	Number of colomn in array

Returns: The ARR2D_SUM macros return the unsigned/signed sum of the array elements.

3.1.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE1; i++)
    for(j = 0; j < SIZE2; j++)
        res += arr1[i][j];
```

Optimization can be done using the following techniques:

1. The elements are accessed as 1d-array elements with number of elements: *size1*size2*, because elements of 2d-array are located in memory sequentally.
2. Loop unrolling by four.
3. Postincrement addressing mode to access input array elements.
4. Descending loop organization.

The following should be noticed:

- The d0 register always holds the sum of array elements.
- The a0 register holds the pointer to input array.

- The d1 register is the counter.

Optimized code:

```
loop1:
    add.l (a0)+,d0
    add.l (a0)+,d0
    add.l (a0)+,d0
    add.l (a0)+,d0
    subq.l #1,d1
bne loop1
```

3.1.4 Differences Between the ARR2D_SUM_UL and the ARR2D_SUM_SL Macros

The type of ARR2D_SUM_UL parameters (*src) is *unsigned long*.

The type of ARR2D_SUM_SL parameters (*src) is *signed long*.

3.2 ARR2D_ADD2_UL, ARR2D_ADD2_SL

3.2.1 Macros Description

These macros compute the elementwise sum of two 2d-arrays of unsigned/signed values. The elementwise sum is computed by the formula:

$$x_{i,j} = x_{i,j} + y_{i,j}$$
$$x_{i,j} \in X, y_{i,j} \in Y, i \in [0, size1 - 1], j \in [0, size2 - 1];$$

where X, Y – input arrays, $x_{i,j}$, $y_{i,j}$ – elements of the corresponding arrays, *size1* – number of rows, *size2* – number of columns

Note:

The type of elements of arrays in the ARR2D_ADD2_UL macro must be unsigned long, and the type of elements of arrays in the ARR2D_ADD2_SL macro must be signed long.

3.2.2 Parameters Description

Call(s):

```
int ARR2D_ADD2_UL(void* dest, void* src, int size1, int size2)

int ARR2D_ADD2_SL(void* dest, void* src, int size1, int size2)
```

Parameters:

Table 3-2. ARR2D_ADD2 Parameters

dest	in/out	Pointer to the destinstion array
src	in	Pointer to the source array
size1	in	Number of rows of matrices
size2	in	Number of columns of matrices

Returns: The ARR2D_ADD2 macro generates unsigned/signed output values, which are stored in the array pointed to by the parameter *dest*.

3.2.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE1; i++)
    for(j = 0; j < SIZE2; j++)
        arr_c[i][j] += arr1[i][j];
```

Optimization can be done using the following techniques:

1. The elements are accessed as 1d-array elements with number of elements: *size1*size2*, because elements of 2d-array are located in memory sequentially.
2. Loop unrolling by four.
3. Every four values of array dest used in each iteration are loaded with only one movem instruction.
4. Every four values of array src used in each iteration are loaded using postincrement addressing mode while performing additons.

5. After perfoming additions, the resulting four values in each iteration are stored with only one movem instruction.

6. If the number of elements is not divisible by four, the tail elements are processed in regular order.

Optimized code:

```
        move.l size1,d1
        move.l size2,d2
        mulu.l d2,d1
        move.l d1,d2
        asr.l #2,d1
        beq l1
    l2:
        movem.l (a0),d3-d6
        add.l (a1)+,d3
        add.l (a1)+,d4
        add.l (a1)+,d5
        add.l (a1)+,d6
        movem.l d3-d6,(a0)
        add.l #16,a0
        subq.l #1,d1
        bne l2
    l1:
        and.l #3,d2
        beq l4
    l3:
        move.l (a0),d3
        add.l (a1)+,d3
        move.l d3,(a0)+
        subq.l #1,d2
        bne l3
    l4:
        add.l (a1)+,d3
        move.l d3,(a0)+
        subq.l #1,d2
        bne l3
    l4:
```

3.2.4 Differences Between the ARR2D_ADD2_UL and the ARR2D_ADD2_SL Macros

There are no differences. The macro was written in two versions to preserve library uniformity.

3.3 ARR2D_ADD3_UL, ARR2D_ADD3_SL

3.3.1 Macros Description

These macros compute the elementwise sum of two 2d-arrays of unsigned/signed values, and store the results in a third 2d-array of unsigned/signed values. The elementwise sum is computed by the formula:

$$z_{i,j} = x_{i,j} + y_{i,j}$$
$$x_{i,j} \in X, y_{i,j} \in Y, z_{i,j} \in Z, i \in [0, size1 - 1], j \in [0, size2 - 1];$$

where *X, Y* – input arrays, $x_{i,j}$, $y_{i,j}$ – elements of the corresponding arrays, *Z* – resultant vestor, $z_{i,j}$ – element of vector Z, *size1* – number of rows, *size2* – number of columns

Note:

The type of elements of arrays in the ARR2D_ADD3_UL macro must be unsigned long, and the type of elements of arrays in the ARR2D_ADD3_SL macro must be signed long.

3.3.2 Parameters Description

Call(s):

```
int ARR2D_ADD3_UL(void* dest, void* src1, void* src2, int size1, int size2);

int ARR2D_ADD3_SL(void* dest, void* src1, void* src2, int size1, int size2);
```

Parameters:

Table 3-3. ARR2D_ADD3 Parameters

dest	in/out	Pointer to the destinstion array
src1	in	Pointer to the source array1
src2	in	Pointer to the source array2
size1	in	Number of rows of matrices
size2	in	Number of columns of matrices

Returns: The ARR2D_ADD3 macro generates unsigned/signed output values, which are stored in the array pointed to by the parameter *dest*.

3.3.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE1; i++)
    for(j = 0; j < SIZE2; j++)
        arr_c[i][j] = arr1[i][j] + arr2[i][j];
```

Optimization can be done using the following techniques:

1. The elements are accessed as 1d-array elements with number of elements: *size1*size2*, because elements of 2d-array are located in memory sequentially.
2. Loop unrolling by four.
3. Every four values of array *src1* used in each iteration are loaded with only one movem instruction.
4. Every four values of array *src2* used in each iteration are loaded using postincrement addressing mode while performing additons.
5. After perfoming additions, the resulting four values in each iteration are stored into the *dest* array with only one movem instruction;
6. If the number of elements is not divisible by four, the tail elements are processed in regular order.

Optimized code:

```
        move.l size1,d1
        move.l size2,d2
        mulu.l d2,d1
        move.l d1,d2
        asr.l #2,d1
        beq l1
    l2:
        movem.l (a1),d3-d6
        add.l (a2)+,d3
        add.l (a2)+,d4
        add.l (a2)+,d5
        add.l (a2)+,d6
        movem.l d3-d6,(a0)
```

```
    add.l #16,a0
    add.l #16,a1
    subq.l #1,d1
    bne l2
l1:
    and.l #3,d2
    beq l4
l3:
    move.l (a1)+,d3
    add.l (a2)+,d3
    move.l d3,(a0)+
    subq.l #1,d2
    bne l3
l4:
```

3.3.4 Differences Between the ARR2D_ADD3_UL and the ARR2D_ADD3_SL Macros

There are no differences. The macro was written in two versions in order to preserve library uniformity.

3.4 ARR2D_ADDSC_UL, ARR2D_ADDSC_SL

3.4.1 Macros Description

These macros compute the elementwise sum of 2d-array of unsigned/signed values with a scalar unsigned/signed value. The elementwise sum is computed by the formula:

$$x_{i,j} = x_{i,j} + scalar$$
$$x_{i,j} \in X, i \in [0, size1 - 1], j \in [size2 - 1];$$

where X – input array, $x_{i,j}$ – element of the array X, *scalar* – variable with unsigned/signed value, *size1* – number of rows, *size2* – number of columns

Note:

The type of elements of array in the ARR2D_ADDSC_UL macro must be unsigned long, and the type of elements of array in the ARR2D_ADDSC_SL macro must be signed long.

3.4.2 Parameters Description

Call(s):

```
int ARR2D_ADDSC_UL(void* arr, int size1, int size2, unsigned long scal);
```

```
int ARR2D_ADDSC_SL(void* arr, int size1, int size2, signed long scal)
```

Parameters:

Table 3-4. ARR2D_ADDSC Parameters

arr	in/out	Pointer to the array
size1	in	Number of rows of matrix
size2	in	Number of columns of matrix
scal	in	Scalar value

Returns: The ARR2D_ADDSC macro generates unsigned/signed output values, which are stored in the array pointed to by the parameter *arr*.

3.4.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE1; i++)
    for(j = 0; j < SIZE2; j++)
        arr_c[i][j] += scalar;
```

Optimization can be done using the following techniques:

1. The elements are accessed as 1d-array elements with number of elements: *size1*size2*, because elements of 2d-array are located in memory sequentially.
2. Loop unrolling by four.
3. Every four values of array *arr* used in each iteration are stored using postincrement addressing mode while performing additons.
4. If the number of elements is not divisible by four, the tail elements are processed in regular order.

Optimized code:

```
        move.l size1,d1
        move.l size2,d2
        mulu.l d2,d1
        move.l d1,d2
        asr.l #2,d1
        beq l1
l2:
        add.l d0,(a0)+
        add.l d0,(a0)+
        add.l d0,(a0)+
        add.l d0,(a0)+
        subq.l #1,d1
        bne l2
l1:
        and.l #3,d2
        beq l4
l3:
        add.l d0,(a0)+
        subq.l #1,d2
        bne l3
l4:
```

3.4.4 Differences Between the ARR2D_ADDSC_UL and the ARR2D_ADDSC_SL Macros

There are no differences. The macro was written in two versions in order to preserve library uniformity.

3.5 ARR2D_PROD_UL, ARR2D_PROD_SL

3.5.1 Macros Description

These macros compute the product of 2d-array with unsigned/signed values. The product is computed by the formula:

$$res = \bigcap_{i,j=0}^{i=size1-1;\, j=size2-1} x_{i,j}$$

$$x_{i,j} \in X;$$

where *res* – result value, *X* – input array, $x_{i,j}$ – element of array *X*, *size1* – number of rows, *size2* – number of columns

Notes:

The type of elements of array in the ARR2D_PROD_UL macro must be unsigned long, and the type of elements of array in the ARR2D_PROD_SL macro must be signed long.

3.5.2 Parameters Description

Call(s):

```
int ARR2D_PROD_SL(void *arr, int size1, int size2);
```

```
int ARR2D_PROD_UL(void *arr, int size1, int size2);
```

Parameters:

Table 3-5. ARR2D_PROD Parameters

arr	in/out	Pointer to the array
size1	in	Number of rows of matrix
size2	in	Number of columns of matrix

Returns: The ARR2D_PROD macro generates an unsigned/signed output value, which is returned by macro.

3.5.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE1; i++)
    for(j = 0; j < SIZE2; j++)
        prod_c *= arr1[i][j];
```

Optimization can be done using the following techniques:

1. The elements are accessed as 1d-array elements with number of elements: *size1*size2*, because elements of 2d-array are located in memory sequentially.
2. Loop unrolling by four.
3. Every four values of array *arr* used in each iteration are loaded using post increment addressing mode while performing multiplications.

4. If the number of elements is not divisible by four, the tail elements are processed in regular order.

Optimized code:

```
        move.l size1,d1
        move.l size2,d2
        mulu.l d2,d1
        move.l d1,d2
        moveq.l #1,d0
        asr.l #2,d1
        beq out1
    loop1:
        muls.l (a0)+,d0
        muls.l (a0)+,d0
        muls.l (a0)+,d0
        muls.l (a0)+,d0
        subq.l #1,d1
        bne loop1
    out1:
        and.l #3,d2
        beq out2
    loop2:
        muls.l (a0)+,d0
        subq.l #1,d2

    bne loop2
    out2:
```

3.5.4 Differences Between the ARR2D_PROD_UL and the ARR2D_PROD_SL Macros

ARR2D_PROD_UL uses instruction *mulu* for multiplication.

ARR2D_PROD_SL uses instruction *muls* for multiplication to keep the signs of operands.

3.6 ARR2D_MUL2_SL, ARR2D_MUL2_UL

3.6.1 Macros Description

These macros perform multiplication of two 2D arrays of unsigned/signed values.

3.6.2 Parameters Description

Call(s):

```
int ARR2D_MUL2_UL(unsigned long* dest,unsigned long* src,long size1, ,long
size2)

int ARR2D_MUL2_SL(long* dest,long* src,long size1,long size2)
```

Parameters:

Table 3-6. ARR2D_MUL2 Parameters

dest	in	Pointer to the destination array
src	in	Pointer to the source array
size1	in	Number of rows in arrays
size2	in	Number of columns in arrays

Returns: The ARR2D_MUL2 macro generates an unsigned/signed output matrix, which is the result of dest and src multiplication, and is pointed to by dest.

3.6.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE1; i++)
    for(j = 0; j < SIZE2; j++)
        arr_c[i][j] *= arr1[i][j];
```

Optimization for MAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
3. The first four values are loaded using one movem instruction.

Optimized code (uses MAC unit):

```
    lea -60(a7),a7
        movem.l d2-d7/a2-a5,(a7)
```

```
    move.l #0,d0
    move.l d0,MACSR
    moveq.l #16,d0
    move.l dest,a0
    move.l src,a1
    move.l size1,d1
    move.l size2,d2
    mulu.l d2,d1
    move.l d1,d2
    asr.l #2,d1
    beq out1
    move.l #0,ACC0
    movem.l (a1),d7/a3-a5
    add.l d0,a1
loop1:
    movem.l (a0),d3-d6
    macl.l d7,d3,(a1)+,d7,ACC0
    move.l ACC0,d3
    move.l #0,ACC0
    macl.l a3,d4,(a1)+,a3,ACC0
    move.l ACC0,d4
    move.l #0,ACC0
    macl.l a4,d5,(a1)+,a4,ACC0
    move.l ACC0,d5
    move.l #0,ACC0
    macl.l a5,d6,(a1)+,a5,ACC0
    move.l ACC0,d6
    move.l #0,ACC0
    movem.l d3-d6,(a0)
    add.l d0,a0
    subq.l #1,d1
    bne loop1
out1:
    and.l #3,d2
    beq out2
    sub.l d0,a1
loop2:
    move.l (a0),d3
    muls.l (a1)+,d3
    move.l d3,(a0)+
    subq.l #1,d2
    bne loop2
```

```
out2:
    movem.l (a7),d2-d7/a2-a5
    lea 60(a7),a7
```

Optimization for eMAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using four accumulators for pipelining.
3. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
4. First four values are loaded using one movem instruction.

Optimized code (uses eMAC unit):

```
lea -60(a7),a7
    movem.l d2-d7/a2-a5,(a7)
    moveq.l #16,d0
    move.l dest,a0
    move.l src,a1
    move.l size1,d1
    move.l size2,d2
    mulu.l d2,d1
    move.l d1,d2
    asr.l #2,d1
    beq out1
    move.l #0,ACC0
    move.l #0,ACC1
    move.l #0,ACC2
    move.l #0,ACC3
    movem.l (a1),d7/a3-a5
    add.l d0,a1
loop1:
    movem.l (a0),d3-d6
    macl.l d7,d3,(a1)+,d7,ACC0
    macl.l a3,d4,(a1)+,a3,ACC1
    macl.l a4,d5,(a1)+,a4,ACC2
    macl.l a5,d6,(a1)+,a5,ACC3
    movclr.l ACC0,d3
    movclr.l ACC1,d4
    movclr.l ACC2,d5
```

```
        movclr.l ACC3,d6
        movem.l d3-d6,(a0)
        add.l d0,a0
        subq.l #1,d1
        bne loop1
    out1:
        and.l #3,d2
        beq out2
        sub.l d0,a1
    loop2:
        move.l (a0),d3
        muls.l (a1)+,d3
        move.l d3,(a0)+
        subq.l #1,d2
        bne loop2
    out2:
        movem.l (a7),d2-d7/a2-a5
        lea 60(a7),a7
```

3.6.4 Differences Between ARR2D_MUL2_UL and ARR2D_MUL2_SL

ARR2D_MUL2_UL macro uses unsigned mode of the MAC unit, while ARR2D_MUL2_SL macro uses signed mode.

3.7 ARR2D_MUL3_SL, ARR2D_MUL3_UL

3.7.1 Macros Description

These macros perform multiplication of two 2D arrays of unsigned/signed values.

3.7.2 Parameters Description

Call(s):

```
int ARR2D_MUL3_UL(unsigned long *dest, unsigned long *src1, unsigned long
*src2, int size1, int size2)

int ARR2D_MUL3_SL(long *dest, long *src1, long *src2, int size, int size2)
```

Parameters:

Table 3-7. ARR2D_MUL3 Parameters

dest	in	Pointer to the destination array
src1	in	Pointer to the source1 array
src2	in	Pointer to the source2 array
size1	In	Number of rows in arrays
size2	In	Number of columns in arrays

Returns: The ARR2D_MUL3 macro generates an unsigned/signed output matrix, which is the result of src1 and src2 multiplication, and is pointed to by dest.

3.7.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE1; i++)
    for(j = 0; j < SIZE2; j++)
        arr_c[i][j] = arr1[i][j] * arr2[i][j];
```

Optimization for MAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
3. The first four values are loaded using one movem instruction.

Optimized code (uses MAC unit):

```
        lea -60(a7),a7
        movem.l d2-d7/a2-a5,(a7)
        move.l #0x40,d0
        move.l d0,MACSR
        moveq.l #16,d0
        move.l dest,a0
        move.l src1,a1
        move.l src2,a2
        move.l size1,d1
```

```
        move.l size2,d2
        mulu.l d2,d1
        move.l d1,d2
        asr.l #2,d1
        beq out1
        move.l #0,ACC0
        movem.l (a1),d7/a3-a5
        add.l d0,a1
    loop1:
        movem.l (a2),d3-d6
        macl.l d7,d3,(a1)+,d7,ACC0
        move.l ACC0,d3
        move.l #0,ACC0
        macl.l a3,d4,(a1)+,a3,ACC0
        move.l ACC0,d4
        move.l #0,ACC0
        macl.l a4,d5,(a1)+,a4,ACC0
        move.l ACC0,d5
        move.l #0,ACC0
        macl.l a5,d6,(a1)+,a5,ACC0
        move.l ACC0,d6
        move.l #0,ACC0
        movem.l d3-d6,(a0)
        add.l d0,a2
        add.l d0,a0
        subq.l #1,d1
        bne loop1
    out1:
        and.l #3,d2
        beq out2
        sub.l d0,a1
    loop2:
        move.l (a2)+,d3
        mulu.l (a1)+,d3
        move.l d3,(a0)+
        subq.l #1,d2
        bne loop2
    out2:
        movem.l (a7),d2-d7/a2-a5
        lea 60(a7),a7
```

Optimization for eMAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using four accumulators for pipelining.
3. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
4. The first four values are loaded using one movem instruction.

Optimized code (uses eMAC unit):

```
    lea -60(a7),a7
    movem.l d2-d7/a2-a5,(a7)
    moveq.l #16,d0
    move.l dest,a0
    move.l src1,a1
    move.l src2,a2
    move.l size1,d1
    move.l size2,d2
    mulu.l d2,d1
    move.l d1,d2
    asr.l #2,d1
    beq out1
    move.l #0,ACC0
    move.l #0,ACC1
    move.l #0,ACC2
    move.l #0,ACC3
    movem.l (a1),d7/a3-a5
    add.l d0,a1
loop1:
    movem.l (a2),d3-d6
    macl.l d7,d3,(a1)+,d7,ACC0
    macl.l a3,d4,(a1)+,a3,ACC1
    macl.l a4,d5,(a1)+,a4,ACC2
    macl.l a5,d6,(a1)+,a5,ACC3
    movclr.l ACC0,d3
    movclr.l ACC1,d4
    movclr.l ACC2,d5
    movclr.l ACC3,d6
    movem.l d3-d6,(a0)
    add.l d0,a2
    add.l d0,a0
    subq.l #1,d1
    bne loop1
```

```
out1:
    and.l #3,d2
    beq out2
    sub.l d0,a1
loop2:
    move.l (a2)+,d3
    mulu.l (a1)+,d3
    move.l d3,(a0)+
    subq.l #1,d2
    bne loop2
out2:
    movem.l (a7),d2-d7/a2-a5
    lea 60(a7),a7
```

3.7.4 Differences Between ARR2D_MUL3_UL and ARR2D_MUL3_SL

ARR2D_MUL3_UL macro uses unsigned mode of the MAC unit, while ARR2D_MUL3_SL macro uses signed mode.

3.8 ARR2D_MULSC_SL, ARR2D_MULSC_UL

3.8.1 Macros Description

These macros perform multiplication of one 2D array by scalar unsigned/signed value.

3.8.2 Parameters Description

Call(s):

```
int ARR2D_MULSC_UL (long* arr, long size1,long size2, unsigned long scal)

int ARR2D_MULSC_SL (long* arr, long size1,long size2, long scal)
```

Parameters:

Table 3-8. ARR2D_MULSC Parameters

Arr	in	Pointer to the destination array

size1	in	Number of rows in arrays
Size2	in	Number of columns in arrays
scal	in	Scalar value

Returns: The ARR2D_MULSC macro generates an unsigned/signed output matrix, which is the result of arr multiplication by scal and is pointed to by arr.

3.8.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE1; i++)
    for(j = 0; j < SIZE2; j++)
        arr_c[i][j] *= scalar;
```

Optimization for MAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
3. The first four values are loaded using one movem instruction.

Optimized code (uses MAC unit):

```
        lea -60(a7),a7
        movem.l d2-d6/a2-a5,(a7)
        move.l #0,d0
        move.l d0,MACSR
        move.l arr,a0
        move.l scal,d0
        move.l size1,d1
        move.l size2,d2
        mulu.l d2,d1
        move.l d1,d2
        asr.l #2,d1
        beq out1
        move.l #0,ACC0
        moveq.l #16,d7
    loop1:
        movem.l (a0),d3-d6
```

```
    mac.l d0,d3,ACC0
    move.l ACC0,d3
    move.l #0,ACC0
    mac.l d0,d4,ACC0
    move.l ACC0,d4
    move.l #0,ACC0
    mac.l d0,d5,ACC0
    move.l ACC0,d5
    move.l #0,ACC0
    mac.l d0,d6,ACC0
    move.l ACC0,d6
    move.l #0,ACC0
    movem.l d3-d6,(a0)
    add.l d7,a0
    subq.l #1,d1
    bne loop1
out1:
    and.l #3,d2
    beq out2
loop2:
    move.l (a0),d3
    muls.l d0,d3
    move.l d3,(a0)+
    subq.l #1,d2
    bne loop2
out2:
    movem.l (a7),d2-d6/a2-a5
    lea 60(a7),a7
```

Optimization for eMAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using four accumulators for pipelining.
3. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
4. The first four values are loaded using one movem instruction.

Optimized code (uses eMAC unit):

```
    lea -60(a7),a7
    movem.l d2-d6/a2-a5,(a7)
    move.l arr,a0
    move.l scal,d0
    move.l size1,d1
    move.l size2,d2
    mulu.l d2,d1
    move.l d1,d2
    asr.l #2,d1
    beq out1
    move.l #0,ACC0
    move.l #0,ACC1
    move.l #0,ACC2
    move.l #0,ACC3
    moveq.l #16,d7
loop1:
    movem.l (a0),d3-d6
    mac.l d0,d3,ACC0
    mac.l d0,d4,ACC1
    mac.l d0,d5,ACC2
    mac.l d0,d6,ACC3
    movclr.l ACC0,d3
    movclr.l ACC1,d4
    movclr.l ACC2,d5
    movclr.l ACC3,d6
    movem.l d3-d6,(a0)
    add.l d7,a0
    subq.l #1,d1
    bne loop1
out1:
    and.l #3,d2
    beq out2
loop2:
    move.l (a0),d3
    muls.l d0,d3
    move.l d3,(a0)+
    subq.l #1,d2
    bne loop2
out2:
    movem.l (a7),d2-d6/a2-a5
    lea 60(a7),a7
```

3.8.4 Differences Between ARR2D_MULSC_UL and ARR2D_MULSC_SL

ARR2D_MULSC_UL macro uses unsigned mode of the MAC unit, while ARR2D_MULSC_SL macro uses signed mode.

3.9 ARR2D_MAX_S, ARR2D_MAX_U

3.9.1 Macros Description

These macros search for a maximum element in a 2D array of signed or unsigned integer values.

3.9.2 Parameters Description

Call(s):

```
ARR2D_MAX_S(void *src, int size1, int size2)

ARR2D_MAX_U(void *src, int size1, int size2)
```

The elements are held in src[] array. The src[] array is searched for maximum from 0 to size-1, where $size = size1 \times size2$. Prior to any call of ARR2D_MAX_S and ARR2D_MAX_U macros, the user must allocate memory for src[] array either in static or in dynamic memory. Types of the array and the invoking macro must correspond. In declaration, src[] array is declared as void for compatibility.

Parameters:

Table 3-9. ARR2D_MAX_S, ARR2D_MAX_U Parameters

src	In	Pointer to the input array.
size1	In	Number of rows
size2	In	Number of columns

Returns: The ARR2D_MAX_S and ARR2D_MAX_U macros return maximum element's index as their result, which is why they can be used in an assignment operation. The index is linear and must be converted to two indices to access C array. The convertion can be done in the following way: $index1 = [index/size2]$; $index2 = index - index1 \times size2$, where *index1* – first C index (row), *index2* – second C index (column), *index* – linear index, *size1* – number of rows, *size2* – number of columns.

3.9.3 Description of Optimization

These macros do not use any multiplication operations. Therefore, it is not suitable to use MAC and eMAC instructions to optimize these macros. This is why instructions from the Integer Instruction Set were used for optimization. For signed and unsigned values, appropriate comparison insructions were used. All optimization issues are the same for both macros.

The following optimization techniques were used:

1. Multiple load/store operations to access array elements.
2. Loop unrolling by four.
3. Descending loop organization.
4. Particular techniques of optimization are reviewed below.

C code:

```
for(i = 0; i <= SIZE1; i++)
    for(j = 0; j <= SIZE2; j++)
      {
           if (arr_c[i][j]>max)
               {   max = arr_c[i][j];
                   i1 = i;
                   i2 = j;
               }
       }
```

Optimized code :

```
;this code is similar to 1D array macro but in preloop operations linear size must be
;calculated and stored
                            l2: ; taken from ARR2D_MAX_S macro
                                movem.l (a0),d1-d4   ;multiple load operations to access
                                cmp.l   d1,d5        ;source array elements
                            bge     c1               ;making comparisons beetwen four
                                move.l  d1,d5        ;elements because of loop unrolling
                                addq.l  #1,d6
                                move.l  d6,a3        ;index is accumulated in d6
                            bra c2
                            c1:
```

```
            addq.l  #1,d6
        c2:
            cmp.l   d2,d5
        bge   c3
            move.l  d2,d5
            addq.l  #1,d6
            move.l  d6,a3
            bra c4
        c3:
            addq.l  #1,d6
        c4:
        cmp.l   d3,d5
        bge   c5
            move.l  d3,d5
            addq.l  #1,d6
            move.l  d6,a3
        bra c6
        c5:
            addq.l  #1,d6
        c6:
        cmp.l   d4,d5
        bge c7
            move.l  d4,d5
            addq.l  #1,d6
            move.l  d6,a3
        bra c8
        c7:
             addq.l  #1,d6
        c8:
        add.l   #16,a0
            subq.l  #1,d0    ;descending loop organization
    bne l2
        l1:
```

3.9.4 Differences Between ARR2D_MAX_U and ARR2D_MAX_S

For signed and unsigned values, appropriate comparison insructions were used.

3.10 ARR2D_MIN_S, ARR2D_MIN_U

3.10.1 Macros Description

The macros search for a minimum element in 2D array of signed or unsigned integer numbers.

3.10.2 Parameters Description

Call(s):

```
ARR2D_MIN_S(void *src, int size1, int size2)

ARR2D_MIN_U(void *src, int size1, int size2)
```

The elements are held in src[] array. The src[] array is searched for maximum from 0 to size-1, where $size = size1 \times size2$. Prior to any call of ARR2D_MIN_S and ARR2D_MIN_U user must allocate memory for src[] array either in static or in dynamic memory. Types of the array and the invoking macro must correspond. In declaration, src[] array is declared as void for compatibility.

Parameters:

Table 3-10. ARR2D_MIN_S, ARR2D_MIN_U Parameters

src	in	Pointer to the input array.
size1	in	Number of rows
size2	in	Number of columns

Returns: ARR2D_MIN_S and ARR2D_MIN_U macros return minimum element's index as their result, which is why they can be used in an assignment operation. The index is linear and must be converted to two indices to access C array. The convertion can be done in the following way: $index1 = [index/size2]$; $index2 = index - index1 \times size2$, where *index1* – first C index (row), *index2* – second C index (column), *index* – linear index, *size1* – number of rows, *size2* – number of columns.

3.10.3 Description of Optimization

These macros does not use any multiply operations. So, it is not suitable to use MAC and eMAC instructions to optimize these macros. This is why instructions from the Integer Instruction Set were used for optimization. For signed and unsigned values, appropriate comparison insructions were used. All optimization issues are the same for both macros.

The following optimization techniques were used:

1. Multiple load/store operations to access array elements.

2. Loop unrolling by four.

3. Decsending loop organization.

Particular techniques of optimization are reviewed below.

C code:

```
for(i = 0; i <= SIZE1; i++)
    for(j = 0; j <= SIZE2; j++)
       {
            if (arr_c[i][j]<min)
                {   min = arr_c[i][j];
                    i1 = i;
                    i2 = j;
                }
        }
```

Optimized code :

```
    ;this code is similar to 1D array macro but in preloop operations linear size ;must be
calculated and stored
                            l2: ;taken from ARR2D_MIN_U macro
                                movem.l (a0),d1-d4  ; multiple load operations to access
                                cmp.l   d1,d5       ; source array elements
                            bls     c1
                                move.l  d1,d5   ;making comparisons beetwen four
                                addq.l  #1,d6   ;elements because of loop unrolling
                                move.l  d6,a3
                            bra c2
                            c1:
                                addq.l  #1,d6   ;index is accumulated in d6
                            c2:
                                cmp.l   d2,d5
                            bls   c3
                                move.l  d2,d5
                                addq.l  #1,d6
                                move.l  d6,a3
                            bra c4
```

```
c3:
    addq.l  #1,d6
c4:
cmp.l   d3,d5
bls   c5
    move.l  d3,d5
    addq.l  #1,d6
    move.l  d6,a3
bra c6
 c5:
    addq.l  #1,d6
 c6:
cmp.l   d4,d5
bls c7
    move.l  d4,d5
    addq.l  #1,d6
    move.l  d6,a3
bra c8
 c7:
    addq.l  #1,d6
 c8:
add.l   #16,a0
    subq.l  #1,d0   ;decsending loop organization
bne l2
```

3.10.4 Differences Between ARR2D_MIN_U and ARR2D_MIN_S

For signed and unsigned values, appropriate comparison insructions were used.

3.11 ARR2D_CAST_SWL, ARR2D_CAST_UWL

3.11.1 Macros Description

These macros convert arrays of word data elements to arrays of long data elements. ARR2D_CAST_SWL is used for signed values, and ARR2D_CAST_UWL for unsigned values. This library of macros only supports arrays of long data elements, so these macros should be used when the programmer needs to use this library with arrays of word data elements. After convertion with these macros, any macro from this library can be used for word.

3.11.2 Parameters Description

Call(s):

```
ARR2D_CAST_SWL(void *src,void *dest, int size1, int size2)

ARR2D_CAST_UWL(void *src,void *dest, int size1, int size2)
```

The original elements are held in src[] array, and the converted elements are stored in array dest[]. Both arrays run from 0 to size-1. Prior to any call of ARR1D_CAST_SWL or ARR1D_CAST_UWL, the user must allocate memory for both src[] and dest[] arrays either in static or dynamic memory. Type void in declaration of these macros is used only for compatibility, so the macro must be called with array of appropriate type.

Parameters:

Table 3-11. ARR2D_CAST_SWL, ARR2D_CAST_UWL Parameters

dest	out	Pointer to the output array of *size* void data elements, but array must have appropriate type depending on the type of a macro.
src	In	Pointer to the input array of *size* signed or unsigned long data elements, but array must have appropriate type depending on the type of a macro.
size1	in	Number of columns
Size2	in	Number of rows

Returns: The ARR2D_CAST_SWL and ARR2D_CAST_UWL macros generate output values, which are stored in the array pointed to by *dest*.

3.11.3 Description of Optimization

These macros do not use any multiplication operations. So it is not suitable to use MAC and eMAC instructions to optimize these macros. This is why instructions from the Integer Instruction Set were used for optimization.

The following optimization techniques were used:

1. Multiple load/store operations to access array elements.
2. Loop unrolling by four.
3. Descending loop organization.

Particular techniques of optimization are reviewed below.

C code:

```
    for(i = 0; i < SIZE1; i++) {
        for(j = 0; j < SIZE2; j++) {
            arr_c[i][j] = (long)arr1[i][j];
        }
    }
```

Optimized code :

```
;this code is similar to 1D array macro but in preloop operations linear size
;must be calculated and stored
                            l2: ;taken from ARR1D_CAST_SWL
                                movem.l (a0),d2/d4  ; multiple load operations to access
                                move.l  d2,d3   ; source array elements
                                move.l  d4,d5   ;convertion performed by four elements
                                swap.w  d2      ;because of loop unrolling
                                swap.w  d4
                                ext.l   d2
                                ext.l   d3  ; in ARR1D_CAST_UWL andi.l   #0xffff,d2
                                ext.l   d4  ; instruction was used
                                ext.l   d5
                                movem.l d2-d5,(a1)  ;multiple stor operation
                                addq.l   #8,a0
                                add.l    #16,a1
                                subq.l   #1,d0  ;decsending loop organization
                            bne    l2
```

3.11.4 Differences Between the ARR1D_SUM_UL and the ARR1D_SUM_SL Macros

ARR2D_CAST_SWL is used for signed values, and ARR2D_CAST_UWL is used for unsigned values. For ARR1D_CAST_SWL, ext.l instruction is used, and for ARR1D_CAST_UWL, andi.l instruction.

Chapter 4
Macros for DSP Algorithms

4.1 DOT_PROD_UL, DOT_PROD_SL

4.1.1 Macros Description

These macros compute the dot product of two vector arrays with unsigned/signed values. The dot product is computed by the following formula:

$$X \cdot Y = \sum_{i=1}^{n} x_i y_i ,$$

where X, Y – input vectors, x_i, y_i – elements of the corresponding vectors, n – size of the vectors

4.1.2 Parameters Description

Call(s):

```
unsigned long DOT_PROD_UL(unsigned long *arr1, unsigned long *arr2, int size)
```

```
signed long DOT_PROD_SL(signed long *arr1, signed long *arr2, int size)
```

Parameters:

Table 4-1. DOT_PROD Parameters

arr1	in	Pointer to the first vector
arr2	in	Pointer to the second vector
size	in	Number of elements in vectors

Returns: The DOT_PROD macro generates an unsigned/signed output value, which is returned by macro.

4.1.3 Description of Optimization

C code:

```
for(i = 0; i < SIZE; i++)
```

```
res_c += arr1[i] * arr2[i];
```

Optimization for MAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
3. The first four values are loaded using one movem instruction.

Optimized code (uses MAC unit):

```
        movem.l (a0), d1-d4
        lea 16(a0),a0
        subq.l  #1, d0
        beq L2
    L3:
        movem.l (a1), a2-a5
        lea 16(a1),a1
        macl.l  d1, a2, (a0)+, d1, ACC0
        macl.l  d2, a3, (a0)+, d2, ACC0
        macl.l  d3, a4, (a0)+, d3, ACC0
        macl.l  d4, a5, (a0)+, d4, ACC0
        subq.l  #1, d0
        bne L3
    L2:
```

There is no need for optimization of the eMAC unit, because there is only one multiply-accumulate sequence in the computations.

4.1.4 Differences Between DOT_PROD_UL and DOT_PROD_SL

DOT_PROD_UL macro uses the unsigned mode of the MAC unit, while DOT_PROD_SL macro uses signed mode.

4.2 RDOT_PROD_UL, RDOT_PROD_SL

4.2.1 Macros Description

These macros compute the reverse dot product of two vector arrays with unsigned/signed values. The reverse dot product is computed by the following formula:

$$X \cdot Y = \sum_{i=1}^{n} x_i y_{n-i+1},$$

where X, Y – input vectors, x_i, y_i – elements of the corresponding vectors, n – size of the vectors.

4.2.2 Parameters Description

Call(s):

```
unsigned long RDOT_PROD_UL(unsigned long *arr1, unsigned long *arr2, int size)
```

```
signed long RDOT_PROD_SL(signed long *arr1, signed long *arr2, int size)
```

Parameters:

Table 4-2. RDOT_PROD Parameters

arr1	in	Pointer to the first vector
arr2	in	Pointer to the second vector
size	in	Number of elements in vectors

Returns: The RDOT_PROD macro generates an unsigned/signed output value, which is returned by macro.

4.2.3 Description of Optimization

Particular techniques of optimization are reviewed below.

C code:

```
for(i = 0; i < SIZE; i++)
   res_c += arr1[i] * arr2[SIZE - i - 1];
```

Optimization for MAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
3. The first four values are loaded using one movem instruction.

Optimized code (uses MAC unit):

```
        lea -16(a0), a0
        movem.l (a0), d1-d4
        subq.l  #1, d0
        beq L2
L3:
        movem.l (a1), a2-a5
        lea 16(a1),a1
        macl.l  d4, a2, -(a0), d4, ACC0
        macl.l  d3, a3, -(a0), d3, ACC0
        macl.l  d2, a4, -(a0), d2, ACC0
        macl.l  d1, a5, -(a0), d1, ACC0
        subq.l  #1, d0
        bne L3
L2:
```

There is no need for optimization of the eMAC unit, because there is only one multiply-accumulate sequence in computations.

4.2.4 Differences Between RDOT_PROD_UL and RDOT_PROD_SL

RDOT_PROD_UL macro uses unsigned mode of the MAC unit, while RDOT_PROD_SL macro uses signed mode.

4.3 MATR_MUL_UL, MATR_MUL_SL

4.3.1 Macros Description

These macros compute the product of two matrices with unsigned/signed values. Matrix multiplication is computed by the following formula:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} \ ,$$

where $c_{i,j}$ is an element of resultant matrix C, $a_{i,k,}$ and $b_{k,j}$ are elements of the input matrices A and B respectively.

4.3.2 Parameters Description

Call(s):

```
void MATR_MUL_UL(void *arrr, void *arr1, void *arr2, int m, int n, int p)
```

```
void MATR_MUL_SL(void *arrr, void *arr1, void *arr2, int m, int n, int p)
```

Parameters:

Table 4-3. MATR_MUL Parameters

arrr	out	Pointer to the resulting matrix (size must be m*p)
arr1	in	Pointer to the first matrix (size must be m*n)
arr1	in	Pointer to the second matrix (size must be n*p)
m	in	Number of raws in the first matrix
n	in	Number of columns in first matrix (number of raws in second matrix)
p	in	Number of columns in second matrix

Returns: The MATR_MUL macro generates an output matrix with unsigned/signed values, which is pointed to by arrr.

4.3.3 Description of Optimization

C code:

```
for(i = 0; i < MSIZE; i++)
    for(j = 0; j < PSIZE; j++)
        for(k = 0; k < NSIZE; k++)
            arr_c[i][j] += arr1[i][k] * arr2[k][j];
```

Optimization for MAC unit: performing multiplication and addition at the same time due to mac instruction.

Optimized code (uses MAC unit):

```
lea    (a0), a1
```

```
            lea (a2), a3
            lea 4(a2), a2
            move.l  n, d2
IN3:
            move.l  (a3), d4
            add.l   d3, a3
            move.l  (a1)+, a4
            mac.l   d4, a4, ACC0
            subq.l  #1, d2
            bne IN3
```

Optimization for MAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
3. Postincremental addressing mode is used for sequential access to matix elements.

Optimized code (uses eMAC unit):

```
OUT1:
            lea (a0), a1
            lea (a2), a3
            lea 16(a2), a2
            move.l  n, d2
IN2:
            movem.l (a3), d4-d7
            add.l   d3, a3

            move.l  (a1)+, a4
            mac.l   d4, a4, ACC0
            mac.l   d5, a4, ACC1
            mac.l   d6, a4, ACC2
            mac.l   d7, a4, ACC3
            subq.l  #1, d2
            bne IN2

            movclr.lACC0, d4
            movclr.lACC1, d5
            movclr.lACC2, d6
            movclr.lACC3, d7
```

```
        movem.l     d4-d7, (a5)
        lea 16(a5), a5

        subq.l  #1, d1
        bne OUT1
```

4.3.4 Differences Between MATR_MUL_UL and MATR_MUL_SL

MATR_MUL_UL macro uses unsigned mode of the MAC unit, while MATR_MUL_SL macro uses signed mode.

4.4 CONV

4.4.1 Macro Description

This macro computes convolution using array of samples and array of coefficients. Convolution is computed by the following formula:

$$y[i] = \sum_{j=0}^{M-1} h[j]x[i-j] ,$$

where *y[i]* is an output sample, *x[i-j]* is an input sample, and *h[j]* is coefficient

There are two algorithms of convolution computing the following:

- Input side algorithm
- Output side algorithm

The macro uses output side algorithm for implementation using MAC unit, because it is more suitable.

To learn more about convolution and its properties, refer to The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/).

Notes:

- Array elements must be of the FRAC32 type.

- The size of the output array must equal the sum of sizes of the input array and array of coefficients.

4.4.2 Parameters Description

Call(s):

```
void CONV(void *y, void *x, void *h, int xsize, int hsize)
```

Parameters:

Table 4-4. CONV Parameters

y	out	Pointer to the output vector, containing computed values
x	in	Pointer to the input vector (array of samples)
h	in	Pointer to the array of coefficients
xsize	in	Size of the input vector
hsize	in	Size of array of coefficients

Returns: The CONV macro generates output samples which are pointed to by y.

4.4.3 Description of Optimization

C code:

```
for(i = 0; i < XSIZE + HSIZE - 1; i++) {
    for(j = 0; j < HSIZE; j++) {
        if((i - j >= 0) && (i - j < XSIZE)) {
            arr_d[i] += xarr_d[i - j] * harr_d[j];
        }
    }
```

Optimization for MAC unit: performing multiplication and addition at the same time due to mac instruction.

Optimized code (uses MAC unit):

```
_OUT1:
            addq.l  #4, d6
            move.l  d6, d2
            movea.l a0, a1
            movea.l a2, a3
            add.l   d2, a3
_IN1:
            move.l  (a1)+, d4
            move.l  -(a3), d5
            mac.w   d4.u, d5.u, <<, acc0
            subq.l  #4, d2
            bne _IN1
            move.l  acc0, d7
            move.l  #0, acc0
            move.l  d7, (a4)+
            subq.l  #1, d1
            bne _OUT1
```

Optimization for eMAC unit can be done using the following techniques:

1. Loop unrolling by four.
2. Reduction of the number of instructions for fetching operand from memory (one element can be used in computation of several output elements).
3. Using macl instruction, which allows multiplying simultaneously with loading four values for the next iteration.
4. Using movclr instruction instead of two instructions to store value in memory and clear the accumulator.
5. Sequential mac operations allow use of eMAC unit pipeline efficiently.

Optimized code (uses eMAC unit):

```
_OUT2:
            movea.l a1, a0
            movea.l a3, a5
            move.l  d0, d5
_IN2:
            move.l  -(a0), d3
            move.l  (a5)+, d4
            macl.l  d3, d4, (a5)+, d4, acc0
            macl.l  d3, d4, (a5)+, d4, acc1
```

```
macl.l  d3, d4, (a5)+, d4, acc2
mac.l   d3, d4, acc3
lea -12(a5), a5
subq.l  #1, d5
bne _IN2

movclr.lacc0, d5
move.l  d5, (a4)+
movclr.lacc1, d5
move.l  d5, (a4)+
movclr.lacc2, d5
move.l  d5, (a4)+
movclr.lacc3, d5
move.l  d5, (a4)+
lea 16(a3), a3
subq.l  #1, d1
bne _OUT2
```

4.5 FIRST_DIFF

4.5.1 Macro Description

This macro peforms a calculation of the *first differences* on input fractional operands, commonly known as discrete derivation. More details on this linear system's characteristic may be found in The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/).

4.5.2 Parameters Description

Call(s):

`FIRST_DIFF(FRAC32* dst, FRAC32* src, long size)`

The original signals are held in array src[], and the first differences are stored in array dst[] . Both arrays run from 0 to size-1. Prior to any call of FIRST_DIFF, the user must allocate memory for both src[] and dst[] arrays, either in static or in dynamic memory.

Parameters:

Table 4-5. FIRST_DIFF Parameters

dst	out	Pointer to the output array of *size* FRAC32 data elements
src	In	Pointer to the input array of of *size* FRAC32 data elements
size	in	Number of elements in input and output arrays

Returns: The FIRST_DIFF macro generates output values, which are stored in the array pointed to by *dst*.

4.5.3 Description of Optimization

This macro does not use any multiplication operations. So it is not suitable to use MAC and eMAC instructions to optimize this macro. Thus, instructions from the Integer Instruction Set were used for optimization.

The following optimization techniques were used:

1. Multiple load/store operations to access arrays elements.
2. Loop unrolling by four.
3. Descending loop organization.

Discussions on particular techniques of optimization is shown below.

C code:

```
for(i = 1; i < SIZE; i++)
    arr_c[i] = arr1d[i] - arr1d[i-1];
```

The following should be noticed:

- The loop is unrolled by four.
- The input operands are fetched from memory in fours and stored in registers d4, d5, d6, d7, a2, a3, a4, and a5.
- The d0 register contains the previously computed value.
- Results are stored in registers a2, a3, a4, and a5.
- The a0 register holds the pointer to output array; the a1 register holds the pointer to input array.

Optimized code :

```
loop1:
        movem.l (a1),d4-d7      ; multiple load operations to access source
        movem.l (a1),a2-a5      ; array's elements

        sub.l d0,a2         ; performing loop body that unrolled by four
        sub.l d4,a3
        sub.l d5,a4
        sub.l d6,a5

        movem.l a2-a5,(a0)      ; multiple store operation to save results

        move.l d7,d0
        add.l #16,a1
        add.l #16,a0
        subq.l #1,d1        ; decsending loop organization
        bne loop1
```

4.6 RUNN_SUM

4.6.1 Macro Description

This macro performs a calculation of the *running sum* of the input fractional operands, commonly known as *discrete integration*. More details on this linear system's characteristic may be found in The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/).

4.6.2 Parameters Description

Call(s):

```
RUNN_SUM(FRAC32* dst, FRAC32* src, long size)
```

The original signals are held in array *src[]*, and the running sum up to the *n*-th element is stored in the corresponding *n*-th element of array *dst[]*. Both arrays run from 0 to *size-1*. Prior to any call of RUNN_SUM, the user must allocate memory for both the *src[]* and *dst[]* arrays, either in static or in dynamic memory.

Parameters:

Table 4-6. RUNN_SUM Parameters

dst	Out	Pointer to the output array of *size* FRAC32 data elements
src	In	Pointer to the input array of of *size* FRAC32 data elements
size	In	Number of elements in input and output arrays

Returns: The RUNN_SUM macro generates output values that are stored in the array, pointed to by *dst*.

4.6.3 Description of Optimization

This macro does not use any multiplication operations. So it is not suitable to use MAC and eMAC instructions to optimize this macro. Thus, instructions from the Integer Instruction Set were used for optimization.

The following optimization techniques were used:

1. Multiple load operations to access array *src* elements.
2. Postincrement addressing mode to store results in array *dst*.
3. Loop unrolling by four.
4. Descending loop organization.

Particular techniques for optimization are reviewed below.

C code:

```
for(i = 1; i < SIZE; i++)
      arr_c[i] = arr_c[i-1] + arr1d[i];
```

The following should be noticed:

- The loop is unrolled by four.
- The input operands are fetched from memory in fours and stored in registers d4, d5, d6, and d7.
- The d0 register contains the latest computed value,
- The a0 register holds the pointer to the output array; register a1 holds the pointer to the input array.

Optimized code :

```
loop1:
    movem.l (a1),d4-d7      ; multiple load operations to access source
                            ; array's elements
    add.l d4,d0         ; cdomputing output value
    move.l d0,(a0)+     ; storing value on output array
    add.l d5,d0
    move.l d0,(a0)+
    add.l d6,d0
    move.l d0,(a0)+
    add.l d7,d0
    move.l d0,(a0)+
    add.l #16,a1
    subq.l #1,d1        ; decsending loop organization
bne loop1
```

4.7 LPASS_1POLE_FLTR

4.7.1 Macros Description

This macro computes a single pole low-pass filter. This recursive filter uses just two coefficients: a_0 and b_1, so the filter can be represented in the following form:

$$y_n = a_0 * x_n + b_1 * y_{n-1}$$

The filter's response characteristics are controlled by the parameter x, a value between zero and one. Physically, x is the amount of decay between adjacent samples.

$$a_0 = 1 - x$$

$$b_1 = x$$

Note: The filter becomes *unstable* if x is made greater than one. Thus, any non zero value on the input will increase the output until an overflow occurs.

More details on this digital recursive filter's characteristic may be found in The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/).

4.7.2 Parameters Description

Call(s):

```
LPASS_1POLE_FLTR(FRAC32 *dst,FRAC32 *src,long size, FRAC32 x)
```

The input signals to the filter are held in array *src[]*, and the output values are stored in array *dst[]*. Both arrays run from 0 to *size-1*. The *x* parameter controls the computation of the a_0 and b_1 filter coefficients. Prior to any call of LPASS_1POLE_FLTR, the user must allocate memory for both the *src[]* and *dst[]* arrays, in either static or dynamic memory.

Parameters:

Table 4-7. LPASS_1POLE_FLTR Parameters

dst	out	Pointer to the output array of *size* FRAC32 data elements
src	In	Pointer to the input array of of *size* FRAC32 data elements
size	in	Number of elements in input and output arrays
x	in	FRAC32 value between zero and one that controls filter coefficients computation

Returns: The LPASS_1POLE_FLTR macro generates output values, which are stored in the array, pointed to by *dst*.

4.7.3 Description of Optimization

This macro frequently performs multiplication and addition operations on fractional values. It is suitable for the eMAC unit, because the eMAC has a fractional mode.

Optimization for the MAC unit is performed as an emulation of the fractional mode, using mac.w with shift to left instruction on the upper 16 bits of operands. So only the upper 16 bits of the resulting signals are valuable.

The following optimization techniques were used:

1. Multiple load operations to access input array elements.
2. Postincrement addressing mode to store output array elements.
3. Loop unrolling by four.
4. Descending loop organization.

Particular techniques for optimization are reviewed below.

C code:

```
arr_c[i] = a0 * arr1d[i] + b1 * arr_c[i-1];
```

Optimization for the MAC unit.

The following should be noticed:

- The loop is unrolled by four.
- Coefficients a_0 and b_1 are pre-computed and held in registers *a3* and *d6* correspondingly.
- Register d0 always holds the last computed output signal.
- Input operands are fetched from memory in fours, and stored in registers d3, d4, d5, and a2.

The MAC unit has only one accumulator and all output elements must be computed sequentially, so the mac instruction pipelining is worse than in the eMAC case. Another aspect is that the MAC unit has no *movclr* instruction, so the accumulator must be cleared explicitly.

Optimized code (uses MAC unit):

```
mac.w a3.u,d3.u,<<,ACC0 ; computes a[0]*x[i] for y[i] ouput element
mac.w d6.u,d0.u,<<,ACC0 ; computes b[1] * y[i-1] to produce y[i]
move.l ACC0,d0              ; moves y[i] to d0
move.l #0,ACC0              ; clear accumulator
move.l d0,(a0)+         ; and stores y[i] to memory

mac.w a3.u,d4.u,<<,ACC0 ; computes a[0]*x[i+1] for y[i+1] ouput element
mac.w d6.u,d0.u,<<,ACC0 ; computes b[1] * y[i] to produce y[i+1]
move.l ACC0,d0              ; moves y[i+1] to d0
move.l #0,ACC0              ; clear accumulator
move.l d0,(a0)+         ; and stores y[i+1] to memory

mac.w a3.u,d5.u,<<,ACC0 ; computes a[0]*x[i+2] for y[i+2] ouput elemen
mac.w d6.u,d0.u,<<,ACC0 ; computes b[1] * y[i+1] to produce y[i+2]
move.l ACC0,d0              ; moves y[i+2] to d0
move.l #0,ACC0              ; clear accumulator
move.l d0,(a0)+         ; and stores y[i+2] to memory

mac.w a3.u,a2.u,<<,ACC0 ; computes a[0]*x[i+3] for y[i+3] ouput element
mac.w d6.u,d0.u,<<,ACC0 ; computes b[1] * y[i+2] to produce y[i+3]
move.l ACC0,d0              ; moves y[i+3] to d0
move.l #0,ACC0              ; clear accumulator
move.l d0,(a0)+         ; and stores y[i+3] to memory
```

Optimization for eMAC unit.

The following should be noticed:

- The loop is unrolled by four.
- Coefficients a_0 and b_1 are pre-computed and held in registers *a3* and *d6* correspondingly.
- The d0 register always holds the last computed output signal.
- Input operands are fetched from memory in fours and stored in registers d3, d4, d5, and a2.

The eMAC unit has four accumulators, so for better pipelining, (a_0*x_i) parts of each output element is computed for all four output elements at the beginning of loop. The rest of the output element computation is performed sequentially, because computation of each output element depends on the value of the previous element.

Optimized code (uses eMAC unit):

```
mac.l a3,d3,ACC0     ; computes a[0]*x[i] for y[i] ouput element
mac.l a3,d4,ACC1     ; computes a[0]*x[i+1] for y[i+1] ouput element
mac.l a3,d5,ACC2     ; computes a[0]*x[i+2] for y[i+2] ouput element
mac.l a3,a2,ACC3     ; computes a[0]*x[i+3] for y[i+3] ouput element

mac.l d6,d0,ACC0    ; computes b[1] * y[i-1] to produce y[i]
movclr.l ACC0,d0    ; moves y[i] to d0
move.l d0,(a0)+     ; and stores y[i] to memory

mac.l d6,d0,ACC1    ; computes b[1] * y[i] to produce y[i+1]
movclr.l ACC1,d0    ; moves y[i+1] to d0
move.l d0,(a0)+     ; and stores y[i+1] to memory

mac.l d6,d0,ACC2    ; computes b[1] * y[i+1] to produce y[i+2]
movclr.l ACC2,d0    ; moves y[i+2] to d0
move.l d0,(a0)+     ; and stores y[i+2] to memory

mac.l d6,d0,ACC3    ; computes b[1] * y[i+2] to produce y[i+3]
movclr.l ACC3,d0    ; moves y[i+3] to d0
move.l d0,(a0)+     ; and stores y[i+3] to memory
```

4.8 HPASS_1POLE_FLTR

4.8.1 Macro Description

The macro computes a single pole high-pass filter. This recursive filter uses three coefficients: a_0, a_1, and b_1, so the filter can be represented in the form:

$$y_n = a_0 * x_n + a_1 * x_{n-1} + b_1 * y_{n-1}$$

The filter's response characteristics are controlled by the parameter x, a value between zero and one. Physically, x is the amount of decay between adjacent samples.

$$a_0 = (1 + x) / 2$$

$$a_1 = -(1 + x) / 2$$

$$b_1 = x$$

Note: The filter becomes *unstable* if x is made greater than one. Thus, any non zero value on the input will increase the output until an overflow occurs.

More details on this digital recursive filter's characteristic may be found in The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/).

4.8.2 Parameters Description

Call(s):

```
HPASS_1POLE_FLTR(FRAC32 *dst,FRAC32 *src,long size, FRAC32 x)
```

The input signals to the filter are held in array *src[]*, and the output values are stored in array *dst[]*. Both arrays run from 0 to size-1. The x parameter controls the computation of the a_0, a_1, and b_1 filter coefficients. Prior to any call to HPASS_1POLE_FLTR, the user must allocate memory for both the *src[]* and *dst[]* arrays either in static or in dynamic memory.

Parameters:

Table 4-8. HPASS_1POLE_FLTR Parameters

dst	Out	Pointer to the output array of *size* FRAC32 data elements
src	In	Pointer to the input array of of *size* FRAC32 data elements

size	In	Number of elements in input and output arrays
x	In	FRAC32 value between zero and one that controls filter coefficients computation

Returns: The HPASS_1POLE_FLTR macro generates output values, which are stored in the array, pointed to by *dst*.

4.8.3 Description of Optimization

This macro frequently performs multiplication and addition operations on fractional values It is suitable for the eMAC unit, because it has a fractional mode.

Optimization for the MAC unit is performed as an emulation of the fractional mode, using mac.w with shift to left instruction on the upper 16 bits of operands. So only the upper 16 bits of the resulting signals are valuable.

The following optimization techniques were used:

1. Mac with load operations to access input array elements.
2. Post-increment addressing mode to store output array elements.
3. Loop unrolling by two.
4. Descending loop organization.

Particular techniques for optimization are reviewed below.

C code:

```
arr_c[i] = a0 * arr1d[i] + a1 * arr1d[i-1] + b1 * arr_c[i-1];
```

Optimization for the MAC unit.

The following should be noticed:

- The loop is unrolled by two.
- Coefficients a_0 and b_1 are pre-computed and held in registers *a3* and *d6* correspondingly.
- The a_1 coefficient is not computed, because $a_1 = -a_0$, so the *msac* operation is used.
- The d0 register always holds the last computed output signal.
- Input operands are fetched from memory in *msac* instructions and stored in registers d3 and d4.
- The a0 register holds the pointer to the output array; the a1 register holds the pointer to the input array.

The MAC unit has only one accumulator and all output elements must be computed sequentially, so *mac* instruction pipelining is worse than in the eMAC unit case. Another aspect is that the MAC unit has no *movclr* instruction, so the accumulator must be cleared explicitly.

Optimized code (uses MAC unit):

```
mac.w a3.u,d3.u,<<,ACC0      ; computes a[0]*x[i] for y[i] ouput element
msac.w a3.u,d4.u,<<,ACC0     ; computes a[1]*x[i-1] for y[i] ouput element
macl.w d6.u,d0.u,<<,(a1)+,d4,ACC0    ; computes b[1] * y[i-1] to produce y[i]
                                ; loads the next input operand
move.l ACC0,d0                  ; moves y[i] to d0
move.l #0,ACC0                  ; clears accumulator
move.l d0,(a0)+              ; and stores y[i] to memory

mac.w a3.u,d4.u,<<,ACC0      ; computes a[0]*x[i+1] for y[i+1] ouput element
msac.w a3.u,d3.u,<<,ACC0     ; computes a[1]*x[i] for y[i+1] ouput element
macl.w d6.u,d0.u,<<,(a1)+,d3,ACC0    ; computes b[1] * y[i] to produce y[i+1]
                                ; loads the next input operand
move.l ACC0,d0                  ; moves y[i+1] to d0
move.l #0,ACC0                  ; clears accumulator
move.l d0,(a0)+              ; and stores y[i] to memory
```

Optimization for the eMAC unit.

The following should be noticed:

- The loop is unrolled by two.
- Coefficients a_0 and b_1 are pre-computed and held in registers *a3* and *d6* correspondingly.
- The a_1 coefficient is not computed, because $a_1 = -a_0$, so thr *msac* operation is used.
- d0 register always holds the last computed output signal.
- Input operands are fetched from memory in *msac* instructions and stored in registers d3 and d4.
- The a0 register holds the pointer to the output array; the a1 register holds the pointer to the input array.

As the loop is unrolled by two, the output values are computed in two eMAC accumulators. The *movclr* instruction is used to clear the accumulators.

Optimized code (uses eMAC unit):

```
mac.l a3,d3,ACC0        ; computes a[0]*x[i] for y[i] ouput element
msac.l a3,d4,ACC0           ; computes a[1]*x[i-1] for y[i] ouput element
macl.l d6,d0,(a1)+,d4,ACC0  ; computes b[1] * y[i-1] to produce y[i]
                                ; loads the next input operand
```

```
movclr.l ACC0,d0    ; moves y[i] to d0
move.l d0,(a0)+     ; and stores y[i] to memory

mac.l a3,d4,ACC1     ; computes a[0]*x[i+1] for y[i+1] ouput element
msac.l a3,d3,ACC1        ; computes a[1]*x[i] for y[i+1] ouput element
macl.l d6,d0,(a1)+,d3,ACC1   ; computes b[1] * y[i] to produce y[i+1]
                                ; loads the next input operand
movclr.l ACC1,d0     ; moves y[i+1] to d0
move.l d0,(a0)+      ; and stores y[i+1] to memory
```

4.9 LPASS_4STG_FLTR

4.9.1 Macros Description

This macro computes a four-stage, low-pass filter. This recursive filter uses five coefficients: a_0, b_1, b_2, b_3, and b_4, so the filter can be represented in the following form:

$$y_n = a_0 * x_n + b_1 * y_{n-1} + b_2 * y_{n-2} + b_3 * y_{n-3} + b_4 * y_{n-4}$$

The filter's response characteristics are controlled by the parameter x, a value between zero and one. The four-stage, low-pass filter is comparable to the Blackman and Gaussian filters (relatives of the moving average), but with a much faster execution speed. The design equations for a four-stage, low-pass filter are the following:

$$a_0 = (1 - x)^4$$

$$b_1 = 4x$$

$$b_2 = -6x^2$$

$$b_3 = 4x^3$$

$$b_4 = -x^4$$

Note: The filter becomes *unstable* if x is made greater than one. Thus, any nonzerovalue on the input will increase the output until an overflow occurs.

More details on this digital recursive filter's characteristic may be found in The Scientist and Engineer’s Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/).

4.9.2 Parameters Description

Call(s):

```
LPASS_4STG_FLTR (FRAC32 *dst, FRAC32 *src, long size, FRAC32 x)
```

The input signals to the filter are held in array *src[]*, and the output values are stored in array *dst[]*. Both arrays run from 0 to size-1. The x parameter controls the computation of the a_0, b_1, b_2, b_3, and b_4 filter coefficients. Prior to any call of LPASS_4STG_FLTR, the user must allocate memory for both the *src[]* and *dst[]* arrays, either in static or in dynamic memory.

Parameters:

Table 4-9. LPASS_4STG_FLTR Parameters

dst	Out	Pointer to the output array of *size* FRAC32 data elements
src	In	Pointer to the input array of of *size* FRAC32 data elements
size	In	Number of elements in input and output arrays
x	In	FRAC32 value between zero and one that controls filter coefficients computation

Returns: The LPASS_4STG_FLTR macro generates output values, which are stored in the array, pointed to by *dst*.

4.9.3 Description of Optimization

This macro frequently performs multiplication and addition operations on fractional values. It is suitable for the MAC unit, because the eMAC has a fractional mode.

Optimization for the MAC unit is performed as an emulation of the fractional mode, using mac.w with shift to left instruction on the upper 16 bits of operands. So only the upper 16 bits of the resulting signals are valuable.

The following optimization techniques were used:

1. Mac with load instructions to access input array elements.
2. Post-increment addressing mode to store output array elements.
3. Loop unrolling by four.
4. Descending loop organization.

Particular techniques for optimization are reviewed below.

C code:

```
arr_c[i] = a0 * arr1d[i] + b1 * arr_c[i-1] +
        b2 * arr_c[i-2] + b3 * arr_c[i-3] + b4 * arr_c[i-4];
```

Optimization for the MAC unit.

The following should be noticed:

- The loop is unrolled by four.
- Coefficients a_0 , b_1 , b_2 , b_3 , and b_4 are pre-computed and held in registers a3 , d6, d7, a4, and a5 correspondingly.
- The a2 register always holds the output sample per each iteration.
- Input operands are fetched from memory one by one and stored in registers d5, d4, d3, and d0.

All add-multiply instructions are performed by the MAC unit. The MAC unit has no *movclr* instruction, so the accumulator must be cleared explicitly. After each computation of an output sample, the data from the accumulator is stored in the register, and the accumulator is cleared explicitly. After, the result is stored into memory.

Optimized code (uses MAC unit):

```
mac.w a3.u,a2.u,<<,ACC0      ; computes a[0]*x[i] for y[i] ouput element
macl.w d6.u,d0.u,<<,(a1)+,a2,ACC0   ; computes b[1]*y[i-1] for y[i+1] ouput
                                ; element   and loads the next input operand
msac.w d7.u,d3.u,<<,ACC0     ; computes b[2]*y[i-2] for y[i] ouput element
mac.w a4.u,d4.u,<<,ACC0      ; computes b[3]*y[i-3] for y[i] ouput element
msac.w a5.u,d5.u,<<,ACC0     ; computes b[4]*y[i-4] to produce y[i]
move.l ACC0,d5                  ; moves y[i] to d5
move.l #0,ACC0                  ; clear accumulator
move.l d5,(a0)+              ; and stores y[i] to memory
```

Optimization for eMAC unit.

The following should be noticed:

- The loop is unrolled by four.
- Coefficients a_0 , b_1 , b_2 , b_3 , and b_4 are pre-computed and held in registers a3 , d6, d7, a4, and a5 correspondingly.
- The a2 register always holds the input sample per each iteration.
- Input operands are fetched from memory one by one and stored in registers d5, d4, d3, and d0.

All add-multiply instructions are performed by the eMAC unit. After each computation of an output sample, the *movclr* instruction is used to clear the accumulator and store the result into the general purpose register. After, the result is stored into memory.

Optimized code (uses eMAC unit):

```
mac.l a3,a2,ACC0             ; computes a[0]*x[i] for y[i] ouput element
macl.l d6,d0,(a1)+,a2,ACC0   ; computes b[1]*y[i-1] for y[i+1] ouput element
                                 ; loads the next input operand
msac.l d7,d3,ACC0                ; computes b[2]*y[i-2] for y[i] ouput element
mac.l a4,d4,ACC0             ; computes b[3]*y[i-3] for y[i] ouput element
msac.l a5,d5,ACC0                ; computes b[4]*y[i-4] to produce y[i]
movclr.l ACC0,d5             ; moves y[i] to d5
move.l d5,(a0)+              ; and stores y[i] to memory
```

4.10 BANDPASS_FLTR

4.10.1 Macro Description

This macro computes a band-pass filter. This recursive filter uses five coefficients: a_0, a_1, a_2, b_1, and b_2. The filter can be represented in the following form:

$$y_n = a_0 * x_n + a_1 * x_{n-1} + a_2 * x_{n-2} + b_1 * y_{n-1} + b_2 * y_{n-2}$$

The filter's response characteristics are controlled by the parameter *f*, a value of center frequency, and *BW*, the bandwidth. Both parameters values must be in the range 0 to 0.5. The design equations for a bandpath filter are the following:

$$a_0 = 1 - K$$

$$a_1 = 2(K-R)\cos(2\pi f)$$

$$a_2 = R^2 - K$$

$$b_1 = 2R\cos(2\pi f)$$

$$b_2 = -R^2$$

where:

$$K = \frac{1 - 2R\cos(2\pi f) + R^2}{2 - 2\cos(2\pi f)}$$

$$R = 1 - 3BW$$

More details on this digital recursive filter's characteristic may be found in The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/).

4.10.2 Parameters Description

Call(s):

```
BANDPASS_FLTR(FRAC32 *dst, FRAC32 *src, long size, FRAC32 freq, FRAC32 bandw)
```

The input signals to the filter are held in array *src[]*, and the output values are stored in array *dst[]*. Both arrays run from 0 to size-1. The freq and bandw parameters control the computation of the a_0, a_1, a_2, b_1, and b_2 filter coefficients. Prior to any call of BANDPASS_FLTR, the user must allocate memory for both the *src[]* and *dst[]* arrays, in either static or dynamic memory.

Parameters:

Table 4-10. BANDPASS_FLTR Parameters

dst	Out	Pointer to the output array of *size* FRAC32 data elements
src	In	Pointer to the input array of of *size* FRAC32 data elements
size	In	Number of elements in input and output arrays
freq	In	FRAC32 value in range of 0 to 0.5 that controls filter coefficients computation
bandw	In	FRAC32 value in range of 0 to 0.5 that controls filter coefficients computation

Returns: The BANDPASS_FLTR macro generates output values, which are stored in the array, pointed to by *dst*.

4.10.3 Description of Optimization

This macro frequently performs multiplication and addition operations on fractional values. It is suitable for the eMAC unit, because the eMAC has a fractional mode.

The optimization for the MAC unit is performed as an emulation of the fractional mode, using mac.w with shift to left instruction on the upper 16 bits of operands. Therefore, only the upper 16 bits of the resulting signals are valuable.

The coefficients are pre-computed using standard C subroutines in the BANDPASS_FLTR macro. Then this macro uses the __IMPL_BAND_FLTR macro to compute output samples.

The following optimization techniques were used:

1. Postincrement addressing mode to load input and store output array elements.
2. Loop unrolling by two.
3. Descending loop organization.

Particular techniques for optimization are reviewed below.

C code:

```
arr_c[i] = a0 * arr1d[i] + a1 * arr1d[i-1] + a2 * arr1d[i-2] +
        b1 * arr_c[i-1] + b2 * arr_c[i-2];
```

Optimization for MAC unit.

The following should be noticed:

- The loop is unrolled by two.
- Coefficients a_0 , a_1 , a_2 , b_1 , and b_2 are pre-computed and held in registers a3 , a4, a5, d6, and d7 correspondingly.
- The a2 and d5 registers always hold the input samples per each iteration.
- The d3 and d0 registers always hold the output samples per each iteration.
- The a1 and a0 registers hold pointers to the *src[]* and *dst[]* arrays.

All add-multiply instructions are performed by the MAC unit. The MAC unit has no *movclr* instruction, so the accumulator must be cleared explicitly. After each computation of the output sample, the data the from accumulator is stored into the register, and the accumulator is cleared explicitly. After, the result is stored into memory.

Optimized code (uses MAC unit):

```
mac.w a3.u,a2.u,<<,ACC0     ; computes a[0]*x[i] for y[i] ouput element
```

```
mac.w a4.u,d4.u,<<,ACC0     ; computes a[1]*x[i-1] for y[i] ouput element
mac.w a5.u,d5.u,<<,ACC0     ; computes a[2]*x[i-2] for y[i] ouput element
mac.w d6.u,d0.u,<<,ACC0     ; computes b[1]*y[i-1] for y[i] ouput element
mac.w d7.u,d3.u,<<,ACC0     ; computes b[2]*y[i-2] to produce y[i]
move.l ACC0,d3                  ; moves y[i] to d3
move.l #0,ACC0                  ; clears accumulator
move.l d3,(a0)+             ; and stores y[i] to memory
```

Optimization for eMAC unit.

The following should be noticed:

- The loop is unrolled by two.
- Coefficients a_0 , a_1 , a_2 , b_1 , and b_2 are pre-computed and held in registers a3 , a4, a5, d6, and d7 correspondingly.
- The a2 and d5 registers always hold the input samples per each iteration.
- The d3 and d0 registers always hold the output samples per each iteration.
- The a1 and a0 registers hold pointers to the *src[]* and *dst[]* arrays.

All add-multiply instructions are performed by the eMAC unit. After each computation of an output sample, the *movclr* instruction is used to clear the accumulator and store the result into the general purpose register. After, the result is stored into memory.

Optimized code (uses eMAC unit):

```
mac.l a3,a2,ACC0            ; computes a[0]*x[i] for y[i] ouput element
mac.l a4,d4,ACC0            ; computes a[1]*x[i-1] for y[i] ouput element
mac.l a5,d5,ACC0            ; computes a[2]*x[i-2] for y[i] ouput element
mac.l d6,d0,ACC0            ; computes b[1]*y[i-1] for y[i] ouput element
mac.l d7,d3,ACC0            ; computes b[2]*y[i-2] to produce y[i]
movclr.l ACC0,d3            ; moves y[i] to d3
move.l d3,(a0)+             ; and stores y[i] to memory
```

4.11 BANDREJECT_FLTR

4.11.1 Macro Description

This macro computes a band-reject filter. This recursive filter uses five coefficients: a_0, a_1, a_2, b_1, and b_2, so the filter can be represented in the following form:

$$y_n = a_0 * x_n + a_1 * x_{n-1} + a_2 * x_{n-2} + b_1 * y_{n-1} + b_2 * y_{n-2}$$

The filter's response characteristics are controlled by the parameter *f*, a value of center frequency, and *BW*, the bandwidth. Both parameters values must be in the range 0 to 0.5. The design equations for a bandpath filter are the following:

$$a_0 = K$$

$$a_1 = -2K\cos(2\pi f)$$

$$a_2 = K$$

$$b_1 = 2R\cos(2\pi f)$$

$$b_2 = -R^2$$

where:

$$K = \frac{1 - 2R\cos(2\pi f) + R^2}{2 - 2\cos(2\pi f)}$$

$$R = 1 - 3BW$$

More details on this digital recursive filters characteristic may be found in The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/).

4.11.2 Parameters Description

Call(s):

```
BANDREJECT_FLTR(FRAC32 *dst, FRAC32 *src, long size, FRAC32 freq, FRAC32
bandw)
```

The input signals to the filter are held in array *src[]*, and the output values are stored in array *dst[]*. Both arrays run from 0 to size-1. The freq and bandw parameters control the computation of the a_0, a_1, a_2, b_1, and b_2 filter coefficients. Prior to any call of BANDREJECT_FLTR, the user must allocate memory for both the *src[]* and *dst[]* arrays, either in static or dynamic memory.

Parameters:

Table 4-11. BANDREJECT_FLTR Parameters

dst	Out	Pointer to the output array of *size* FRAC32 data elements
src	In	Pointer to the input array of of *size* FRAC32 data elements
size	In	Number of elements in input and output arrays

freq	In	FRAC32 value in range of 0 to 0.5 that controls filter coefficients computation
bandw	In	FRAC32 value in range of 0 to 0.5 that controls filter coefficients computation

Returns: The BANDREJECT_FLTR macro generates output values, which are stored in the array pointed to by *dst*.

4.11.3 Description of Optimization

This macro frequently performs multiplication and addition operations on fractional values. It is suitable for the eMAC unit, because the eMAC has a fractional mode.

The optimization for the MAC unit is performed as an emulation of the fractional mode, using mac.w with shift to left instruction on the upper 16 bits of operands. So only the upper 16 bits of the resulting signals are valuable.

The coefficients are pre-computed using standard C subroutines in the BANDREJECT_FLTR macro. Then this macro uses the __IMPL_BAND_FLTR macro to compute output samples.

4.12 MOV_AVG_FLTR

4.12.1 Macros Description

This macro computes the moving average filter. As the name implies, the moving average filter operates by averaging a number of points from the input signal to produce each point in the output signal. In the equation form, this filter can be represented as the following:

$$y[i] = \frac{1}{M}\sum_{j=0}^{M-1} x[i+j]$$

M is the number of points used in the moving average.

More details on this digital filter's characteristic may be found in The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D. California Technical Publishing (http://www.dspguide.com/).

4.12.2 Parameters Description

Call(s):

```
MOV_AVG_FLTR(FRAC32 *dst, FRAC32 *src, long size, long M)
```

The input signals to the filter are held in array *src[]*, and the output values are stored in array *dst[]*. Both arrays run from *0* to *size-1*. M is the number of points used in the moving average. Prior to any call of MOV_AVG_FLTR, the user must allocate memory for both the *src[]* and *dst[]* arrays, either in static or dynamic memory.

Parameters:

Table 4-12. MOV_AVG_FLTR Parameters

dst	out	Pointer to the output array of *size* FRAC32 data elements
src	In	Pointer to the input array of of *size* FRAC32 data elements
size	in	Number of elements in input and output arrays
M	in	M is the number of points used in moving average.

Returns: The MOV_AVG_FLTR macro generates output values, which are stored in the array, pointed to by *dst*.

4.12.3 Description of Optimization

This macro frequently performs multiplication and addition operations on fractional values. It is suitable for the eMAC unit, because the eMAC has a fractional mode.

Optimization for the MAC unit is performed as an emulation of the fractional mode, using mac.w with shift to left instruction on the upper 16 bits of operands. So only the upper 16 bits of the resulting signals are valuable.

The standard C macro MOV_AVG_FLTR computes the 1/M value and uses the IMPL_MOV_AVG_FLTR macro to compute output samples.

- Optimization of IMPL_MOV_AVG_FLTR macro:

The following optimization techniques were used:

1. Post-increment addressing mode to load input and store output array elements.

2. Descending loop organization.

Particular techniques for optimization are reviewed below.

C code:

```
for(i = 0; i < SIZE - M + 1; i++)
   {
```

```
    for  (j = 0; j < M; j++)
        arr_c[i] += arr1d[i+j];
    arr_c[i] /= md;
}
```

Optimization for MAC unit.

The following should be noticed:

- The 1/M value is stored in register a3.
- To calculate the y[i+1] value, the y[i] value is used. The first item of y[i] value is subtracted from the accumulator, and the last item of y[i+1] is added to the accumulator. Then the accumulator value is stored as y[i+1].
- The a1 and a0 registers hold pointers to the *src[]* and *dst[]* arrays.

All add-multiply instructions are performed by the MAC unit.

Optimized code (uses MAC unit):

```
mac.w d4.u,a3.u,<<,ACC0     ; adds the last item of y[i+1] to accumulator
msac.w d0.u,a3.u,<<,ACC0    ; substructs the first item of y[i] from
                                ; accumulator
    ...
move.l ACC0,d5          ; stores the y[i] to d5 from accumulator
move.l d5,(a0)+     ; stores the y[i] into memory
```

Optimization for eMAC unit.

The following should be noticed:

- The 1/M value is stored in register a3;.
- To calculate the y[i+1] value, the y[i] value is used. The first item of y[i] value is subtracted from the accumulator and the last item of y[i+1] is added to the accumulator. Then accumulator value is stored as y[i+1]
- The a1 and a0 registers hold pointers to the *src[]* and *dst[]* arrays.

All add-multiply instructions are performed by the eMAC unit.

Optimized code (uses eMAC unit):

```
mac.l d4,a3,ACC0    ; adds the last item of y[i+1] to accumulator
```

```
msac.l d0,a3,ACC0       ; substructs the first item of y[i] from accumulator
    ...
move.l ACC0,d5          ; stores the y[i] to d5 from accumulator
move.l d5,(a0)+     ; stores the y[i] into memory
```

Chapter 5
Macros for Mathematical Functions

5.1 SIN

5.1.1 Macro Description

This macro performs some arithmetical operations with the angle parameter to reduce the angle value to the range of [0.. π/4], and then calls the SIN_F or COS_F macro to compute the sine function.

Notes:

- Value of the angle parameter must be in [0..2*π].
- SIN and COS macros have a common header file “sin_cos.h.”

5.1.2 Parameters Description

Call(s):

```
FIXED64 SIN(FIXED64 ang)
```

Parameters:

Table 5-1. SIN Parameters

ang	in	an angle value

Returns: sine value of the angle.

5.1.3 Description of Optimization

Because the SIN macro only performs some simple arithmetical operations with the ang parameter before invoking the SIN_F/COS_F functions, no optimization is needed.

5.2 COS

5.2.1 Macro Description

This macro performs some arithmetical operations with the angle parameter to reduce the angle value to the range of [0.. π/4], and then calls the SIN_F or COS_F macro to compute the cosine function.

Notes:

- Value of the angle parameter must be in [0..2*π].
- SIN and COS macros have a common header file "sin_cos.h."

5.2.2 Parameters Description

Call(s):

```
FIXED64 COS(FIXED64 ang)
```

Parameters:

Table 5-2. COS Parameters

ang	In	an angle value

Returns: cosine value of the angle.

5.2.3 Description of Optimization

Because the COS macro only performs some simple arithmetical operations with the ang parameter before invoking the SIN_F/COS_F functions, no optimization is needed.

5.3 SIN_F

5.3.1 Macro Description

This macro computes the sine of an angle from the range [0..π/4]. Computation is done by Teylor's series consisting of 6 elements:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!}$$

Notes:

- Value of the angle parameter must be in [0..π/4].
- SIN_F and COS_F macros have a common header file "sin_cos.h."

5.3.2 Parameters Description

Call(s):

```
FRAC32 SIN_F(FRAC32 ang)
```

Parameters:

Table 5-3. SIN_F Parameters

ang	in	An angle value

Returns: value of the sine function of the angle.

5.3.3 Description of Optimization

C code:

```
res_c = sin(tstvalc);
```

Optimization for the MAC unit can be done using the following techniques:

1. Sequential mac instructions that allow efficient use of the MAC pipeline.
2. Quick multiplication and subtraction due to the msac instruction.
3. Quick multiplication due to the MAC unit.

Optimized code (uses MAC unit):

```
move.l  #0, ACC0
mac.w   d0.u, d0.u, <<, ACC0
move.l  ACC0, d1
move.l  #0, ACC0
mac.w   d1.u, d0.u, <<, ACC0
move.l  ACC0, d2
move.l  #0, ACC0
mac.w   d1.u, d2.u, <<, ACC0
move.l  ACC0, d3
move.l  #0, ACC0
mac.w   d1.u, d3.u, <<, ACC0
move.l  ACC0, d4
move.l  #0, ACC0
mac.w   d1.u, d4.u, <<, ACC0
move.l  ACC0, d5
move.l  #0, ACC0
mac.w   d1.u, d5.u, <<, ACC0
move.l  ACC0, d6
dc.w    0xa100      //move.ld0, ACC0
movea.l #357913941, a0
movea.l #17895697, a1
movea.l #426088, a2
movea.l #5917, a3
movea.l #53, a4
msac.w  d2.u, a0.u, <<, ACC0
mac.w   d3.u, a1.u, <<, ACC0
msac.w  d4.u, a2.u, <<, ACC0
mac.w   d5.u, a3.u, <<, ACC0
msac.w  d6.u, a4.u, <<, ACC0
move.l  ACC0, d0
```

Optimization for the eMAC unit includes the same optimization techniques as the MAC unit, as well as the following:

1. Using fractional mode of the eMAC unit, which allows using 32x32 multiplication without lack of precision.

2. Using the movclr instruction to store a value in a register and clear an accumulator at the same time.

Optimized code (uses eMAC unit):

```
mac.l   d0, d0, ACC0
movclr.lACC0, d1
mac.l   d1, d0, ACC0
movclr.lACC0, d2
mac.l   d1, d2, ACC0
movclr.lACC0, d3
mac.l   d1, d3, ACC0
movclr.lACC0, d4
mac.l   d1, d4, ACC0
movclr.lACC0, d5
mac.l   d1, d5, ACC0
movclr.lACC0, d6
dc.w 0xa100         //move.ld0, ACC0
movea.l #357913941, a0
movea.l #17895697, a1
movea.l #426088, a2
movea.l #5917, a3
movea.l #53, a4
msac.l  d2, a0, ACC0
mac.l   d3, a1, ACC0
msac.l  d4, a2, ACC0
mac.l   d5, a3, ACC0
msac.l  d6, a4, ACC0
move.l  ACC0, d0
```

5.4 COS_F

5.4.1 Macro Description

This macro computes the cosine of an angle from the range [0..π/4]. Computation is done by Teylor’s series consisting of 7 elements:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \frac{x^{12}}{12!}$$

Notes:

- Value of the angle parameter must be in [0..$\pi/4$].
- SIN_F and COS_F macros have a common header file “sin_cos.h.”

5.4.2 Parameters Description

Call(s):

```
FRAC32 COS_F(FRAC32 ang)
```

Parameters:

Table 5-4. COS_F Parameters

ang	in	An angle value

Returns: value of the cosine function of the angle.

5.4.3 Description of Optimization

C code:

```
res_c = cos(tstvalc);
```

Optimization for the MAC unit can be done using the following techniques:

1. Sequential mac instructions that allow efficient use of the MAC pipeline.
2. Quick multiplication and subtraction due to the msac instruction.
3. Quick multiplication due to the MAC unit.

Optimized code (uses MAC unit):

```
        move.l  #0, ACC0
        mac.w   d0.u, d0.u, <<, ACC0
        move.l  ACC0, d1
        move.l  #0, ACC0
        mac.w   d1.u, d1.u, <<, ACC0
        move.l  ACC0, d2
        move.l  #0, ACC0
        mac.w   d1.u, d2.u, <<, ACC0
        move.l  ACC0, d3
```

```
move.l  #0, ACC0
mac.w   d2.u, d2.u, <<, ACC0
move.l  ACC0, d4
move.l  #0, ACC0
mac.w   d2.u, d3.u, <<, ACC0
move.l  ACC0, d5
move.l  #0, ACC0
mac.w   d3.u, d3.u, <<, ACC0
move.l  ACC0, d6
move.l  #0x7fffffff, ACC0
movea.l #1073741824, a0
movea.l #89478485, a1
movea.l #2982616, a2
movea.l #53261, a3
movea.l #591, a4
movea.l #4, a5
msac.w  d1.u, a0.u, <<, ACC0
mac.w   d2.u, a1.u, <<, ACC0
msac.w  d3.u, a2.u, <<, ACC0
mac.w   d4.u, a3.u, <<, ACC0
msac.w  d5.u, a4.u, <<, ACC0
mac.w   d6.u, a5.u, <<, ACC0
move.l  ACC0, d0
```

Optimization for the eMAC unit includes the same optimization techniques as the MAC unit, as well as following:

1. Using fractional mode of the eMAC unit, which allows using 32x32 multiplication without lack of precision.
2. Using the movclr instruction to store a value in a register and clear an accumulator at the same time.
3. Using two accumulators for quickly raising the operand to the needed power.

Optimized code (uses eMAC unit):

```
move.l  #0, ACC0
move.l  #0, ACC1
mac.l   d0, d0, ACC0
movclr.lACC0, d1
mac.l   d1, d1, ACC0
movclr.lACC0, d2
```

```
mac.l   d1, d2, ACC0
mac.l   d2, d2, ACC1
movclr.lACC0, d3
movclr.lACC1, d4
mac.l   d2, d3, ACC0
mac.l   d3, d3, ACC1
movclr.lACC0, d5
movclr.lACC1, d6
move.l  #0x7fffffff, ACC0
movea.l #1073741824, a0
movea.l #89478485, a1
movea.l #2982616, a2
movea.l #53261, a3
movea.l #591, a4
movea.l #4, a5
msac.l  d1, a0, ACC0
mac.l   d2, a1, ACC0
msac.l  d3, a2, ACC0
mac.l   d4, a3, ACC0
msac.l  d5, a4, ACC0
mac.l   d6, a5, ACC0
move.l  ACC0, d0
```

5.5 MUL

5.5.1 Macro Description

This macro computes a product of two fixed point numbers.

5.5.2 Parameters Description

Call(s):

```
FIXED64 MUL(FIXED64 m1, FIXED64 m2)
```

Parameters:

Table 5-5. MUL Parameters

m1	in	Multiplicand

m2	in	Multiplier

Returns: product of m1 and m2.

5.5.3 Description of Optimization

C code:

```
res_c = a * b;
```

Optimization for the MAC unit is unsuitable, because of the absence of fractional mode in the MAC unit.

Optimization for the eMAC unit can be done using the following techniques:

1. Using both integer and fractional modes of the eMAC unit to get all 64 bits of the result with only 6 mac instructions.
2. Using the eMAC rounding mode to gain a suitable precision without additional mac instructions.

Optimized code (uses eMAC unit):

```
lsr.l   %1, d1
lsr.l   %1, d3
mac.l   d1, d3, ACC0
mac.l   d0, d3, ACC1
move.l  %0, ACCEXT01
mac.l   d1, d2, ACC1
lsl.l   %1, d1
lsl.l   %1, d3
move.l  %0x40, d5
move.l  d5, MACSR
mac.l   d0, d3, ACC2
mac.l   d1, d2, ACC3
mac.l   d0, d2, ACC1
```

Chapter 6
QuickStart for CodeWarrior

The Library of Macros is very easy to use and test. Altough all macros are written in assembly, they were developed in such a way that they can be easily integrated in a C program.

The purpose of this chapter is to guide an user on the steps required to add, compile, test, and use the Library of Macros. The CONV function will be used for demonstration purposes. The example was developed in CodeWarrior for ColdFire V6.0 using the MCF5282 microprocessor, and the same steps may be applied to other derivatives and versions.

6.1 Creating a New Project

a) Open CodeWarrior. Usually in “Start→Programs→Metrowerks CodeWarrior→CW for ColdFire 6.0→CodeWarrior IDE.” CodeWarrior main window should appear.

b) From the main menu bar, select File→New. The “New” dialog box should appear.

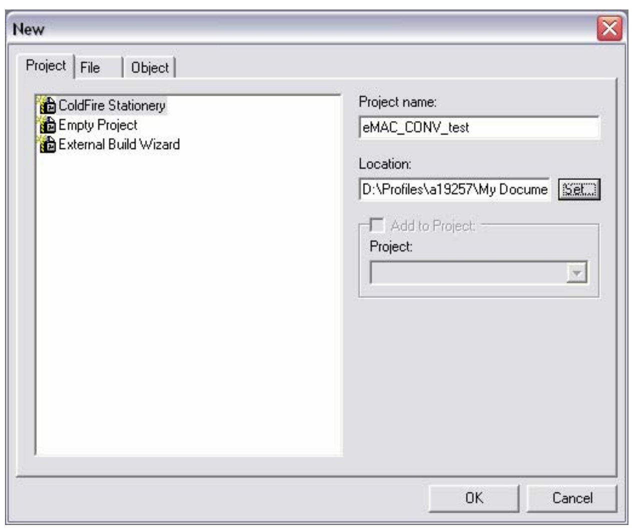

Figure 6-1. “New” Dialog Box

c) Select ColdFire Stationery as the type of project.

d) Select a project name in the “Project name” textbox. I.e. eMAC_CONV_test.

e) Select an appropiate location for your project using the “Location” textbox.

f) Click “OK.” The “New Project” Dialog Box should appear.

g) Select the appropiate stationary. I.e: expand CF_M5282EVB and select “C.”

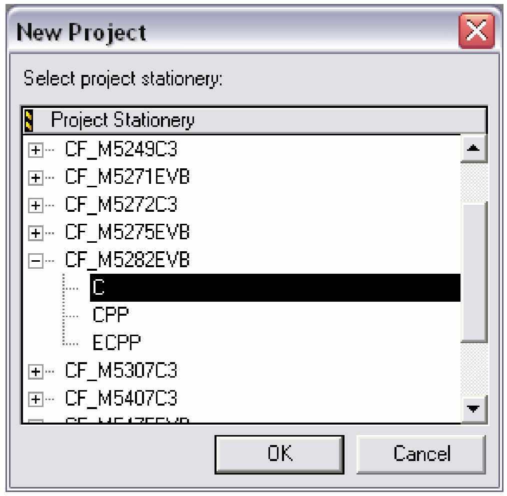

Figure 6-2. “New Project” Dialog Box

h) Click OK. A new folder will be created for your project and the project window appears, docked at the left side of the main window.

6.2 Modifying the Settings of your Project

a) Select an appropiate target to debug your code. I.e. “M5282EVB UART Debug.”

b) Open the Settings window of your project by selecting “Menu→Edit→your_target Settings” or Alt+F7 or clicking the  button. The Settings window should appear.

c) Enable the processor to use MAC or eMAC by selecting clicking on the appropiate checkbox in the “Language Settins→ColdFire Assembler” section. I.e. check the “Processor has EMAC” checkbox.

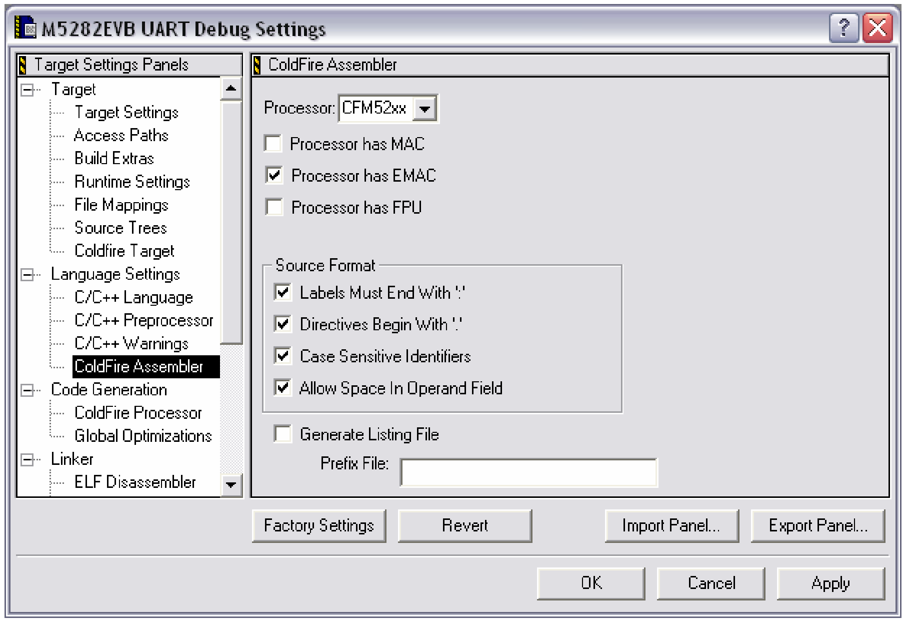

Figure 6-3. "Settings" Window in "ColdFire Assembler" Selection

d) Change to the "Debugger→Remote Debugging" section.

e) A message will appear informing that the project must be rebuilt. Click OK.

f) Select an appropiate Connection for your EVB. I.e. "PEMICRO USB" if you are using the P&E USB wiggler.

g) Click OK. Your project is now configured to use and debug the Library of Macros.

6.3 Adding the Library of Macros

h) Using windows explorer, copy the unzipped folder "library_macros" into your project. I.e. the final path for your libraries can be "..\eMAC_CONV_test\Source\library_macros."

i) Drag-and-drop the copied "library_macros" folder from windows explorer to your CodeWarrior project window inside the "source folder." This will add all files and folders from the library of macros to your current project. You can also add each file and folder by right-clicking in the project window and selecting "Add files" and "Create Group."

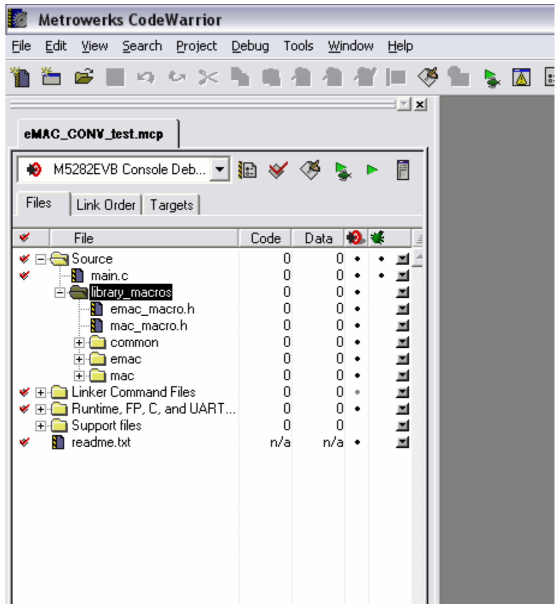

Figure 6-4. Library of Macros added to Project Explorer

j) Click on the “Make” button  in order to compile and link your project.

k) You shouldn’t get any errors. Otherwise verify previous steps.

l) Now you can use any desired macro from the library.

6.4 Using a Macro

a) Include the appropate header into your main.c file

- Using a microprocessor with eMAC:

```
#include “emac_macro.h”
```

- Using a microprocessor with MAC:

```
#include “mac_macro.h”
```

b) Using the prototype declaration described in this document, add the your function call. I.e. using the CONV macro, described in Section 4.4 CONV the prototype is the following:

```
void CONV(void *y, void *x, void *h, int xsize, int hsize)
```

So, the call of your function can be something like:

```
CONV(f32_y, f32_x, f32_h, X_SIZE, H_SIZE);
```

c) Create the arrays for testing purposes. I.e:

```
#define X_SIZE 20
#define H_SIZE 10

FRAC32 f32_y[X_SIZE+H_SIZE-1];
FRAC32 f32_x[X_SIZE] = {
   D_TO_F32(0),                     D_TO_F32(0.309016994374947),
   D_TO_F32(0.587785252292473),     D_TO_F32(0.809016994374947),
   D_TO_F32(0.951056516295154),     D_TO_F32(0.99999),
   D_TO_F32(0.951056516295154),     D_TO_F32(0.809016994374947),
   D_TO_F32(0.587785252292473),     D_TO_F32(0.309016994374948),
   D_TO_F32(1.22514845490862E-16),  D_TO_F32(-0.309016994374948),
   D_TO_F32(-0.587785252292473),    D_TO_F32(-0.809016994374947),
   D_TO_F32(-0.951056516295154),    D_TO_F32(-1),
   D_TO_F32(-0.951056516295154),    D_TO_F32(-0.809016994374948),
   D_TO_F32(-0.587785252292473),    D_TO_F32(-0.309016994374948) };
FRAC32 f32_h[H_SIZE] = {
   D_TO_F32(.1),  D_TO_F32(.2),  D_TO_F32(.3), D_TO_F32(.4),
   D_TO_F32(.5),  D_TO_F32(.6),  D_TO_F32(.7),  D_TO_F32(.8),
   D_TO_F32(.9),  D_TO_F32(.99) };
```

d) Click the Make button. You shouldn't have any errors. Otherwise, review the errors and fix them.

e) Now you can debug or execute your application. You can also use the serial terminal to display the results of your function as follows:

```
for (i=0; i < (X_SIZE+H_SIZE-1); i++){
       printf("Y%d = %d\n\r", i, f32_y[i]);
}
```

Note that this `printf` function will send output data in FRAC32 format (multiplied by 2^{31}). In order to get the real value, the result must be divided by 2^{31}.

f) For this example, the result will be as follows:

f32_x			f32_h			f32_y		
	frac32	decimal		frac32	decimal		frac32	decimal
X0	0	0	H0	21474836	0.01	Y0	0	0
X1	6.64E+08	0.309017	H1	42949672	0.02	Y1	6636089	0.00309
X2	1.26E+09	0.587785	H2	64424509	0.03	Y2	25894770	0.012058
X3	1.74E+09	0.809017	H3	85899345	0.04	Y3	62526959	0.029116
X4	2.04E+09	0.951057	H4	1.07E+08	0.05	Y4	1.2E+08	0.055685
X5	2.15E+09	0.99999	H5	1.29E+08	0.06	Y5	1.98E+08	0.092254
X6	2.04E+09	0.951057	H6	1.5E+08	0.07	Y6	2.97E+08	0.138333
X7	1.74E+09	0.809017	H7	1.72E+08	0.08	Y7	4.13E+08	0.192502
X8	1.26E+09	0.587785	H8	1.93E+08	0.09	Y8	5.42E+08	0.25255
X9	6.64E+08	0.309017	H9	2.15E+08	0.1	Y9	6.78E+08	0.315687
X10	0	0				Y10	8.14E+08	0.378824
X11	-6.6E+08	-0.30902				Y11	8.69E+08	0.40488
X12	-1.3E+09	-0.58779				Y12	8.4E+08	0.391303
X13	-1.7E+09	-0.80902				Y13	7.29E+08	0.339422
X14	-2E+09	-0.95106				Y14	5.46E+08	0.254316
X15	-2.1E+09	-1				Y15	3.1E+08	0.144317
X16	-2E+09	-0.95106				Y16	43358789	0.020191
X17	-1.7E+09	-0.80902				Y17	-2.3E+08	-0.10591
X18	-1.3E+09	-0.58779				Y18	-4.8E+08	-0.22165
X19	-6.6E+08	-0.30902				Y19	-6.8E+08	-0.31569
						Y20	-8.1E+08	-0.37883
						Y21	-8.8E+08	-0.40797
						Y22	-8.7E+08	-0.40336
						Y23	-7.9E+08	-0.36854
						Y24	-6.7E+08	-0.31
						Y25	-5.1E+08	-0.23657
						Y26	-3.4E+08	-0.15852
						Y27	-1.9E+08	-0.08659
						Y28	-6.6E+07	-0.0309

Table 6-1. Result of CONV Example

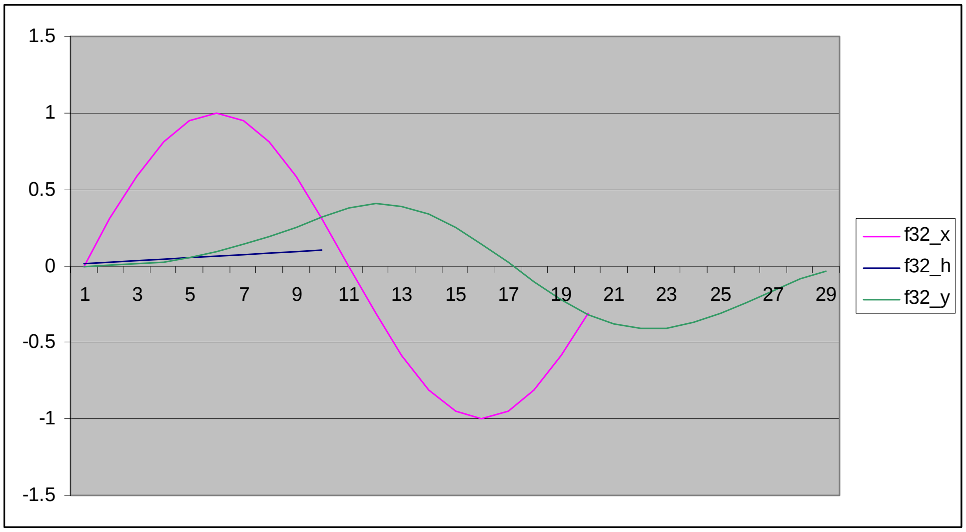

Figure 6-5. Resulting Graph of CONV Example