DSPLib

Reference Manual

Atmel Roma • Via Vito Giuseppe Galati 91 • Roma, 00155 Italy
TEL (+39) 06-40901420 • FAX (+39) 06-40501613 • Web Site: ***http://www.atmelroma.it***

Change track

Rev.	Reason for Revision/ Description of change	Author	Date
Draft	Confidential	EPA	10/06

Table of Contents

Function cfbyfdivide

`#include "magic_chess.h"`

Functions

- long **cfByfDivide** (_c_float *rpacfIn1, int riStrideInp1, float *rpafIn2, int riStrideInp2, _c_float *rpacfOut, int riStrideOut, int riLen)
  cfByfDivide: complex float array division by a float array

Function Documentation

long cfByfDivide (_c_float * *rpacfIn1*, int *riStrideInp1*, float * *rpafIn2*, int *riStrideInp2*, _c_float * *rpacfOut*, int *riStrideOut*, int *riLen*)

cfByfDivide: complex float array division by a float array

Division of Input complex float Array by a float Array (23 bits of precision)

```
Z(k) = (Re(X(k))+Im(X(k)))/Y(k)

k=0...riLen-1
```

Parameters:

rpacfIn1 : Pointer to a floating point input Array 1
rpafIn2 : Pointer to a floating point input Array 2
rpacfOut : Pointer to a floating point output Array
riStrideInp1 : Stride for the input Array 1
riStrideInp2 : Stride for the input Array 2
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 19+N*7

Number of VLIW: 26

Restrictions: Element count should be a multiple of 1

Function cfconjscaleoffset

`#include "magic_chess.h"`

Functions

- long **cfConjScaleOffset** (float *gfpIn, int giStrideInp, float *gfpOut, int giStrideOut, float *gfpScale, float *gfpOffset, int giLen)
  cfConjScaleOffset: The conjugate of the contents of the complex float array is multiplied with the complex float scale value and then added with the complex offset

---

Function Documentation

long cfConjScaleOffset (float * *gfpIn*, int *giStrideInp*, float * *gfpOut*, int *giStrideOut*, float * *gfpScale*, float * *gfpOffset*, int *giLen*)

cfConjScaleOffset: The conjugate of the contents of the complex float array is multiplied with the complex float scale value and then added with the complex offset

```
gfpOut(k)=gfpScale*conj(gfpIn(k))+gfpOffset

k=0.......giLen - 1
```

Parameters:

gfpIn : Pointer to a floating point input Array
gfpOut : Pointer to a floating point output Array
giStrideInp : Stride for the input Array
giStrideOut : Stride for the Output Array
gfpScale : Pointer to floating point Scale Value
gfpOffset : Pointer to floating point OffsetValue
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 17+N*1.25

Number of VLIW: 22

Restrictions: Element count should be a multiple of 4

Function cfconv

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)
- #define **ODD_VECLEN**

Functions

- long **cfConv** (float *rpafInX, float *rpafInCoeffH, float *rpafOut, int riVecLen, int riFilterLen, int riTransient)
  cfConv: Computing Complex vector Convolution

Define Documentation

#define ODD_VECLEN

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long cfConv (float * *rpafInX*, float * *rpafInCoeffH*, float * *rpafOut*, int *riVecLen*, int *riFilterLen*, int *riTransient*)

cfConv: Computing Complex vector Convolution

Convolution of input complex float array is calculated with the input Coefficent

```
Y (k) = sumi( X(i)*H(k-i) )

i = 0 to M-1
k = 0 to (N+M-1)
N:- input vector length
M:- input filter length
```

Parameters:

rpafInX : Pointer to a floating point input array
rpafInCoeffH : Pointer to a floating point Coefficent array
rpafOut : Pointer to a floating point Array
riVecLen : Number of input N
riFilterLen : Number of Coefficent M
riTransient : Integer value used to compute or not the transient codes of the convolution: if riTransient=0 the transient isn't computed, otherwise it's calculated

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: Input/Output transient: (M-1) * (19+2.5*M)

Number of cycles: Steady state:

- Case1 N is odd : 45 + 2.00*M + (N-M-1)*(26.5 + 1*(M-8))
- Case2 N is even: 64 + 3.25*M + (N-M-2)*(28.0 + 1*(M-8))

Number of VLIW:

- Case1 N is odd :138
- Case2 N is even:165

Restrictions: M must be a multiple of 4 and should be less than N

User Info:

1. Inorder to have a better performance enable/disable ODD_VECLEN depending on the even/odd nature of N
2. #define UNROLL_FAC_2:disable this to activate the code with unroll factor of 4

Function cfconv2d

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **cfConv2d** (_c_float *rpafInA, int riRowLenA, int riColLenA, _c_float *rpafInKernelH, _c_float *rpafOut, int riKOrder)
  cfConv2d: Computing Complex vector 2D-Convolution

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long cfConv2d (_c_float * *rpafInA*, int *riRowLenA*, int *riColLenA*, _c_float * *rpafInKernelH*, _c_float * *rpafOut*, int *riKOrder*)

cfConv2d: Computing Complex vector 2D-Convolution

2D-Convolution of input complex float array is calculated with the input Coefficent

```
    C(r,c) = sumi(sumj( H[k-1-i][k-1-j] * A[r+i][c+j] ))

r = 0 to M-k+1  i = 0 to k-1
c = 0 to N-k+1  j = 0 to k-1

M = No. of rows in A
N = No. of columns in A
k = Order of H
```

Parameters:

rpafInA : Pointer to a complex floating point input matrix
riRowLenA : Number of rows in matrix, A, to be convolved M
riColLenA : Number of columns in matrix, A, to be convolved N
rpafInKernelH : Pointer to a complex floating point input matrix
rpafOut : Pointer to the complex floating point output matrix C of the order (M-k+1) * (N-k+1)
riKOrder : Kernel size k

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 32+(M-k+1) * (15+(N-k+1) * (9+k*(6.5+1.5*k)))

Number of VLIW: 72

Restrictions: k must be even and greater than 2

User Info:

1. #define PIPE: define, when [N >= M > k >= 24], for better performances
2. Set input lengths which satisfy N >= M > k

Function cfdist

```
#include "magic_chess.h"
```

Functions

- long **vfSqrt** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfSqrt: *Square Root Computation*

- long **cfDist** (float *rpafIn1, int riStrideInp1, float *rpafIn2, int riStrideInp2, float *rpafOut, int riStrideOut, int riLen)
  cfdist: Distance between two complex array.

Variables

- long **glRetSqrt_s**
- long **glRetSqrt**

Function Documentation

long cfDist (float * *rpafIn1*, int *riStrideInp1*, float * *rpafIn2*, int *riStrideInp2*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

cfdist: Distance between two complex array.

The Output array contains the distance between two complex array.

```
Z(k) = Sqrt((re.X(k)-re.Y(k))^2+(im.X(k)-im.Y(k))^2))

k=0...riLen-1
```

Parameters:

rpafIn1 : Pointer to a floating point Input Array 1
rpafIn2 : Pointer to a floating point Input Array 2
rpafOut : Pointer to a floating point Output Array
riStrideInp1 : Stride for the input Array 1
riStrideInp2 : Stride for the input Array 2
riStrideOut : Stride for the Output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles:

```
 54   +  3.00 * N
 55   +   7.50 * N     For Sqrt with domain check enabled
------------------------
109   +   10.50 * N
```

Number of VLIW:

```
 66
116                    For Sqrt with domain check enabled
------------------------
182
```

Number of cycles:

```
 32   +   3.00 * N
 45   +   7.00 * N     For Sqrt with domain check disabled
------------------------
 77    +   10.00 * N
```

Number of VLIW:

```
 66
 59                    For Sqrt with domain check disabled
------------------------
125
```

Restrictions: riLen should be a multiple of 4

User Info:

1. Note: The square-root computation happens only for riLen/2 values.
2. Enable #define DOMAIN_CHECK in the file ..\..\..\dsplibrary\vfsqrt.c to include domain check (default setting)

long vfSqrt (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfSqrt: Square Root Computation

Computes the square root of the input vector float array

```
Y(k) = sqrt(X(k)       k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point input Array
rpafOut : Pointer to a floating point output Array
riStrideInp : Stride for the input Array
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+N*15

Number of VLIW: 116

Restrictions: Array should have at least 1 vector element

User Info: The domain check flag is set whenever the input is negative.

Variable Documentation

long glRetSqrt

long glRetSqrt_s

Function cfdot

`#include "magic_chess.h"`

Functions

- long **cfDot** (float *rfpaIn1, int riStrideInp1, float *rfpaIn2, int riStrideInp2, float *rfpaOut, int riLen)
  cfDot: Dot Product between two complex float arrays

Function Documentation

long cfDot (float * *rfpaIn1*, int *riStrideInp1*, float * *rfpaIn2*, int *riStrideInp2*, float * *rfpaOut*, int *riLen*)

cfDot: Dot Product between two complex float arrays

```
rfpaOut = Sum(rfpaIn1(k)*rfpaIn1(k))

k=0.......giLen - 1
```

Parameters:

rfpaIn1 : Pointer to a floating point input Array 1
rfpaIn2 : Pointer to a floating point input Array 2
rfpaOut : Pointer to a floating point output Array
riStrideInp1 : Stride for the input Array 1
riStrideInp2 : Stride for the input Array 2
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 28+1.25*riLen

Number of VLIW: 33

Restrictions: riLen count should be a multiple of 4 and greater than 4

Function cfexp

```
#include "magic_chess.h"
```

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **cfExp** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  cfExp: Complex exponential of a vectorial input array

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long cfExp (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

cfExp: Complex exponential of a vectorial input array

Computation of Complex exponential

```
Y(k)= e^i(X(k))

k = 0,1,.....riLen-1
```

Parameters:

rpafIn : Pointer to the input array of type vector float
riStrideInp : Stride to be applied on input array
rpafOut : Pointer to the output array of type vector float
riStrideOut : Stride to be applied on output array
riLen : Number of input vectors

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: : 101 + 30*riLen

Number of VLIW: : 161

Restrictions: riLen should be a multiple of 2

User Info: The relative error (value: 2.1678180205308e-007) is high for the value of the input 11.0000 radians which is very close to 90 degrees; otherwise for the rest of the input the relative error is of the order of e-009.

Function cflaststage

```
#include "magic_chess.h"
```

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **cfLastStage** (float *rpafInX1, float *rpafInX2, float *rpafInW, float *rpafOutY1, float *rpafOutY2, int riButterFlyNum)
  cfLastStage: LastStage calculates last FFT stage of a complete FFT

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long cfLastStage (float * *rpafInX1*, float * *rpafInX2*, float * *rpafInW*, float * *rpafOutY1*, float * *rpafOutY2*, int *riButterFlyNum*)

cfLastStage: LastStage calculates last FFT stage of a complete FFT

The function LastStage can be used as last FFT stage of a complete FFT

```
Y1[k] = X1[k] + W[k] * X2[k]
Y2[k] = X1[k] - W[k] * X2[k]

where k = 0....N-1
```

Parameters:

rpafInX1 : Pointer to a complex floating point input1 array
rpafInX2 : Pointer to a complex floating point input2 array
rpafInW : Pointer to a complex floating point coefficent array
rpafOutY1 : Pointer to a complex floating point output1 array
rpafOutY2 : Pointer to a complex floating point output2 array
riButterFlyNum : Number of butterfly to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 22+2.75*riButterFlyNum

Number of VLIW: 33

Restrictions: riButterFlyNum must be a multiple of 4

Function cfmagn

`#include "magic_chess.h"`

Functions

- long **vfSqrt** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfSqrt: *Square Root Computation*

- long **cfMagn** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  cfmagn: Complex Magnitude

Function Documentation

long cfMagn (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

cfmagn: Complex Magnitude

The Output array contains magnitude of the complex input array.

```
Z(k) = Sqrt((Re.X(k)+Im.X(k)^2)

k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point Input Array
rpafOut : Pointer to a floating point Output Array
riStrideInp : Stride for the Input Array
riStrideOut : Stride for the Output array
riLen : Element count N

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles:

```
 48  +  1.25 * N
 45  +  7.0  * N       For Sqrt
-----------------
 93  +  8.25 * N
```

Number of VLIW:

```
 53
 59                    For Sqrt
------
112
```

Restrictions: riLen (N) should be a multiple of 4

User Info: Note: square-root computation happens only for N/2 values

long vfSqrt (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfSqrt: Square Root Computation

Computes the square root of the input vector float array

```
Y(k) = sqrt(X(k)        k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point input Array
rpafOut : Pointer to a floating point output Array
riStrideInp : Stride for the input Array
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+N*15

Number of VLIW: 116

Restrictions: Array should have at least 1 vector element

User Info: The domain check flag is set whenever the input is negative.

Function cfmatrix2determ

`#include "magic_chess.h"`

Functions

- long **cfMatrix2Determ** (float *rpafIn, float *rpafOut)
  cfMatrix2Determ: Determinant of 2x2 Matrix

Function Documentation

long cfMatrix2Determ (float * *rpafIn*, float * *rpafOut*)

cfMatrix2Determ: Determinant of 2x2 Matrix

Computation of the Determinant of 2x2 Matrix

```
   Output = Det A

Algorithm:

   Matrix   A   =   |a     b |
                    |c     d |

   Determinant  =  a * d - b * c
```

Parameters:

rpafIn : Pointer to a floating point Input Array
rpafOut : Pointer to a floating point Output Array

Returns:

0 if succesful, !=0 otherwhise

Number of cycles: 17

Number of VLIW: 17

Restrictions: None

Function cfmatrix3determ

#include "magic_chess.h"

Functions

- long **cfMatrix3Determ** (float *rpafIn, float *rpafOut)
  cfMatrix3Determ: Determinant of 3x3 Matrix

Function Documentation

long cfMatrix3Determ (float * *rpafIn*, float * *rpafOut*)

cfMatrix3Determ: Determinant of 3x3 Matrix

Computation of determinant of a 3*3 Matrix

```
   Output = Det A

Algorithm:

   Matrix A=      |0   1   2|
                  |3   4   5|
                  |6   7   8|

   Determinant = 0(4.8-5.7)-1(3.8-5.6)+2(3.7-4.6)
```

Parameters:

rpafIn : Pointer to a floating point Input Array
rpafOut : Pointer to a floating point Output Array

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 31

Number of VLIW: 31

Restrictions: None

Function cfmatrix3x3byvectsmul

```
#include "magic_chess.h"
```

Functions

- long **cfMatrix3x3ByVectsMul** (_c_float *rpafIn1, _c_float *rpafIn2, _c_float *rpafOut, int riVnum)
  cfMatrix3x3ByVectsMul: Multiplication of a 3x3 Matrix and a vector

---

Function Documentation

long cfMatrix3x3ByVectsMul (_c_float * *rpafIn1*, _c_float * *rpafIn2*, _c_float * *rpafOut*, int *riVnum*)

cfMatrix3x3ByVectsMul: Multiplication of a 3x3 Matrix and a vector

Product of a complex 3x3 Matrix with a set of complex vectors of size 3

```
  Matrix A= |A11    A12   A13 |         Vector B= |B1|
            |A21    A22   A23 |                   |B2|
            |A31    A32   A33 |                   |B3|
                                                  |B4|
                                                  |B5|
                                                  |B6|
                                                   .
                                                   .
                                                   .
                                                  |BN|

1.Output C11,C21 and C31=Matrix A* First  Vector of B(B1,B2 and B3)
2.Output C21,C22 and C32=Matrix A* Second Vector of B(B4,B5 and B6)
3.....
4......          and C3N
```

Parameters:

rpafIn1 : Pointer to a floating point Input Matrix
rpafIn2 : Pointer to a floating point Input Array
rpafOut : Pointer to a floating point Output Matrix
riVnum : Number of input vectors

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 31+9*riVnum

Number of VLIW: 40

Restrictions: riVnum should be a multiple of 3

Function cfmatrix4x4byvectsmul

```
#include "magic_chess.h"
```

Functions

- long **cfMatrix4x4ByVectsMul** (_c_float *rpafIn1, _c_float *rpafIn2, _c_float *rpafOut, int riVnum)
  cfMatrix4x4ByVectsMul: Multiplication of a 4x4 Matrix and a vector

---

Function Documentation

long cfMatrix4x4ByVectsMul (_c_float * *rpafIn1*, _c_float * *rpafIn2*, _c_float * *rpafOut*, int *riVnum*)

cfMatrix4x4ByVectsMul: Multiplication of a 4x4 Matrix and a vector

The product of the 4x4 Matrix and a vector of dimension 4*N is computed

```
  Matrix A= |A11   A12   A13    A14|    Vector B=      |B1|
            |A21   A22   A23    A24|            |B2|
            |A31   A32   A33    A34|            |B3|
            |A41   A42   A43    A44|            |B4|
                                                |B5|
                                                |B6|
                                                |B7|
                                                |B8|
                                                 .
                                                 .
                                                 .
                                                |BN|

1.Output C11,C21,C31 and C41=Matrix A* First  Vector of B(B1,B2 and B3)
2.Output C21,C22,C32 and C42=Matrix A* Second Vector of B(B4,B5 and B6)
3.....
4......          and C4N
```

Parameters:

rpafIn1 : Pointer to a floating point Input Matrix
rpafIn2 : Pointer to a floating point Input Array
rpafOut : Pointer to a floating point Output Matrix
riVnum : Number of input vectors.

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 38+16*riVnum

Number of VLIW: 54

Restrictions: No of Vectors should be a multiple of 4

Function cfmatrix8x8byvectsmul

```
#include "magic_chess.h"
```

Functions

- long **cfMatrix8x8ByVectsMul** (_c_float *rpafIn1, _c_float *rpafIn2, _c_float *rpafOut, int riVnum)
  cfMatrix8x8ByVectsMul: Multiplication of a 8x8 Matrix and a vector

---

Function Documentation

long cfMatrix8x8ByVectsMul (_c_float * *rpafIn1*, _c_float * *rpafIn2*, _c_float * *rpafOut*, int *riVnum*)

cfMatrix8x8ByVectsMul: Multiplication of a 8x8 Matrix and a vector

The product of the 8x8 Matrix and a vector of dimension 8*N is computed

```
     Matrix A= |A11 A12 A13 A14 A15 A16 A17 A18|       Vector B=      |B1|
               |A21 A22 A23 A24 A25 A26 A27 A28|               |B2|
               |A31 A32 A33 A34 A33 A36 A37 A38|               |B3|
               |A41 A42 A43 A44 A45 A46 A47 A48|               |B4|
               |A51 A52 A53 A54 A55 A56 A57 A58|               |B5|
               |A61 A62 A63 A64 A65 A66 A67 A68|               |B6|
               |A71 A72 A73 A74 A75 A76 A77 A78|               |B7|
               |A81 A82 A83 A84 A85 A86 A87 A88|               |B8|
                                                               |B9|
                                                                .
                                                                .
                                                                .
                                                               |BN|

 1.Output C11,C21,C31,C41......C81=Matrix A* First  Vector of B(B1,B2 ... B8)
 2.Output C21,C22,C32,C42......C82=Matrix A* Second Vector of B(B4,B5 ... B8)
 3.....
 4......                             and C8N
```

Parameters:

rpafIn1 : Pointer to a floating point Input Matrix
rpafIn2 : Pointer to a floating point Input Vector array
rpafOut : Pointer to a floating point Output Array
riVnum : Number of Vector

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 44+65*riVnum

Number of VLIW: 109

Restrictions: riVnum should be a multiple of 8

Function cfmatrixadd

```
#include "magic_chess.h"
```

Defines

- #define **CHESS_UNROLL_4**

Functions

- long **cfMatrixAdd** (_c_float *rpacfIn1, _c_float *rpacfIn2, int riRows, int riCols, _c_float *rpacfOut)
  cfMatrixAdd: Matrix Addition

Define Documentation

#define CHESS_UNROLL_4

Function Documentation

long cfMatrixAdd (_c_float * *rpacfIn1*, _c_float * *rpacfIn2*, int *riRows*, int *riCols*, _c_float * *rpacfOut*)

cfMatrixAdd: Matrix Addition

Addition of 2 complex floating point matrices

```
C[riRows][riCols] = a[riRows][riCols]+b[riRows][riCols]            k=0...riRows*riCols-1
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Matrix A
rpacfIn2 : Pointer to a complex floating point input Matrix B
riRows : No of rows of Matrix A/Matrix B
riCols : No of cols of Matrix A/Matrix B
rpacfOut : Pointer to a complex floating point output Matrix C

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 36+(riRows*riCols*1.75)

Number of VLIW: 43

Restrictions: Minimum Matrix Length(riRows*riCols) should be 4

User Info: #define CHESS_UNROLL_4: enable this in order to activate the code with unroll factor of 4

Function cfmatrixbyvectsmul

```
#include "magic_chess.h"
```

Functions

- long **cfMatrixByVectsMul** (_c_float *rpacfIn1, int riRow, int riCol, _c_float *rpacfIn2, _c_float *rpacfOut, int riVnum)
  cfMatrixByVectsMul: Matrix by Vector multiplication

Function Documentation

long cfMatrixByVectsMul (_c_float * *rpacfIn1*, int *riRow*, int *riCol*, _c_float * *rpacfIn2*, _c_float * *rpacfOut*, int *riVnum*)

cfMatrixByVectsMul: Matrix by Vector multiplication

Product of a complex floating point matrix with a set of complex vectors

```
C[riRow][riVnum] = a[riRow][riCol]*b[riCol][riVnum]
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Matrix
riRow : No of Rows of Matrix A
riCol : No of coloumns of Matrix A/No of Rows of Vector
rpacfIn2 : Pointer to a complex floating point Vector arrray
rpacfOut : Pointer to a complex floating point Array
riVnum : Number of Vector

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 14 + ((((4*(riCol-1)+16)*riRow)+14)*riVnum)

Number of VLIW: 48

Restrictions: No of rows of Matrix A should be >=2

User Info: The even and odd flavors give a better performance when compared to this general optimization. This code works for both even and odd values of input

Function cfmatrixbyvectsmul_odd and cfmatrixbyvectsmul_even

```
#include "magic_chess.h"
```

Functions

- long **cfMatrixByVectsMul_odd** (_c_float *rpacfIn1, int riRow, int riCol, _c_float *rpacfIn2, _c_float *rpacfOut, int riVnum)
  cfMatrixByVectsMul_odd: Matrix by Vector multiplication -Odd Flavour

- long **cfMatrixByVectsMul_even** (_c_float *rpacfIn1, int riRow, int riCol, _c_float *rpacfIn2, _c_float *rpacfOut, int riVnum)
  cfMatrixByVectsMul_Even: Matrix by Vector multiplication -Even Flavour

---

Function Documentation

long cfMatrixByVectsMul_even (_c_float * *rpacfIn1*, int *riRow*, int *riCol*, _c_float * *rpacfIn2*, _c_float * *rpacfOut*, int *riVnum*)

cfMatrixByVectsMul_Even: Matrix by Vector multiplication -Even Flavour

Product of a complex floating point matrix with a set of complex vectors

```
C[riRow][riVnum] = a[riRow][riCol]*b[riCol][riVnum]
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Matrix
riRow : No of Rows of Matrix A
riCol : No of coloumns of Matrix A
rpacfIn2 : Pointer to a complex floating point Vector array
rpacfOut : Pointer to a complex floating point Output Matrix
riVnum : Number of Vector

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 16 + ((((4*((riCol/2)-1)+14)*riRow)+8)*riVnum)

Number of VLIW: 42

Restrictions: No of rows of Matrix A should be Even and >=4

long cfMatrixByVectsMul_odd (_c_float * *rpacfIn1*, int *riRow*, int *riCol*, _c_float * *rpacfIn2*, _c_float * *rpacfOut*, int *riVnum*)

cfMatrixByVectsMul_odd: Matrix by Vector multiplication -Odd Flavour

Product of a complex floating point matrix with a set of complex vectors

```
C[riRow][riVnum] = a[riRow][riCol]*b[riCol][riVnum]
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Matrix
riRow : No of Rows of Matrix A
riCol : No of coloumns of Matrix A
rpacfIn2 : Pointer to the complex floating point Vectors
rpacfOut : Pointer to a complex floating point Output Matrix
riVnum : Number of Vectors

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 16+ (((((4*((riCol/2)-1)+16)*riRow)+9)*riVnum)

Number of VLIW: 45

Restrictions: No of rows of Matrix A should be Odd and >=5

Function cfmatrixchol

```
#include "magic_chess.h"
```

Functions

- long **cfMatrixChol** (_c_float *rpacfInp, _c_float *rpacfOut_1, _c_float *rpacfOut_2, int riN)
  cfMatrixChol: Cholesky Decomposition

Function Documentation

long cfMatrixChol (_c_float * *rpacfInp*, _c_float * *rpacfOut_1*, _c_float * *rpacfOut_2*, int *riN*)

cfMatrixChol: Cholesky Decomposition

Decomposition of a Square matrix to get lower and upper triangular matrices

```
Y = Chol |A|
```

Parameters:

rpacfInp : Pointer to a complex floating point Input Matrix
rpacfOut_1 : Pointer to the complex floating point Output Matrix-I
rpacfOut_2 : Pointer to the complex floating point Output Matrix-II
riN : Dimension of the input matrix N

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 24+0.6667*N^3+19.0*N^2+68.833*N

Number of VLIW: 201

Restrictions: N should be > 4

User Info: #define UNROLL_FAC_1: disable this in order to activate the code with unroll factor of 2

Function cfmatrixdeterm

```
#include "magic_chess.h"
```

Functions

- long **cfMatrixDeterm** (_c_float *rpacfInp, int riN, _c_float *rpacfOut)
  cfMatrixDeterm: Determinant Computation

Function Documentation

long cfMatrixDeterm (_c_float * *rpacfInp*, int *riN*, _c_float * *rpacfOut*)

cfMatrixDeterm: Determinant Computation

Determinant of a N*N matrix can be computed

```
    Y = Det |A|

Algorithm:
     1.Partial pivoting
     2.Gaussian Elimination
```

Parameters:

rpacfInp : Pointer to a complex floating point Input Matrix
rpacfOut : Pointer to the complex floating point Output
riN : Dimension of the input matrix N

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 1.667*N^3 +15.5*N^2 + 111.833*N +76

Number of VLIW: 264

Restrictions: N should be > 3

User Info:

1. If N is even, then the preprocessor definition #define N_EVEN can be enabled, which would reduce the VLIW to 248 and will have a performance improvement
2. #define UNROLL_FAC_2: diasble this in order to de-activate the code with an unrolling factor of 2

Function cfmatrixinvert

`#include "magic_chess.h"`

Functions

- long **cfMatrixInvert** (_c_float *rpacfInp, int riN, _c_float *rpacfOut)
  fMatrixInvert: Inverse Computation

Function Documentation

long cfMatrixInvert (_c_float * *rpacfInp*, int *riN*, _c_float * *rpacfOut*)

fMatrixInvert: Inverse Computation

Computation of inverse of a Square Matrix

```
Y = Inverse |A|
```

Parameters:

rpafInp : Pointer to a floating point Input Matrix
rpafOut : Pointer to the floating point Output Matrix
riN : Dimension of the input matrix (N)

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 2.666*N^3+101*N^2+129.833*N+77

Number of VLIW: 448

Restrictions: N should be > 3

User Info: If N is even, then #define N_EVEN can be enabled which would reduce the net VLIW to 424 and some performance improvement with respect to cycle

Function cfmatrixmul

```
#include "magic_chess.h"
```

Functions

- long **cfMatrixMul** (_c_float *rpacfIn1, int riX, int riY, _c_float *rpacfIn2, int riZ, _c_float *rpacfOut)
  cfMatrixMul: Matrix multiplication

Function Documentation

long cfMatrixMul (_c_float * *rpacfIn1*, int *riX*, int *riY*, _c_float * *rpacfIn2*, int *riZ*, _c_float * *rpacfOut*)

cfMatrixMul: Matrix multiplication

multiplication of 2 complex floating point matrices

```
C[riX][riZ] = a[riX][riY]*b[riY][riZ]
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Matrix A
riX : No of Rows of Matrix A x
rpacfIn2 : Pointer to a complex floating point Matrix B
riY : No of coloumns of Matrix A/No of Rows of Matrix B y
riZ : No of coloumns of Matrix B z
rpacfOut : Pointer to a complex floating point Matrix C

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 19 + (((((4*(y-1)+14)*x)+11)*z)

Number of VLIW: 48

Restrictions: No of rows of Matrix B/no of coloumns of Matrix A can be Even/Odd and >=2

User Info: The function cfMatrixMul works for both even and odd values of CR_M1M2

Function cfmatrixmul_even

```
#include "magic_chess.h"
```

Functions

- long **cfMatrixMul_even** (_c_float *rpacfIn1, int riX, int riY, _c_float *rpacfIn2, int riZ, _c_float *rpacfOut)
  cfMatrixMul_even: Matrix multiplication

Function Documentation

long cfMatrixMul_even (_c_float * *rpacfIn1*, int *riX*, int *riY*, _c_float * *rpacfIn2*, int *riZ*, _c_float * *rpacfOut*)

cfMatrixMul_even: Matrix multiplication

Multiplication of 2 complex floating point matrices

```
C[riX][riZ] = a[riX][riY]*b[riY][riZ]
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Matrix A
riX : No of Rows of Matrix A x
rpacfIn2 : Pointer to a complex floating point Matrix B
riY : No of coloumns of Matrix A/No of Rows of Matrix B y
riZ : No of coloumns of Matrix B z
rpacfOut : Pointer to a complex floating point Matrix C

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 20+ ((((4*((y/2)-1)+14)*x)+13)*z)

Number of VLIW: 51

Restrictions: No of rows of Matrix B/No of coloumns of Matrix A should be Even and >=4

User Info: enable this code in main whenever CR_M1M2 is even in order to get better performance

Function cfmatrixmul_odd

`#include "magic_chess.h"`

Functions

- long **cfMatrixMul_odd** (_c_float *rpacfIn1, int riX, int riY, _c_float *rpacfIn2, int riZ, _c_float *rpacfOut)
  cfMatrixMul_odd: Matrix multiplication

Function Documentation

long cfMatrixMul_odd (_c_float * *rpacfIn1*, int *riX*, int *riY*, _c_float * *rpacfIn2*, int *riZ*, _c_float * *rpacfOut*)

cfMatrixMul_odd: Matrix multiplication

Multiplication of 2 complex floating point matrices

```
C[riX][riZ] = a[riX][riY]*b[riY][riZ]
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Matrix A
riX : No of Rows of Matrix A x
rpacfIn2 : Pointer to a complex floating point Matrix B
riY : No of coloumns of Matrix A/No of Rows of Matrix B y
riZ : No of coloumns of Matrix B z
rpacfOut : Pointer to a complex floating point Matrix C

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 20+ (((((4*((y/2)-1)+16)*x)+13)*z)

Number of VLIW: 53

Restrictions: No of rows of Matrix B/no of coloumns of Matrix A should be Odd and >=5

User Info: enable this code in main whenever CR_M1M2 is odd in order to get better performance

Function cfmatrixtrace

```
#include "magic_chess.h"
```

Functions

- long **cfMatrixTrace** (_c_float *rpacfIn, int riDim, _c_float *rpacfOut)
  cfMatrixTrace: Trace computation

Function Documentation

long cfMatrixTrace (_c_float * *rpacfIn*, int *riDim*, _c_float * *rpacfOut*)

cfMatrixTrace: Trace computation

Computation of trace of N*N complex matrix

```
          Output=Sum of all the diagonal elments.
```

Parameters:

rpacfIn : Pointer to a floating point Input Array
riDim : Dimension N of the square matrix
rpacfOut : Pointer to a floating point Output Array

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 51+(4*(N/4))+4*(N4)

Number of VLIW: 56

Restrictions: N should be >=4

Function cfmovescaleoffset

```
#include "magic_chess.h"
```

Functions

- long **cfMoveScaleOffset** (float *gfpIn, int giStrideInp, float *gfpOut, int giStrideOut, float *gfpScale, float *gfpOffset, int giLen)

  cfMoveScaleOffset: The values in the complex float array are multiplied with the complex scale value and a complex offset is added to it.

Function Documentation

long cfMoveScaleOffset (float * *gfpIn*, int *giStrideInp*, float * *gfpOut*, int *giStrideOut*, float * *gfpScale*, float * *gfpOffset*, int *giLen*)

cfMoveScaleOffset: The values in the complex float array are multiplied with the complex scale value and a complex offset is added to it.

```
gfpOut(k)=(gfpIn(k)*gfpScale + gfpOffset)

k=0.......giLen - 1
```

Parameters:

gfpIn : Pointer to a floating point input Array
gfpOut : Pointer to a floating point output Array
giStrideInp : Stride for the input Array
giStrideOut : Stride for the Output Array
gfpScale : Pointer to a floating point Scale value
gfpOffset : Pointer to a floating point Offset value
giLen : Element count N

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 17+N*1.25

Number of VLIW: 22

Restrictions: giLen should be a multiple of 4

Function cfmul

```
#include "magic_chess.h"
```

Functions

- long **cfMul** (float *gfpIn1, int giStrideInp1, float *gfpIn2, int giStrideInp2, float *gfpOut, int giStrideOut, int giLen)
  cfMul: Complex Multiplication of input Array1 and Array2 and the output is stored in the output Array

---

Function Documentation

long cfMul (float * *gfpIn1*, int *giStrideInp1*, float * *gfpIn2*, int *giStrideInp2*, float * *gfpOut*, int *giStrideOut*, int *giLen*)

cfMul: Complex Multiplication of input Array1 and Array2 and the output is stored in the output Array

```
gfpOut(k)=gfpIn1(k) * gfpIn2(k)

k=0.......giLen - 1
```

Parameters:

gfpIn1 : Pointer to a floating point input Array 1
gfpIn2 : Pointer to a floating point input Array 2
gfpOut : Pointer to a floating point output Array
giStrideInp1 : Stride for the input Array 1
giStrideInp2 : Stride for the input Array 2
giStrideOut : Stride for the Output Array
giLen : Element count N

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 18+N*1.5

Number of VLIW: 24

Restrictions: N should be a multiple of 4

Function cfmuladd

`#include "magic_chess.h"`

Functions

- long **cfMulAdd** (float *rpafIn1, int riStrideInp1, float *rpafIn2, int riStrideInp2, float *rpafIn3, int riStrideInp3, float *rpafOut, int riStrideOut, int riLen)
  cfmuladd: The two complex arrays are multiplied and added with another complex array.

Function Documentation

long cfMulAdd (float * *rpafIn1*, int *riStrideInp1*, float * *rpafIn2*, int *riStrideInp2*, float * *rpafIn3*, int *riStrideInp3*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

cfmuladd: The two complex arrays are multiplied and added with another complex array.

```
M(k)=A(k)*B(k);
Z(K)=C(k)+M(k); where  k=0...riLen-1
```

Parameters:

rpafIn1 : Pointer to a floating point Input Array 1
rpafIn2 : Pointer to a floating point Input Array 2
rpafIn3 : Pointer to a floating point Input Array 3
rpafOut : Pointer to a floating point Output Array
riStrideInp1 : Stride for the input Array 1
riStrideInp2 : Stride for the input Array 2
riStrideInp3 : Stride for the input Array 3
riStrideOut : Stride for the Output Array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 19+N*2

Number of VLIW: 27

Restrictions: Element count should be a multiple of 4

Function cfmulconj

```
#include "magic_chess.h"
```

Functions

- long **cfMulConj** (float *gfpIn1, int giStrideInp1, float *gfpIn2, int giStrideInp2, float *gfpOut, int giStrideOut, int giLen)
  cfMulConj: Complex Multiplication of input Array1 and conjuagate of Array2 and the output is stored in the output Array

---

Function Documentation

long cfMulConj (float * *gfpIn1*, int *giStrideInp1*, float * *gfpIn2*, int *giStrideInp2*, float * *gfpOut*, int *giStrideOut*, int *giLen*)

cfMulConj: Complex Multiplication of input Array1 and conjuagate of Array2 and the output is stored in the output Array

```
gfpOut(k)=gfpIn1(k) * conj(gfpIn2(k))

k=0.......giLen - 1
```

Parameters:

gfpIn1 : Pointer to a floating point input Array 1
gfpIn2 : Pointer to a floating point input Array 2
gfpOut : Pointer to a floating point output Array
giStrideInp1 : Stride for the input Array 1
giStrideInp2 : Stride for the input Array 2
giStrideOut : Stride for the Output Array
giLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 18+N*1.5

Number of VLIW: 24

Restrictions: Element count should be a multiple of 4

Function cfmulconjconj

```
#include "magic_chess.h"
```

Functions

- long **cfMulConjConj** (float *gfpIn1, int giStrideInp1, float *gfpIn2, int giStrideInp2, float *gfpOut, int giStrideOut, int giLen)
  cfMulConjConj: Complex conjugate Multiplication of input Array1 and Array2 and the output is stored in the output Array

---

Function Documentation

long cfMulConjConj (float * *gfpIn1*, int *giStrideInp1*, float * *gfpIn2*, int *giStrideInp2*, float * *gfpOut*, int *giStrideOut*, int *giLen*)

cfMulConjConj: Complex conjugate Multiplication of input Array1 and Array2 and the output is stored in the output Array

```
gfpOut(k)=conj(gfpIn1(k)) * conj(gfpIn2(k))

k=0.......giLen - 1
```

Parameters:

gfpIn1 : Pointer to a floating point input Array 1
gfpIn2 : Pointer to a floating point input Array 2
gfpOut : Pointer to a floating point output Array
giStrideInp1 : Stride for the input Array 1
giStrideInp2 : Stride for the input Array 2
giStrideOut : Stride for the Output Array
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 18+N*1.5

Number of VLIW: 24

Restrictions: Element count should be a multiple of 4

Function cfsquaremagn

```
#include "magic_chess.h"
```

Functions

- long **cfSquareMagn** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  cfsquaremagn: Complex number square magnitude

Function Documentation

long cfSquareMagn (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

cfsquaremagn: Complex number square magnitude

The Output array contains square magintude of the input complex array.

```
Z(k) = ((re.X(k)+re.X(k))^2+(im.X(k)+im.X(k))^2))          k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point Input Array
rpafOut : Pointer to a floating point Output Array
riStrideInp : Stride for the Input Array
riStrideOut : Stride for the Output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 17+N*1.25

Number of VLIW: 22

Restrictions: Element count should be a multiple of 4

Function cfvectbymatrixmul

```
#include "magic_chess.h"
```

Functions

- long **cfVectByMatrixMul** (_c_float *rpacfIn1, _c_float *rpacfIn2, int riRow, int riCol, _c_float *rpacfOut)
  cfVectByMatrixMul: Vector by Matrix multiplication

Function Documentation

long cfVectByMatrixMul (_c_float * *rpacfIn1*, _c_float * *rpacfIn2*, int *riRow*, int *riCol*, _c_float * *rpacfOut*)

cfVectByMatrixMul: Vector by Matrix multiplication

Product of a complex vector with a complex matrix

```
C[riCol] = a[riRow]*b[riRow][riCol]
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Vector
rpacfIn2 : Pointer to a complex floating point Matrix
riRow : No of coloumns of Vector/No of Rows of Matrix
riCol : No of coloumns of Matrix
rpacfOut : Pointer to a complex floating point Matrix

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 14 + riCol*(10+4*riRow)

Number of VLIW: 32

Restrictions: No of rows of Matrix /no of Vector length A can be Even/Odd and >=2

User Info: The function cfVectByMatrixMul works for both even and odd values of riRow. Even and Odd flavors gives better performance when compared to this.

Function cfvectbymatrixmul_even and cfvectbymatrixmul_odd

```
#include "magic_chess.h"
```

Functions

- long **cfVectByMatrixMul_even** (_c_float *rpacfIn1, _c_float *rpacfIn2, int riRow, int riCol, _c_float *rpacfOut)
  cfVectByMatrixMul_even: Vector by Matrix multiplication Even Flavour

- long **cfVectByMatrixMul_odd** (_c_float *rpacfIn1, _c_float *rpacfIn2, int riRow, int riCol, _c_float *rpacfOut)
  cfVectByMatrixMul_odd: Vector by Matrix multiplication Odd Flavour

---

Function Documentation

long cfVectByMatrixMul_even (_c_float * *rpacfIn1*, _c_float * *rpacfIn2*, int *riRow*, int *riCol*, _c_float * *rpacfOut*)

cfVectByMatrixMul_even: Vector by Matrix multiplication Even Flavour

Product of a complex vector with a complex matrix

```
C[riCol] = a[riRow]*b[riRow][riCol]
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Vector
rpacfIn2 : Pointer to a complex floating point Matrix
riRow : No of coloumns of Vector/No of Rows of Matrix
riCol : No of coloumns of Matrix
rpacfOut : Pointer to a complex floating point Matrix

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 14 + riCol*(11+2*riRow)

Number of VLIW: 33

Restrictions: No of rows of Matrix B/No of coloumns of Matrix A should be Even and >=4

long cfVectByMatrixMul_odd (_c_float * *rpacfIn1*, _c_float * *rpacfIn2*, int *riRow*, int *riCol*, _c_float * *rpacfOut*)

cfVectByMatrixMul_odd: Vector by Matrix multiplication Odd Flavour

Product of a complex vector with a complex matrix

```
C[riCol] = a[riRow]*b[riRow][riCol]
```

Parameters:

rpacfIn1 : Pointer to a complex floating point input Vector
rpacfIn2 : Pointer to a complex floating point Matrix
riRow : No of coloumns of Vector/No of Rows of Matrix
riCol : No of coloumns of Matrix
rpacfOut : Pointer to a complex floating point Matrix

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: : 14 + riCol*(11+2*riRow)

Number of VLIW: : 35

Restrictions: : No of rows of Matrix B/no of coloumns of Matrix A should be Odd and >=5

Function cfxcorr

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **cfXcorr** (float *rpafX, int riStrideX, float *rpafY, int riStrideY, float *rpafZ, int riStrideZ, int riLen, int riNcoeff)
  cfXcorr: cross correlation or auto correlation computation

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long cfXcorr (float * *rpafX*, int *riStrideX*, float * *rpafY*, int *riStrideY*, float * *rpafZ*, int *riStrideZ*, int *riLen*, int *riNcoeff*)

cfXcorr: cross correlation or auto correlation computation

Computation of correlation between two float arrays

```
                  _
                 |
                 |  sumk( X(k-i) * conj(Y(k)) ); k= 0 to N-i-1, i= 0 to Ncorr/2
Rxy(i) =|
                 |  Ryx(-i); i= -Ncorr/2 to -1
                 |_

 Z(i)   = Rxy(i-Ncorr/2)
```

Parameters:

rpafX : Pointer to the first floating point input array
riStrideX : Stride for the first input array
rpafY : Pointer to the second floating point input array
riStrideY : Stride for the second input array
rpafZ : Pointer to the floating point output array
riStrideZ : Stride for the output array
riLen : (N in the formula above) Minimum of the size of X and Y
riNcoeff : (Ncorr in the formula above) Number of outputs to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles:

- case 1: riLen is odd : 26 + 9.50*riNcoeff + riLen*riNcoeff - 0.25*riNcoeff^2
- case 2: riLen is even: 26 + 8.75*riNcoeff + riLen*riNcoeff - 0.25*riNcoeff^2

Number of VLIW: 89

Restrictions: riNcoeff must be a multiple of 4 and riLen must be greater than or equal to 3

User Info:

1. Minimum value of VSIZEX and VSIZEY should be 3
2. Ncorr should be a muliple of 4 and in the range [1,((2*min(VSIZEX,VSIZEY))-1)]
3. For autocorrelation include the same files for gafX and gafY

Function clconjscaleoffset

```
#include "magic_chess.h"
```

Functions

- long **clConjScaleOffset** (long *glpIn, int giStrideInp, long *glpOut, int giStrideOut, long *glpScale, long *glpOffset, int giLen)
  clConjScaleOffset: The conjugate of the contents of the complex array is computed and then multiplied with the scale value and then offset is added to it

---

Function Documentation

long clConjScaleOffset (long * *glpIn*, int *giStrideInp*, long * *glpOut*, int *giStrideOut*, long * *glpScale*, long * *glpOffset*, int *giLen*)

clConjScaleOffset: The conjugate of the contents of the complex array is computed and then multiplied with the scale value and then offset is added to it

```
glpOut(k)=glpScale*conj(glpIn(k))+glpOffset

k=0.......giLen - 1
```

Parameters:

glpIn : Pointer to a long input Array
glpOut : Pointer to a long output Array
giStrideInp : Stride for the input Array
giStrideOut : Stride for the Output Array
glpScale : Pointer to long Scale Value
glpOffset : Pointer to long OffsetValue
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 17+N*1.25

Number of VLIW: 22

Restrictions: Element count should be a multiple of 4

Function clmovescaleoffset

```
#include "magic_chess.h"
```

Functions

- long **clMoveScaleOffset** (long *glpIn, int giStrideInp, long *glpOut, int giStrideOut, long *glpScale, long *glpOffset, int giLen)
  clMoveScaleOffset:The complex long value of the input array is multiplied with the complex scale value and then added with the complex scale offset.

Function Documentation

long clMoveScaleOffset (long * *glpIn*, int *giStrideInp*, long * *glpOut*, int *giStrideOut*, long * *glpScale*, long * *glpOffset*, int *giLen*)

clMoveScaleOffset:The complex long value of the input array is multiplied with the complex scale value and then added with the complex scale offset.

```
glpOut(k)=(glpIn(k)*glpScale + glpOffset)

k=0.......giLen - 1
```

Parameters:

glpIn : Pointer to a long input Array
glpOut : Pointer to a long output Array
giStrideInp : Stride for the input Array
giStrideOut : Stride for the Output Array
glpScale : Pointer to a long Scale value
glpOffset : Pointer to a long Offset value
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 17+N*1.25

Number of VLIW: 22

Restrictions: Element count should be a multiple of 4

Function vfexp

```
#include "magic_chess.h"
```

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfExp** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfExp: Computing Exponential of vector float array

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfExp (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfExp: Computing Exponential of vector float array

Exponential of input vector float array is calculated

```
    Y (k) = e^X(k)          k=Nelements
```

Parameters:

rpafIn : Pointer to a vector floating point array
riStrideInp : Input Stride
rpafOut : Pointer to a vector floating point Array
riStrideOut : Output Stride
riLen : Number of Elements

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+22*riLen

Number of VLIW: 99

Restrictions: Array should have at least 2 vector elements

Function vflog

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)
- #define **DOMAIN_CHECK**

Functions

- long **vfLog** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, float rfBaseConversion, int riLen)
  vfLog: Computing Logarithm of vector float array

Variables

- VFLOAT **gfCoeff** [22]
- long **glDomainChkFlagLn**

Define Documentation

#define DOMAIN_CHECK

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfLog (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, float *rfBaseConversion*, int *riLen*)

vfLog: Computing Logarithm of vector float array

Logarithm of input vector float array is calculated

```
    Y (k) = log(X(k))              k=Nelements
```

Parameters:

rpafIn : Pointer to the vector floating point Input array
riStrideInp : Stride for the input Array
rpafOut : Pointer to the vector floating point Output Array
riStrideOut : Stride for the Output Array
rfBaseConversion,: Base conversion Parameter to convert base of logarithm
riLen : Number of Input elements

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 93+26*riLen

Number of VLIW: 146

Restrictions: Array should have at least 2 vector elements

User Info: The valid inputs for log(x) are x>0. For all invalid inputs this code sets the domain check flag and outputs a zero. To disable the domain check, recompile the code undefining the DOMAIN_CHECK variable

---

Variable Documentation

VFLOAT gfCoeff[22]

```
Initial value:
{
        1.0000000038428374 ,    1.0000000038428374,
        -0.49999998873765605,   -0.49999998873765605,
        0.33333257583292664,    0.33333257583292664,
        -0.25000052334758038,   -0.25000052334758038,
         0.20004242170148245,   0.20004242170148245,
        -0.16668554940406466,   -0.16668554940406466,
        0.14188553811982274,    0.14188553811982274,
        -0.12376045290147886,   -0.12376045290147886,
        0.12045167665928602 ,   0.12045167665928602,
        -0.11897297506220639,   -0.11897297506220639,
        0.067073686048388481 ,  0.067073686048388481
}
```

long glDomainChkFlagLn

Function vfsqrt

`#include "magic_chess.h"`

Defines

- #define **DOMAIN_CHECK**

Functions

- long **vfSqrt** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfSqrt: *Square Root Computation*

Variables

- long **glDomainChkFlagSqrt**

Define Documentation

#define DOMAIN_CHECK

Function Documentation

long vfSqrt (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfSqrt: Square Root Computation

Computes the square root of the input vector float array

```
Y(k) = sqrt(X(k)        k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point input Array
rpafOut : Pointer to a floating point output Array
riStrideInp : Stride for the input Array
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+N*15

Number of VLIW: 116

Restrictions: Array should have at least 1 vector element

User Info: The domain check flag is set whenever the input is negative.

Variable Documentation

long glDomainChkFlagSqrt

Function fconv

```
#include "magic_chess.h"
```

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)
- #define **ODD_VECLEN**

Functions

- long **fConv** (float *rpafInX, float *rpafInCoeffH, float *rpafOut, int riVecLen, int riFilterLen, int riTransient)
  fConv: Computing float Convolution

Define Documentation

#define ODD_VECLEN

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long fConv (float * *rpafInX*, float * *rpafInCoeffH*, float * *rpafOut*, int *riVecLen*, int *riFilterLen*, int *riTransient*)

fConv: Computing float Convolution

Convolution of input float array is calculated with the input float Coefficent

```
Y (k) = sumi( X(i)*H(k-i) )

i = 0 to M-1
k = 0 to (N+M-1)

N:- input vector length
M:- input filter length
```

Parameters:

rpafInX : Pointer to a floating point input array
rpafInCoeffH : Pointer to a floating point Coefficent array
rpafOut : Pointer to a floating point Array
riVecLen : Number of input N
riFilterLen : Number of Coefficent M
riTransient : Integer value used to compute or not the transient codes of the convolution: if riTransient=0 the transient isn't computed, otherwise it's calculated

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: Input/Output transients: (M-1) * (15+2*M)

Number of cycles: Steady state:

- Case1 N is odd : 65 + 2*M + (N-M-1)*(10.5 + 1*(M-2))
- Case1 N is even: 94 + 4*M + (N-M-2)*(10.5 + 1*(M-2))

Number of VLIW:

- Case1 N is odd :113
- Case2 N is even:146

Restrictions: M must be even and should be less than N

User Info: In order to have better performances, enable/disable ODD_VECLEN depending on the even/odd nature of N

Function fconv2d

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)
- #define **PIPE**

Functions

- long **fConv2d** (float *rpafInA, int riRowLenA, int riColLenA, float *rpafInKernelH, float *rpafOut, int riKOrder)
  fConv2d: Computing 2D-Convolution

Define Documentation

#define PIPE

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long fConv2d (float * *rpafInA*, int *riRowLenA*, int *riColLenA*, float * *rpafInKernelH*, float * *rpafOut*, int *riKOrder*)

fConv2d: Computing 2D-Convolution

2D-Convolution of input float array is calculated with the input Coefficent

```
        C(r,c) = sumi(sumj( H[k-1-i][k-1-j] * A[r+i][c+j] ))

    r = 0 to M-k+1
    i = 0 to k-1
    c = 0 to N-k+1
    j = 0 to k-1

    M = No. of rows in A
    N = No. of columns in A
    k = Order of H
```

Parameters:

rpafInA : Pointer to a floating point input matrix
riRowLenA : Number of rows in matrix, A, to be convolved (M)
riColLenA : Number of columns in matrix, A, to be convolved (N)
rpafInKernelH : Pointer to a floating point input matrix

rpafOut : Pointer to the floating point output matrix c of the order (M-k+1)*(N-k+1)
riKOrder : Kernel size (k)

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 30+(M-k+1) * (13+(N-k+1) * (12+k*(8.5+k)))

Number of VLIW: 80

Restrictions: k must be even and greater than 2

User Info:

1. #define PIPE: define, when k >= 5, for better performances
2. Set input lengths which satisfy N >= M > k

Function ffirnlms

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **fFIRnlms** (float *rpafInX, float *rpafKerCoeffH, float *rpafRefOut, float *rpafDelayBuff, int riSampleLen, int riKernelLen, float rfAdapCoeff)
  fFIRnlms: Computation of a pair of adaptive FIR filter

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long fFIRnlms (float * *rpafInX*, float * *rpafKerCoeffH*, float * *rpafRefOut*, float * *rpafDelayBuff*, int *riSampleLen*, int *riKernelLen*, float *rfAdapCoeff*)

fFIRnlms: Computation of a pair of adaptive FIR filter

FIR filter coefficient computed using Least Mean Square Algorithm

```
T [n] = rpafDelayBuff[n- k]*rpafKerCoeffH[k]

e=T[n]- rpafRefOut[n]

E =Summation (rpafDelayBuff[k]^2)

C =B*e/E

rpafKerCoeffH[k] = rpafKerCoeffH[k] +C*rpafDelayBuff[k]

where
k=0...riKernelLen-1
n=riKernelLen.....riSampleLen-1
```

Parameters:

rpafInX : Pointer to a real floating point input
rpafKerCoeffH : Pointer to a real floating point FIR kernel
rpafRefOut : Pointer to a real floating point vector containing the desired output
rpafDelayBuff : Pointer to delay memory of length riKernelLen

riSampleLen : Number of input samples
riKernelLen : Order of the filter
rfAdapCoeff : Adaption coefficient

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles : 49 + 2*riKernelLen + (26*riKernelLen+49)*(riSampleLen-riKernelLen+1)

Number of VLIW : 236

Restrictions: riKernelLen must be an even value multiple of 4.

User Info:

Function fft1024

`#include "magic_chess.h"`

Functions

- void **fft32m** (_c_float *, _c_float *, _c_float *, int)
  fft32m: Performs the second layer of a N-point FFT by means of M executions of the modified 32-point FFT.

- long **fft1024** (_c_float *W, _c_float *in, _c_float *temp, _c_float *out)
  fft1024: Computes the 1024-point FFT of a complex vector.

Function Documentation

long fft1024 (_c_float * *W*, _c_float * *in*, _c_float * *temp*, _c_float * *out*)

fft1024: Computes the 1024-point FFT of a complex vector.

Function **fft1024()** is the mixed radix implementation of the 1024-point FFT. Function **fft32m()** is used as a component block. If more than one fft size is used in an application, the module **fft32m()** is shared among them.

Parameters:

W : Pointer to the array of 512 trigonometric coefficients: exp(-i*2*pi*n/1024) with n = 0, 1, .. 511.
in : Pointer to the input complex vector (size 1024).
temp : Pointer to a temporary auxiliary vector (size 1024) for FFT computation.
out : Pointer to the output complex vector (size 1024); after function call, the vector pointed by out contains the FFT of the vector pointed by in.

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 5504

Number of VLIW: 158 (+ 194 for called function fft32m)

Restrictions: vector pointed by temp cannot coincide with vector pointed by out, so only the following vector combinations are allowed: in != temp != out (the 3 vectors are all different); in = temp != out; in = out != temp;

Number of stack locations used: 12

void fft32m (_c_float * *W_base_addr*, _c_float * *input_base_addr*, _c_float * *output_base_addr*, int *M*)

fft32m: Performs the second layer of a N-point FFT by means of M executions of the modified 32-point FFT.

Parameters:

W_base_addr : Pointer to the array of N/2 trigonometric coefficients: exp(-i*2*pi*n/N) with n = 0, 1, .. (N/2 - 1).
input_base_addr : Pointer to the array (size N) of input complex data.

output_base_addr : Pointer to the array (size N) where output data will be written.
M : Requested number of executions of the modified 32-point FFT.

Returns:

void.

Number of cycles: 124 + 84*M

Number of VLIW: 194

Restrictions: vector pointed by input_base_addr cannot coincide with vector pointed by output_base_addr.

Number of stack locations used: 6

Function fft32m

Functions

- void **fft32m** (_c_float *W_base_addr, _c_float *input_base_addr, _c_float *output_base_addr, int M)
  fft32m: Performs the second layer of a N-point FFT by means of M executions of the modified 32-point FFT.

Function Documentation

void fft32m (_c_float * *W_base_addr*, _c_float * *input_base_addr*, _c_float * *output_base_addr*, int *M*)

fft32m: Performs the second layer of a N-point FFT by means of M executions of the modified 32-point FFT.

Parameters:

W_base_addr : Pointer to the array of N/2 trigonometric coefficients: exp(-i*2*pi*n/N) with n = 0, 1, .. (N/2 - 1).
input_base_addr : Pointer to the array (size N) of input complex data.
output_base_addr : Pointer to the array (size N) where output data will be written.
M : Requested number of executions of the modified 32-point FFT.

Returns:

void.

Number of cycles: 124 + 84*M

Number of VLIW: 194

Restrictions: vector pointed by input_base_addr cannot coincide with vector pointed by output_base_addr.

Number of stack locations used: 6

Function flevinson

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **fLevinson** (float *rpafInAutoCorR, float *rpafLPCOut, float *rpafScalarOut, int riCoeffNum)
  fLevinson: Solving of Levinson-Durbin equations.

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long fLevinson (float * *rpafInAutoCorR*, float * *rpafLPCOut*, float * *rpafScalarOut*, int *riCoeffNum*)

fLevinson: Solving of Levinson-Durbin equations.

The function levinson solves the p th order system of linear equations

```
E(0) = rpafInAutoCorR(0)

k(i)={rpafInAutoCorR(i)-sumation [rpafLPCOut(j)(i- 1) x rpafInAutoCorR(i-j)]}/E(i-1)

rpafLPCOut(i)(i)= k(i)

rpafLPCOut(j)(i)= rpafLPCOut(j)(i - 1)- k(i)x rpafLPCOut(i - j)(i - 1)

E(i)= (1- k(i)^2) x E(i-1)

where
i=0...riKernelLen-1
j=1.....i-1
```

Parameters:

rpafInAutoCorR : pointer to the autocorrelation input vector.
rpafLPCOut : pointer to the output vector.
rpafScalarOut : pointer to the scalar output.
riCoeffNum : Number of coefficients to be computed.

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles : 995 (for riCoeffNum = 11)

Number of VLIW : 148

Restrictions : riCoeffNum should be less than the length of InAutoCorR vector

Function flpc2cepstr

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **fLPC2Cepstr** (float *rpafLPC, float *rpafCC, int riLPCLen, int riCCLen)
  fLPC2Cepstr: LPC coeffecients to Cepstral Coefficients computation

---

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long fLPC2Cepstr (float * *rpafLPC*, float * *rpafCC*, int *riLPCLen*, int *riCCLen*)

fLPC2Cepstr: LPC coeffecients to Cepstral Coefficients computation

Computation of Cepstral coeffecients from LPC coefficients

```
          _
         |
         | -a[0]                              ; m=0
         |
C[m] =   | -a[m] + sumk( (k/m)*C[k]*(-a[m-k]) ); k= 1 to m-1, 1<= m <LPCLENGTH
          |
         |  sumk( (k/m)*C[k]*(-a[m-k]) )       ; k= 1 to m-1, LPCLENGTH<= m <CCLENGTH
         |_
```

Parameters:

rpafLPC : Pointer to the floating point input LPC coefficients
rpafCC : Pointer to the floating point output Cepstral coefficients
riLPCLen : Number of input coefficients
riCCLen : Number of output coefficients

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles:

- case 1> riLPCLen is odd : 2*riCCLen*riLPCLen - riLPCLen^2 + 40*riCCLen + 9*riLPCLen - 71
- case 2> riLPCLen is even: 2*riCCLen*riLPCLen - riLPCLen^2 + 42*riCCLen + 7*riLPCLen - 72

Number of VLIW: 163

Restrictions: Array should have at least 1 vector element

User Info: CEPSTRAL_LENGTH should be greater than LPC_LENGTH

Function fmatrixadd

`#include "magic_chess.h"`

Defines

- #define **ODD**

Functions

- long **fMatrixAdd** (float *rpafIn1, float *rpafIn2, int riRows, int riCols, float *rpafOut)
  fMatrixAdd: Matrix Addition

Define Documentation

#define ODD

Function Documentation

long fMatrixAdd (float * *rpafIn1*, float * *rpafIn2*, int *riRows*, int *riCols*, float * *rpafOut*)

fMatrixAdd: Matrix Addition

Addition of 2 floating point matrices

```
C[riRows][riCols] = a[riRows][riCols]+b[riRows][riCols]            k=0...riRows*riCols-1
```

Parameters:

rpafIn1 : Pointer to a floating point input Matrix A
rpafIn2 : Pointer to a floating point input Matrix B
riRows : No of rows of Matrix A and Matrix B
riCols : No of cols of Matrix A and Matrix B
rpafOut : Pointer to a floating point output Matrix C

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: Case1: riRows*riCols is odd : 33 + 0.875*(riRows*riCols-1) Case2: riRows*riCols is even: 19 + 0.875*(riRows*riCols)

Number of VLIW: Case1: riRows*riCols is odd : 40 Case2: riRows*riCols is even: 26

Restrictions: Minimum elements in the matrix (riRows*riCols) should be 8, and (riRows*riCols) or (riRows*riCols-1) should be a multiple of 8.

User Info: #define ODD:Disable when the product riRows*riCols is even to save VLIW

Function fmatrixbyvectsmul

```
#include "magic_chess.h"
```

Functions

- long **fMatrixByVectsMul** (float *rpafIn1, int riRow, int riCol, float *rpafIn2, float *rpafOut, int riVnum)
  fMatrixByVectsMul: Matrix by Vector multiplication

- long **fMatrixByVectsMul_even** (float *rpafIn1, int riRow, int riCol, float *rpafIn2, float *rpafOut, int riVnum)
  fMatrixByVectsMul_even: Matrix by Vector multiplication -Even Flavour

---

Function Documentation

long fMatrixByVectsMul (float * *rpafIn1*, int *riRow*, int *riCol*, float * *rpafIn2*, float * *rpafOut*, int *riVnum*)

fMatrixByVectsMul: Matrix by Vector multiplication

Product of a floating point matrix with a set of vectors

```
C[riRow][riVnum] = a[riRow][riCol]*b[riCol][riVnum]
```

Parameters:

rpafIn1 : Pointer to a floating point input Matrix
riRow : No of Rows of Matrix A
riCol : No of coloumns of Matrix A/No of Rows of Vector
rpafIn2 : Pointer to a floating point Vector arrray
rpafOut : Pointer to a floating point Array
riVnum : Number of Vector

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 15 + (((4*riCol+9)*riRow)+8)*riVnum

Number of VLIW: 40

Restrictions: No of rows of Matrix A should be >=2

User Info: The even flavors give a better performance when (COL is even) compared to this general optimization. This code works for both even and odd values of input

long fMatrixByVectsMul_even (float * *rpafIn1*, int *riRow*, int *riCol*, float * *rpafIn2*, float * *rpafOut*, int *riVnum*)

fMatrixByVectsMul_even: Matrix by Vector multiplication -Even Flavour

Product of a floating point matrix with a set of vectors

```
C[riRow][riVnum] = a[riRow][riCol]*b[riCol][riVnum]
```

Parameters:

rpafIn1 : Pointer to a floating point input Matrix
riRow : No of Rows of Matrix A
riCol : No of coloumns of Matrix A
rpafIn2 : Pointer to a floating point Vector array
rpafOut : Pointer to a floating point Output Matrix
riVnum : Number of Vector

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 17 + (((2*riCol+13)*riRow)+8)*riVnum

Number of VLIW: 42

Restrictions: No of rows of Matrix A should be Even and >=4

User Info: #define UNROLL_FAC_2:disable this to activate the code with no-unroll

Function fmatrixdeterm

`#include "magic_chess.h"`

Defines

- #define **UNROLL_FAC2**

Functions

- long **fMatrixDeterm** (float *rpafInp, int riN, float *rpafOut)
  fMatrixDeterm: Determinant Computation

Define Documentation

#define UNROLL_FAC2

Function Documentation

long fMatrixDeterm (float * *rpafInp*, int *riN*, float * *rpafOut*)

fMatrixDeterm: Determinant Computation

Determinant of a N*N matrix can be computed

```
    Y = Det |A|

Algorithm:
    1.Partial pivoting
    2.Gaussian Elimination
```

Parameters:

rpafInp : Pointer to a floating point Input Matrix
rpafOut : Pointer to the floating point Output
riN : Dimension of the input matrix (N)

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: N^3 + 24.75*N^2 + 131.0*N + 79.25

Number of VLIW: 293

Restrictions: N should be > 3

User Info: #define UNROLL_FAC_2: enable this in order to activate the code with an unrolling factor of 2

Function fmatrixinvert

`#include "magic_chess.h"`

Functions

- long **fMatrixInvert** (float *rpafInp, int riN, float *rpafOut)
  fMatrixInvert: Inverse Computation

Variables

- long **glZeroDeterminant**

Function Documentation

long fMatrixInvert (float * *rpafInp*, int *riN*, float * *rpafOut*)

fMatrixInvert: Inverse Computation

Computation of inverse of a Square Matrix

```
Y = Inverse |A|
```

Parameters:

rpafInp : Pointer to a floating point Input Matrix
rpafOut : Pointer to the floating point Output Matrix
riN : Dimension of the input matrix (N)

Returns:

0 if succesful, !=0 otherwise.

Number of cycles:

- case1: riN is odd: 2.667*N^3 + 96.5*N^2 + 138.833*N + 103.5
- case2: riN is even: 2.667*N^3 + 97.5*N^2 + 139.833*N + 106.0

Number of VLIW: 448

Restrictions: N should be > 3

User Info:

1. If N is even, then #define N_EVEN can be enabled which would reduce the net VLIW to 418 and some performance improvement with respect to cycle
2. The cycle calculation is for the worst case, assuming that the input is such that all the if conditions are true
3. If the input matrix has a zero determinant, it implies that an inverse of such matrix does not exist. If this is the case the function sets the global variable glZeroDeterminant and the output values are all not proper

---

Variable Documentation

long glZeroDeterminant

Function fmatrixmul

```
#include "magic_chess.h"
```

Functions

- long **fMatrixMul** (float *rpafIn1, int riX, int riY, float *rpafIn2, int riZ, float *rpafOut)
  fMatrixMul: Matrix multiplication

Function Documentation

long fMatrixMul (float * *rpafIn1*, int *riX*, int *riY*, float * *rpafIn2*, int *riZ*, float * *rpafOut*)

fMatrixMul: Matrix multiplication

multiplication of 2 floating point matrices

```
C[riX][riZ] = a[riX][riY]*b[riY][riZ]
```

Parameters:

rpafIn1 : Pointer to a floating point input Matrix A
riX : No of rows of Matrix A (x)
riY : No of coloumns of Matrix A and No of rows of Matrix B (y)
rpafIn2 : Pointer to a floating point input Matrix B
riZ : No of coloumns of Matrix B (z)
rpafOut : Pointer to a floating point output Matrix C

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 16 + z(11+x(9+4y))

Number of VLIW: 44

Restrictions: No of rows of Matrix B/no of coloumns of Matrix A can be Even/Odd and >=2

User Info: The function fMatrixMul works for both even and odd values of CR_M1M2

Function fmatrixmul_even and fmatrixmul_odd

```
#include "magic_chess.h"
```

Functions

- long **fMatrixMul_even** (float *rpafIn1, int riX, int riY, float *rpafIn2, int riZ, float *rpafOut)
  fMatrixMul_even: Matrix multiplication

- long **fMatrixMul_odd** (float *rpafIn1, int riX, int riY, float *rpafIn2, int riZ, float *rpafOut)
  fMatrixMul_odd: Matrix multiplication

Function Documentation

long fMatrixMul_even (float * *rpafIn1*, int *riX*, int *riY*, float * *rpafIn2*, int *riZ*, float * *rpafOut*)

fMatrixMul_even: Matrix multiplication

Multiplication of 2 floating point matrices

```
C[riX][riZ] = a[riX][riY]*b[riY][riZ]
```

Parameters:

rpafIn1 : Pointer to a floating point input Matrix A
riX : No of rows of Matrix A (x)
riY : No of coloumns of Matrix A and No of rows of Matrix B (y)
rpafIn2 : Pointer to a floating point input Matrix B
riZ : No of coloumns of Matrix B (z)
rpafOut : Pointer to a floating point output Matrix C

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 18 + z(13+x(10+2y))

Number of VLIW: 49

Restrictions: No of columns of Matrix A should be even and >=4

long fMatrixMul_even_vf (float * *rpafIn1*, int *riX*, int *riY*, float * *rpafIn2*, int *riZ*, float * *rpafOut*)

fMatrixMul_even_vf: Matrix multiplication

Multiplication of 2 floating point matrices

```
C[riX][riZ] = a[riX][riY]*b[riY][riZ]
```

Parameters:

rpafIn1 : Pointer to a floating point input Matrix A
riX : No of rows of Matrix A (x)
riY : No of coloumns of Matrix A and No of Rows of Matrix B (y)
rpafIn2 : Pointer to a floating point input Matrix B
riZ : No of coloumns of Matrix B (z)
rpafOut : Pointer to a floating point output Matrix C

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 20 + x(16+y(10.5 + z))

Number of VLIW: 71

Restrictions: No of columns of Matrix A and No of coloumns of Matrix B should be Even and >2

User Info: Note: This function is a further optimization whenever both the matrices have even number of columns. All operations are done using vector float numbers.

long fMatrixMul_odd (float * *rpafIn1*, int *riX*, int *riY*, float * *rpafIn2*, int *riZ*, float * *rpafOut*)

fMatrixMul_odd: Matrix multiplication

Multiplication of 2 point matrices

```
C[riX][riZ] = a[riX][riY]*b[riY][riZ]
```

Parameters:

rpafIn1 : Pointer to a floating point input Matrix A
riX : No of rows of Matrix A (x)
riY : No of coloumns of Matrix A and No of rows of Matrix B (y)
rpafIn2 : Pointer to a floating point Matrix B
riZ : No of coloumns of Matrix B (z)
rpafOut : Pointer to a floating point output Matrix C

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 18 + z(13+x(8 + 2y))

Number of VLIW: 49

Restrictions: No of columns of Matrix A should be Odd and >=3

Function fmatrixtrace

```
#include "magic_chess.h"
```

Functions

- long **fMatrixTrace** (float *rpafIn, int riDim, float *rpafOut)
  fMatrixTrace: Trace computation

Function Documentation

long fMatrixTrace (float * *rpafIn*, int *riDim*, float * *rpafOut*)

fMatrixTrace: Trace computation

Computation of trace of N*N matrix

```
        Output=Sum of all the diagonal elments.
```

Parameters:

rpafIn : Pointer to a floating point Input Array
riDim : Dimension of the square matrix(Row/coloumn)>=2
rpafOut : Pointer to a floating point Output Array

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 44 + 4* [(int(riDim/4)) + (riDim/4)]

Number of VLIW: 52

Restrictions: N should be >=4

Function fmean

`#include "magic_chess.h"`

Defines

- #define **UNROLL_FAC_8**

Functions

- long **fMean** (float *gfpIn, int giStrideInp, float *gfpOut, int giLen)
  fMean: Computation of Mean of the float input Array

Define Documentation

#define UNROLL_FAC_8

Function Documentation

long fMean (float * *gfpIn*, int *giStrideInp*, float * *gfpOut*, int *giLen*)

fMean: Computation of Mean of the float input Array

```
gfpOut=Summation(gfpIn(k))/N

k=0.......giLen - 1
N= The number of input(giLen) by 4
```

Parameters:

gfpIn : Pointer to a floating point input Array
gfpOut : Pointer to a floating point output
giStrideInp : Stride for the input Array
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise

Number of cycles: 41+N*0.5

Number of VLIW: 45

Restrictions: Element count should be a multiple of 8

User Info: #define UNROLL_FAC_8 : Enable this to have unroll factor 8. This gives better performance when compared to unroll factor 4

Function fsum

```
#include "magic_chess.h"
```

Defines

- #define **UNROLL_FAC_8**

Functions

- long **fSum** (float *gfpIn, int giStrideInp, float *gfpOut, int giLen)
  fSum: Sum of the elements of input Array

Define Documentation

#define UNROLL_FAC_8

Function Documentation

long fSum (float * *gfpIn*, int *giStrideInp*, float * *gfpOut*, int *giLen*)

fSum: Sum of the elements of input Array

```
gfpOut=Summation(gfpIn(k))

k=0.......giLen - 1
```

Parameters:

gfpIn : Pointer to a floating point input Array
gfpOut : Pointer to a floating point output
giStrideInp : Stride for the input Array
giLen : Element count (N)

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 32+N*0.5

Number of VLIW: 36

Restrictions: Element count should be a multiple of 8

User Info: #define UNROLL_FAC_8 : Enable this to have unroll factor 8. This gives better performances when compared to unroll factor 4

Function fvectbymatrixmul

```
#include "magic_chess.h"
```

Functions

- long **fVectByMatrixMul** (float *rpafIn1, float *rpafIn2, int riRow, int riCol, float *rpafOut)
  fVectByMatrixMul: Vector by Matrix multiplication

- long **fVectByMatrixMul_even** (float *rpafIn1, float *rpafIn2, int riRow, int riCol, float *rpafOut)
  fVectByMatrixMul_even: Vector by Matrix multiplication Even Flavour

- long **fVectByMatrixMul_odd** (float *rpafIn1, float *rpafIn2, int riRow, int riCol, float *rpafOut)
  fVectByMatrixMul_odd: Vector by Matrix multiplication Odd Flavour

---

Function Documentation

long fVectByMatrixMul (float * *rpafIn1*, float * *rpafIn2*, int *riRow*, int *riCol*, float * *rpafOut*)

fVectByMatrixMul: Vector by Matrix multiplication

Product of a float vector with a float matrix

```
C[riCol] = a[riRow]*b[riRow][riCol]
```

Parameters:

rpafIn1 : Pointer to a floating point input Vector
rpafIn2 : Pointer to a floating point Matrix
riRow : No of coloumns of Vector/No of Rows of Matrix
riCol : No of coloumns of Matrix
rpafOut : Pointer to a floating point output Matrix

Returns:

0 if succesful, !=0 otherwhise

Number of cycles: : 12 + riCol*(15+4*riRow)

Number of VLIW: 35

Restrictions: None

User Info: The function fVectByMatrixMul works for both even and odd values of riRow. Even and Odd flavors give better performance when compared to this.

long fVectByMatrixMul_even (float * *rpafIn1*, float * *rpafIn2*, int *riRow*, int *riCol*, float * *rpafOut*)

fVectByMatrixMul_even: Vector by Matrix multiplication Even Flavour

Product of a float vector with a float matrix

```
C[riCol] = a[riRow]*b[riRow][riCol]
```

Parameters:

rpafIn1 : Pointer to a floating point input Vector
rpafIn2 : Pointer to a floating point Matrix
riRow : No of coloumns of Vector/No of Rows of Matrix
riCol : No of coloumns of Matrix
rpafOut : Pointer to a floating point output Matrix

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 17 + riCol*(10+2*riRow)

Number of VLIW: : 41

Restrictions: No of rows of Matrix B/No of coloumns of Matrix A should be Even and >=4

User Info: This flavour works only when No of rows of Matrix B/No of coloumns of Matrix A are even

long fVectByMatrixMul_odd (float * *rpafIn1*, float * *rpafIn2*, int *riRow*, int *riCol*, float * *rpafOut*)

fVectByMatrixMul_odd: Vector by Matrix multiplication Odd Flavour

Product of a float vector with a float matrix

```
C[riCol] = a[riRow]*b[riRow][riCol]
```

Parameters:

rpafIn1 : Pointer to a floating point input Vector
rpafIn2 : Pointer to a floating point Matrix
riRow : No of coloumns of Vector/No of Rows of Matrix
riCol : No of coloumns of Matrix
rpafOut : Pointer to a floating point output Matrix

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 18 + riCol*(8+2*riRow)

Number of VLIW: 42

Restrictions: No of rows of Matrix B/no of coloumns of Matrix A should be Odd and >=5

User Info: This flavour works only when No of rows of Matrix B/No of coloumns of Matrix A are odd

Function fxcorr

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **fXcorr** (float *rpafX, int riStrideX, float *rpafY, int riStrideY, float *rpafZ, int riStrideZ, int riLen, int riNcoeff)
  fXcorr: cross correlation or auto correlation computation

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long fXcorr (float * *rpafX*, int *riStrideX*, float * *rpafY*, int *riStrideY*, float * *rpafZ*, int *riStrideZ*, int *riLen*, int *riNcoeff*)

fXcorr: cross correlation or auto correlation computation

Computation of correlation between two float arrays

```
          _
         |
         |  sumk( X(k-i) * Y(k) ); k= 0 to N-i-1, i= 0 to Ncorr/2
Rxy(i) =|
         |  Ryx(-i); i= -Ncorr/2 to -1
         |_

         Z(i)   = Rxy(i-Ncorr/2)
```

Parameters:

rpafX : Pointer to the first floating point input array
riStrideX : Stride for the first input array
rpafY : Pointer to the second floating point input array
riStrideY : Stride for the second input array
rpafZ : Pointer to the floating point output array
riStrideZ : Stride for the output array
riLen : (N in the formula above) Minimum of the size of X and Y
riNcoeff : (Ncorr in the formula above) Number of outputs to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles:

- case 1> riLen is odd : 22 + 9.50*riNcoeff + riLen*riNcoeff - 0.25*riNcoeff^2
- case 2> riLen is even: 22 + 10.0*riNcoeff + riLen*riNcoeff - 0.25*riNcoeff^2

Number of VLIW: 87

Restrictions: riNcoeff must be a multiple of 4 and riLen must be greater than or equal to 3

User Info:

1. Minimum value of VSIZEX and VSIZEY should be 3
2. Ncorr should be a muliple of 4 and in the range [1,((2*min(VSIZEX,VSIZEY))-1)]
3. For autocorrelation include the same files for gafX and gafY

Function ifft1024

`#include "magic_chess.h"`

Defines

- #define **DIVISION_BY_N_FACTOR** 9.765625e-4

Functions

- void **ifft32m** (_c_float *, _c_float *, _c_float *, int)
  ifft32m: Performs the second layer of a N-point IFFT by means of M executions of the modified 32-point IFFT.

- long **ifft1024** (_c_float *W, _c_float *in, _c_float *temp, _c_float *out)
  ifft1024: Computes the 1024-point IFFT of a complex vector.

Define Documentation

#define DIVISION_BY_N_FACTOR 9.765625e-4

Function Documentation

long ifft1024 (_c_float * *W*, _c_float * *in*, _c_float * *temp*, _c_float * *out*)

ifft1024: Computes the 1024-point IFFT of a complex vector.

Function **ifft1024()** is the mixed radix implementation of the 1024-point IFFT. Function **ifft32m()** is used as a component block. If more than one fft size is used in an application, the module **ifft32m()** is shared among them. Note: in order to optimize the execution time, the division by 1024 has been performed in the first layer by including the 1/1024 factor in the coefficients of some of the butterflies.

Parameters:

W : Pointer to the array of 512 trigonometric coefficients: exp(-i*2*pi*n/1024) with n = 0, 1, .. 511.
in : Pointer to the input complex vector (size 1024).
temp : Pointer to a temporary auxiliary vector (size 1024) for FFT computation.
out : Pointer to the output complex vector (size 1024); after function call, the vector pointed by out contains the IFFT of the vector pointed by in.

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 5515

Number of VLIW: 169 (+ 194 for called function fft32m)

Restrictions: vector pointed by temp cannot coincide with vector pointed by out, so only the following vector combinations are allowed: in != temp != out (the 3 vectors are all different); in = temp != out; in = out != temp;

Number of stack locations used: 14

void ifft32m (_c_float * *W_base_addr*, _c_float * *input_base_addr*, _c_float * *output_base_addr*, int *M*)

ifft32m: Performs the second layer of a N-point IFFT by means of M executions of the modified 32-point IFFT.

Parameters:

W_base_addr Pointer to the array of N/2 trigonometric coefficients: exp(-i*2*pi*n/N) with n = 0, 1, .. (N/2 - 1).
input_base_addr Pointer to the array (size N) of input complex data.
output_base_addr Pointer to the array (size N) where output data will be written.
M Requested number of executions of the modified 32-point IFFT.

Returns:

void.

Number of cycles: 124 + 84*M

Number of VLIW: 194

Restrictions: vector pointed by input_base_addr cannot coincide with vector pointed by output_base_addr.

Number of stack locations used: 6

Function ifft32m

Functions

- void **ifft32m** (_c_float *W_base_addr, _c_float *input_base_addr, _c_float *output_base_addr, int M)
  ifft32m: Performs the second layer of a N-point IFFT by means of M executions of the modified 32-point IFFT.

Function Documentation

void ifft32m (_c_float * *W_base_addr*, _c_float * *input_base_addr*, _c_float * *output_base_addr*, int *M*)

ifft32m: Performs the second layer of a N-point IFFT by means of M executions of the modified 32-point IFFT.

Parameters:

W_base_addr Pointer to the array of N/2 trigonometric coefficients: exp(-i*2*pi*n/N) with n = 0, 1, .. (N/2 - 1).
input_base_addr Pointer to the array (size N) of input complex data.
output_base_addr Pointer to the array (size N) where output data will be written.
M Requested number of executions of the modified 32-point IFFT.

Returns:

void.

Number of cycles: 124 + 84*M

Number of VLIW: 194

Restrictions: vector pointed by input_base_addr cannot coincide with vector pointed by output_base_addr.

Number of stack locations used: 6

Function vfacos

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)
- #define **DOMAIN_CHECK**

Functions

- long **vfSqrt** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfSqrt: *Square Root Computation*

- long **vfAcos** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfAcos: Inverse Cos computation

Variables

- long **glRetDomainCheckAcos**

Define Documentation

#define DOMAIN_CHECK

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfAcos (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfAcos: Inverse Cos computation

Inverse Cos of a vector float input array

```
Y(k) = acos(X(k))              k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a vector floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose sin has to be computed

Returns:

0 if succesful, !=0 otherwhise

Number of cycles:

```
147  +   32.0 * N
 55  +   15.0 * N      For vfsqrt
-------------------
202  +   47.0 * N
```

Number of VLIW:

```
212
116                    For vfsqrt
-----
328
```

Restrictions: Array should have at least 2 vector elements

User Info: The valid input range for acos is [-1,1]. However if the input is not in this range the function outputs a zero. Enable #define DOMAIN_CHECK to activate this piece of code

```
Inner Loop cycles for Loop Above Sqrt :              9.0 * N
Inner Loop cycles for Loop Below Sqrt :             23.0 * N
                                             ----------
                                             32.0 * N
```

long vfSqrt (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfSqrt: Square Root Computation

Computes the square root of the input vector float array

```
Y(k) = sqrt(X(k)       k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point input Array
rpafOut : Pointer to a floating point output Array
riStrideInp : Stride for the input Array
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+N*15

Number of VLIW: 116

Restrictions: Array should have at least 1 vector element

User Info: The domain check flag is set whenever the input is negative.

Variable Documentation

long glRetDomainCheckAcos

Function vfacosh

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)
- #define **DOMAIN_CHECK**

Functions

- long **vfLog** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, float rfBaseConversion, int riLen)
  vfLog: Computing Logarithm of vector float array

- long **vfSqrt** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfSqrt: *Square Root Computation*

- long **vfAcosh** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfAcosh: Inverse Hyperbolic Cosine

Variables

- long **glDomainChkFlagACosh**

Define Documentation

#define DOMAIN_CHECK

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfAcosh (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfAcosh: Inverse Hyperbolic Cosine

Computation of Inverse Hyperbolic Cosine of vector float input array

```
    Y(k) = vfAcosh(X(k))          k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a vector floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose inverse hyperbolic cos has to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles:

```
             81  +  15.0 * riLen
             55  +  15.0 * riLen   For vfSqrt (with DOMAIN CHECK enabled)
             93  +  26.0 * riLen   For vfLog (with DOMAIN CHECK enabled)
            -----------------------
            229  +  56.0 * riLen
```

Number of VLIW:

```
            112
            116                    For vfSqrt (with DOMAIN CHECK enabled)
            146                    For vfLog (with DOMAIN CHECK enabled)
            -----
            374
```

Restrictions: Array should have at least 2 vector elements

User Info:

1. The formula used for the computation of acosh is: acosh(x)=ln(x+sqrt(x^2-1)). The valid input range for this function is [1,+inf]. Instead of +inf the upper limit is taken as 1e+17. For any input above this value the formula is approximated to acosh(x)=ln(2x)

2. For any invalid input the domain check flags are set and the output is zero

long vfLog (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, float *rfBaseConversion*, int *riLen*)

vfLog: Computing Logarithm of vector float array

Logarithm of input vector float array is calculated

```
    Y (k) = log(X(k))              k=Nelements
```

Parameters:

rpafIn : Pointer to the vector floating point Input array
riStrideInp : Stride for the input Array
rpafOut : Pointer to the vector floating point Output Array
riStrideOut : Stride for the Output Array
rfBaseConversion,: Base conversion Parameter to convert base of logarithm
riLen : Number of Input elements

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 93+26*riLen

Number of VLIW: 146

Restrictions: Array should have at least 2 vector elements

User Info: The valid inputs for log(x) are x>0. For all invalid inputs this code sets the domain check flag and outputs a zero. To disable the domain check, recompile the code undefining the DOMAIN_CHECK variable

long vfSqrt (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfSqrt: Square Root Computation

Computes the square root of the input vector float array

```
Y(k) = sqrt(X(k)        k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point input Array
rpafOut : Pointer to a floating point output Array
riStrideInp : Stride for the input Array
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+N*15

Number of VLIW: 116

Restrictions: Array should have at least 1 vector element

User Info: The domain check flag is set whenever the input is negative.

Variable Documentation

long glDomainChkFlagACosh

Function vfadd

#include "magic_chess.h"

Functions

- long **vfAdd** (float *gfpIn1, int giStrideInp1, float *gfpIn2, int giStrideInp2, float *gfpOut, int giStrideOut, int giLen)
  vfAdd: Addition of Input Array1 and Array2 and the output is stored in the output Array

---

Function Documentation

long vfAdd (float * *gfpIn1*, int *giStrideInp1*, float * *gfpIn2*, int *giStrideInp2*, float * *gfpOut*, int *giStrideOut*, int *giLen*)

vfAdd: Addition of Input Array1 and Array2 and the output is stored in the output Array

```
gfpOut(k)=gfpIn1(k) + gfpIn2(k)                k=0.......giLen - 1
```

Parameters:

gfpIn1 : Pointer to a floating point input Array 1
gfpIn2 : Pointer to a floating point input Array 2
gfpOut : Pointer to a floating point output Array
giStrideInp1 : Stride for the input Array 1
giStrideInp2 : Stride for the input Array 2
giStrideOut : Stride for the Output Array
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 16+N*1.75

Number of VLIW: 23

Restrictions: Element count should be a multiple of 4

Function vfasin

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)
- #define **DOMAIN_CHECK**

Functions

- long **vfSqrt** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfSqrt: *Square Root Computation*

- long **vfAsin** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)

Variables

- long **glRetDomainCheck**
  vfAsin: Inverse Sin computation

Define Documentation

#define DOMAIN_CHECK

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfAsin (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

long vfSqrt (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfSqrt: Square Root Computation

Computes the square root of the input vector float array

```
Y(k) = sqrt(X(k)       k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point input Array
rpafOut : Pointer to a floating point output Array

riStrideInp : Stride for the input Array
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 55+N*15

Number of VLIW: 116

Restrictions: Array should have at least 1 vector element

User Info: The domain check flag is set whenever the input is negative.

Variable Documentation

long glRetDomainCheck

vfAsin: Inverse Sin computation

Inverse Sin of a vector float input array

```
    Y(k) = asin(X(k))             k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a vector floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose inverse sin has to be computed

Returns:

0 if succesful, !=0 otherwise.

Number of cycles:

```
           145  +  32.0 * N
            55  + 15.0 * N       For vfSqrt with DOMAIN_CHECK enabled
           -------------------
           200  +  47.0 * N
```

Number of VLIW:

```
           210
           116                   For vfSqrt with DOMAIN_CHECK enabled
           -----
           326
```

Restrictions: Array should have at least 2 vector elements

```
  cycles for Loop Above Sqrt:      9.0 * N
  cycles for Loop Below Sqrt:     23.0 * N
                        ----------
                        32.0 * N
```

User Info: The valid input range for acos is [-1,1]. However if the input is not in this range the function outputs a zero. Enable #define DOMAIN_CHECK to activate this piece of code.

Function vfasinh

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfLog** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, float rfBaseConversion, int riLen)
  vfLog: Computing Logarithm of vector float array

- long **vfSqrt** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfSqrt: *Square Root Computation*

- long **vfAsinh** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfaSinh: Inverse Sin Hyperbolic computation

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfAsinh (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfaSinh: Inverse Sin Hyperbolic computation

Inverse Hyperbolic sin of vector float input array

```
Y(k) = vfasinh(X(k))          k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a vector floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose sin has to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles:

```
 83  +  15.0 * riLen
 45  +  14.0 * riLen          For vfSqrt with Domain Check disabled
 74  +  23.0 * riLen          For vfLog with Domain Check disabled
     -----------------------
202  +  52.0 * riLen
```

Number of VLIW:

```
118
 59                           For vfSqrt with Domain Check disabled
 97                           For vfLog with Domain Check disabled
-----
274
```

Number of cycles:

```
 83  +  15.0 * riLen
 55  +  15.0 * riLen          For vfSqrt with Domain Check enabled
 93  +  26.0 * riLen          For vfLog with Domain Check enabled
-----------------------
231  +  56.0 * riLen
```

Number of VLIW:

```
118
116                           For vfSqrt with Domain Check enabled
146                           For vfLog with Domain Check enabled
-----
380
```

Restrictions: Array should have at least 3 vector elements

User Info:

1. The formula used for the computation of asinh is: asinh(x)=ln(x+sqrt(x^2+1)). The valid input range for this function is (-inf,+inf). Instead of +inf the upper limit is taken as 1e+17 and similarly the lower limtit is -1e+17. For any x for which |x| > 1e+17 the formula is approximated to asinh(x)=ln(2x)
2. Enable #define DOMAIN_CHECK in the files ..\..\..\dsplibrary\vfsqrt.c and ..\..\..\dsplibrary\vflog.c to check wether invalid inputs are passed to the vfSqrt and vfLog functions

long vfLog (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, float *rfBaseConversion*, int *riLen*)

vfLog: Computing Logarithm of vector float array

Logarithm of input vector float array is calculated

```
Y (k) = log(X(k))             k=Nelements
```

Parameters:

rpafIn : Pointer to the vector floating point Input array
riStrideInp : Stride for the input Array
rpafOut : Pointer to the vector floating point Output Array
riStrideOut : Stride for the Output Array
rfBaseConversion,: Base conversion Parameter to convert base of logarithm
riLen : Number of Input elements

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 93+26*riLen

Number of VLIW: 146

Restrictions: Array should have at least 2 vector elements

User Info: The valid inputs for log(x) are x>0. For all invalid inputs this code sets the domain check flag and outputs a zero. To disable the domain check, recompile the code undefining the DOMAIN_CHECK variable

long vfSqrt (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfSqrt: Square Root Computation

Computes the square root of the input vector float array

```
    Y(k) = sqrt(X(k)        k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point input Array
rpafOut : Pointer to a floating point output Array
riStrideInp : Stride for the input Array
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 55+N*15

Number of VLIW: 116

Restrictions: Array should have at least 1 vector element

User Info: The domain check flag is set whenever the input is negative.

Function vfatan2

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfAtan2** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfAtan2: atan2 computation

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfAtan2 (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfAtan2: atan2 computation

atan2 of a complex float input array

```
 Y(k) = atan2(X(k))

where
 k=0...riLen-1
 X(k)=a+ib is a complex number
```

Parameters:

rpafIn : Pointer to a complex floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose atan2 has to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 227 + 24 * N n

Number of VLIW: 274

Restrictions: Array should have at least 2 vector elements

Function vfatanh

```
#include "magic_chess.h"
```

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)
- #define **DOMAIN_CHECK**

Functions

- long **vfLog** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, float rfBaseConversion, int riLen)
  vfLog: Computing Logarithm of vector float array

- long **vfAtanh** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfAtanh: Inverse Hyperbolic Tan

Variables

- long **glDomainChkFlagATanh**

Define Documentation

#define DOMAIN_CHECK

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfAtanh (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfAtanh: Inverse Hyperbolic Tan

Computation of Inverse Hyperbolic Tan of vector float input array

```
    Y(k) = vfatanh(X(k))          k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a vector floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose inverse hyperbolic tan has to be computed

Returns:

0 if succesful, !=0 otherwise.

Number of cycles:

```
 71  +  13.0 * riLen
 93  +  26.0 * riLen   For vfLog (with DOMAIN CHECK enabled)
----------------------
164  +  39.0 * riLen
```

Number of VLIW:

```
 94
146                    For vfLog
-----
240
```

Restrictions: Array should have at least 2 vector elements

User Info:

1. The formula used for the computation of atanh is: atanh(x)=0.5*ln((1+x)/(1-x))
2. The valid input range for this function is (-1,1). Enable #define DOMAIN_CHECK in the file ..\..\..\dsplibrary\vflog.c to set the domain check flag whenever the input is out of this range

long vfLog (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, float *rfBaseConversion*, int *riLen*)

vfLog: Computing Logarithm of vector float array

Logarithm of input vector float array is calculated

```
Y (k) = log(X(k))              k=Nelements
```

Parameters:

rpafIn : Pointer to the vector floating point Input array
riStrideInp : Stride for the input Array
rpafOut : Pointer to the vector floating point Output Array
riStrideOut : Stride for the Output Array
rfBaseConversion,: Base conversion Parameter to convert base of logarithm
riLen : Number of Input elements

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 93+26*riLen

Number of VLIW: 146

Restrictions: Array should have at least 2 vector elements

User Info: The valid inputs for log(x) are x>0. For all invalid inputs this code sets the domain check flag and outputs a zero. To disable the domain check, recompile the code undefining the DOMAIN_CHECK variable

---

Variable Documentation

long glDomainChkFlagATanh

Function vfbubblesort

```
#include "magic_chess.h"
```

Functions

- long **vfBubbleSort** (float *rpafIn, int riStrideInp, int riLen)
  vfBubbleSort: Sorting

Function Documentation

long vfBubbleSort (float * *rpafIn*, int *riStrideInp*, int *riLen*)

vfBubbleSort: Sorting

Sorting a vector float array using the Bubble sort Technique

```
X(k) = Sort(X(k))              k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a floating point input Array
riStrideInp : Stride for the input Array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 5*N^2 + 9*N - 4

Number of VLIW: 34

Restrictions: None

Function vfclip

```
#include "magic_chess.h"
```

Functions

- long **vfClip** (float *rpafX, int riStrideX, float *rpafY, int riStrideY, float *rfClipUp, float *rfClipDown, int riLen)
  vfClip: Vector Clipping between the two values rfClipUp and rfClipDown

---

Function Documentation

long vfClip (float * *rpafX*, int *riStrideX*, float * *rpafY*, int *riStrideY*, float * *rfClipUp*, float * *rfClipDown*, int *riLen*)

vfClip: Vector Clipping between the two values rfClipUp and rfClipDown

```
    Y = clip X(k)           k=0,1,2...riLen-1
```

Parameters:

rpafX : Pointer to a floating point Vector Input Array
riStrideX : Stride for the Input Array
rpafY : Pointer to the vector floating point Output Array
riStrideY : Stride for the Output Array
rfClipUp : Value to be used as the upper limit for the Input
rfClipDown : Value to be used as the lower limit for the Input
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 15 + 2*N

Number of VLIW: 19

Restrictions: Element count should be a multiple of 2

Function vfconv.c

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfConv** (float *rpafInX, float *rpafInCoeffH, float *rpafOut, int riVecLen, int riFilterLen, int riTransient)
  vfConv: Computing vector float Convolution

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfConv (float * *rpafInX*, float * *rpafInCoeffH*, float * *rpafOut*, int *riVecLen*, int *riFilterLen*, int *riTransient*)

vfConv: Computing vector float Convolution

Convolution of input vector float array is calculated with the input vector Coefficent

```
Y (k) = sumi( X(i) * H(k-i) )

i = 0 to M-1
k = 0 to (N+M-1)
N:- input vector length
M:- input filter length
```

Parameters:

rpafInX : Pointer to a floating point input array
rpafInCoeffH : Pointer to a floating point Coefficent array
rpafOut : Pointer to a floating point Array
riVecLen : Number of input N
riFilterLen : Number of Coefficent M
riTransient : Integer value used to compute or not the transient codes of the convolution: if riTransient=0 the transient isn't computed, otherwise it's calculated

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: Input/Output transient: (M-1) * (15+2*M)

Number of cycles: Steady state:

- Case N is odd: 76 + 2*M + (N-M-1)*(10.5 + 1*(M-2))
- Case N is even: 102 + 4*M + (N-M-2)*(10.5 + 1*(M-2))

Number of VLIW:

- Case N is odd: 125
- Case N is even: 154

Restrictions: M must be even and should be less than N

User Info: In order to have a better performance enable/disable ODD_VECLEN depending on the even/odd nature of N

Function vfcos

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfCos** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfCos: Cosine of a vectorial input array

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfCos (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfCos: Cosine of a vectorial input array

```
    Y(k)= cos(X(k))                k = 0,1,.....riLen-1;
```

Parameters:

rpafIn : Pointer to the input array of type vector float
riStrideInp : Stride to be applied on input array
rpafOut : Pointer to the output array of type vector float
riStrideOut : Stride to be applied on output array
riLen : Number of input vectors

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 60 + 19*riLen

Number of VLIW: 98

Restrictions: Array should have at least 2 vector elements

Function vfcosh

```
#include "magic_chess.h"
```

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfExp** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfExp: Computing Exponential of vector float array

- long **vfCosh** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfCosh: Hyperbolic Cosine computation

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfCosh (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfCosh: Hyperbolic Cosine computation

Hyperbolic Cosine of vector float input array

```
    Y(k) = cosh(X(k))            k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a vector floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose cosh has to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles:

```
           75  +  10.0 * N
           55  +  22.0 * N       For vfExp
          ------------------
```

```
            130  +  32.0 * N
```

Number of VLIW:

```
            106
             99                      For vfExp
            -----
            205
```

Restrictions: Array should have at least 3 vector elements

User Info: The formula used for cosh computation is cosh(x)= (e^(2x) + 1)/(2*e^x). Whenever x>20 this is approximated to cosh(x)= (e^x)/2

long vfExp (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfExp: Computing Exponential of vector float array

Exponential of input vector float array is calculated

```
    Y (k) = e^X(k)         k=Nelements
```

Parameters:

rpafIn : Pointer to a vector floating point array
riStrideInp : Input Stride
rpafOut : Pointer to a vector floating point Array
riStrideOut : Output Stride
riLen : Number of Elements

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+22*riLen

Number of VLIW: 99

Restrictions: Array should have at least 2 vector elements

Function vfdiv32

`#include "magic_chess.h"`

Functions

- long **vfDiv32** (float *rpafIn1, int riStrideInp1, float *rfapIn2, int riStrideInp2, float *rfapOut, int riStrideOut, int riLen)
  vfDiv32: Vector Float Division

Function Documentation

long vfDiv32 (float * *rpafIn1*, int *riStrideInp1*, float * *rfapIn2*, int *riStrideInp2*, float * *rfapOut*, int *riStrideOut*, int *riLen*)

vfDiv32: Vector Float Division

Divides one vector float value by another vector float array with 23 bits of precision

```
Z(k) = X(k)/Y(k)                 k=0...riLen-1
```

Parameters:

rpafIn1 : Pointer to a floating point input Array 1
rfapIn2 : Pointer to a floating point input Array 2
rfapOut : Pointer to a floating point output Array
riStrideInp1 : Stride for the input Array 1
riStrideInp2 : Stride for the input Array 2
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 24+N*6

Number of VLIW: 30

Restrictions: Element count should be a multiple of 1

Function vfdiv40

```
#include "magic_chess.h"
```

Functions

- long **vfDiv40** (float *rpafIn1, int riStrideInp1, float *rpafIn2, int riStrideInp2, float *rpafOut, int riStrideOut, int riLen)
  vfDiv40: Vector Float Division

Function Documentation

long vfDiv40 (float * *rpafIn1*, int *riStrideInp1*, float * *rpafIn2*, int *riStrideInp2*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfDiv40: Vector Float Division

Divides a vector float array by another vector float array with 40 bits of precison

```
Z(k) = X(k)/Y(k)              k=0...riLen-1
```

Parameters:

rpafIn1 : Pointer to a floating point input Array 1
rpafIn2 : Pointer to a floating point input Array 2
rpafOut : Pointer to a floating point output Array
riStrideInp1 : Stride for the input Array 1
riStrideInp2 : Stride for the input Array 2
riStrideOut : Address stride in words for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 27+N*8

Number of VLIW: 35

Restrictions: Element count should be a multiple of 1

Function vfdot

```
#include "magic_chess.h"
```

Functions

- long **vfDot** (float *rfpaIn1, int riStrideInp1, float *rfpaIn2, int riStrideInp2, float *rfpaOut, int riLen)
  vfDot: Dot Product between two vector float arrays

Function Documentation

long vfDot (float * *rfpaIn1*, int *riStrideInp1*, float * *rfpaIn2*, int *riStrideInp2*, float * *rfpaOut*, int *riLen*)

vfDot: Dot Product between two vector float arrays

```
    rfpaOut = Sum(rfpaIn1(k)*rfpaIn1(k))          k=0.......giLen - 1
```

Parameters:

rfpaIn1 : Pointer to a floating point input Array 1
rfpaIn2 : Pointer to a floating point input Array 2
rfpaOut : Pointer to a floating point output Array
riStrideInp1 : Stride for the input Array 1
riStrideInp2 : Stride for the input Array 2
riLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 24+1*riLen

Number of VLIW: 28

Restrictions: riLen count should be a multiple of 4 and greater than 4

Function vfexp10

```
#include "magic_chess.h"
```

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfExp10** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfExp10: Computing Exponential of vector float array

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfExp10 (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfExp10: Computing Exponential of vector float array

Exponential of input vector float array is calculated

```
    Y (k) = exp10^(X(k))          k=Nelements
```

Parameters:

rpafIn : Pointer to a vector floating point array
riStrideInp : Input Stride
rpafOut : Pointer to a vector floating point Array
riStrideOut : Output Stride
riLen : Number of Vector input

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+22*riLen

Number of VLIW: 99

Restrictions: Array should have at least 2 vector elements

Function vffill

#include "magic_chess.h"

Functions

- long **vfFill** (float *gfpIn, float *gfpOut, int giStride_out, int giLen) property(functional)
  vfFill: Copies a vector float value into all the locations of the output Array

---

Function Documentation

long vfFill (float * *gfpIn*, float * *gfpOut*, int *giStride_out*, int *giLen*)

vfFill: Copies a vector float value into all the locations of the output Array

```
gfpOut(k)=gfpIn(k)            k=0.......giLen - 1
```

Parameters:

gfpIn : Pointer to a floating point input
giStride_out : Address stride in words for the output array c
gfpOut : Pointer to a floating point output array
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 15+N*1

Number of VLIW: 19

Restrictions: Element count should be a multiple of 4

Function vffirnlms

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfFIRnlms** (float *rpafInX, float *rpafKerCoeffH, float *rpafRefOut, float *rpafDelayBuff, int riSampleLen, int riKernelLen, float rfAdapCoeff)
  vfFIRnlms: Computation of a pair of adaptive FIR filter

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfFIRnlms (float * *rpafInX*, float * *rpafKerCoeffH*, float * *rpafRefOut*, float * *rpafDelayBuff*, int *riSampleLen*, int *riKernelLen*, float *rfAdapCoeff*)

vfFIRnlms: Computation of a pair of adaptive FIR filter

FIR filter coefficient computed using Least Mean Square Algorithm

```
T [n] = rpafDelayBuff[n- k]*rpafKerCoeffH[k]

e=T[n]- rpafRefOut[n]

E =Summation (D[k]^2)

C =B*e/E

rpafKerCoeffH[k] = rpafKerCoeffH[k] +C*rpafDelayBuff[k]

where
k=0...riKernelLen-1
n=riKernelLen.....riSampleLen-1
```

Parameters:

rpafInX : Pointer to a real floating point input vector
rpafKerCoeffH : Pointer to a real floating point FIR kernel
rpafRefOut : Pointer to a real floating point vector containing the desired output
rpafDelayBuff : Pointer to delay memory of length riKernelLen

riSampleLen : Number of input samples
riKernelLen : Order of the filter
rfAdapCoeff : Adaption coefficient

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 55 + 2*riKernelLen + (27*riKernelLen+49)*(riSampleLen-riKernelLen+1)

Number of VLIW: 247

Restrictions: riKernelLen must be an even value multiple of 4

Function vffixscaleoffset

`#include "magic_chess.h"`

Functions

- long **vfFixScaleOffset** (float *gfpIn, int giStrideInp, long *glpOut, int giStrideOut, float *gfpScale, float *gfpOffset, int giLen)
  vfFixScaleOffset: The input Array is scaled, then added with an offset and then converted to a long value

---

Function Documentation

long vfFixScaleOffset (float * *gfpIn*, int *giStrideInp*, long * *glpOut*, int *giStrideOut*, float * *gfpScale*, float * *gfpOffset*, int *giLen*)

vfFixScaleOffset: The input Array is scaled, then added with an offset and then converted to a long value

```
glpOut(k)=long(gfpIn(k)*gfpScale + gfpOffset)               k=0.......giLen - 1
```

Parameters:

gfpIn : Pointer to a floating point input Array
glpOut : Pointer to a long output Array
giStrideInp : Stride for the input Array
giStrideOut : Stride for the Output Array
gfpScale : Pointer to a floating point Scale value
gfpOffset : Pointer to a floating point Offset value
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 17+N*2

Number of VLIW: 21

Restrictions: Element count should be a multiple of 2

Function vffixscaleoffsetclip

```
#include "magic_chess.h"
```

Functions

- long **vfFixScaleOffsetClip** (float *rpafX, int riStrideX, long *rpalY, int riStrideY, float *rfScale, float *rfOffset, float *rfClipUp, float *rfClipDown, int riLen)
  vfFixScaleOffsetClip:Scale,Offset and Clipping of Vector float values between two limits

---

Function Documentation

long vfFixScaleOffsetClip (float * *rpafX*, int *riStrideX*, long * *rpalY*, int *riStrideY*, float * *rfScale*, float * *rfOffset*, float * *rfClipUp*, float * *rfClipDown*, int *riLen*)

vfFixScaleOffsetClip:Scale,Offset and Clipping of Vector float values between two limits

```
    Y = truncate(clip X(k)*scale+offset)          k=0,1,2,3...riLen-1
```

Parameters:

rpafX : Pointer to a vector floating point Input Array
riStrideX : Stride for the Input Array
rpalY : Pointer to the vector long Output Array
riStrideY : Stride for the Output Array
rfScale : Vector floating point multiply factor
rfOffset : Vector floating point Offset Value
rfClipUp : Value to be used as the upper limit for the Input
rfClipDown : Value to be used as the lower limit for the Input
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 22 + 3*riLeN

Number of VLIW: 28

Restrictions: Element count should be a multiple of 2

Function vflog_base

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfLog** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, float rfBaseConv, int riLen)
  vfLog: Computing Logarithm of vector float array

- long **vfLog_Base** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, long rlBaseConversion, int riLen)
  vfLog_base: Logarithm to a base

Variables

- long **glRetVal_s_Ln_1**
- long **glRetVal_s_Ln_2**
- long **glRetVal_Ln_1**
- long **glRetVal_Ln_2**
- VFLOAT **lfBaseConversionIn** [2]
- VFLOAT **lfBaseConversionOut** [2]
- float **lfBaseConversion**

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfLog (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, float *rfBaseConversion*, int *riLen*)

vfLog: Computing Logarithm of vector float array

Logarithm of input vector float array is calculated

```
    Y (k) = log(X(k))             k=Nelements
```

Parameters:

rpafIn : Pointer to the vector floating point Input array
riStrideInp : Stride for the input Array

rpafOut : Pointer to the vector floating point Output Array
riStrideOut : Stride for the Output Array
rfBaseConversion,: Base conversion Parameter to convert base of logarithm
riLen : Number of Input elements

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 93+26*riLen

Number of VLIW: 146

Restrictions: Array should have at least 2 vector elements

User Info: The valid inputs for log(x) are x>0. For all invalid inputs this code sets the domain check flag and outputs a zero. To disable the domain check, recompile the code undefining the DOMAIN_CHECK variable

long vfLog_Base (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, long *rlBaseConversion*, int *riLen*)

vfLog_base: Logarithm to a base

Logarithm of a Base of input vector float array , given the base is calculated

```
    Y (k) = log(X(k)) to Base b            k=0,1,2......Nelements
```

Parameters:

rpafIn : Pointer to a vector floating point array
riStrideInp : Stride for the input Array
rpafOut : Pointer to a vector floating point Array
riStrideOut : Stride for the Output Array
rlBaseConversion : The base at which the log has to be computed
riLen : Number of Vector input

Returns:

0 if succesful, !=0 otherwise.

Number of cycles:

```
             50  +   0.0 * riLen
             74  +  23.0 * riLen         vfLog without without domain check
            -----------------------
            124  +  23.0 * riLen
```

Number of VLIW:

```
             50
             97                          vfLog without Domain Check
            -----
            147
```

Number of cycles:

```
             50  +   0.0 * riLen
             93  +  26.0 * riLen         vfLog with Domain Check
            -----------------------
            143  +  26.0 * riLen
```

Number of VLIW:

```
             50
```

```
146                            vfLog with Domain Check
-----
196
```

Restrictions: Array should have at least 2 vector elements

User Info:

1. To set the domain check falgs for invalid inputs, enable #define DOMAIN_CHECK in the log code
2. The rlBaseConversion parameter should be passed as desired

---

Variable Documentation

long glRetVal_Ln_1

long glRetVal_Ln_2

long glRetVal_s_Ln_1

long glRetVal_s_Ln_2

float lfBaseConversion

VFLOAT lfBaseConversionIn[2]

VFLOAT lfBaseConversionOut[2]

Function vfmax

`#include "magic_chess.h"`

Functions

- long **vfMax** (float *rpavfIn, int riStrideInp, float *rpavfOut, int riLen)
  vfMax: Finding the maximum of a vector float array

Function Documentation

long vfMax (float * *rpavfIn*, int *riStrideInp*, float * *rpavfOut*, int *riLen*)

vfMax: Finding the maximum of a vector float array

```
Y = max (rpavfIn)
```

Parameters:

rpavfIn : Pointer to the Input Array of type vector float
riStrideInp : Stride for the Input Array
rpavfOut : Pointer to the Output Array of type vector float
riLen : Dimension of the input matrix

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 24 + 1*N

Number of VLIW: 28

Restrictions: N should be a multiple of 4

Function vfmaxandindex

```
#include "magic_chess.h"
```

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfMaxAndIndex** (float *rpafIn, int riStrideInp, float *rpafMax, long *rpalMaxIdx, int riLen)
  vfMaxAndIndex: Max and Index computation

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfMaxAndIndex (float * *rpafIn*, int *riStrideInp*, float * *rpafMax*, long * *rpalMaxIdx*, int *riLen*)

vfMaxAndIndex: Max and Index computation

Finding the maximum of a set of given vectors and extracting the index

```
Max     = max(X(k))
Idx_Max = index(max(X(k)))
```

Parameters:

rpafIn : Pointer to the input array of type vector float
riStrideInp : Stride for the input array
rpafMax : Pointer to the output location for storing max element
rpalMaxIdx : Pointer to the output location for storing the index
riLen : Dimension of the input array

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 78 + 4.5*riLen

Number of VLIW: 96

Restrictions: riLen should be a multple of 2

Function vfmean

`#include "magic_chess.h"`

Functions

- long **vfMean** (float *gfpIn, int giStrideInp, float *gfpOut, int giLen)
  vfMean: Computation of Mean of the vector float input Array

---

Function Documentation

long vfMean (float * *gfpIn*, int *giStrideInp*, float * *gfpOut*, int *giLen*)

vfMean: Computation of Mean of the vector float input Array

```
                          ____
                          \
gfpOut= 1/giLen * /___ gfpIn(k)                    k=0.......giLen - 1
```

Parameters:

gfpIn : Pointer to a floating point input Array
gfpOut : Pointer to a floating point output
giStrideInp : Stride for the input Array
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 38+N*1

Number of VLIW: 42

Restrictions: Element count should be a multiple of 4

Function vfmovescaleoffset

```
#include "magic_chess.h"
```

Functions

- long **vfMoveScaleOffset** (float *gfpIn, int giStrideInp, float *gfpOut, int giStrideOut, float *gfpScale, float *gfpOffset, int giLen)

  vfMoveScaleOffset: *The input array is multiplied with the scale value and added with the offset value*

Function Documentation

long vfMoveScaleOffset (float * *gfpIn*, int *giStrideInp*, float * *gfpOut*, int *giStrideOut*, float * *gfpScale*, float * *gfpOffset*, int *giLen*)

vfMoveScaleOffset: The input array is multiplied with the scale value and added with the offset value

The function vfMoveScaleOffset moves vectorial floating point data with scale and offset. Note that the simple move is obtained multiplying by the complex unity (1.0 + 1.0i) and adding with complex zero (0.0 + 0.0i)

```
    gfpOut(k)=(gfpIn(k)*gfpScale + gfpOffset)           k=0.......giLen - 1
```

Parameters:

gfpIn : Pointer to a floating point input Array
gfpOut : Pointer to a floating point output Array
giStrideInp : Stride for the input Array
giStrideOut : Stride for the Output Array
gfpScale : Pointer to a floating point Scale value
gfpOffset : Pointer to a floating point Offset value
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 16+N*1

Number of VLIW: 20

Restrictions: Element count should be a multiple of 2

Function vfmul

`#include "magic_chess.h"`

Functions

- long **vfMul** (float *rpafIn1, int riStrideInp1, float *rpafIn2, int riStrideInp2, float *rpafOut, int riStrideOut, int riLen)
  vfMul: *Vector Float Multipliaction*

Function Documentation

long vfMul (float * *rpafIn1*, int *riStrideInp1*, float * *rpafIn2*, int *riStrideInp2*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfMul: Vector Float Multipliaction

The function vfMul performs vectorial element-by-element multiplication

```
    Z(k) = X(k)*Y(k)                k=0...riLen-1
```

Parameters:

rpafIn1 : Pointer to a vector floating point input Array 1
rpafIn2 : Pointer to a vector floating point input Array 2
rpafOut : Pointer to a floating point output Array
riStrideInp1 : Stride for the input Array 1
riStrideInp2 : Stride for the input Array 2
riStrideOut : Stride for the output array
riLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 16+N*1.75

Number of VLIW: 23

Restrictions: Element count should be a multiple of 4

Function vfmuladd

```
#include "magic_chess.h"
```

Functions

- long **vfMulAdd** (float *rpafIn1, int riStrideInp1, float *rpafIn2, int riStrideInp2, float *rpafIn3, int riStrideInp3, float *rpafOut, int riStrideOut, int riLen)

  vfmuladd: *Product of two vector floating point arrays and sum with a third vector floating point array.*

Function Documentation

long vfMulAdd (float * *rpafIn1*, int *riStrideInp1*, float * *rpafIn2*, int *riStrideInp2*, float * *rpafIn3*, int *riStrideInp3*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfmuladd: Product of two vector floating point arrays and sum with a third vector floating point array.

The function vfmuladd computes the product of two vector floating point arrays and the product obtained is added with a third vector floating point array.

```
M(k)=A(k)*B(k)+C(k)            k=0...riLen-1
```

Parameters:

rpafIn1 : Pointer to a floating point Input Array 1
rpafIn2 : Pointer to a floating point Input Array 2
rpafIn3 : Pointer to a floating point Input Array 3
rpafOut : Pointer to a floating point Output Array
riStrideInp1 : Stride for the input Array 1
riStrideInp2 : Stride for the input Array 2
riStrideInp3 : Stride for the input Array 3
riStrideOut : Stride for the Output Array
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 19+N*2

Number of VLIW: 27

Restrictions: Element count should be a multiple of 4

Function vfquicksort

`#include "magic_chess.h"`

Defines

- #define **MAX_LENGTH** 512
- #define **STACK_SIZE** (MAX_LENGTH/2)
- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfQuickSort** (float *rpafIn, int riStrideInp, int riStart, int riEnd)
  vfQuickSort: *Sorting of an input vector float array*

Define Documentation

#define MAX_LENGTH 512

#define STACK_SIZE (MAX_LENGTH/2)

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfQuickSort (float * *rpafIn*, int *riStrideInp*, int *riStart*, int *riEnd*)

vfQuickSort: Sorting of an input vector float array

The function vfQuickSort sorts an input array using the Quick Sort algorithm

```
Y(k)=quicksort(X(k))
```

Parameters:

rpafIn : Pointer to the input array of type vector float
riStrideInp : Stride for the input array
riStart : Pointer to the strating location of the input vector
riEnd : Pointer to the last location of the input vector

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: data depending

Number of VLIW: 280

Restrictions: STACK_SIZE indicates maximum number of items that can be sorted in each array, if required set MAX_LENGTH greater than 512 constrained by the available data memory

Function vfrand

```
#include "magic_chess.h"
```

Functions

- long **vfRand** (float *rpafNorm, float *rpafOffset, float *rpafOut, int riStrideOut, int riLen)
  vfRand: *Random Number generation*

Function Documentation

long vfRand (float * *rpafNorm*, float * *rpafOffset*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfRand: Random Number generation

The function vfRand generates a vectorial float array of random numbers using a linear congruential method: a seed is multiplied for a vectorial float normalization factor and added to a vectorial float offset.

```
SEED = Norm*SEED[k] + Offset  k=0...riLen-1
SEED[k] = (A*SEED[k-1]+C) MOD 2^32  A = 69069 C = 1
```

Parameters:

rpafNorm : Pointer to a floating point Norm
rpafOffset : Pointer to a floating point Offset
rpafOut : Pointer to a floating point output Array
riStrideOut : Address stride in words for the output array c
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 27+N*3

Number of VLIW: 33

Restrictions: None

Function vfrms

#include "magic_chess.h"

Functions

- long **vfRMS** (float *rpafIn, int riStrideInp, float *rpafOut, int riLen)
  vfRMS: *Root Mean Square*

Function Documentation

long vfRMS (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riLen*)

vfRMS: Root Mean Square

The function vfRMS computes the Root Mean Square of a vector floating point input array

```
Z = Sum(X(k)/N                    k=0...riLen-1

Output=Sqrt(Z)
```

Parameters:

rpafIn : Pointer to a floating point input Array
rpafOut : Pointer to a floating point output Value
riStrideInp : Stride for the input Array
riLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 76 + 1.0 * N

Number of VLIW: 80

Restrictions: Element count should be a multiple of 4

Function vfsin

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfSin** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfSin: *Sin computation*

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfSin (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfSin: Sin computation

The function vfSin computes the sine of all the elements of a vectorial floating point input array

```
Y(k) = Sin(X(k))              k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a vector floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose sin has to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 58+20*N

Number of VLIW: 98

Restrictions: Array should have at least 2 vector element

Function vfsinh

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfExp** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfExp: Computing Exponential of vector float array

- long **vfSinh** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfSinh: *Hyperbolic Sin computation*

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfExp (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfExp: Computing Exponential of vector float array

Exponential of input vector float array is calculated

```
    Y (k) = e^X(k)          k=Nelements
```

Parameters:

rpafIn : Pointer to a vector floating point array
riStrideInp : Input Stride
rpafOut : Pointer to a vector floating point Array
riStrideOut : Output Stride
riLen : Number of Elements

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+22*riLen

Number of VLIW: 99

Restrictions: Array should have at least 2 vector elements

long vfSinh (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfSinh: Hyperbolic Sin computation

The function vfSinh computes the hyperbolic sine of all the elements of a vectorial floating point input array

```
Y(k) = sinh(X(k))              k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a vector floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose Hyperbolic sin has to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles:

```
 75  +  10.0 * N
 55  +  22.0 * N              For vfExp
------------------
130  +  32.0 * N
```

Number of VLIW:

```
116
 99                           For vfExp
-----
205
```

Restrictions: Array should have at least 4 vector elements

User Info: The formula used for cosh computation is sinh(x)= (e^(2x) - 1)/(2*e^x). Whenever x>20 this is approximated to sinh(x)= (e^x)/2

Function vfsub

```
#include "magic_chess.h"
```

Functions

- long **vfSub** (float *gfpIn1, int giStrideInp1, float *gfpIn2, int giStrideInp2, float *gfpOut, int giStrideOut, int giLen)
  vfSub: *Subtraction of two vectorial floating point array*

Function Documentation

long vfSub (float * *gfpIn1*, int *giStrideInp1*, float * *gfpIn2*, int *giStrideInp2*, float * *gfpOut*, int *giStrideOut*, int *giLen*)

vfSub: Subtraction of two vectorial floating point array

The function vfSub performs the element by element subtruction of the second input array from the first input array

```
gfpOut(k)=gfpIn1(k) - gfpIn2(k)              k=0.......giLen - 1
```

Parameters:

gfpIn1 : Pointer to a floating point input Array 1
gfpIn2 : Pointer to a floating point input Array 2
gfpOut : Pointer to a floating point output Array
giStrideInp1 : Stride for the input Array 1
giStrideInp2 : Stride for the input Array 2
giStrideOut : Stride for the Output Array
giLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 16+N*1.75

Number of VLIW: 23

Restrictions: Element count should be a multiple of 4

Function vfsum

`#include "magic_chess.h"`

Functions

- long **vfSum** (float *gfpIn, int giStrideInp, float *gfpOut, int giLen)
  vfSum: *Sum of the elements of input Array*

Function Documentation

long vfSum (float * *gfpIn*, int *giStrideInp*, float * *gfpOut*, int *giLen*)

vfSum: Sum of the elements of input Array

```
            ____
            \
gfpOut= /___  (gfpIn(k))               k=0.......giLen - 1
```

Parameters:

gfpIn : Pointer to a floating point input Array
gfpOut : Pointer to a floating point output
giStrideInp : Stride for the input Array
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 24+N*1

Number of VLIW: 28

Restrictions: Element count should be a multiple of 4

Function vftan

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfTan** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfTan: *Tan conputation*

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfTan (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfTan: Tan conputation

The vfTan function performs the tan computation of Vector float input array

```
Y(k)= tan(X(k))                 k = 0,1,.....riLen-1;
```

Parameters:

rpafIn : Pointer to the input array of type vector float
riStrideInp : Stride to be applied on input array
rpafOut : Pointer to the output array of type vector float
riStrideOut : Stride to be applied on output array
riLen : Number of input vectors

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 99 + 31*riLen

Number of VLIW: 161

Restrictions: Array should have at least 2 vector elements

User Info: Near 1.57(90 degree) the error is high, otherwise its of the order of e-008

Function vftanh

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfExp** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfExp: Computing Exponential of vector float array

- long **vfTanh** (float *rpafIn, int riStrideInp, float *rpafOut, int riStrideOut, int riLen)
  vfTanh: *Hyperbolic Tan computation*

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfExp (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfExp: Computing Exponential of vector float array

Exponential of input vector float array is calculated

```
Y (k) = e^X(k)          k=Nelements
```

Parameters:

rpafIn : Pointer to a vector floating point array
riStrideInp : Input Stride
rpafOut : Pointer to a vector floating point Array
riStrideOut : Output Stride
riLen : Number of Elements

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 55+22*riLen

Number of VLIW: 99

Restrictions: Array should have at least 2 vector elements

long vfTanh (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, int *riStrideOut*, int *riLen*)

vfTanh: Hyperbolic Tan computation

Hyperbolic Tan of vector float input array

```
    Y(k) = tanh(X(k))              k=0...riLen-1
```

Parameters:

rpafIn : Pointer to a vector floating point input Array
riStrideInp : Stride for the input array
rpafOut : Pointer to the vector floating point output Array
riStrideOut : Stride for the output array
riLen : No of vector elements whose tanh has to be computed

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles:

```
             65  +  10.0 * riLen
             55  +  22.0 * riLen          For vfExp
            ----------------------
            120  +  32.0 * riLen
```

Number of VLIW:

```
             96
             99                           For vfExp
            -----
            195
```

Restrictions: Array should have at least 3 vector elements

User Info: The formula used for tanh computation is; tanh(x)= ((e^2x)-1)/((e^2x)+1). If x>20 the formula is approximated as tanh(x)=1

Function vfvar

```
#include "magic_chess.h"
```

Functions

- long **vfVar** (float *rpafIn, int riStrideInp, float *rpafOut, float *rpafMean, float *rpafInvlen, int riLen)
  vfvar: *Variance computation.*

Function Documentation

long vfVar (float * *rpafIn*, int *riStrideInp*, float * *rpafOut*, float * *rpafMean*, float * *rpafInvlen*, int *riLen*)

vfvar: Variance computation.

The varaince of the input array is computed with mean passed as a parameter.

```
    variance = mean(X^2)-{mean(x)}^2
```

Parameters:

rpafIn : Pointer to a floating point Input Array
riStrideInp : Stride for the input Array
rpafOut : Pointer to a floating point Output
rpafMean : Pointer to the floating point Mean Value of Input
rpafInvlen : Pointer to the floating point Inv of Element count
riLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 36+N*1

Number of VLIW: 40

Restrictions: Element count should be a multiple of 4

Function vfxcorr

`#include "magic_chess.h"`

Defines

- #define **VFLOAT** float chess_storage(DATA%2)
- #define **VLONG** long chess_storage(DATA%2)

Functions

- long **vfXcorr** (float *rpafX, int riStrideX, float *rpafY, int riStrideY, float *rpafZ, int riStrideZ, int riLen, int riNcoeff)
  vfXcorr *: cross correlation or auto correlation computation*

Define Documentation

#define VFLOAT float chess_storage(DATA%2)

#define VLONG long chess_storage(DATA%2)

Function Documentation

long vfXcorr (float * *rpafX*, int *riStrideX*, float * *rpafY*, int *riStrideY*, float * *rpafZ*, int *riStrideZ*, int *riLen*, int *riNcoeff*)

vfXcorr : cross correlation or auto correlation computation

Computation of correlation between two float arrays

```
                _
               |
               |  sumk( X(k-i) * Y(k) ); k= 0 to N-i-1, i= 0 to Ncorr/2
Rxy(i)  =      |
               |  Ryx(-i); i= -Ncorr/2 to -1
               |_

Z(i)    =      Rxy(i-Ncorr/2)
```

Parameters:

rpafX : Pointer to the first floating point input array
riStrideX : Stride for the first input array
rpafY : Pointer to the second floating point input array
riStrideY : Stride for the second input array
rpafZ : Pointer to the floating point output array
riStrideZ : Stride for the output array
riLen : (N in the formula above) Minimum of the size of X and Y
riNcoeff : (Ncorr in the formula above) Number of outputs to be computed

Returns:

0 if succesful, !=0 otherwhise

Number of cycles:

- case 1> riLen is odd : 26 + 8.75*riNcoeff + riLen*riNcoeff - 0.25riNcoeff^2
- case 2> riLen is even: 26 + 9.00*riNcoeff + riLen*riNcoeff - 0.25riNcoeff^2

Number of VLIW: 87

Restrictions: riNcoeff must be a multiple of 4 and riLen must be greater than or equal to 3

User Info:

1. Minimum value of VSIZEX and VSIZEY should be 3
2. Ncorr should be a muliple of 4 and in the range [1,((2*min(VSIZEX,VSIZEY))-1)]
3. For autocorrelation include the same files for gafX and gafY

Function vladd

#include "magic_chess.h"

Functions

- long **vlAdd** (long *glpIn1, int giStrideInp1, long *glpIn2, int giStrideInp2, long *glpOut, int giStrideOut, int giLen)
  vlAdd: *Addition of two vectorial long arrays*

Function Documentation

long vlAdd (long * *glpIn1*, int *giStrideInp1*, long * *glpIn2*, int *giStrideInp2*, long * *glpOut*, int *giStrideOut*, int *giLen*)

vlAdd: Addition of two vectorial long arrays

The function vlAdd performs vectorial element-by-element addition of the two vectorial long input arrays

```
gfpOut(k)=gfpIn1(k) + gfpIn2(k)                k=0.......giLen - 1
```

Parameters:

glpIn1 : Pointer to a long input Array 1
glpIn2 : Pointer to a long input Array 2
glpOut : Pointer to a long output Array
giStrideInp1 : Stride for the input Array 1
giStrideInp2 : Stride for the input Array 2
giStrideOut : Stride for the Output Array
giLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 16+N*1.75

Number of VLIW: 23

Restrictions: Element count should be a multiple of 4

Function vlmovescaleoffset

`#include "magic_chess.h"`

Functions

- long **vlMoveScaleOffset** (long *glpIn, int giStrideInp, long *glpOut, int giStrideOut, long *glpScale, long *glpOffset, int giLen)

  vlMoveScaleOffset: *The input array is multiplied with the scale value and then the offset is added to it*

Function Documentation

long vlMoveScaleOffset (long * *glpIn*, int *giStrideInp*, long * *glpOut*, int *giStrideOut*, long * *glpScale*, long * *glpOffset*, int *giLen*)

vlMoveScaleOffset: The input array is multiplied with the scale value and then the offset is added to it

The function vlMoveScaleOffset moves vectorial long data with scale and offset. Note that the simple move is obtained multiplying by the complex unity (1 + 1i) and adding with complex zero (0 + 0i)

```
    glpOut(k)=(glpIn(k)*glpScale + glpOffset)            k=0.......giLen - 1
```

Parameters:

glpIn : Pointer to a long input Array
glpOut : Pointer to a long output Array
giStrideInp : Stride for the input Array
giStrideOut : Stride for the Output Array
glpScale : Pointer to a long Scale value
glpOffset : Pointer to a long Offset value
giLen : Element count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 16+N*1

Number of VLIW: 20

Restrictions: Element count should be a multiple of 2

Function vlrotate

`#include "magic_chess.h"`

Defines

- #define **_M_MASK_SHL_32** (float)3.68934881474e+019

Functions

- long **vlRotate** (long *rpalIn, int riStrideIn, long *rpalOut, int riStrideOut, long rlLShift, long rlRShift, int riLen)
  vlRotate: *Rotate mod 32 (Left/Right)*

Define Documentation

#define _M_MASK_SHL_32 (float)3.68934881474e+019

Function Documentation

long vlRotate (long * *rpalIn*, int *riStrideIn*, long * *rpalOut*, int *riStrideOut*, long *rlLShift*, long *rlRShift*, int *riLen*)

vlRotate: Rotate mod 32 (Left/Right)

The function vlRotate performs a left or right rotate mod.32 of the long input array. The number of rotated bits is respectively equal to rlLShift for the real part and rlRShift for the imaginary part. rlLShift and rlRShift can be positive o negative. If they are positive the function performs a left retate otherwise a right rotate.

```
Y(k)= rotate32 X(k)          k = 0,1,.....riLen-1;
```

Parameters:

rpalIn : Pointer to the input array of type vector long
riStrideIn : Stride to be applied on input array
rpalOut : Pointer to the output array of type vector long
riStrideOut : Stride to be applied on output array
rlLShift : Number of bit rotations for the real part of the vector
rlRShift : Number of bit rotations for the imaginary part of the vector
riLen : Number of input vectors

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 36 + 1*riLen

Number of VLIW: 40

Restrictions: riLen must be a multiple of 4

Function vlshand

`#include "magic_chess.h"`

Defines

- #define **_M_MASK_SHL_32** (float)3.68934881474e+019

Functions

- long **vlShand** (long *rpalIn, int riStrideIn, long *rpalOut, int riStrideOut, long rlLShift, long rlRShift, long rlLMask, long rlRMask, int riLen)

  vlShand: *Logical "AND" and Vectorial Shift(Left/Right)*

Define Documentation

#define _M_MASK_SHL_32 (float)3.68934881474e+019

Function Documentation

long vlShand (long * *rpalIn*, int *riStrideIn*, long * *rpalOut*, int *riStrideOut*, long *rlLShift*, long *rlRShift*, long *rlLMask*, long *rlRMask*, int *riLen*)

vlShand: Logical "AND" and Vectorial Shift(Left/Right)

The function vlShand performs on the long input array:

1. a left or right arithmetic shift
2. a logical AND with a mask

The number of shifts and the mask for the logical AND are respectively equal to rlLShift and rlLMask for the real part and rlRShift and rlRMask for the imaginary part. rlLShift and rlRShift can be positive o negative. If they are positive the function performs a left shift otherwise a right shift.

```
    Y(k)= vshand X(k)              k = 0,1,.....riLen-1;
```

Parameters:

rpalIn : Pointer to the input array of type vector long
riStrideIn : Stride to be applied on input array
rpalOut : Pointer to the output array of type vector long
riStrideOut : Stride to be applied on output array
rlLShift : Number of bit shifts for the real part of the vector
rlRShift : Number of bit shifts for the imaginary part of the vector
rlLMask : Mask for the logical AND for the real part of the vector
rlRMask : Mask for the logical AND for the imaginary part of the vector
riLen : Element Count

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 42 +1*riLen

Number of VLIW: 46

Restrictions: riLen must be a multiple of 4

Function vlshift

```
#include "magic_chess.h"
```

Defines

- #define **_M_MASK_SHL_32** (float)3.68934881474e+019

Functions

- long **vlShift** (long *rpalIn, int riStrideIn, long *rpalOut, int riStrideOut, long rlLShift, long rlRShift, int riLen)
  vlShift: *Vectorial shift (Left/Right)*

Define Documentation

#define _M_MASK_SHL_32 (float)3.68934881474e+019

Function Documentation

long vlShift (long * *rpalIn*, int *riStrideIn*, long * *rpalOut*, int *riStrideOut*, long *rlLShift*, long *rlRShift*, int *riLen*)

vlShift: Vectorial shift (Left/Right)

The function vlShift performs a left or right arithmetic shift on the long input array. The number of shifts are respectively equal to rlLShift for the real part and rlRShift for the imaginary part. rlLShift and rlRShift can be positive o negative. If they are positive the function performs a left shift otherwise a right shift.

```
    Y(k)= vshift X(k)             k = 0,1,.....riLen-1;
```

Parameters:

rpalIn : Pointer to the input array of type vector long
riStrideIn : Stride to be applied on input array
rpalOut : Pointer to the output array of type vector long
riStrideOut : Stride to be applied on output array
rlLShift : Number of bit shifts for the real part of the vector
rlRShift : Number of bit shifts for the imaginary part of the vector
riLen : Number of input vectors

Returns:

0 if succesful, !=0 otherwhise.

Number of cycles: 42+ 1*riLen

Number of VLIW: 46

Restrictions: riLen must a multiple of 4

Function vltofloatscaleoffset

```
#include "magic_chess.h"
```

Functions

- long **vlToFloatScaleOffset** (long *glpIn, int giStrideInp, float *gfpOut, int giStrideOut, float *gfpScale, float *gfpOffset, int giLen)

  vlToFloatScaleOffset: *The long values in the input Array is converted to a float value, scaled then added with an offset.*

Function Documentation

long vlToFloatScaleOffset (long * *glpIn*, int *giStrideInp*, float * *gfpOut*, int *giStrideOut*, float * *gfpScale*, float * *gfpOffset*, int *giLen*)

vlToFloatScaleOffset: The long values in the input Array is converted to a float value, scaled then added with an offset.

```
    gfpOut(k)=float(glpIn(k))*gfpScale + gfpOffset              k=0.......giLen - 1
```

Parameters:

glpIn : Pointer to a long input Array
gfpOut : Pointer to a floating point output Array
giStrideInp : Stride for the input Array
giStrideOut : Stride for the Output Array
gfpScale : Pointer to a floating point Scale value
gfpOffset : Pointer to a floating point Offset value
giLen : Element count

Returns:

0 if succesful, !=0 otherwise.

Number of cycles: 23+N*2

Number of VLIW: 27

Restrictions: Element count should be a multiple of 2