DSP Library

User Manual (draft)

Table of Contents

Section 1

Introduction

This document describes the functions contained in the basic DSP function library for mAgic.

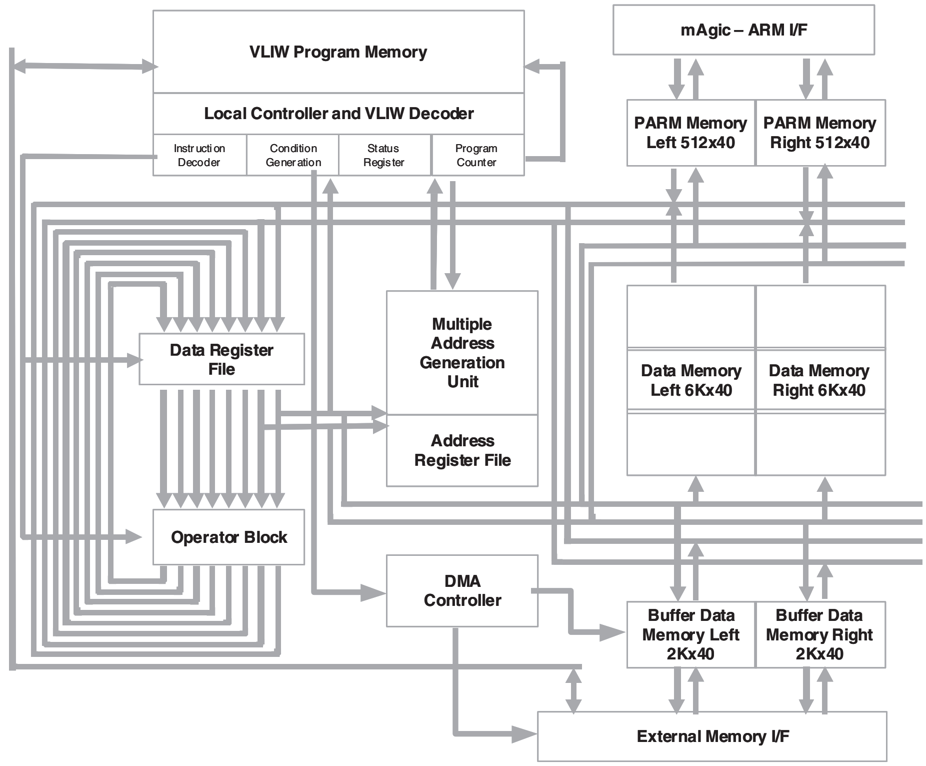
Notes: ***1.*** All the number of cycles given in each function description includes the C-calling protocol (register push-pop and stack management as appropriate).

2. Some further optimization can be obtained by appropriately modifying the code at micro assembler level.

The functions are C-callable and respect the C-calling protocol (refer to [4] in Section “Related Documents” on page 4-1).

An overview of mAgic DSP is given in the next paragraphs. For details refer to [2] in Section “Related Documents” on page 4-1.

Figure 1-1. mAgic DSP Block Diagram



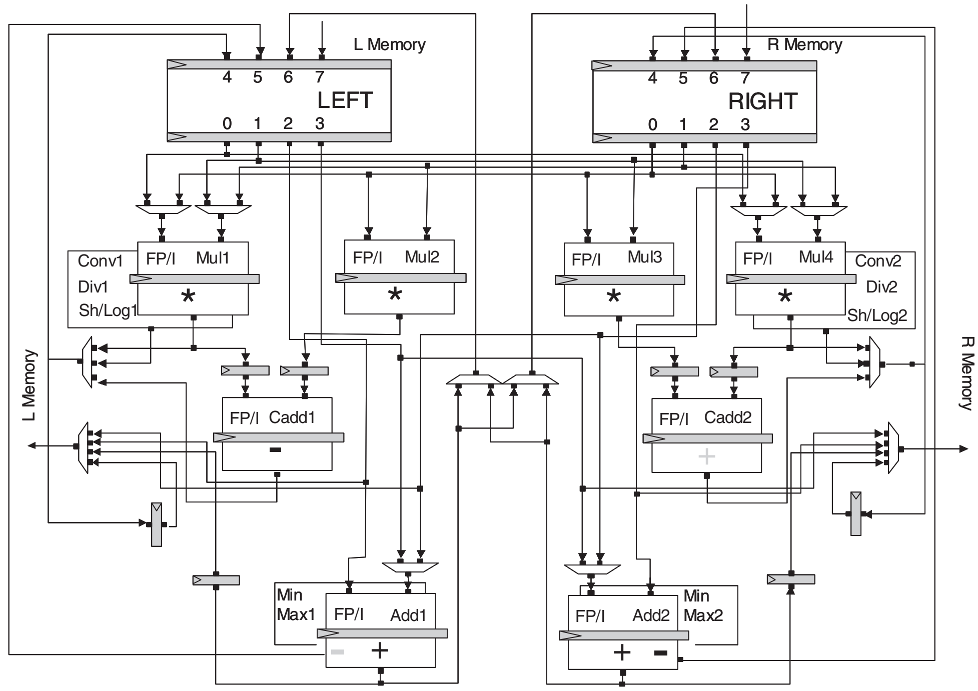
1.1 mAgic DSP Processor

The mAgic DSP is the VLIW numeric processor of the D740. It operates on IEEE 754 40-bit extended precision floating-point and 32-bit integer numeric format. The main components of the DSP subsystem are the core processor, the on-chip memories and the interfaces to and from the ARM subsystem. The operators block, the register file, the address generation unit and the program decoding and sequencing unit compose the core processor. In the following paragraphs a short description of each block is given. For detailed information refer to the specific section in document [2] in Section "Related Documents" on page 3-1.

1.1.1 Core processor

mAgic is a VLIW engine but, from an user point of view, it works like a RISC machine, implementing triadic computing operations on data coming from the register file, and data move operations between the local memories and the register file. The operators are pipe-lined for maximum performance. The pipe-line depth depends on the operator used. The operations scheduling and parallelism are automatically defined and managed at compile time by the assembler-optimizer, allowing efficient code execution. To give the best support to the RISC-like programming model, mAgic is equipped with a complex 256-entry register file. It can be used as a complex register file (real and imaginary part), or as dual register file for vectorial operations. When performing single instructions the register file can be used as an ordinary 512 register file. Both the left and right side of the register file are 8-ported, making a total of 16 I/O port available for the data move to and from the operator block and the memory. The total data bandwidth between the register file and the operator block is 70 bytes per clock cycle, avoiding bottlenecks in the data flow between the two units.The operators' block, the register file, the address generation unit and the program-sequencing unit compose the core processor.The hardware that performs arithmetical operations is contained in the Operators Block. It works on 32-bit integers and IEEE 754 extended precision 40-bit floating-point data.

Figure 1-2. Register Files and Operators Block.



The Operator Block is composed of four integer/floating point multipliers: an adder, a subtractor and two add-subtract integer/floating point units. It has two shift/logic units, a Min/Max operator and two seed generators for efficient division and inverse square root computation also. The operator block is arranged to support complex arithmetic (single cycle complex multiply or multiply and add), fast FFT (single cycle butterfly computation) and vectorial computations. The mAgic peak performance is achieved during single cycle FFT butterfly execution, when it delivers 10 floating-point operations per clock cycle.

mAgic is equipped with two independent address generation units. It is able to generate up to two couple of addresses, one to access the left and right memory for reading and one to access the left and right memory for writing. It is also used in the loop control to test if the end of a loop is reached. The Multiple Address Generation Unit (MAGU) supports indexed addressing, linear addressing with stride, circular addressing and bit reversed addressing. The address generation unit is composed by 16 registers.

The Program Address Generation Unit is devoted to manage the correct Program Counter generation according to the program flow. It generates addresses for linear code execution as well as for non-sequential program flow. The Condition Generation Unit combines the flags generated by the operators to produce complex conditions flags used to control the program execution. Predicated instruction execution is supported for different groups of instructions: arithmetical instructions, memory write, immediate load, or all of them. The Program Address Generation Unit allows also to perform conditioned and unconditioned branch instructions, loops, call to subroutines and return from subroutines.

1.1.2 Internal Memories, External Memories and DMA

mAgic has four on chip memory blocks: the Program Memory, the Data Memory, the Data Buffer, and the dual ported memory shared with the ARM processor. An External Memory Interface multiplexes the Data accesses and the Program accesses to and from the External Memory.The Program Memory stores the VLIW program to be executed by mAgic. It is 8K words by 128-bit single port memory. When mAgic is in System mode ARM can modify the content of the mAgic Program Memory in two different ways. ARM can directly write a Program Memory location by accessing the memory address space assigned to the mAgic Program Memory in the ARM memory map. In this access mode ARM writes four 32-bit words to four consecutive addresses at correct address boundaries, in order to properly complete a single VLIW word write cycle. ARM can also modify the content of the mAgic Program Memory by initiating a DMA transfer from the external memory to the mAgic Program Memory. In this access mode a single VLIW word is transferred from the mAgic external memory to the mAgic Program Memory 64-bit per cycle, that is one complete word every two clock cycles. Due to the program compression scheme used (see later), allowing average program compression between 2 and 3, the code accessing capability of mAgic from its external memory is greater then one instruction per clock cycle. When mAgic is in Run mode, ARM can't get access to the mAgic Program Memory. When in Run mode mAgic can initiate a DMA transfer from the external memory to the mAgic Program Memory to load a new code segment.

In order to optimize the internal Program Memory usage and the code bandwidth from the external Program Memory to the internal Program Memory, a code compression mechanism has been implemented. The code for mAgic can be generated and executed in compressed or encompassed form. When the code stored in Program Memory is compressed, the decompression is done "on flight" just after the Instruction Fetch. The current code compression scheme allows getting compression factors between 2 and 3, depending on the code structure without performance loss.

Anyway the classic DSP execution determinism is maintained: only the amount of program memory used can change, as function of the compression factor achieved, not the program execution timing. Thanks to the code compression, the code density obtained for mAgic is similar to the code density available on other non VLIW DSP, while maintaining the advantage in terms of instruction level parallelism.

The mAgic internal Data Memory is made of three memory pages, 2K words by 40-bit for the left data memory and 2K words by 40-bit for the right data memory, giving a total of 6K word left and right memory banks (12 Kword total). Each Data Memory bank is a dual port memory that allows four simultaneous accesses, two accesses in reading mode and two in writing. The core can access vectorial and single data stored in data memory. Accessing a complex data is equivalent to accessing a vectorial data. During simultaneous read and write memory accesses, the MAGU generates two independent read and write addresses common to the left and right memory banks. The total available bandwidth between the register file and the data memory is 20 bytes per clock cycle, allowing full speed implementation of numerically intensive algorithms (e.g. complex FFT and FIR).

The Buffer Memory is 2K words by 40-bit for both the left and right memory. The Buffer Memory is a dual port memory. One port is connected to the core processor. The MAGU generates the Buffer Memory addresses for transferring data to and from the core. The second port of the Buffer Memory is connected to the External Memory Interface. The Buffer Memory doesn't support dual read and write accesses neither from the core nor from the External Memory Interface. The available bandwidth between the core processor and the Buffer Memory is equal to the available bandwidth between the External Memory Interface and the Buffer Memory: 10 bytes per clock cycle. The maximum external memory size of mAgic is 16 Mword Left and Right (equivalent to 32 Mword or 160 Mbytes; 24-bit address bus). A DMA controller manages the data transfer between the external memory and the Buffer Memory. The DMA controller can generate accesses with stride both for the External Memory and the Buffer Memory. The DMA transfers to and from the Buffer Memory can be executed in parallel with the full speed core instructions execution with zero-overhead and without the intervention of the core processor used only to initiate it.

Two kind of DMA transfer are allowed: non-blocking transfers and blocking transfers. The first type (non-blocking transfers) consists of a transfer that is immediately launched if the DMA machine is idle. If the DMA machine is busy, the transfer request is queued into a FIFO. The second type of transfers (blocking transfers) consists of a transfer that is immediately launched if the DMA machine is idle. If the DMA machine is busy, the end of the current transfer is waited and then the burst is started. In this case the execution of core instruction is suspended until the requested transfer is started. The core can be synchronized with the DMA engine through the usage of specific synchronization instructions.

The last memory block in the address space of mAgic DSP is the memory shared (PARM) between mAgic and the ARM processor. It is a dual port memory 512 words by 40-bit for the left and right banks (total 1K by 40-bit). This memory can be used to efficiently transfer data between the two processors. The available bandwidth between the core processor and the shared memory is 10 bytes per clock cycle. On the ARM side the available bandwidth is limited by the bus size of the ARM processor (32 bits) giving a bandwidth of 4 bytes per ARM clock cycle.

1.1.3 ARM Interface

The DIOPSIS 740 master is the ARM7 RISC processor. mAgic behaves as standard AMBA ASB slave device, allowing access to different resources depending on the operating mode (Run or System).

In System Mode, mAgic halts its execution and ARM takes control on it. When mAgic is in System mode ARM can access many mAgic internal devices. The ability of ARM to access internal mAgic resources in System Mode can be used for initialization and debugging purposes. Accessing the Command Register, ARM can change the operating status of the DSP (Run/System Mode), initiate DMA transactions, force single or multiple step execution, or simply read the DSP operating status.

In Run Mode, mAgic works under direct control of its own VLIW program and ARM has access only to the 1K x 40-bit dual ported shared memory (PARM) and to the mAgic Command Register.

In order to allow a tight coupling between the operations of mAgic and ARM at run time, they can exchange synchronization signals, based on interrupts.

1.1.4 VLIW Program Word

The mAgic VLIW program word can assume different configurations according to the kind of instructions it contains.

In the first configuration, that is also the most typical one, the VLIW is divided in four fields, corresponding to the building blocks of the VLIW core: Flow Control Unit, Multiple Address Generation Unit, Data Register File Addresses, and Operators Block. In this configuration each field directly drives the architectural blocks to which it’s connected.

A second kind of mAgic instruction uses all the bits in the long instruction word to perform a single cycle, multiple loading of immediate data, multiple addressing initialization and looping set up.

A third kind of instruction contains the parameters for launching DMAs between the external memory interface and the local buffers. This instruction is passed to the DMA engine and is executed in complete parallelism with the activities of the VLIW core.

1.1.5 Instruction Set

The operands supported in the instruction set are different for the different kind of instruction. The available operands types are summarized in Table 1-1.

Table 1-1. Operands Data Type

Complex (Float or Integer)
Complex Conjugated (Float or Integer)
Complex Double Conjugated (Float or Integer)
Complex with Real (Float or Integer)
Vectorial (Float or Integer)
Single Operand (Float or Integer)

mAgic treats complex numbers as couples of 40-bit floating-point. The real part is stored in the left (L) memory bank and the imaginary part is stored at the same address of the right (R) memory bank. The Register File is also divided in real (L) and imaginary parts (R).

mAgic instruction set supports the kind of instruction summarized in Table 1-2.

Table 1-2. Instructions Summary

Add-Sub Instructions
Address Register File Management Instructions
Branch Instructions
DMA (Burst Transfer) Instructions
Compare Instructions
Condition Code and Loop Instructions
Control and Miscellaneous Instructions
Convertion Instructions
Interrupt Management Instructions
Logical and Shift Instructions
Mathematical Seed Generation Instructions
Miscellaneous Arithmetic Instructions
Move Instructions
Multiply Instructions
Repeat Instructions

Some assembly instructions operate on complex conjugated numbers. They can be of two types: the CJ ones in which the first operand is a complex number while the second is conjugated before its use and the CJJ in which both operands are conjugated.

It is also possible the execution of additions and multiplications between a complex number and a real number (40-bit floating point or 32-bit integer). This kind of instructions are obtained with a complex additions or products in which the second complex operand has the imaginary part masked with zero.

The vectorial numbers are couple of data of the same type (40-bit floating point, or 32-bit integer). The first element of the couple must be in the L memory or registers; the second element must be in the R memory or registers. On vectorial numbers, two operations of the same type (two additions, two products, etc.) are performed (Vectorial Operations). The operands for vectorial instructions are couple of registers. Real numbers (40-bit floating point and 32-bit integer) can be placed either in the L or R space. The single arithmetical operations are performed exclusively on one path (L or R depending on the destination register). The input and destination registers can be in any bank.

The combination of the available computing operations and the different kind of operands for the complex domain operations allows implementing in a very natural way many common signal-processing operations (e.g. a sampled correlation computation is simply a multiply with conjugate and add; Inverse FFT is a scaled FFT with conjugate coefficients). The operations scheduling and parallelization is automatically defined and managed at compile time by the assembler-optimizer, allowing efficient code execution and substantially simplifying the code development.

Section 2

List of the DSP Library Function

2.1 General Restrictions

The library functions are designed to work with the mAgic C compiler mcc. The functions make use of the C stack to push the used registers when appropriate. In the chapter 3 are listed for each function the number of locations of the stack used. The library functions can also be called from an assembly code using the same conventions used by the C compiler to pass the parameters. In this case, it is advisable to copy the registers with the passed parameters of the function who calls the leaf function, in not scratch registers and push them.

Sometime the functions rely on the value of the C initialized registers (e.g. the register already initialized to 1.0f or to 1 or to 0). Thus to correctly execute a function from the library the mcc runtime initialization code must be executed to appropriately initialize the constant register values and to initialize a stack. Moreover the mcc register usage conventions are adopted. Refer to the mcc manual for all the details.

The vectorial functions operates on arrays that have a size less or equal to 2K locations, independently if they are of type int, float, __vector__ float, __vector__ int, __complex__ float or __complex__int. Arrays defined in Parm Memory must have a maximum of 512 elements. The arrays used in the DSP library functions can be allocated in Internal Memory. It is also possible to declare an array in Buffer Memory or in Parm Memory, but the simultaneous access in reading and writing mode to input/temporary/output arrays must be granted. For example, if the user defines an input array in Parm Memory, any other array of that function can't be defined in the same Parm Memory. As a general rule, for each function it is possible to allocate a maximum of one array in Parm Memory, a maximum of one array in Buffer Memory and as many arrays as required in Internal Memory.

Note that the Internal Memory and the Buffer Memory corrispond respectively to the Data Memory and Buffer Data Memory indicated in the Figure 1-1 on page 2. The mAgic C compiler mcc refers to the Internal Memory with: P0, P1, P2, to the Buffer Memory with P3 and to the Parm Memory with P4.

The __vector__ int value returned by some functions described in the chaper 3 has the following meaning: it stores the content of the two Sticky Status registers in the return registers (498 and 499) of the Register File after the computation. If an operation has happened on invalid values or arithmetic operation has resulted in an exception, the relevant bits of these registers are set. For more details on the Status Flags and Exceptions refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

2.2 Alphabetical DSP Function List

The DSP functions are all C-callable and comply with the mAgic C Compiler (MCC) protocol. The execution cycles listed include the C-calling protocol overhead.

Table 2-1. DSP Function List

Function name	Execution Cycles	Code Size (in VLIW)	Notes
cmulcxcy	24 + 2 × Nelements	20	Complex conjugate element by element multiplication
cmulxy	25 + 2 × Nelements	21	Complex element by element multiplication with the first input conjugate
cmulxy	25 + 2 × Nelements	21	Complex element by element multiplication
conv	Initialization: 25 Input transient: 14 + 34 × (M - 1) + 6 × M / 2 × (M - 1) Steady state: 38 + 44 × L / 2 +13 × M / 4 × L / 2 --> L = N-- M + 1 Output transient: 6 + 35 × (M - 1) + 6 × M / 2 × (M - 1)	123	Convolution with complex vectors
conv2d	171 + 4 × K + 3 × (M - K + 1) × (N - K + 1) + (((9 × K / 2 + 30) × (N - K + 1) + 7 × K / 2 + 25) × K / 2 + 28) × (M - K+1)	165	2-dimensional convolution of complex matrix A with complex kernel matrix H
cvexp	137 + 23.5 × Nelements	67	Complex exponential of an input array stored in left memory
cvma	37 + 3 × Nelements	33	Product of 2 complex input arrays and sum with the third complex input array
cvrdiv	83 + 8 × Nelements	51	Division of a complex array by a real array stored in left memory element by element
FD_RealFIR_Pair	268 + 20 × (N / 4 - 5) + fft cycles + ifft cycles	164	FIR filter on two real signals using two different filter sequences
fft1024	6405	230	Complex FFT on 1024 points
fft128	1053	183	Complex FFT on 128 points
fft256	1729	175	Complex FFT on 256 points
fft288	2623	193	Complex FFT on 288 points
fft512	3251	178	Complex FFT on 512 points
fft64	769	148	Complex FFT on 64 points
FIR	136 + (79 + 13 × (M / 4 - 3)) × L / 2	99	Complex FIR filter
FirNlmsll	77+ (94+ 4.25 × (P-4)) × (N-P+1) +8.0 × P	130	Fir filter using Least Mean Square Algorithm

Table 2-1. DSP Function List (Continued)

FirNlmsv	78 + (94+ 4.25 × (P-4)) × (N-P+1) +8.0 × P - 7	135	Pair of FIR filters using Least Mean Square Algorithm
getvq	65 + 1× Nelements	39	Extraction of vectorial data from a vector queue to the destination vector
getvq_f2i	60 + 1 × Nelements	36	Extraction of vectorial data from a vector queue to the destination vector and float to integer conversion
getvq_i2f	71 + 1 × Nelements	40	Extraction of vectorial data from a vector queue to the destination vector and integer to floatconversion
getvqelem	12	4	Number of unread elements in a vector queue
getvqfree	12	4	Number of free positions in a vector queue
hilbert	174 + 2.6875 × N + fft cycles + ifft cycles	113	Discrete time hilbert function on a complex input vector of N elements
ifft1024	6527	233	Complex inverse FFT on 1024 points
ifft128	1112	176	Complex inverse FFT on 128 points
ifft256	1829	183	Complex inverse FFT on 256 points
ifft288	2836	179	Complex inverse FFT on 288 points
ifft512	3487	181	Complex inverse FFT on 512 points
ifft64	767	151	Complex inverse FFT on 64 points
IIR1	189 + [47 + 14 × (Stages_Nr - 2)] × Ch_Nr × Samples_Nr/2	109	Cascaded vectorial IIR biquad section with pipeline on sections
IIR2	187 + [66 + 20 × (Stages_Nr × Ch_Nr -4)/ 2] × Samples_Nr	122	Cascaded vectorial IIR biquad section on input sequences
Init_IIR1_struct	277 + 6 × Stages_Nr × Ch_Nr × 2	49	Initialization procedure for IIR1 function
Init_IIR2_struct	204 + 6 × Stages_Nr × Ch_Nr × 2	64	Initialization procedure for IIR2 function
initFIR	35 + 3 × M	23	Initialization procedure for FIR function
initvq	45	22	Initialization of the data structure used to manage a vector circular buffer
LastStage	137 + 3.25 × N	71	Plain radix two butterfly
levinson	3297 (P = 11)	131	Levinson-Durbin recursion
lpc2cep	5074 (N = 11 and M = 32)	122	Cepstral coefficients of a real float array in left memory
madd	35 + 7 × (M × N / 2 -1)	25	Sum of two complex matrices
mchol	0.4166 × + 23.75 × +47.84 × N + 138	212	L-U decomposition of a positive definite square matrix using Cholesky algorithm

Table 2-1. DSP Function List (Continued)

mdeterm	28 + 1.33 × + 23 × + 36.5 × N + Cycles for swap operation, which is data dependent	195	Determinant of a complex matrix of the order $N \times N$
mdeterm2	29	9	Determinant of a complex matrix of the order 2×2
mdeterm3	22	22	Determinant of a complex matrix of the order 3×3
minvert	4.66 × + 68.5 × - N × 18.17 - 44 + 130 + Cycles for swap operation which is data dependent	400	Inverse of a complex square matrix of the order $N \times N$
mmul	112+ (((((6 × (N-1)+13) × M)+11) × P)	56	Product of 2 complex matrices
mtrace	35 + 5 × N / 2	22	Trace of $N \times N$ complex matrix
mvmul	46 + ((((6 × (N-1)) + 17) × M) + 11) × P	48	Product of a complex matrix with a set of complex vectors
mvmul3x3	59 + 9 × Nelements	44	Product of a complex 3×3 matrix with a set of complex vectors of size 3
mvmul4x4	125 + 16 × Nelements	68	Product of a complex 4×4 matrix with a set of complex vectors of size 4
mvmul8x8	461 + 69 × Nelements	203	Product of a complex 8×8 matrix with a set of complex vectors of size 8
pack40to16ll	39 + 6 × Nelements	40	Multiplication by a float value, addition of a float offset, clipping in a float range of a pair of data in left memory and conversion of the results in a 16 bit integer arranged in a 32 bit word in left memory
pack40to16lr	39 + 6 × Nelements	41	Multiplication by a float value, addition of a float offset, clipping in a float range of a pair of data in left memory and conversion of the results in a 16 bit integer arranged in a 32 bit word in right memory
pack40to16rl	42 + 6 × Nelements	41	Multiplication by a float value, addition of a float offset, clipping in a float range of a pair of data in right memory and conversion of the results in a 16 bit integer arranged in a 32 bit word in left memory
pack40to16rr	41 + 6 × Nelements	42	Multiplication by a float value, addition of a float offset, clipping in a float range of a pair of data in right memory and conversion of the results in a 16 bit integer arranged in a 32 bit word in right memory
putvq	64 + 1× Nelements	37	Filling of a vector queue with vectorial data
putvq_f2i	72 + 1 × Nelements	38	Filling of a vector queue with vectorial data converted from float to integer

Table 2-1. DSP Function List (Continued)

putvq_i2f	72 + 1 × Nelements	38	Filling of a vector queue with vectorial data converted from integer to float
v2magnlrl	24 + 14 × Nelements	18	Vector squared magnitude
v2magnv	26 + 2.75 × Nelements	24	Vectorial complex squared magnitude
vacoshll	400 + 27.75 × Nelements	251	Inverse hyperbolic cosine of a float input array and left to left move
vacoshlr	389 + 27.75 × Nelements	254	Inverse hyperbolic cosine of a float input array and left to right move
vacoshrl	400 + 27.75 × Nelements	252	Inverse hyperbolic cosine of a float input array and right to left move
vacoshrr	391 + 27.75 × Nelements	254	Inverse hyperbolic cosine of a float input array and right to right move
vacoshv	354 + 50.5 × Nelements	220	Inverse hyperbolic cosine of a vectorial input array
vacosll	310 + 26.25 × Nelements	232	Inverse cosine of a float input array and left to left move
vacoslr	300 + 26.75 × Nelements	232	Inverse cosine of a float input array and left to right move
vacosrl	308 + 26 × Nelements	233	Inverse cosine of a float input array and right to left move
vacosrr	298 + 26.5 × Nelements	232	Inverse cosine of a float input array and right to right move
vacosv	292 + 52 × Nelements	208	inverse cosine of vectorial input array
vaddintv	39 + 2 × Nelements	34	Sum of 2 vectorial integer arrays
vaddlll	31+ 2 × Nelements	24	Sum of 2 input float array stored in left memory and output in left memory
vaddllr	32 + 2.25 × Nelements	36	Sum of 2 input float array stored in left memory and output in right memory
vaddlrl	31+ 2 × Nelements	25	Sum of 2 input float array : the first is stored in left memory while the second in right memory. The output is written in left memory
vaddlrr	31+ 2 × Nelements	25	Sum of 2 input float array: the first is stored in left memory while the second in right memory. The result is written in right memory
vaddrrl	40 + 2 × Nelements	36	Sum of 2 input float array stored in right memory and output in left memory
vaddrrr	35 + 2 × Nelements	25	Sum of 2 input float array stored in right memory and output in right memory
vaddv	32 + 2.75 × Nelements	27	Sum of 2 vectorial float array

Table 2-1. DSP Function List (Continued)

varll	53 + 1.75 × Nelements	33	Variance of a float array
vasinhll	400 + 27.75 × Nelements	249	Inverse hyperbolic sine of a float input array and left to left move
vasinhlr	389 + 27.75 × Nelements	252	Inverse hyperbolic sine of a float input array and left to right move
vasinhrl	400 + 27.75 × Nelements	250	Inverse hyperbolic sine of a float input array and right to left move
vasinhrr	390 + 27.75 × Nelements	252	Inverse hyperbolic sine of a float input array and right to right move
vasinhv	354 + 50.5 × Nelements	219	Inverse hyperbolic sine of a vectorial input array
vasinll	310 + 26.25 × Nelements	233	Inverse sine of a float input array and left to left move
vasinlr	299 + 26.75 × Nelements	231	Inverse sine of a float input array and left to right move
vasinrl	290 + 26 × Nelements	232	Inverse sine of a float input array and right to left move
vasinrr	297 + 26.5 × Nelements	236	Inverse sine of a float input array and right to right move
vasinv	290 + 51 × Nelements	210	Inverse sine of a vectorial input array
vatan2	339 + 26.5 × Nelements	224	argument (arctan2) of a complex input array and result in a float array in left memory
vatanhll	323 +19.25 × Nelements	184	Inverse hyperbolic tangent of a float input array and left to left move
vatanhlr	320 +19.25 × Nelements	186	Inverse hyperbolic tangent of a float input array and left to right move
vatanhrl	321 +19.25 × Nelements	182	Inverse hyperbolic tangent of a float input array and right to left move
vatanhrr	318 +19.25 × Nelements	184	Inverse hyperbolic tangent of a float input array and right to right move
vatanhv	300 + 35 × Nelements	161	Inverse hyperbolic tangent of a vectorial input array
vbyvmulv	25 + 2 × Nelements	19	Vectorial element by element multiplication
vclipll	25 + 2 × Nelements	26	Clipping of a float array in left memory between two float values ClipUp and ClipDown and left to left move
vcliprr	31 + 2 × Nelements	27	Cipping of a float array in right memory between two float values ClipUp and ClipDown and right to right move
vclipv	36 + 2 × Nelements	30	Vectorial clipping between the two values ClipUp and ClipDown
vcoshll	307 + 19 × Nelements	165	Hyperbolic cosine of a float input array and left to left move

Table 2-1. DSP Function List (Continued)

vcoshlr	306 +18.5 × Nelements	159	Hyperbolic cosine of a float input array and left to right move
vcoshrl	304 +19 × Nelements	166	Hyperbolic cosine of a float input array and right to left move
vcoshrr	306 + 18.5 × Nelements	161	Hyperbolic cosine of a float input array and right to right move
vcoshv	320 + 31× Nelements	156	Hyperbolic cosine of a vectorial input array
vcosll	125 + 13.25 × Nelements	65	Cosine of a float input array and left to left move
vcoslr	124 + 13 × Nelements	66	Cosine of a float input array and left to right move
vcosrl	125 + 13 × Nelements	67	Cosine of a float input array and right to left move
vcosrr	123 + 13 × Nelements	66	Cosine of a float input array and right to right move
vcosv	107 + 20.5 × Nelements	58	Cosine of a vectorial input array
vdist	173 + 10.5 × Nelements	109	Euclidean distance between two input complex arrays
vdiv0rll	32 + 25 × Nelements	27	Float array division element by element
vdiv40lll	78 + 7.75 × Nelements	64	Float array division element by element with Y and X il left memory and precision equal to 31 bit of mantissa
vdiv40lrl	79 + 7.75 × Nelements	68	Float array division element by element with Y in left memory and X in right memory and precision equal to 31 bit of mantissa
vdiv40rll	78 + 7.75 × Nelements	66	Float array division element by element with Y in right memory and X in left memory and precision equal to 31 bit of mantissa
vdiv40rrl	80 + 7.75 × Nelements	65	Float array division element by element with Y and X in right memory and precision equal to 31 bit of mantissa
vdivlll	96 + 3.75 × Nelements	59	Float array division element by element with Y and X in left memory and precision equal to 23 bit of mantissa
vdivlrl	98 + 3.25 × Nelements	61	Float array division division element by element with Y in left memory and X in right memory and precision equal to 23 bit of mantissa
vdivrll	98 + 3.5 × Nelements	59	Float array division element by element with Y in right memory and X in left memory and precision equal to 23 bit of mantissa

Table 2-1. DSP Function List (Continued)

vdivrrl	93 + 3.75 × Nelements	59	Float array division element by element with X and Y in right memory and precision equal to 23 bit of mantissa
vdivv	90 + 6.75 × Nelements	51	Vectorial float division element by element
vexp10ll	124 + 10 × Nelements	69	exponential to base 10 (10^x) of a float input array and left to left move
vexp10lr	126 + 10 × Nelements	69	exponential to base 10 (10^x) of a float input array and left to right move
vexp10rl	123 + 10 × Nelements	69	exponential to base 10 (10^x) of a float input array and right to left move
vexp10rr	123 + 10 × Nelements	69	exponential to base 10 (10^x) of a float input array and right to right move
vexp10v	115 + 18.5 × Nelements	60	exponential to base 10 (10^x) of a vectorial input array
vexpll	125 + 10 × Nelements	70	exponential to base **e** (e^x) of a float input array and left to left move
vexplr	124 + 9.75 × Nelements	66	exponential to base **e** (e^x) of a float input array and left to right move
vexprl	124 + 10 × Nelements	70	exponential to base **e** (e^x) of a float input array and right to left move
vexprr	123 + 9.75 × Nelements	66	exponential to base **e** (e^x) of a float input array and right to right move
vexpv	116 + 18.5 × Nelements	61	exponential to base **e** (e^x) of a vectorial input array
vfillll	20 + 1.5 × Nelements	18	Filling of an array in left memory with a constant stored in left memory
vfilllr	20 + 1.5 × Nelements	18	Filling of an array in right memory with a constant stored in left memory
vfillrl	22 + 1.5 × Nelements	19	Filling of an array in left memory with a constant stored in right memory
vfillrr	22 + 1.5 × Nelements	19	Filling of an array in right memory with a constant stored in right memory
vfillv	22 + 1.5 × Nelements	19	Filling of a vectorial array with a vectorial constant
vfix1ll	42 + 1 × Nelements	29	Addition of a float offset, float to integer conversion and left to left move
vfix1lr	42 + 1 × Nelements	29	Addition of a float offset, float to integer conversion and left to right move

Table 2-1. DSP Function List (Continued)

vfix1rl	43 + 1 × Nelements	29	Addition of a float offset, float to integer conversion and right to left move
vfix1rr	43 + 1 × Nelements	29	Addition of a float offset, float to integer conversion and right to right move
vfix1v	53 + 1× Nelements	30	Addition of a vectorial float offset, float to integer conversion and vectorial move
vfix2ll	34 + 2 × Nelements	36	Multiplication by a float value, addition of a float offset, float to integer conversion and left to left move
vfix2lr	34 + 2 × Nelements	36	Multiplication by a float value, addition of a float offset, float to integer conversion and left to right move
vfix2rl	36 + 2 × Nelements	35	Multiplication by a float value, addition of a float offset, float to integer conversion and right to left move
vfix2rr	36 + 2 × Nelements	35	Multiplication by a float value, addition of a float offset, float to integer conversion and right to right move
vfix2v	36 + 2 × Nelements	35	Multiplication by a vectorial float value, addition of a vectorial float offset and float to integer conversion
vfix3ll	24 + 3.75 × Nelements	55	Multiplication by a float value, addition of a float offset, clipping in a float range, float to integer conversion and left to left move
vfix3lr	24 + 3.75 × Nelements	57	Multiplication by a float value, addition of a float offset, clipping in a float range, float to integer conversion and left to right move
vfix3rl	27 + 3.75 × Nelements	55	Multiplication by a float value, addition of a float offset, clipping in a float range, float to integer conversion and right to left move
vfix3rr	27 + 3.75 × Nelements	57	Multiplication by a float value, addition of a float offset, clipping in a float range, float to integer conversion and right to right move
vfix3v	44 + 3 × Nelements	61	Multiplication by a vectorial float value, addition of a vectorial float offset, clipping in a vectorial float range and float to integer conversion
vfloat1ll	36 + 1 × Nelements	28	Integer to float conversion, addition of a float offset and left to left move

Table 2-1. DSP Function List (Continued)

vfloat1lr	36 + 1 × Nelements	28	Integer to float conversion, addition of a float offset and left to right move
vfloat1rl	39 + 1 × Nelements	29	Integer to float conversion, addition of a float offset and right to left move
vfloat1rr	39 + 1 × Nelements	29	Integer to float conversion, addition of a float offset and right to right move
vfloat1v	39 + 1 × Nelements	29	Vectorial integer to float conversion and addition of a vectorial float offset
vfloat2ll	37 + 2 × Nelements	33	Integer to float conversion, multiplication by a float scale factor, addition of a float offset and left to left move
vfloat2lr	37 + 2 × Nelements	33	Integer to float conversion, multiplication by a float scale factor, addition of a float offset and left to right move
vfloat2rl	39 + 2 × Nelements	34	Integer to float conversion, multiplication by a float scale factor, addition of a float offset and right to left move
vfloat2rr	39 + 2 × Nelements	34	Integer to float conversion, multiplication by a float scale factor, addition of a float offset and right to right move
vfloat2v	39 + 2 × Nelements	34	Vectorial integer to vectorial float conversion, multiplication by a vectorial float scale factor and addition of a vectorial float offset
vlog10ll	156 + 13 × Nelements	85	Logarithm to base **10** of a float input array and left to right move
vlog10lr	156 + 13 × Nelements	85	Logarithm to base **10** of a float input array and right to left move
vlog10rl	156 + 13 × Nelements	85	Logarithm to base **10** of a float input array and right to left move
vlog10rr	154 + 13 × Nelements	86	Logarithm to base **10** of a float input array and right to right move
vlog10v	143 + 24.5 × Nelements	74	Logarithm to base **10** of a vectorial input array
vlogll	157 + 13 × Nelements	85	Natural logarithm of a float input array and left to left move
vloglr	156 + 13 × Nelements	82	Natural logarithm of a float input array and left to right move
vlogrl	157 + 13 × Nelements	86	Natural logarithm of a float input array and right to left move
vlogrr	154 + 13 × Nelements	86	Natural logarithm of a float input array and right to right move

Table 2-1. DSP Function List (Continued)

vlogv	143 + 24.5 × Nelements	74	Natural logarithm of a vectorial input array
vmagnlrl	30 + 41 × Nelements	31	Vector magnitude
vmagnv	115 + 8.75 × Nelements	84	Complex magnitude
vmaxv	43 + 1 × Nelements	29	Vectorial maximum
vmax1v	54 + 7.25 × Nelements	63	Pipelined vectorial maximum with indexes extraction
vmax2v	33 + 8 × Nelements	35	Vectorial maximum with indexes extraction
vmmul	50 + ((6 × (M - 1)) + 18) × N	42	Product of a complex vector with a complex matrix
vmove2cx	30 + 1× Nelements	26	Complex conjugate vector move with scale factor and offset
vmove2cxint	32 + 2.25 × Nelements	31	Complex conjugate vector integer move with scale factor and offset
vmove2v	28 + 1× Nelements	25	Vectorial move with scale factor and offset
vmove2vint	30 + 2 × Nelements	30	Vectorial integer move with scale factor and offset
vmove2x	30 + 1× Nelements	27	Complex vector move with scale factor and offset
vmove2xint	32 + 2.25 × Nelements	31	Complex integer vector move with scale factor and offset
vmovell	20 + 1× Nelements	18	Left to left float array move
vmovelr	20 + 1 × Nelements	18	Left to right float array move
vmoverl	24 + 1 × Nelements	18	Right to left float array move
vmoverr	23 + 1× Nelements	19	Right to right float array move
vmovev	19 + 1× Nelements	18	Vectorial move
vmvell	54 + 1 × Nelements	29	Mean stored in left memory of a float input array stored in left memory
vmvelr	54 + 1 × Nelements	29	Mean stored in right memory of a float input array stored in left memory
vmverl	54 + 1 × Nelements	30	Mean stored in left memory of a float input array stored in right memory
vmverr	55 + 1 × Nelements	30	Mean stored in right memory of a float input array stored in right memory
vmvev	55 + 1 × Nelements	31	Mean of a vectorial input array
vq2vq	132 + 1 × Nelements	56	Copy of vectorial data from the vector queue 1 to vector queue 2
vrandl	37 + 2.5 × Nelements	41	Random numbers generator in left memory

Table 2-1. DSP Function List (Continued)

vrandr	41 + 2.25 × Nelements	41	Random numbers generator in right memory
vrandv	35 + 4.5 × Nelements	37	Vectorial float array random numbers generator
vrmvesqll	104 + 1 × Nelements	46	Root mean square stored in left memory of an input array stored in left memory
vrmvesqlr	104 + 1 × Nelements	46	Root mean square stored in right memory of an input array stored in left memory
vrmvesqrl	104 + 1 × Nelements	47	Root mean square stored in left memory of an input array stored in right memory
vrmvesqrr	105 + 1 × Nelements	47	Root mean square stored in right memory of an input array stored in right memory
vrmvesqv	109 + 1× Nelements	47	Root mean square of a vectorial input array
vrotate32v	47 + 1 × Nelements	31	Vectorial integer array left or right shift mod.32 with number of shifts (0 to 31)
vshandv	57 + 1 × Nelements	33	Vectorial integer array left or right shift with number of shifts (0 to 31) and logical AND
vshiftv	44 + 1 × Nelements	30	Vectorial integer array left or right shift with number of shifts (0 to 31)
vsinhll	307 + 19 × Nelements	164	Hyperbolic sine of a float input array and left to left move
vsinhlr	303 + 18.5 × Nelements	161	Hyperbolic sine of a float input array and left to right move
vsinhrl	304 + 19 × Nelements	165	Hyperbolic sine of a float input array and right memory to left move
vsinhrr	306 + 18.5 × Nelements	161	Hyperbolic sine of a float input array and right to right move
vsinhv	313 + 31 × Nelements	167	Hyperbolic sine of a vectorial input array
vsinll	117 + 11.25 × Nelements	63	Sine of a float input array and left to left move
vsinlr	117 + 11.25 × Nelements	63	Sine of a float input array and left to right move
vsinrl	119 + 11.25 × Nelements	64	Sine of a float input array and right to left move
vsinrr	118 + 11.25 × Nelements	64	Sine of a float input array and right to right move
vsinv	109 + 21.5 × Nelements	58	Sine of a vectorial input array
vsqrt0ll	118 + 22 × Nelements	55	Single vector square root computation and left to left move

Table 2-1. DSP Function List (Continued)

vsqrt0lr	118 + 22 × Nelements	55	Single vector square root computation and left to right move
vsqrt0rl	118 + 22 × Nelements	55	Single vector square root computation and right to left move
vsqrt0rr	118 + 22 × Nelements	55	Single vector square root computation and right to right move
vsqrt0v	118 + 22 × Nelements	55	Vectorial square root computation
vsqrtll	130 + 7.75 × Nelements	74	Pipelined single vector square root computation and left to left move
vsqrtlr	130 + 7.75 × Nelements	74	Pipelined single vector square root computation and left to right move
vsqrtrl	122 + 7.75 × Nelements	74	Pipelined single vector square root computation and right to left move
vsqrtrr	122 + 7.75 × Nelements	74	Pipelined single vector square root computation and right to right move
vsqrtv	115 + 15.5 × Nelements	66	Pipelined vectorial square root computation
vsubll	27 + 2 × Nelements	22	Subtraction of 2 float array in left memory
vsubrr	32 + 2 × Nelements	20	Subtraction of 2 float array in right memory
vsubv	29 + 2.75 × Nelements	24	Subtraction of 2 vectorial float array
vsumv	44 + 1 × Nelements	27	Sum of vector elements
vtanhll	309 +19.75 × Nelements	165	Hyperbolic tan of a float input array and left to left move
vtanhlr	304 + 18.75 × Nelements	161	Hyperbolic tan of a float input array and left to right move
vtanhrl	302 + 18.75 × Nelements	165	Hyperbolic tan of a float input array and right to left move
vtanhrr	308 + 19 × Nelements	162	Hyperbolic tan of a float input array and right to right move
vtanhv	325 + 30 × Nelements	178	Hyperbolic tan of a vectorial input array
vtanll	142 + 18 × Nelements	79	Tan of a float input array and left to left move
vtanlr	140 + 17.5 × Nelements	79	Tan of a float input array and left to right move
vtanrl	141+ 17.5 × Nelements	79	Tan of a float input array and right to left move
vtanrr	143 + 18 × Nelements	74	Tan of a float input array and right to right move
vtanv	134 + 34.5 × Nelements	74	Tan of a vectorial input array
xcorrc	80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))	94	Cross-correlation between complex float array or auto-correlation of a complex float array

Table 2-1. DSP Function List (Continued)

xcorrlll	80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))	94	Cross-correlation between 2 float float array stored in left memory or auto-correlation of a float array stored in left memory. The result is stored in left memory
xcorrllr	80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))	94	Cross-correlation between 2 float float array stored in left memory or auto-correlation of a float array stored in left memory. The result is stored in right memory
xcorrlrl	80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))	94	Cross-correlation between 2 float array: the first stored in left memory and the second in right memory. The result is stored in left memory
xcorrlrl	80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))	94	Cross-correlation between 2 float array: the first stored in left memory and the second in right memory. The result is stored in right memory
xcorrlrr	80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))	94	
xcorrrrl	80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))	94	Cross-correlation between 2 float array stored in right memory or auto-correlation of a float array stored in right memory. The result is stored in left memory
xcorrrrr	80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))	94	Cross-correlation between 2 float array stored in right memory or auto-correlation of a float array stored in right memory. The result is stored in right memory
xcorrv	80 + (26+20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))	94	Cross-correlation between vectorial float array or auto-correlation of a vectorial float array

Section 3

DSP Functions Description

3.1 cmulcxcy

Function: complex conjugate element by element multiplication

$$Z(k) = conj(X(k)) \times conj(Y(k)) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int cmulcxcy(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: __complex__ float*

strideX**: stride to be used for the X data. ***Type: int

Y**: pointer to the second input vector. ***Type: __complex__ float*

strideY**: stride to be used for the Y data.Type: int***

Z**: pointer to the output vector. ***Type: __complex__ float*

strideZ**: stride to be used for the Z data. ***Type: int

Nelements: number of elements to be computed.***Type: int***

The function cmulcxcy performs complex conjugate element-by-element multiplication on complex vectors only.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Number of cycles:

24 + 2 × Nelements

Number of VLIW:

20

File: cmulcxcy.mas

3.2 cmulcxy

Function: complex element by element multiplication with the first input conjugate

$$Z(k) = conj(X(k)) \times Y(k) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int cmulcxy(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: __complex__ float*

strideX**: stride to be used for the X data. ***Type: int

Y**: pointer to the second input vector. ***Type: __complex__ float*

strideY**: stride to be used for the Y data.Type: int***

Z**: pointer to the output vector. ***Type: __complex__ float*

strideZ**: stride to be used for the Z data. ***Type: int

Nelements: number of elements to be computed.***Type: int***

The function cmulcxy performs complex element-by-element multiplication on complex vectors with first vector conjugate.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Number of cycles:

25 + 2 × Nelements

Number of VLIW:

21

File: cmulcxy.mas

3.3 cmulxy

Function: complex element by element multiplication

$$Z(k) = X(k) \times Y(k) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int cmulxy(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: __complex__ float*

strideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the second input vector. ***Type: __complex__ float*

strideY: stride to be used for the Y data.***Type: int***

Z**: pointer to the output vector. ***Type: __complex__ float*

strideZ: stride to be used for the Z data. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function cmulxy performs complex element-by-element multiplication on complex vectors.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Number of cycles:

25 + 2 × Nelements

Number of VLIW:

21

File: cmulxy.mas

3.4 conv

Function: convolution with complex vectors

$$Y(k) = \sum_{n=0}^{M-1} X(n) \times H(k-n) \qquad k = 0 \ldots N+M-1$$

Synopsis: __vector__ int conv(*X, *H, *Y, N, M, Transient)

Include file: DSPlib.h.

***X**: pointer to the input vector (size N). ***Type: __complex__ float ***

H**: pointer to the filter coefficients (size M). They must be stored in ordinary sequence, i.e. starting from index 0 to M-1.Type: __complex__ float ***

***Y**: pointer to the output vector (size N + M - 1). After function call, Y contains the result of the X vector convolved with the filter. ***Type: __complex__ float ***

N**: input vectors length. ***Type: int

M**: filter length. ***Type: int

Transient**: integer value used to compute or not the transient codes of the convolution:if Transient=0 the transient isn't computed, otherwise it's calculated. ***Type: int

The function conv is the implementation of the convolution of the input vector X with the filter H. The function corresponds to the Matlab conv(a,b) function. The conv function can compute or not the transient states, according to the value set with the Transient parameter: if $Transient = 0$ the transient isn't computed, otherwise it's calculated. For the continuous FIR filtering on an infinite stream of input data, see the function “FIR” on page 3-17.

Restrictions:

N must be an odd value

M must be an even value multiple of 4

Number of cycles:

Initialization:	25
Input transient:	14 + 34 × (M - 1) + 6 × M / 2 × (M - 1)
Steady state:	38 + 44 × L / 2 +13 × M / 4 × L / 2 --> L = N-- M + 1
Output transient:	6 + 35 × (M - 1) + 6 × M / 2 × (M - 1)

Number of VLIW:

123

File: conv.mas

3.5 conv2d

Function: 2-dimensional convolution of complex matrix A with complex kernel matrix H

$$C(r,c) = \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} H[K-1-i][K-1-j] \times A[r+i][c+j] \qquad \begin{cases} r = 0 \ldots M-K+1 \\ c = 0 \ldots N-K+1 \end{cases}$$

Synopsis: __vector__ int conv2d(*A, M, N, *H, *C, K)

Include file: DSPlib.h

A**: pointer to the input complex matrix. ***Type: __complex__ float*

M**: number of rows of matrix AType: int***

N**: number of columns of matrix AType: int***

H**: pointer to the complex kernel matrix. ***Type: __complex__ float*

C**: pointer to the output complex matrix ***Type: __complex__ float*

K**: order of the complex kernel square matrix H. ***Type: int

The function conv2d performs 2-dimensional convolution of matrix A of the order $M \times N$ with matrix H of the order $K \times K$ without the zero-padded edges. It is equivalent to the Matlab function conv2(a,b,'valid). For this reason the output matrix C is of the order. $(M-K+1) \times (N-K+1)$ and not $(M+K-1) \times (N+K-1)$.

Restrictions:

K must be multiple of 2

M must be greater or equal to K

N must be greater or equal to K

Number of cycles:

171 + 4 × K + 3 × (M - K + 1) × (N - K + 1) + (((9 × K / 2 + 30) × (N - K + 1) + 7 × K / 2 + 25) × K / 2 + 28) × (M - K+1)

Number of VLIW:

165

File: conv2d.mas

3.6 cvexp

Function: complex exponential of an input array stored in left memory

$$Y(k) = e^{jX(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int cvexp (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array in vector memory space into which the computed value is written.Type: ***__complex__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function cvexp computes the complex exponential. The complex value obtained is written to a complex array.

Precision:

see Table 3-14 on page 211, Table 3-8 on page 109

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

X must be in memory left

Number of cycles:

137 + 23.5 × Nelements

Number of VLIW:

67

File: cvexp.mas, sinCosCoeff.mas

3.7 cvma

Function: product of 2 complex input arrays and sum with the third complex input array

$$W(k) = X(k) \times Y(k) + Z(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int cvma (*X, strideX, *Y, strideY, *Z, strideZ,*W, strideW, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: __complex__ float *****

strideX: stride to be used for input array X. ***Type: int***

Y**: pointer to the input array . ***Type: __complex__ float *****

strideY: stride to be used for input array Y. ***Type: int***

Z**: pointer to the input array . ***Type: __complex__ float *****

strideZ: stride to be used for input array Z. ***Type: int***

W**: pointer to the output array . ***Type: __complex__ float *****

strideW: stride to be used for output array W. ***Type: int***

Nelements: Number of elements to be computed. ***Type: int***

The function cvma computes the product of two complex arrays and the product obtained is added with the third complex array.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 2

Number of cycles:

37 + 3 × Nelements

Number of VLIW:

33

File: cvma.mas

3.8 cvrdiv

Function: division of a complex array by a real array stored in left memory element by element

$$\begin{cases} Re(Z(k)) = \dfrac{Re(X(k))}{Y(k)} \\ Im(Z(k)) = \dfrac{Im(X(k))}{Y(k)} \end{cases}$$

Synopsis: __vector__ int cvrdiv (*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h

X**: pointer to the complex input array. ***Type: __complex__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the real input array. ***Type: float*

strideY: stride to be used for the real input array. ***Type: int***

Z**: pointer to the complex output array. ***Type: __complex__ float*

strideZ: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be divided.***Type: int***

The function cvrdiv performs the division of a complex array by a real array, element by element.

Restrictions:

Nelements must be multiple of 2

Y must be in left memory

Number of cycles:

83 + 8 × Nelements

Number of VLIW:

51

File: cvrdiv.mas

3.9 FD_RealFIR_Pair

Function: FIR filter on two real signals using two different filter sequences

$$Y(k) = \sum_{n=0}^{M-1} X(n) \times H(k-n) \qquad k = 0\ldots N-1$$

Synopsis: __vector__ int FD_RealFIR_Pair(*W,*X,*data_temp, *Y, *H1, *H2, fft_ptr, ifft_ptr, N)

Include file: DSPlib.h

W**: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/N), with n=0..N/2-1. ***Type: __complex__ float*

X**: pointer to the input vector (size N). ***Type: __complex__ float*

*__data_temp__: pointer to a temporary vector for FFT computation (size N).

Type: __complex__ float*

H1:** pointer to the first filter vector in the frequency domain (size N/2+1)**.** ***Type: __complex__ float*

H2:** pointer to the second filter vector in the frequency domain (size N/2+1)**.** ***Type: __complex__ float*

Y_ptr**: pointer to the output vector (size N). ***Type: __complex__ float*

fft_ptr: memory address for the FFT function to be called. Note that the function depends from the input vector length N. If $N = 256$, then the called function will be ifft256, if $N = 64$, it will be fft64, etc. For the fft_ptr initialization see Section 3.9.1 on page 3-10. ***Type: int***

ifft_ptr: memory address for the IFFT function to be called. Note that the function depends from the input vector length N. If $N = 256$, then the called function will be ifft256, if $N = 64$, it will be ifft64, etc. For the ifft_ptr initialization see Section 3.9.1 on page 3-10. ***Type: int***

N: input vector length. ***Type: int***

Called files:

fft and ifft functions (with fft32M.mas and ifft32M.mas) of the required length.

The function FD_RealFIR_Pair is a library routine used for the computation of couples of real independent FIRs of length M using complex FFTs. This implementation is equivalent to the FIR computation on two real input sequences s1 and s2, both of length N, with the filter coefficients respectively h1 and h2. The difference from a linear convolution implementation is that the one using complex FFTs allows an increase of performances whose amount depends from the length of the filter and from the number of computed elements. It is responsibility of the caller to extract from the output sequence the subsequence corresponding to the desired output (typically the part corresponding to the linear convolution discarding the part corresponding to the circular convolution).

The processing follows the following steps:

1- compute the FFT of a pair of real signals (s1 and s2) using a single complex FFT on $s = s1 + j \times s2$. The complex s signal is obtained storing the s1 real vector in the left memory bank at the address X and the s2 real vector in the right memory bank at the same address of s1

2- FFT post-processing to extract the two complex sequences S1 and S2

3- element by element (.*) product between the FFT of the signal and the FFTs of the filters $(O1 = S1, \times H1)$ and $(O2 = S2, \times H2)$

4- build a complex signal composed by the superposition of the two signals in the frequency domain $(O = O1 + j \times O2)$

5- compute the IFFT of the signal O, obtaining the complex signal o. The result of the FIR filtering of the two real sequences is available as the real and the imaginary part of o:

$$\textbf{\textit{real}}(0) = conv(s1,h1)$$

$$\textbf{\textit{imag}}(0) = conv(s2,h2)$$

Due to the circular convolution implementation, only a subset of the output o data will be equal to the one computed using linear convolution. Note that it is possible to exploit the hermitianity of the FFT of a real signal in order to compute only ½ + 1 points of the post-processed sequence O1 and O2; moreover, due to the same reason, it is possible to store only ½ + 1 of the point of the transform of the filters H1 and H2 in the frequency domain.

3.9.1 C initialization for realFIR function.

Before the FD_RealFIR_Pair call, the integer variables fft_ptr and ifft_ptr must be initialized with the fft and ifft functions pointers. To do this, the following Macro must be used :

```
__GetFuncPtrMem__(name,funcname)
```

where: name is the integerer variable (global or local) initialized with the funcname function pointer

funcname is the function called.

In particular FD_RealFIR_Pair calls 2 functions: fft and ifft , so you need to use the previous Macro for both:

```
__GetFuncPtrMem__(name1,funcname1)
__GetFuncPtrMem__(name2,funcname2)
```

where: name1 is the seventh parameter passed to the FD_RealFIR_Pair function (fft_ptr)

funcname1 is one of the following functions: fft1024, fft512, fft256, fft288, fft128, fft64

name2 is the eighth parameter passed to the FD_RealFIR_Pair function (ifft_ptr)

funcname2 is one of the following functions: ifft1024, ifft512, ifft256, ifft228, ifft128, ifft64

Note: the function FD_RealFIR_Pair uses 75 locations of the stack included that utilized by the fft and ifft functions

Restrictions:

N must be one of the following values: 1024, 128, 256, 288, 512, 64

see the restrictions for the fft and ifft functions

Number of cycles:

268 + 20 × (N / 4 - 5) + fft cycles + ifft cycles

Number of VLIW:

164

File: FD_RealFIR_Pair.mas

3.10 fft1024

Function: complex FFT on 1024 points

$$X(k) = \sum_{n=0}^{1023} W_{1024}^{n \times k} \times x(n) \qquad k = 0 \ldots 1024$$

Synopsis: __vector__ int fft1024(*W, *x, *data_temp, *X)

Include file: DSPlib.h.

****W***: pointer to the ordinary trigonometric coefficients table exp(-i2 × pi × n/1024), with n=0..511. ***Type: __complex__ float****

****x***: pointer to the input vector (size 1024). ***Type: __complex__ float****

****data_temp***: pointer to a temporary vector for FFT computation (size 1024).

Type: __complex__ float*

****X***: pointer to the output vector (size 1024). After function call X contains the FFT of x vector. ***Type: __complex__ float****

The function fft1024 is the mixed radix implementation of the 1024 points FFT. The fft32m assembly function is used as component block. If more than one fft size is used in an application the module fft32m is shared among them.

Note: the function fft1024 uses 75 locations of the stack

Restrictions:

only the following vectors combinations are allowed:

x ≠ data_temp ≠ X

x = data_temp ≠ X

x = X ≠ data_temp

x and X can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

6405

Number of VLIW:

230

File: fft1024.mas

3.11 fft128

Function: complex FFT on fft128 points

$$X(k) = \sum_{n=0}^{127} W_{128}^{n \times k} \times x(n) \qquad k = 0 \dots 127$$

Synopsis: __vector__ int fft128(*W, *x, *data_temp, *X)

Include file: DSPlib.h.

****W***: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/128),with n=0..63. ***Type: __complex__ float****

****x***: pointer to the input vector (size 128). ***Type: __complex__ float****

****data_temp***: pointer to a temporary vector for FFT computation (size 128).

Type: __complex__ float*

*X: pointer to the output vector (size 128). After function call X contains the FFT of x vector. ***Type: __complex__ float****

The function fft128 is the mixed radix implementation of the 128 points FFT. The fft32m assembly function is used as component block. If more than one fft size is used in an application the module fft32m is shared among them.

Note: the function fft128 uses 75 locations of the stack

Restrictions: only the following vectors combinations are allowed:

x ≠ data_temp ≠ X

x = data_temp ≠ X

x = X ≠ data_temp

x and X can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

1053

Number of VLIW:

183

File: fft128.mas

3.12 fft256

Function: complex FFT on fft256 points

$$X(k) = \sum_{n=0}^{255} W_{256}^{x \times k} \times x(n) \qquad k = 0 \ldots 255$$

Synopsis: __vector__ int fft256(*W, *x, *data_temp, *X)

Include file: DSPlib.h.

****W***: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n /256), with n=0...127. ***Type: __complex__ float****

****x***: pointer to the input vector (size 256). ***Type: __complex__ float****

****data_temp***: pointer to a temporary vector for FFT computation (size 256).

Type: __complex__ float*

****X***: pointer to the output vector (size 256). After function call X contains the FFT of x vector. ***Type: __complex__ float****

The function fft256 is the mixed radix implementation of the 256 points FFT. The fft32m assembly function is used as component block. If more than one fft size is used in an application the module fft32m is shared among them.

Note: the function fft256 uses 75 locations of the stack

Restrictions. only the following vectors combinations are allowed:

x ≠ data_temp ≠ X

x = data_temp ≠ X

x = X ≠ data_temp

x and X can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

1729

Number of VLIW:

175

File: fft256.mas

3.13 fft288

Function: complex FFT on 288 points

$$X(k) = \sum_{n=0}^{287} W_{288}^{n \times k} \times x(n) \qquad k = 0 \ldots 287$$

Synopsis: __vector__ int fft288 (*W, *x, *data_temp, *X)

Include file: DSPlib.h.

****W***: pointer to the ordinary trigonometric coefficients table exp(-i*2*pi*n/288), with n=0..143. ***Type: __complex__ float****

****x***: pointer to the input vector (size 288). ***Type: __complex__ float****

****data_temp***: pointer to a temporary vector for FFT computation (size 288).

Type: __complex__ float*

X**: pointer to the output vector (size 288). After function call X contains the FFT of x vector. ***Type: __complex__ float*

The function fft288 is the mixed radix implementation of the 288 points FFT. The fft32m assembly function is used as component block. If more than one fft size is used in an application the module fft32m is shared among them.

Note: the function fft288 uses 75 locations of the stack

Restrictions. only the following vectors combinations are allowed:

x ≠ data_temp ≠ X

x = data_temp ≠ X

x = X ≠ data_temp

x and X can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be in always in Internal Memory

Number of cycles:

2623

Number of VLIW:

193

File: fft288.mas

3.14 fft512

Function: complex FFT on 512 points

$$X(k) = \sum_{n=0}^{511} W_{512}^{n \times k} \times x(n) \qquad k = 0 \ldots 511$$

Synopsis: __vector__ int fft512 (*W, *x, *data_temp, *X)

Include file: DSPlib.h.

W**: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/512), with n=0..255. ***Type: __complex__ float*

x**: pointer to the input vector (size 512). ***Type: __complex__ float*

data_temp**: pointer to a temporary vector for FFT computation (size 512). ***Type: __complex__ float*

X**: pointer to the output vector (size 512). After function call X contains the FFT of x vector. ***Type: __complex__ float*

The function fft512 is the mixed radix implementation of the 512 points FFT. The fft32m assembly function is used as component block. If more than one fft size is used in an application the module fft32m is shared among them.

Note: the function fft512 uses 75 locations of the stack

Restrictions.

only the following vectors combinations are allowed:

x ≠ data_temp ≠ X

x = data_temp ≠ X

x = X ≠ data_temp

x and X can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

3251

Number of VLIW:

178

File: fft512.mas

3.15 fft64

Function: complex FFT on 64 points

$$X(k) = \sum_{n=0}^{63} W_{64}^{n \times k} \times x(n) \qquad k = 0 \ldots 31$$

Synopsis: __vector__ int fft64(*W, *x, *data_temp, *X)

Include file: DSPlib.h.

W**: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/64), with n=0.31. ***Type: __complex__ float*

x**: pointer to the input vector (size 64). ***Type: __complex__ float*

data_temp**: pointer to a temporary vector for FFT computation (size 64) ***Type: __complex__ float*

*X: pointer to the output vector (size 64). After function call X contains the FFT of x vector. ***Type: __complex__ float****

The function fft64 is the mixed radix implementation of the 64 points FFT. The fft32m assembly function is used as component block. If more than one fft size is used in an application the module fft32m is shared among them.

Note: the function fft64 uses 75 locations of the stack

Restrictions.

only the following vectors combinations are allowed:

x ≠ data_temp ≠ X

x = data_temp ≠ X

x = X ≠ data_temp

x and X can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

769

Number of VLIW:

148

File: fft64.mas

3.16 FIR

Function: complex FIR filter

$$Y(k) = \sum_{n=0}^{M-1} X(n) \times H(k-n) \qquad k = 0 \ldots L-1$$

Synopsis: __vector__ int FIR(*X, **address_buffer, *H, *Y, L, M)

Include file: DSPlib.h.

*__*X*__: pointer to the input vector (size L). ***Type: __complex__ float****

__*address_buffer:*__ pointer to the pointer to the delay_line (size M). ***Type: __complex__ float***

*__*H*__: pointer to the FIR filter coefficients (size M). They must be stored in ordinary sequence, i.e. starting from index 0 to M-1. ***Type: __complex__ float****

Y: pointer to the output vector (size L). After function call, Y contains the filtered sequence of data input vector X. ***Type: __complex__float****

L: input and output vectors length. ***Type: int***

M: filter length. ***Type: int***

The function FIR is a FIR filter implementation able to filter complex input vectors of length L with a filter of length M. A running filter can be obtained making infinite calls to the FIR function. In this way it's allowed the computation of a complex vector of infinite length. The input data pointed by "X" are copied in the circular delay-line during the function execution: thus the delay-line is kept updated from function call to call. The assembly function “initFIR” on page 3-40, is used to initialize the FIR computation. It must be called only once, before the first FIR call.For the single execution of the FIR filter function see the function “conv” on page 3-4, which allows computing the FIR filter without maintaining a delay-line (less memory occupation).

Note: the function FIR uses 3 locations of the stack

Restrictions:

L must be an even value

M must be an even value multiple of 4 and greater or equal to 16

L must be less-equal M

Number of cycles:

136 + (79 + 13 × (M / 4 - 3)) × L / 2

Number of VLIW:

99

File: FIR.mas, initFIR.mas

3.17 FirNlmsll

Function: FIR filter computed using Least Mean Square Algorithm

Synopsis: __vector__ int FirNlmsll (*X, *H, *Y, *D, N, P, B)

Include file: DSPlib.h

X**: pointer to the input buffer in vector memory space. ***Type: float*

H**: pointer to the buffer containing filter kernel coefficients. ***Type: float*

Y**: pointer to the buffer containing reference output. ***Type: float*

D**: pointer to the delay buffer of length P. ***Type: float*

N: number of samples over which the fitler is adapted (adaptation time). ***Type: int***

P: filter kernel size. ***Type: int***

B: adaption coefficient. ***Type: float****

The function FirNlmsll computes a FIR filter using coefficients stored in the float array H applied to the elements of the input float array X. The float array H has to be initialized to zero or to meaningful values. The adapted filter coefficients are available in the same buffer at the end of the execution of the function. The Algorithm for the filter is as given below:

1. copy of 1 sample from the input buffer **X** into the delay buffer **D**
2. convolution of **D** by the filter kernel **H** to obtain the output value **T**

$$T[n] = \sum_{k=0}^{P-1} D[n-k] \times H[k]$$

3. compute of the difference between the obtained output and the desired output

$$e = T[n] - Y[n]$$

4. compute of the energy of the previous **P-1** samples stored in the delay buffer

$$E = \sum_{k=0}^{P-1} D[k]^2$$

5. compute of the correction factor by the expression

$$C = \frac{B \times e}{E}$$

6. applying of the correction factor to the filter kernel according as follow

$$H[k] = H[k] + C \times D[k] \ldots k = 0 \ldots P-1$$

Restrictions:

P must be multiple of 4

X must be in left memory

H must be in left memory

Y must be in left memory

D must be in left memory

Number of cycles:

77+ (94+ 4.25 × (P-4)) × (N-P+1) +8.0 × P

Number of VLIW:

130

File: FirNlmsll.mas

3.18 FirNlmsv

Function: pair of FIR filters computed using Least Mean Square Algorithm

Synopsis: __vector__ int FirNlmsv (*X, *H, *Y, *D, N, P, B)

Include file: DSPlib.h

X**: pointer to the input buffer in vector memory space. ***Type: __vector__ float*

H**: pointer to the buffer containing filter kernel coefficients. ***Type: __vector__ float*

Y**: pointer to the buffer containing reference output. ***Type: __vector__ float*

D**: pointer to the delay buffer of length P. ***Type: __vector__ float*

N: number of samples over which the fitler is adapted (adaptation time). ***Type: int***

P: filter kernel size. ***Type: int***

B: adaption coefficient. ***Type: __vector__ float****

The function FirNlmsv computes a pair of FIR filters using coefficients stored in the vectorial float array H applied to the elements of the vectorial float input array X. The vectorial float array H has to be initialized to zero or to meaningful values. The adapted filter coefficients are available in the same buffer at the end of the execution of the function. The Algorithm for the filter is as given below:

1. copy of 1 sample from the input buffer **X** into the delay buffer **D**
2. convolution of **D** by the filter kernel **H** to obtain the output value **T**

$$T[n] = \sum_{k=0}^{P-1} D[n-k] \times H[k]$$

3. compute of the difference between the obtained output and the desired output

$$e = T[n] - Y[n]$$

4. compute of the energy of the previous **P-1** samples stored in the delay buffer

$$E = \sum_{k=0}^{P-1} D[k]^2$$

5. compute of the correction factor by the expression

$$C = \frac{B \times e}{E}$$

6. applying of the correction factor to the filter kernel according as follow

$$H[k] = H[k] + C \times D[k] \ldots k = 0 \ldots P-1$$

Restrictions:

P must be multiple of 4

Number of cycles:

78 + (94+ 4.25 × (P-4)) × (N-P+1) +8.0 × P - 7

Number of VLIW:

135

File: FirNlmsv.mas

3.19 getvq

Function: extraction of vectorial (left - right) data from a vector queue to the destination vector X

Synopsis: int getvq(*q, *X, StrideX, Nelements)

q**: pointer to a queue structure defined using the vqdef macro. ***Type: void *

X**: pointer to the destination vector where the data are copied. ***Type: void *

StrideX**: stride used to write data to the X vector. ***Type: int

Nelements**: number of elements copied. ***Type: int

The function getvq copies the data from the vector queue (q) to the destination buffer (X). If the number of elements in the vector queue is lower than Nelements a -1 is returned (q underrun), but the copy is anyway done. This allows using the getvq also in a non-strictly queued structure, but in structures where circular addressing is used over a vector. A vector queue is a structure defined using the macro "vqdef" and explicitly declared using that macro: see the function “initvq” on page 3-40. If the return code is not checked the structure is simply a circular buffer and the user must guarantee consistency.

Restrictions:

Nelement must be greater than 12 and multiple of 4

Recall:

Nelement can be 2047 elements max

Number of cycles:

65 + 1× Nelements

Number of VLIW:

39

File: getvq.mas

3.20 getvq_f2i

Function: extraction of vectorial (left - right) data from a vector queue to the destination vector and float to integer conversion

Synopsis: int getvq_f2i(*q, *X, StrideX, Nelements)

q**: pointer to a vector queue structure defined using the vqdef macro. ***Type: __vector__ float *

X**: pointer to the destination vector where the data are copied. ***Type: __vector__ int *

StrideX**: stride used to write data to the X vector. ***Type: int

Nelements**: number of elements copied. ***Type: int

The function getvq_f2i copies data from the vector queue to the destination buffer after their conversion from float to integer. If the number of elements in the vector queue is lower than Nelements a -1 is returned (q underrun), but the copy is anyway done. This allows using the getvq_f2i also in a non-strictly queued structure, but in structures where circular addressing is used over a vector. A vector queue is a structure defined using the macro “vqdef” explicitly declared using that macro see the function: “initvq” on page 3-40. If the return code is not checked the structure is simply a circular buffer and the user must guarantee a consistency.

Restrictions:

Nelements must be greater than 12 and multiple of 4

Recall:

Nelements can be 2047 elements max

Number of cycles:

60 + 1 × Nelements

Number of VLIW:

36

File: getvq_f2i.mas

3.21 getvq_i2f

Function: extraction of vectorial (left - right) data from a vector queue to the destination vector and integer to float conversion

Synopsis: int getvq_i2f (*q, *Z, StrideZ, Nelements)

****q***: pointer to a vector queue structure defined using the vqdef macro. ***Type: __vector__ int****

Z**: pointer to the destination vector where the data are copied. ***Type: __vector__ float *

StrideZ**: stride used to write data to the X vector. ***Type: int

Nelements**: number of elements copied. ***Type: int

The function getvq_i2f copies data from the vector queue to the destination buffer after their conversion from integer to float. If the number of elements in the vector queue is lower than Nelements a -1 is returned (q underrun), but the copy is anyway done. This allows using the getvq_i2f also in a non-strictly queued structure, but in structures where circular addressing is used over a vector. A vector queue is a structure defined using the macro “vqdef” explicitly declared using that macro see the function: “initvq” on page 3-40. If the return code is not checked the structure is simply a circular buffer and the user must guarantee consistency.

Restrictions:

Nelements must be greater than 12 and multiple of 4

Recall:

Nelements can be 2047 elements max

Number of cycles:

71 + 1 × Nelements

Number of VLIW:

40

File: getvq_i2f.mas

3.22 getvqelem

Function: number of unread elements in a vector queue

Synopsis: int getvqelem(*q)

q**: pointer to a vector queue structure defined using the vqdef macro: ***Type: void *

A vector queue is a structure defined using the macro "vqdef" and explicitly declared using that macro see the function: “initvq” on page 3-40.

Recall:

the vector queue length can be 2047 elements max

Number of cycles***:***

12

Number of VLIW***:***

4

File: getvqelem.mas

3.23 getvqfree

Function: number of free positions in a vector queue

Synopsis: int getvqfree(*q)

q**: pointer to a vector queue structure defined using the vqdef macro: ***Type: void *

A vector queue is a structure defined using the macro "vqdef" and explicitly declared using that macro see the function: “initvq” on page 3-40.

Recall:

the vector queue length can be 2047 elements max

Number of cycles:

12

Number of VLIW:

4

File: getvqfree.mas

3.24 hilbert [(1)]

Function: discrete time Hilbert function on a complex input vector of N elements

$$Z(k) = ifft[fft(Re(X(k))) \times z(k)] \qquad k = 0 \ldots N-1$$

where ***z(k)*** is a sequence defined as:

$$z(k) = \begin{cases} 1 & for & k = 0 \\ 2 & for & 1 \le k \le N/2-1 \\ 1 & for & k = N/2 \\ 0 & for & N/2+1 \le k < N \end{cases}$$

Synopsis: __vector__ int hilbert(*W, *X, *data_temp, *Y, *Z, fft_ptr, ifft_ptr, N)

Include file: DSPlib.h.

W**: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/N), with n=0..N/2-1.Type: __complex__ float****

X**: pointer to the input vector (size N). ***Type: __complex__ float*

1. See S. Lawrence Marple, Jr. "Computing the Discrete-Time 'Analytic' Signal via FFT", IEEE Transactions on Signal Processing, Vol 47, No 9, September 1999, page 2600.

data_temp**: pointer to a temporary vector for FFT computation (size N) ***Type: __complex__ float*

Y**: pointer to the first output vector (size N). ***Type: __complex__ float*

Z**: pointer to the second output vector (size N). ***Type: __complex__ float*

fft_ptr: integer containing the program memory address for the FFT function to be called. Note that the function depends from the input vector length N. If $N = 256$, then the called function will be fft256, if $N = 64$, it will be fft64, etc.See "C initialization for hilbert function" in the follow, for the fft_ptr initialization. ***Type: int***

ifft_ptr: integer containing the program memory address for the IFFT function to be called. Note that the function depends from the input vector length N. If $N = 256$, then the called function will be ifft256, if $N = 64$, it will be ifft64, etc. See "C initialization for hilbert function" in the follow, for the ifft_ptr initialization. ***Type: int***

N: input and output vectors length. ***Type: int***

The function hilbert computes the Hilbert transform of the real part of a complex input vector (X). It calls the function vmove2v to build a temporary complex vector input in wich the real part is equal to the real part of X and the imaginary part is equal to 0. The real part of the second complex output vector (Z) is the original data input, the imaginary part contains the Hilbert transform.

3.24.1 C initialization for hilbert function.

Before the hilbert call, the integer variables fft_ptr and ifft_ptr must be initialized with the fft and ifft functions pointers. To do this, the following Macro must be used:

```
__GetFuncPtrMem__(name,funcname)
```

where:

name is the integerer variable (global or local) initialized with the funcname function pointer

funcname is the function called.

In particular hilbert calls 2 functions: fft and ifft , so you need to use the previous Macro for both:

```
__GetFuncPtrMem__(name1,funcname1)
__GetFuncPtrMem__(name2,funcname2)
```

where:

name1 is the fifth parameter passed to the hilbert function (fft_ptr)

funcname1 is one of the following functions: fft1024, fft512, fft256, fft288, fft128, fft64 name2 is the sixth parameter passed to the hilbert function (ifft_ptr)

funcname2 is one of the following functions: ifft1024, ifft512, ifft256, ifft228, ifft128, ifft64

In order to use the previous Macro, the file “magic.h” must be included in your project.

Note: the function hilbert uses 75 locations of the stack included that utilized by the fft and ifft functions

Restrictions:

can be 2 differents configuration of the parameter passed to the hilbert function:

1.hilbert(*W, *X, *data_temp, *Z, *Z, fft_ptr, ifft_ptr, N)

2.hilbert(*W, *X, *data_temp, *Y, *Z, fft_ptr, ifft_ptr, N)

the configuration 1 can be used only to store the output of the hilbert function. In this case the output of the fft function is lost. The output of the hilbert function is memorized in the data array Z.

the configuration 2 can be used to store both the output of the fft function and the output of the hilbert function . The first is saved in the data array Y, the second in the data array Z.

N must be one of the following values: 1024, 128, 256, 288, 512, 64

see the restrictions for the fft and ifft functions

Number of cycles:

174 + 2.6875 × N + fft cycles + ifft cycles

Number of VLIW:

113

File: hilbert.mas, vmove2v.mas

3.25 ifft1024

Function: complex IFFT on 1024 points

$$x(k) = \frac{1}{1024}\sum_{n=0}^{1023} W_{1024}^{-n \times k} \times X(n) \quad k = 0 \ldots 1024$$

Synopsis: __vector__ int ifft1024(*W, *X, *data_temp, *x)

Include file: DSPlib.h.

W: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/1024), with n=0..511. ***Type: __complex__ float****

X: pointer to the input vector (size 1024). ***Type: __complex__ float****

data_temp: pointer to a temporary vector for IFFT computation (size 1024).

Type: __complex__ float*

x: pointer to the output vector (size 1024). After function call x contains the FFT of X vector. ***Type: __complex__ float****

The function ifft1024 is the mixed radix implementation of the 1024 points IFFT. The ifft32m assembly function is used as component block. If more than one fft size is used in an application the module ifft32m is shared among them.

Note: the function ifft1024 uses 75 locations of the stack

Restrictions: only the following vectors combinations are allowed:

X ≠ data_temp ≠ x

X = data_temp ≠ x

X = x ≠ data_temp

X and x can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of Cycles:

6527

Number of VLIW:

233

File: ifft1024.mas

3.26 ifft128

Function: complex IFFT on 128 points

$$x(k) = \frac{1}{128}\sum_{n=0}^{128} W_{128}^{-n \times k} \times X(n) \qquad k = 0 \ldots 1024$$

Synopsis: __vector__ int ifft128(*W, *X, *data_temp, *x)

Include file: DSPlib.h.

W**: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/128), with n=0..63.Type: __complex__ float****

X**: pointer to the input vector (size 128). ***Type: __complex__ float*

***data_temp**: pointer to a temporary vector for IFFT computation (size 128).

Type: __complex__ float*

*__x__: pointer to the output vector (size 128). After function call x contains the FFT of X vector. ***Type: __complex__ float****

The function ifft128 is the mixed radix implementation of the 128 points IFFT. The ifft32m assembly function is used as component block. If more than one fft size is used in an application the module ifft32m is shared among them.

Note: the function ifft128 uses 75 locations of the stack

Restrictions.

only the following vectors combinations are allowed:

X ≠ data_temp ≠ x

X = data_temp ≠ x

X = x ≠ data_temp

X and x can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

1112

Number of VLIW:

176

File: ifft128.mas

3.27 ifft256

Function: complex IFFT on 256 points

$$x(k) = \frac{1}{256}\sum_{n=0}^{255} W_{256}^{-n \times k} \times X(n) \quad k = 0 \ldots 255$$

Synopsis: __vector__ int ifft256(*W, *X, *data_temp, *x)

Include file: DSPlib.h.

W**: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/256), with n=0..127. ***Type: __complex__ float*

X**: pointer to the input vector (size 256). ***Type: __complex__ float*

***data_temp**: pointer to a temporary vector for IFFT computation (size 256).

Type: __complex__ float*

*___x___: pointer to the output vector (size 256). After function call x contains the FFT of X vector. ***Type: __complex__ float****

The function ifft256 is the mixed radix implementation of the 256 points IFFT. The ifft32m assembly function is used as component block. If more than one fft size is used in an application the module ifft32m is shared among them.

Note: the function ifft256 uses 75 locations of the stack

Restrictions.

only the following vectors combinations are allowed:

X ≠ data_temp ≠ x

X = data_temp ≠ x

X = x ≠ data_temp

X and x can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

1829

Number of VLIW:

183

File: ifft256.mas

3.28 ifft288

Function: complex IFFT on 288 points

$$x(k) = \frac{1}{288}\sum_{n=0}^{287} W_{288}^{-n \times k} \times X(n) \quad k = 0\ldots287$$

Synopsis: __vector__ int ifft288 (*W, *X, *data_temp, *x)

Include file: DSPlib.h.

*W: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/288), with n=0..143. ***Type: __complex__ float****

X**: pointer to the input vector (size 288). ***Type: __complex__ float*

****data_temp***: pointer to a temporary vector for IFFT computation (size 288).

Type: __complex__ float*

****x***: pointer to the output vector (size 288). After function call x contains the FFT of X vector. ***Type: __complex__ float****

The function ifft288 is the mixed radix implementation of the 288 points FFT. The ifft32m assembly function is used as component block. If more than one fft size is used in an application the module ifft32m is shared among them.

Note: the function ifft288 uses 75 locations of the stack

Restrictions.

only the following vectors combinations are allowed:

X ≠ data_temp ≠ x

X = data_temp ≠ x

X = x ≠ data_temp

X and x can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

2836

Number of VLIW:

179

File: ifft288.mas

3.29 ifft512

Function: complex IFFT on 512 points

$$x(k) = \frac{1}{512}\sum_{n=0}^{511} W_{512}^{-n \times k} \times X(n) \qquad k = 0 \ldots 512$$

Synopsis: __vector__ int ifft512 (*W, *X, *data_temp, *x)

Include file: DSPlib.h.

W**: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/512), with n=0..255. ***Type: __complex__ float*

X**: pointer to the input vector (size 512). ***Type: __complex__ float*

****data_temp***: pointer to a temporary vector for IFFT computation (size 512).

Type: __complex__ float*

****x***: pointer to the output vector (size 512). After function call x contains the FFT of X vector. ***Type: __complex__ float****

The function ifft512 is the mixed radix implementation of the 512 points IFFT. The ifft32m assembly function is used as component block. If more than one fft size is used in an application the module ifft32m is shared among them.

Note: the function ifft512 uses 75 locations of the stack

Restrictions.

only the following vectors combinations are allowed:

X ≠ data_temp ≠ x

X = data_temp ≠ x

X = x ≠ data_temp

X and x can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

3487

Number of VLIW:

181

File: ifft512.mas

3.30 ifft64

Function: complex IFFT on 64 points

$$x(k) = \frac{1}{64}\sum_{n=0}^{63} W_{64}^{-n \times k} \times X(k) \qquad k = o \ldots 31$$

Synopsis: __vector__ int ifft64(*W, *X, *data_temp, *x)

Include file: DSPlib.h.

W**: pointer to the ordinary trigonometric coefficients table exp(-i × 2 × pi × n/64), with n=0..31. ***Type: __complex__ float*

X**: pointer to the input vector (size 64). ***Type: __complex__ float*

***data_temp**: pointer to a temporary vector for IFFT computation (size 64).

Type: __complex__ float*

****x***: pointer to the output vector (size 64). After function call x contains the FFT of X vector. ***Type: __complex__ float****

The function ifft64 is the mixed radix implementation of the 64 points IFFT. The ifft32m assembly function is used as component block. If more than one fft size is used in an application the module ifft32m is shared among them.

Note: the function ifft64 uses 75 locations of the stack

Restrictions.

only the following vectors combinations are allowed:

X ≠ data_temp ≠ x

X = data_temp ≠ x

X = X ≠ data_temp

X and x can be allocated in Internal Memory, in Buffer Memory or in Parm memory

data_temp must be always in Internal Memory

Number of cycles:

767

Number of VLIW:

151

File: ifft64.mas

3.31 IIR1

Function: cascaded vectorial IIR biquad sections on input sequences

Synopsis: __vector__ int IIR1(*chan, *in, *out)

Include file: DSPlib.h.

chan**: pointer to an "iir_biquad_struct" structure. ***Type: pointer to the structure name type used in the declaration (see later for description)

****in***: pointer to the input vector. ***Type: __vector__ float ****

***out**: pointer to the output vector. ***Type: __vector__ float ***

A running filter can be obtained making infinite calls to the IIR1 function. This allows filtering infinite length vectors. This implementation is pipelined on the stages parameter, i.e. it is best suited when the number of stages is greater than the number of input data channel and of the number of output samples to be computed at each call. See the function "IIR2" on page 3-36 for other function optimization flavors. At least one data structure of the type "type_name" must be declared to allow proper function execution. The structure of type "type_name" can be declared using the macro ***IIR1_struct_def*** (see DSPlib.h). The structure "type_name" contains the coefficients and the status values of the different biquad sections for each stage and processing channel, the gain value for the different processing channels and the pointers to the array declared with structure.

```
typedef struct type_name
{
        float *a_circ_ptr;
        float *b_circ_ptr;
        float *k_circ_ptr;
        float *s_circ_ptr;
        float *G_circ_ptr;
        float a[channel_Nr][stage_Nr][2];
        float b[channel_Nr][stage_Nr][2];
        float k[channel_Nr][stage_Nr][2];
        float s[channel_Nr][stage_Nr][2];
        float Gain[channel_Nr];
                                        ]type_name;
#define IIR1_struct_def(tag, stage_Nr, channel_Nr) typedef struct tag {  \
  __vector__ float *a_circ_ptr;\
  __vector__ float *b_circ_ptr;\
  __vector__ float *k_circ_ptr;\
  __vector__ float *s_circ_ptr;\
  __vector__ float *G_circ_ptr;\
  __vector__ float a[channel_Nr][stage_Nr][2];\
  __vector__ float b[channel_Nr][stage_Nr][2];\
  __vector__ float k[channel_Nr][stage_Nr][2];\
  __vector__ float s[channel_Nr][stage_Nr][2];\
  __vector__ float Gain[channel_Nr];} tag;
```

The IIR1_struct_def has as parameters the type name (type_name) to be assigned to the structure, the number of IIR channels to be processed (channel_Nr) and the IIR filter stage number (stage_Nr).

The input and output vectors must be ordered as follows:

x1(k), x1(k+1), x2(k), x2(k+1), x3(k), x3(k+1), ... xNch(k), xNch(k+1),

x1(k+2), x1(k+3), x2(k+2), x2(k+3), x3(k+2), x3(k+3), ... xNch(k+2), xNch(k+3),

....

x1(k+Ns-1), x1(k+Ns), x2(k+Ns-1), x2(k+Ns), x3(k+Ns-1), x3(k+Ns), ... xNch(k+Ns-1), xNch(k+Ns)

i.e. the input and output vectors are composed by Nch pair of data (vectorial processing) with consecutive indexes k and k+1. If more than two samples must be computed then the different samples will follow in the array.The input and output vector structure is:

x[Ns/2][Nch][2].

The ordinary (anyway vectorial) filtering of an input sequence like the Matlab "filter(b,a,x)" can be performed using stage_Nr = length(x), channel_Nr = 1 and initializing the coefficent arrays as described later. A single biquad section has the following form:

$$H(z) = gain \times \frac{1 + b_1 \times z^{-1} + b_2 \times z^{-2}}{1 - a_1 \times z^{-1} - a_2 \times z^{-2}}$$

The Gain term is the DC gain of the biquad section.

The equations implementing the canonical form II of the IIR filter are:

$$w(n) = x(n) + a1 \times w(n-1) + a2 \times w(n-2)$$

$$y(n) = w(n) + b1 \times w(n-1) + b2 \times w(n-2)$$

In the actual implementation of a cascade of sections, one single multiply is performed at the end, combining all the gains of the cascade.

In order to decouple the output computation equation from the state evolution equation, allowing better computation pipelining, the following modification has been done on the biquad equation:

w(k) = x(k)   + a1 × w(k-1) + a2 × w(k-2);

w(k+1)   = x(k+1) + a1 × x(k) + k1 × w(k-1) + a1 × a2 × w(k-2);

y(k) = x(k)   + b1 × w(k-1) + b2 × w(k-2);

y(k+1) = x(k+1) + b1 × x(k) + k2 × w(k-1) + b1 × a2 × w(k-2);

Here w(k), w(k+1), y(k) and y(k+1) depends only on w(k-1) and w(k-2).

If the coefficients are expressed in the Matlab format, the following equations have to be used to compute the coefficients a1, a2, b1, b2, k1 and k2:

a1 = -a(2);

a2 = -a(3);

b1 = (-a(2)+b(2)/b(1));

b2 = (-a(3)+b(3)/b(1));

k1 = a1^2 + a2;

k2 = a1 × b1 + b2;

The coefficients must be stored in the respective arrays ordered as follows:

a[Channel][Stage][0] = a2

a[Channel][Stage][1] = a1

b[Channel][Stage][0] = b2

b[Channel][Stage][1] = b1

k[Channel][Stage][0] = k1

k[Channel][Stage][1] = k2

Note the index inversion for the array "a" and "b". This coefficients are used in the computation performed by the function "IIR1".The assembly function “Init_IIR1_struct” on page 3-38 is used to initialize the private IIR filter data structure. It must be called before the first "IIR1" call. Then the function "init_IIR1_struct" can be called to clear the status of the IIR filter; the coefficient values will not be affected. The init_IIR1_struct function must be called to initialize the pointer to the status locations and coefficient locations with the correct values of the S,L,A,M and P fields for appropriate looping and circular addressing control.

Note: the function IIR1 uses 10 locations of the stack

Restrictions:

the number of stages must be greater or equal to 3

the number of input and output samples pair must be multiple of 2

Number of cycles:

189 + [47 + 14 × (Stages_Nr - 2)] × Ch_Nr × Samples_Nr/2

Number of VLIW***:***

109

File: IIR1.mas, Init_IIR1_struct.mas

3.32 IIR2

Function: cascaded vectorial IIR biquad sections on input sequences (one sample on the left and one on the right data memory)

Synopsis: __vector__ int IIR2(*chan, *in)

Include file: DSPlib.h.

chan: pointer to an "iir_biquad_struct" structure. ***Type: pointer to the structure name type used in the declaration (see later for description)***

****in***: pointer to the input vector. ***Type: __vector__ float ****

A running filter can be obtained making infinite calls to the “IIR1” on page 3-33 function. This allows filtering infinite length vectors. This function works "in place", overwriting the input data with the output results.

The initialization function “Init_IIR2_struct” on page 3-39 initializes the coefficient for independent biquad section coefficient values. All the biquad sections can be different allowing multiple-input-multiple-output computation in true multichannel way. The function filters N_IIR_CH channel with N_IIR_CH different filter cascade.

This implementation is pipelined on the channels and stages parameter, i.e. it is best suited when the number of stages is greater than the number of input data channel and of the number of output samples to be computed at each call. See also IIR1 for other function optimization flavors.

At least one data structure of the type "type_name" must be declared to allow proper function execution. The structure of type "type_name" can be declared using the macro IIR2_struct_def (see DSPlib.h). The structure "type_name" contains the coefficients and the status values of the different biquad sections for each stage and processing channel, the gain value for the different processing channels and the pointers to the array declared with structure.

The structure is:

```
typedef struct type_name
{
        __vector__ float × a_circ_ptr;
        __vector__ float × k_circ_ptr;
        __vector__ float × w_circ_ptr;
        __vector__ float a[stage_Nr][channel_Nr][2];
        __vector__ float k[stage_Nr][channel_Nr][3];
        __vector__ float w[stage_Nr][channel_Nr][2]
                                        } type_name;
```

The IIR2_struct_def has as parameters a type name (type_name) to be assigned to the structure, a number of IIR channels to be processed (channel_Nr) and a IIR filter stage number (stage_Nr).

Note: the function IIR2 uses 10 locations of the stack

Restrictions:

the number of samples must be greater than 0;

the number of input channel must even and greater than 5 (6 min) the product Number_of_Stages × Number_of_Samples must be lower than 2048 ince data are stored in an array of contiguous locations and thus the restriction on max array size applies

the init_IIR2_struct function must be called to initialize the pointer to the status locations and coefficient locations with the correct values of the S,L,A,M and P fields for appropriate looping and circular addressing control.

Number of cycles:

187 + [66 + 20 × (Stages_Nr × Ch_Nr -4)/ 2] × Samples_Nr

Number of VLIW:

122

File: IIR2.mas, Init_IIR2_struct.mas

3.33 Init_IIR1_struct

Function: initialization procedure for the IIR1 function

Synopsis: void init_IIR1_struct(*bq_ptr, Ch_Nr, Stages_Nr, Samples_Nr)

Include file: DSPlib.h.

bq_ptr**: pointer to a structure of the type defined using the IIR1_struct_def macro (see IIR1 description) containing the coefficient and status values for the IIR1 function. ***Type: type name assigned using the IIR1_struct_def *

Ch_Nr: number of independent input channels to be processed. ***Type: int***

Stages_Nr: number of biquad stages composing the desired filter. ***Type: int***

Samples_Nr: number of samples to be filtered (equal to the number of samples produced in output).***Type: int***

The function init_IIR1_struct is used to initialize the structure used by the function “IIR1” on page 3-33. The operations performed are:

initialization of the status locations with all elements equal to 0.0f (vectorial)

initialization of the pointer to the status locations and coefficient locations with the correct values of the S,L,A,M and P fields for appropriate looping and circular addressing control.

Restrictions:

the restrictions are the same of the function IIR1, but they are not checked by the function

the number of stages (Stages_Nr) must be greater than 3 the number of input and output samples pair (Samples_Nr) must be multiple of 2.

Number of cycles:

277 + 6 × Stages_Nr × Ch_Nr × 2

Number of VLIW:

49

File: Init_IIR1_struct.mas

3.34 Init_IIR2_struct

Function: initialization procedure for the IIR2 function

Synopsis: void init_IIR2_struct(*bq_ptr, Ch_Nr, Stages_Nr, Samples_Nr)

Include file: DSPlib.h.

bq_ptr**: pointer to a structure of the type defined using the IIR1_struct_def macro (see IIR2 description) containing the coefficient and status values for the IIR2 function. ***Type: type name assigned using the IIR2_struct_def *

Ch_Nr: number of independent input channels to be processed. ***Type: int***

Stages_Nr: number of biquad stages composing the desired filter. ***Type: int***

Samples_Nr: number of samples to be filtered (equal to the number of samples produced in output).***Type: int***

init_IIR2_struct is used to initialize the structure used by the IIR2 function.The operations performed are:

initialization of the status locations with all elements equal to 0.0f (vectorial);

initialization of the pointer to the status locations and coefficient locations with the correct values of the S,L,A,M and P fields for appropriate looping and circular addressing control.

Restrictions:

the restrictions are the same of the function IIR2, but they are not checked by the function

Number of cycles:

204 + 6 × Stages_Nr × Ch_Nr × 2

Number of VLIW:

64

File: Init_IIR2_struct.mas

3.35 initFIR

Function: initialization procedure for the FIR function

Synopsis: __vector__ int initFIR(**address_buffer, M)

Include file: DSPlib.h.

**address_buffer: pointer to the pointer to the delay_line (size M) used in the FIR filter.
Type: __complex__ float*

M: delay_line length. ***Type: int***

The function initFIR is used to initialize the variables of a FIR filter. The operations performed are:

- initialization of the delay_line with all elements equal to (0+ 0i)
- initialization of the pointer to the delay_line (**address_buffer) with the correct values of L (length = M), A (index = last element in the vector, because it's used in decrement mode) and M (increment = -1, because it's used in decrement mode)

Restrictions:

M must be an even value multiple of 4 and greater or equal to 16

Number of cycles:

$35 + 3 \times M$

Number of VLIW:

23

File: initFIR.mas

3.36 initvq

Function: initialization of the data structures used to manage a vector circular buffer (vector queue)

Synopsis: void initvq(*q, Nelements)

Include file: DSPlib.h.

****q***: pointer to a vector queue structure defined using the macro vqdef.
Type: void *

Nelements: length of the queue array contained in the structure: ***Type: int***

A vector queue is a structure defined using the macro "vqdef" explicitly declared:

```
#define vqdef(tag, type, size) typedef struct tag {  \
        int qlen;                     \
        int Nelements __attribute__((packed)); \
        void *wptr;                     \
        void *rptr;                     \
        type queue[Nelements];} tag;
```

The pointer locations are set to the proper values in order to be used by the put and get functions. Specifically the S field of q.wptr and q.rptr are the address of queue[0] while A is set to 0.

This pseudo code describes the function performed:

```
q.wptr(S field) = &(q.queue[0]);
q.rptr(S field) = &(q.queue[0]);
q.wptr(L field) = Nelements;
q.rptr(L field) = Nelements;
q.wptr(A field) = 0;
q.rptr(A field) = 0;
q.wptr(M field) = 1;
q.rptr(M field) = 1;
```

Restrictions:

the vector queue length can be 2047 elements max

Number of cycles:

45

Number of VLIW:

22

File: initvq.mas

3.37 LastStage

Function: "plain" radix two butterfly

$$Y1(k) = X1(k) + W(k) \times X2(k) \qquad k = 0 \ldots N-1$$
$$Y2(k) = X1(k) + W(k) \times X2(k)$$

Synopsis: __vector__ int LastStage(*X1, *X2, *W, *Y1, *Y2, N)

Include file: DSPlib.h.

X1**: pointer to X1 input vector. ***Type: __complex__ float*

X2**: pointer to X2 input vector. ***Type: __complex__ float*

W**: pointer to W coefficient vector. ***Type: __complex__ float*

Y1**: pointer to Y1 output vector. ***Type: __complex__ float*

Y2**: pointer to Y2 output vector. ***Type: __complex__ float*

N**: number of butterfly to be computed. ***Type: int

The function LastStage can be used as last FFT stage of a complete FFT by proving the proper coefficients.

Note: the function LastStage uses 3 locations of the stack

Restrictions:

N must be greater or equal to 8 and multiple of 4

Number of cycles:

137 + 3.25 × N

Number of VLIW:

71

File: LastStage.mas

3.38 levinson

Function: Levinson-Durbin recursion

$$M \times LPC = B \rightarrow \begin{bmatrix} R(0) & R(1) & \dots & R(P-1) \\ R(1) & R(0) & \dots & R(P-2) \\ \dots & \dots & \dots & \dots \\ R(P-1) & R(P-2) & \dots & R(0) \end{bmatrix} \times \begin{bmatrix} LPC(1) \\ LPC(2) \\ \dots \\ LPC(P) \end{bmatrix} = \begin{bmatrix} R(1) \\ R(2) \\ \dots \\ R(P) \end{bmatrix}$$

The previous set of equations computes the predictor coefficients LPC[P]. This set is solved using the Levinson-Durbin recursion:

$$E^{(0)} = R(0)$$

$$k^{(i)} = \frac{R(i) - \sum_{j=1}^{i-1} LPC(j)^{(i-1)} \times R(i-j)}{E^{(i-1)}} \qquad 1 \le j \le i-1 \qquad 1 \le i \le p$$

$$LPC(i)^{(i)} = k^{(i)}$$

$$LPC(j)^{(i)} = LPC(j)^{(i-1)} - k^{(i)} \times LPC(i-j)^{(i-1)}$$

$$E^{(i)} = (1 - (k^{(i)})^2) \times E^{(i-1)}$$

Synopsis: levinson(float *R, float *LPC, float *d, int P)

Include file: DSPlib.h.

R**: pointer to the autocorrelation input vector. ***Type: float *

LPC**: pointer to the output vector. ***Type: float*

*d: pointer to the scalar output. It stores the value of E calculated in the last iteration : $E^{(P)}$. ***Type: float****

P: number of coefficients to be computed. ***Type: int***

The function levinson solves the P^{th} order system of linear equations: $M \times LPC = B$ described above in matrix format, for the particular case where M is a real symmetric, Toeplitz matrix and B is identical to the first coloumn of M shifted by one element.

Restrictions:

R must be in the left memory

LPC must be in the left memory

d must be in left memory

Number of cycles:

3297 (P = 11)

Number of VLIW:

131

File: levinson.mas

3.39 lpc2cep

Function: cepstral coefficients of a real float array in left memory

Synopsis: lpc2cep (float *X, float *CEP, int N, int M)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: float *

CEP**: pointer to the output vector. ***Type: float*

N: length of the input vector X. ***Type: int***

P: number of coefficients to be computed. ***Type: int***

The function lpc2cep returns int the float arry CEP, the cepstrum of the real float array X.

Restrictions:

X must be in the left memory

CEP must be in the left memory

Number of cycles:

5074 (N = 11 and M = 32)

Number of VLIW:

122

File: lpc2cep.mas

3.40 madd

Function: sum of two complex matrices

$$C(r,c) = A(r,c) + B(r,c) \qquad \begin{cases} r = 0 \ldots M-1 \\ c = 0 \ldots N-1 \end{cases}$$

Synopsis: __vector__ int madd(*A, *B, M, N, *C)

Include file: DSPlib.h

A**: pointer to the first input matrix . ***Type: __complex__ float *****

B**: pointer to the second input matrix . ***Type: __complex__ float *****

M**: number of rows of matrix A and matrix B. ***Type: int

N: number of columns of matrix A and matrix B ***Type: int***

****C***: pointer to the output matrix . ***Type:*** __complex__ float *****

The function madd computes the sum of 2 complex matrices of order $M \times N$.

Restrictions:

the product of $M \times N$ must be > 1.

Number of cycles:

35 + 7 × (M × N / 2 -1)

Number of VLIW:

25

File: madd.mas

3.41 mchol

Function: L - U decomposition of a positive definite square matrix using Cholesky algorithm

$$A(r,c) = \sum_{i=0}^{N-1} L(r,i) \times U(i,c) \qquad \begin{cases} r = 0 \ldots N-1 \\ c = 0 \ldots N-1 \end{cases}$$

Synopsis: __vector__ int mchol (*A, *L, *U, N)

Include file: DSPlib.h

****A***: pointer to the input square matrix . ***Type:*** __complex__ float *****

****L***: pointer to the output square matrix into which the decomposed lower triangular matrix is written. ***Type:*** __complex__ float *****

****U***: pointer to the output square matrix into which the decomposed upper triangular matrix is written . ***Type:*** __complex__ float *****

N: order of matrix A. ***Type: int***

The function mchol decomposes a positive definite complex square matrix A into the lower and upper triangular complex matrices L and U respectively using Cholesky decomposition algorithm.

Note: the function mchol uses 3 locations of the stack

Restrictions:

N should be > 3

Number of cycles:

$0.4166 \times N^3 + 23.75 \times N^2 + 47.84 \times N + 138$

Number of VLIW:

212

File: mchol.mas

3.42 mdeterm

Function: determinant of a complex matrix of the order $N \times N$

$$C = det\ A$$

Synopsis: __vector__ int mdeterm (*A, N, *C)

Include file: DSPlib.h

A**: pointer to the input square matrix . ***Type: __complex__ float *****

N**: order of matrix A. ***Type: int

C**: pointer to the output scalar . ***Type: __complex__ float *****

The function mdeterm computes the determinant of a complex square matrix A of the order $N \times N$. Gaussian elimination with partial pivoting is used for finding the determinant. In place decomposition of matrix A into upper triangular matrix takes place and hence the original values of input matirx is lost. The computed determinant value is written to a complex scalar value whose pointer is passed to the function.

Note: the function mdeterm uses 5 locations of the stack

Restrictions:

N should be > 3

Number of cycles:

$28 + 1.33 \times N^3 + 23 \times N^2 + 36.5 \times N$ + Cycles for swap operation, which is data dependent

Number of VLIW:

195

File: mdeterm.mas

3.43 mdeterm2

Function: determinant of a complex matrix of the order 2×2

$$C = det\ A$$

Synopsis: __vector__ int mdeterm2 (*A, *C)

Include file: DSPlib.h

A**: pointer to the input square matrix . ***Type: __complex__ float *

C**: pointer to the output complex scalar . ***Type: __complex__ float *

The function mdeterm2 computes the determinant of a complex square matrix A of the order 2×2 .

Number of cycles:
29

Number of VLIW:
9

File: mdeterm2.mas

3.44 mdeterm3

Function: determinant of a complex matrix of the order 3×3

$$C = det\ A$$

Synopsis: __vector__ int mdeterm3 (*A, *C)

Include file: DSPlib.h

A**: pointer to the input square matrix . ***Type: __complex__ float *

C**: pointer to the output complex scalar . ***Type: __complex__ float *

The function mdeterm3 computes the determinant of a complex square matrix A of the order 3×3.

Number of cycles:

70

Number of VLIW:

22

File: mdeterm3.mas

3.45 minvert

Function: inverse of a complex square matrix of the order $N \times N$ matrix

$$C = A^{-1}$$

Synopsis: __vector__ int minvert (*A, *C, N)

Include file: DSPlib.h

A**: pointer to the input square matrix . ***Type: __complex__ float *

C**: pointer to the output square matrix . ***Type: __complex__ float *

N**: order of matrix A. ***Type: int

The function minvert computes the inverse of a complex square matrix A of the order $N \times N$. Gaussian-Jordan elimination with partial pivoting is used for finding the inverse. In place decomposition of matrix A into upper triangular matrix takes place and hence the original values of input matirx A is lost. The inverse of the matrix A computed is written to the complex output vector C.

Note: the function minvert uses 9 locations of the stack

Restrictions:

N should be > 3

Number of cycles:

4.66 × N^3 + 68.5 × N^2 - N × 18.17 - 44 + 130 + Cycles for swap operation which is data dependent

Number of VLIW:

400

File: minvert.mas

3.46 mmul

Function: product of two complex matrices

$$C(r,c) = \sum_{i=0}^{N-1} A(r,i) \times B(i,c) \qquad \begin{cases} r = 0 \ldots M-1 \\ c = 0 \ldots P-1 \end{cases}$$

Synopsis: __vector__ int mmul(*A, M, N, *B, P, *C)

Include file: DSPlib.h

A**: pointer to the first input matrix . ***Type: __complex__ float *****

M: number of rows of matrix A. ***Type: int***

N: number of columns of matrix A and rows of matrix B. ***Type: int***

B**: pointer to the second input matrix . ***Type: __complex__ float *****

P: number of columns of matrix B. ***Type: int***

C**: pointer to the output matrix . ***Type: __complex__ float *****

The function mmul computes the product of 2 complex matrices of order $M \times N$ and $N \times P$ respectively. The output matrix is of the order $M \times P$.

Restrictions:

N should be > 1.

Number of cycles:

112+ (((((6 × (N-1)+13) × M)+11) × P)

Number of VLIW:

56

File: mmul.mas

3.47 mtrace

Function: trace of $N \times N$ complex matrix

$$Y = \sum_{i=0}^{N-1} A(i,i)$$

Synopsis: __vector__ int mtrace(*A, N, *Y)

Include file: DSPlib.h

A**: pointer to the input matrix . ***Type: __complex__ float *****

N: order of the input matrix . ***Type: int***

Y**: pointer to the output complex scalar. ***Type: __complex__ float *****

The function mtrace computes the trace of a complex matrix of order $N \times N$.

Number of cycles:

35 + 5 × N / 2

Number of VLIW:

22

File: mtrace.mas

3.48 mvmul

Function: product of a complex matrix with a set of complex vectors

$$Y_k(i) = \sum_{j=0}^{N-1} A(i,j) \times X_k(j) \qquad i = 0 \ldots M-1 \qquad k = 0 \ldots P$$

Synopsis: __vector__ int mvmul (*A, M, N, *B, *C, P)

Include file: DSPlib.h

A**: pointer to the input matrix A(i,j). The matrix must be stored by row (row-major order). ***Type: __complex__ float *****

M: number of rows of matrix A. ***Type: int***

N: number of columns of matrix A and rows of matrix B. ***Type: int***

X**: pointer to the second input vector. ***Type: __complex__ float *****

Y**: pointer to the output vector . ***Type: __complex__ float *****

P**: number of vectors. ***Type: int

The function mvmul computes the product of a matrix of order $M \times N$ with P vectors each of length N. The matrix A(i, j) is loaded into the register file and then is used to multiply the vectors. The input vector X must be stored in memory in subsequent locations i.e. data storage must be equivalent to an array of vectors: X[P][N]. The output vector Y will be equivalent to an array of vectors: X[P][M].

Restrictions:

N should be > 1

Number of cycles:

46 + ((((6 × (N-1)) + 17) × M) + 11) × P

Number of VLIW:

48

File: mvmul.mas

3.49 mvmul3x3

Function: product of a complex 3×3 matrix with a set of complex vectors of size 3

$$Y_k(i) = \sum_{j=0}^{2} A(i,j) \times X_k(j) \qquad i = 0 \ldots 2 \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int mvmul3x3(*A, *X, *Y, Nelements)

Include file: DSPlib.h.

A**: pointer to the input matrix A(i,j). The matrix must be stored by row (row-major order).Type: __complex__ float****

X**: pointer to the second input vector. ***Type: __complex__ float*

Y** pointer to the output vector. ***Type: __complex__ float*

Nelement**s: number of input vectors. ***Type: int

The function mvmul3x3 executes the multiply of a matrix by a set of vectors. The matrix A(i, j) is loaded into the register file and then is used to multiply the vectors. The input vector X must be stored in memory in subsequent locations (i.e. data storage must be equivalent to an array of vectors: X[Nelements][3]). The output vector Y will be written in memory like the input vector X.

Number of cycles:

$59 + 9 \times Nelements$

Number of VLIW:

44

File: mvmul3x3.mas

3.50 mvmul4x4

Function: product of a complex 4×4 matrix with a set of complex vectors of size 4

$$Y_k(i) = \sum_{j=0}^{3} A(i,j) \times X_k(j) \qquad i = 0 \ldots 3 \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int mvmul4x4(*A, *X, *Y, Nelements)

Include file: DSPlib.h.

*A: pointer to the input matrix A(i,j). The matrix must be stored by row (row-major order). ***Type: __complex__ float****

*X: pointer to the second input vector. ***Type: __complex__ float****

*Y: pointer to the output vector. ***Type: __complex__ float****

Nelements: number of input vectors. ***Type: int***

The function mvmul4x4 executes the multiply of a matrix by a set of vectors. The matrix A(i, j) is loaded into the register file and then is used to multiply the vectors. The input vector X must be stored in memory in subsequent locations (i.e. data storage must be equivalent to an array of vectors:X[Nelements][4]). The output vector Y will be written in memory like the input vector X.

Number of cycles:

$125 + 16 \times Nelements$

Number of VLIW:

68

File: mvmul4x4.mas

3.51 mvmul8x8

Function: product of a complex 8×8 matrix with a set of complex vectors of size 8

$$Y_k(i) = \sum_{j=0}^{7} A(i,j) \times X_k(j) \qquad i = 0\ldots7 \qquad k = 0\ldots Nelements$$

Synopsis: __vector__ int mvmul8x8(*A, *X, *Y, Nelements)

Include file: DSPlib.h.

A**: pointer to the input matrix A(i,j). The matrix must be stored by row (row-major order). ***Type: __complex__ float*

X**: pointer to the second input vector. ***Type: __complex__ float*

*__*Y*__: pointer to the output vector. ***Type: __complex__ float****

Nelements**: number of input vectors. ***Type: int

The function mvmul8x8 executes the multiplication of a matrix by a set of vectors. The matrix A(i, j) is loaded into the register file and then is used to multiply the vectors. The input vector X must be stored in memory in subsequent locations (i.e. data storage must be equivalent to an array of vectors: X[Nelements][8]). The output vector Y will be written in memory like the input vector X.

Note: the function mvmul8x8 uses 168 locations of the stack

Number of cycles:

461 + 69 × Nelements

Number of VLIW:

203

File: mvmul8x8.mas

3.52 pack40to16ll

Function: multiplication by a float value, addition of a float offset, clipping in a float range of a pair of data in left memory: X1 and X2 and conversion of the results in 16 bit integer arranged in a 32 bit word in left memory

$$Y(k) = round(clip(X1 \times Scale + Offset)) \ll 16 \quad or \quad round(clip(X2 \times Scale + Offset))$$

Synopsis: __vector__ int pack40to16ll(*X, strideX, *Y, strideY, scale, offset, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

***X**: pointer to the input vector (size N). ***Type: float ***

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the output vector (size N/2). ***Type: int*

StrideY: stride to be used for the Y data. ***Type: int***

Scale: scalar multiply factor to scale the input vector. T***ype: float***

Offset: scalar offset to be added to the input vector.***Type: floa***t

ClipUp: value to be used as upper limit for the data. ***Type: float***

ClipDown: value to be used as lower limit for the data. ***Type: float***

Nelements: number of elements to be computed. ***Type: int***

The function pack40to16ll takes pair of data X: X1 and X2, scales them by a float factor, adds a float offset, clips the values in a float range and converts the results to a pair of 16 bit integer arranged in a 32 bit word Y.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in the left memory

Y must be in the left memory

ClipUp must be less or equal to 2^15 - 1

ClipDown must be greater or equal to -2^(16-1)

Number of cycles:

39 + 6 × Nelements

Number of VLIW:

40

File: pack40to16ll.mas

3.53 pack40to16lr

Function: multiplication by a float value, addition of a float offset, clipping in a float range of a pair of data in left memory: X1 and X2 and conversion of the results in 16 bit integer arranged in a 32 bit word in right memory

$$Y(k) = round(clip(X1 \times Scale + Offset)) \ll 16 \quad or \quad round(clip(X2 \times Scale + Offset))$$

Synopsis: __vector__ int pack40to16lr(*X, strideX, *Y, strideY, scale, offset, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector (size N). ***Type: float *

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the output vector (size N/2). ***Type: int*

StrideY: stride to be used for the Y data. ***Type: int***

Scale: scalar multiply factor to scale the input vector. T***ype: float***

Offset: scalar offset to be added to the input vector.***Type: floa***t

ClipUp: value to be used as upper limit for the data. ***Type: float***

ClipDown: value to be used as lower limit for the data. ***Type: float***

Nelements: number of elements to be computed. ***Type: int***

The function pack40to16lr takes pair of data X: X1 and X2, scales them by a float factor, adds a float offset, clips the values in a float range and converts the results to a pair of 16 bit integer arranged in a 32 bit word Y.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in the left memory

Y must be in the right memory

ClipUp must be less or equal to 2^15 - 1

ClipDown must be greater or equal to -2^(16-1)

Number of cycles:

39 + 6 × Nelements

Number of VLIW:

41

File: pack40to16lr.mas

3.54 pack40to16rl

Function: multiplication by a float value, addition of a float offset, clipping in a float range of a pair of data in right memory: X1 and X2 and conversion of the results in 16 bit integer arranged in a 32 bit word in left memory

$$Y(k) = round(clip(X1 \times Scale + Offset)) \ll 16 \qquad or \qquad round(clip(X2 \times Scale + Offset))$$

Synopsis: __vector__ int pack40to16rl(*X, strideX, *Y, strideY, scale, offset, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector (size N). ***Type: float *

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the output vector (size N/2). ***Type: int*

StrideY: stride to be used for the Y data. ***Type: int***

Scale: scalar multiply factor to scale the input vector. T***ype: float***

Offset: scalar offset to be added to the input vector.***Type: floa***t

ClipUp: value to be used as upper limit for the data. ***Type: float***

ClipDown: value to be used as lower limit for the data. ***Type: float***

Nelements: number of elements to be computed. ***Type: int***

The function pack40to16rl takes pair of data X: X1 and X2, scales them by a float factor, adds a float offset, clips the values in a float range and converts the results to a pair of 16 bit integer arranged in a 32 bit word Y.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in the right memory

Y must be in the left memory

ClipUp must be less or equal to 2^15 - 1

ClipDown must be greater or equal to -2^(16-1)

Number of cycles:

42 + 6 × Nelements

Number of VLIW:

41

File: pack40to16rl.mas

3.55 pack40to16rr

Function: multiplication by a float value, addition of a float offset, clipping in a float range of a pair of data in right memory: X1 and X2 and conversion of the results in 16 bit integer arranged in a 32 bit word in right memory

$$Y(k) = round(clip(X1 \times Scale + Offset)) \ll 16 \quad or \quad round(clip(X2 \times Scale + Offset))$$

Synopsis: __vector__ int pack40to16rr(*X, strideX, *Y, strideY, scale, offset, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

***X**: pointer to the input vector (size N). ***Type: float ***

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the output vector (size N/2). ***Type: int*

StrideY: stride to be used for the Y data. ***Type: int***

Scale: scalar multiply factor to scale the input vector. T***ype: float***

Offset: scalar offset to be added to the input vector.***Type: floa***t

ClipUp: value to be used as upper limit for the data. ***Type: float***

ClipDown: value to be used as lower limit for the data. ***Type: float***

Nelements: number of elements to be computed. ***Type: int***

The function pack40to16rr takes pair of data X: X1 and X2, scales them by a float factor, adds a float offset, clips the values in a float range and converts the results to a pair of 16 bit integer arranged in a 32 bit word Y.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in the right memory

Y must be in the right memory

ClipUp must be less or equal to 2^15 - 1

ClipDown must be greater or equal to -2^(16-1)

Number of cycles:

41 + 6 × Nelements

Number of VLIW:

42

File: pack40to16rr.mas

3.56 putvq

Function: filling of a vector queue with vecorial (left - right) data stored in the vectorial array X

Synopsis: int putvq(*q, *X, StrideX, Nelements)

q: pointer to a vector queue structure defined using the vqdef macro. ***Type: void ****

X: pointer to the destination vector where the data are copied. ***Type: void ****

StrideX: stride used to read data to the X vector. ***Type: int***

Nelements: number of elements copied. ***Type: int***

The function putvq copies data to the vector queue from the X buffer. If the number of elements available in the vector queue is lower than Nelements a -1 is returned (q over-run), but the copy is anyway done. This allows using the putvq also in a non-strictly queued structure, but in structures where circular addressing is used over a vector. A vector queue is a structure defined using the macro “vqdef” explicitly declared using that macro see the function: “initvq” on page 3-40. If the return code is not checked the structure is simply a circular buffer and consistency must be guaranteed by the user.

Recall:

Nelements can be 2047 elements max

Restrictions:

Nelements must be greater than 12 and multiple of 4

Number of cycles:

64 + 1× Nelements

Number of VLIW:

37

File: putvq.mas

3.57 putvq_f2i

Function: filling of a vector queue with vecorial (left - right) data stored in the vectorial array X after their conversion from float to integer

Synopsis: int putvq_f2i(*q, *X, StrideX, Nelements)

q**: pointer to a vector queue structure defined using the vqdef macro. ***Type: __vector__ int *

X**: pointer to the destination vector where the data are copied. ***Type: __vector__ float*

StrideX**: stride used to read data to the X vector. ***Type: int

Nelements**: number of elements copied. ***Type: int

The function putvq_f2i copies data to the vector queue from the X buffer after their conversion from float to integer. If the number of elements available in the vector queue is lower than Nelements a -1 is returned (q overrun), but the copy is anyway done. This allows using the putvq_f2i also in a non-strictly queued structure, but in structures where circular addressing is used over a vector. A vector queue is a structure defined using the macro “vqdef” explicitly declared using that macro see the function: “initvq” on page 3-40. If the return code is not checked the structure is simply a circular buffer and consistency must be guaranteed by the user.

Recall:

Nelements can be 2047 elements max

Restrictions:

Nelement must be greater than 12 and multiple of 4

Number of cycles:

72 + 1 × Nelements

Number of VLIW:

38

File: putvq_f2i.mas

3.58 putvq_i2f

Function: filling of a vector queue with vecorial (left - right) data stored in the vectorial array X after their conversion from integer to float

Synopsis: putvq_i2f (*q, *X, StrideX, Nelements)

q**: pointer to a vector queue structure defined using the vqdef macro. ***Type: __vector__ float *

X**: destination vector where the data are copied. ***Type: __vector__ int *

StrideX**: stride used to read data to the X vector. ***Type: int

Nelements**: number of elements copied. ***Type: int

The function putvq_i2f copies data to the vector queue from the X buffer after their conversion from integer to float. If the number of elements available in the vector queue is lower than Nelements a -1 is returned (q overrun), but the copy is anyway done. This allows using the putvq_i2f also in a non-strictly queued structure, but in structures where circular addressing is used over a vector. A vector queue is a structure defined using the macro "vqdef" explicitly declared using that macro see the function: "initvq" on page 3-40. If the return code is not checked the structure is simply a circular buffer and consistency must be guaranteed by the user.

Recall:

Nelements can be 2047 elements max

Restrictions:

Nelements must be greater than 12 and multiple of 4

Number of cycles:

72 + 1 × Nelements

Number of VLIW:

38

File: putvq_i2f.mas

3.59 v2magnlrl

Function: vector squared magnitude

$$Z(k) = X(k)^2 + Y(k)^2 \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int v2magnlrl(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

strideX**: stride to be applied on X vector. ***Type: int

Y**: pointer to the second input vector. ***Type: float *

strideY**: stride to be applied on Y vector. ***Type: int

Z**: pointer to the output vector. ***Type: float *

strideZ**: stride to be applied on Z vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The function v2magnlrl computes the square magnitude of a pair of float array: X and Y. The first must be stored in left memory, the second in right memory. The result is written in left memory.

Restrictions:

Nelements can be any number greater or equal to 1

vector X must be in left data memory

vector Y must be in right data memory

vector Z must be in left data memory

Number of cycles:

24 + 14 × Nelements

Number of VLIW:

18

File: v2magnlrl.mas

3.60 v2magnv

Function: vectorial complex squared magnitude

$$Z(k) = (ReX(k))^2 + (ImX(k))^2 \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int v2magnv(*X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the complex input vector. ***Type: __complex__ float *

strideX**: stride to be applied on X vector. ***Type: int

Z**: pointer to the output vector. ***Type: float *

StrideZ**: stride to be applied on Z vector. ***Type: int

Nelements: number elements to be computed. ***Type: int***

The function v2magnv computes the square magnitude of a complex vector and writes the result in left memory.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

Z must be in left memory

Number of cycles:

26 + 2.75 × Nelements

Number of VLIW:

24

File: v2magnv.mas

3.61 vacoshll

Function: inverse hyperbolic cosine of a float input array and left to left move

$$Y(k) = \boldsymbol{acosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vacoshll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacoshll computes the inverse hyperbolic cosine of an input array stored in left memory space and writes the output to an array in left memory space.

Precision:

see Table 3-1 on page 66

Restrictions:

Nelements must be multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

$400 + 27.75 \times Nelements$

Number of VLIW:

251

File: vacoshll.mas, vlogll.mas, vsqrtll.mas, lnCoeff.mas

3.62 vacoshlr

Function: inverse hyperbolic cosine of a float input array and left to right move

$$Y(k) = \boldsymbol{acosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vacoshlr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacoshlr computes the inverse hyperbolic cosine of an input array stored in left memory space and writes the output to an array in right memory space.

Precision:

see Table 3-1 on page 66

Restrictions:

Nelements must be multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

$389 + 27.75 \times$ Nelements

Number of VLIW:

254

File: vacoshlr.mas, vlogrr.mas, vsqrtrr.mas, lnCoeff.mas

3.63 vacoshrl

Function: inverse hyperbolic cosine of a float input array and right to left move

$$Y(k) = \boldsymbol{acosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vacoshrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacoshrl computes the inverse hyperbolic cosine of an input array stored in right memory space and writes the output to an array in left memory space.

Precision:

see Table 3-1 on page 66

Restrictions:

Nelements must be multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

400 + 27.75 × Nelements

Number of VLIW:

252

File: vacoshrl.mas, vlogll.mas, vsqrtll.mas, lnCoeff.mas

3.64 vacoshrr

Function: inverse hyperbolic cosine of a float input array and right to right move

$$Y(k) = \textbf{acosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vacoshrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacoshrr computes the inverse hyperbolic cosine of an input array stored in right memory space and writes the output to an array in right memory space.

Precision:

see Table 3-1 on page 66

Restrictions:

Nelements must be multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

391 + 27.75 × Nelements

Number of VLIW:

254

File: vacoshrr.mas, vlogrr.mas, vsqrtrr.mas, lnCoeff.mas

3.65 vacoshv

Function: inverse hyperbolic cosine of a vectorial input array

$$Y(k) = \boldsymbol{acosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vacoshv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacoshv computes the inverse hyperbolic cosine of an input array stored in vector space and writes the output to an array in vector space. For computing the Inverse hyperbolic cosine, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: “vacoshll” on page 3-62, “vacoshlr” on page 3-63, “vacoshrl” on page 3-64 and “vacoshrr” on page 3-65.

Precision:

the following table provides the information about the precision for this function

Table 3-1.

Range of input values	Absolute error	Relative error
1 to 1.414	3.35331e-009	Inf
1 to 10^{18}	1.19466e-009	1.74353e-009
10^{18} to 10^{38}	1.03627e-009	1.38434e-009
0 to 1	1.96387e-009	Inf
-1 to- 10^{8}	3.01231e-009	1.72469e-010
-10^{8} to -10^{38}	163.56	60.8094

Restrictions:

Nelements must be multiple of 2

Number of cycles:

354 + 50.5 × Nelements

Number of VLIW:

220

File: vacoshv.mas, vlogv.mas, vsqrtv.mas, lnCoeff.mas

3.66 vacosll

Function: inverse cosine of a float input array and left to left move

$$Y(k) = \boldsymbol{acos}(X(k)) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vacosll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacosll computes the arc cosine of an input array stored in left memory space and writes the output to an array in left memory space.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

see Table 3-2 on page 71

Restrictions:

Nelements must be multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

310 + 26.25 × Nelements

Number of VLIW:

232

File: vacosll.mas, vsqrtll.mas,acosCoeff.mas

3.67 vacoslr

Function: inverse cosine of a float input array and left to right move

$$Y(k) = \boldsymbol{acos}(X(k)) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vacoslr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacoslr computes the arc cosine of an input array stored in left memory space and writes the output to an array in right memory space.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

see Table 3-2 on page 71

Restrictions:

Nelements must be multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

300 + 26.75 × Nelements

Number of VLIW:

232

File: vacoslr.mas, vsqrtrr.mas,acosCoeff.mas

3.68 vacosrl

Function: inverse cosine of a float input array and right to left move

$$Y(k) = \boldsymbol{acos}(X(k)) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vacosrl (*X, strideX, *Y, strideY, Nelements)

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacosrl computes the arc cosine of an input array stored in right memory space and writes the output to an array in left memory space.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

see Table 3-2 on page 71

Restrictions:

Nelements must be multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

308 + 26 × Nelements

Number of VLIW:

233

File: vacosrl.mas, vsqrtll.mas, acosCoeff.mas

3.69 vacosrr

Function: inverse cosine of a float input array and right to right move

$$Y(k) = \boldsymbol{acos}(X(k)) \qquad k = 0 \dots Nelements$$

Synopsis: __vector__ int vacosrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacosrr computes the arc cosine of an input array stored in right memory space and writes the output to an array in right memory space.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

see Table 3-2 on page 71

Restrictions:

Nelements must be multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

298 + 26.5 × Nelements

Number of VLIW:

232

File: vacosrr.mas, vsqrtrr.mas, acosCoeff.mas

3.70 vacosv

Function: inverse cosine of vectorial input array

$$Y(k) = \boldsymbol{acos}(X(k)) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vacosv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vacosv computes the arc cosine of an input array stored in vector space and writes the output to an array in vector space. For computing the arc cosine, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: “vacosll” on page 3-67, “vacoslr” on page 3-68, “vacosrl” on page 3-69, “vacosrr” on page 3-70.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

the following table provides the information about the precision for this function

Table 3-2.

Description of input values	Absolute error	Relative error
0.01 to 0.5	5.64143e-009	5.27317e-008
0.5 to 0.9999	5.45383e-009	5.27317e-008
-0.9999 to -0.0001	5.64143e-009	5.27317e-008

Restrictions:

Nelements must be multiple of 2

Number of cycles:

292 + 52 × Nelements

Number of VLIW:

208

File: vacosv.mas, vsqrtv.mas, acosCoeff.mas

3.71 vaddintv

Function: sum of 2 vectorial integer arrays

$$Z(k) = X(k) + Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vaddintv(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: __vector__ int *

strideX**: stride to be used for the X the data. ***Type: int

Y**: pointer to the second input vector. ***Type: __vector__ int *

strideY**: stride to be used for the Y the data. ***Type: int

Z**: pointer to the output vector Z. ***Type: __vector__ int *

strideZ**: stride to be used for the Z data. ***Type: int

Nelements**: Number of element to be computed. ***Type: int

The function vaddintv performs the sum between two vectorial integer data: X and Y. They can be complex vectors or two vectorial streams of real vectors that will be procesed in parallel.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

Number of cycles:

39 + 2 × Nelements

Number of VLIW:

34

File: vaddintv.mas

3.72 vaddlll

Function: sum of 2 input float array stored in left memory and output in left memory

$$Z(k) = X(k) + Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vaddlll(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

strideX: stride to be used for the X the data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float *

strideY: stride to be used for the Y the data. ***Type: int***

Z**: pointer to the output vector. ***Type: float *

strideZ: stride to be used for the Z data. ***Type: int***

Nelements: Number of element to be computed. ***Type: int***

The function vaddlll adds two float vectors stored in left memory and writes the output in left memory.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in left memory

Z must be in left memory

Number of cycles:

31+ 2 × Nelements

Number of VLIW:

24

File: vaddlll.mas

3.73 vaddllr

Function: sum of 2 input float array stored in left memory and output in right memory

$$Z(k) = X(k) + Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vaddllr(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X: pointer to the first input vector. ***Type: float ****

strideX: stride to be used for the X the data. ***Type: int***

Y: pointer to the second input vector. ***Type: float ****

strideY: stride to be used for the Y the data. ***Type: int***

Z: pointer to the output vector. ***Type: float ****

strideZ: stride to be used for the Z data. ***Type: int***

Nelements: Number of element to be computed. ***Type: int***

The function vaddllr adds two float vectors stored in left memory and writes the output in right memory.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

X must be in left memory

Y must be in left memory

Z must be in right memory

Number of cycles:

32 + 2.25 × Nelements

Number of VLIW:

36

File: vaddllr.mas

3.74 vaddlrl

Function: sum of 2 input float array : the first is stored in left memory while the second in right memory. The output is written in left memory

$$Z(k) = X(k) + Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vaddlrl(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float*

strideX: stride to be used for the X the data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float*

strideY: stride to be used for the Y the data. ***Type: int***

Z**: pointer to the output vector. ***Type: float*

strideZ: stride to be used for the Z data. ***Type: int***

Nelements: Number of element to be computed. ***Type: int***

The function vaddlrl adds two float vectors: the first is stored in left memory while the second in right memory. It writes the output in left memory.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

Z must be in left memory

Number of cycles:

31+ 2 × Nelements

Number of VLIW:

25

File: vaddlrl.mas

3.75 vaddlrr

Function: sum of 2 input float array: the first is stored in left memory while the second in right memory. The result is written in right memory

$$Z(k) = X(k) + Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vaddlrr(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

strideX: stride to be used for the X the data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float *

strideY: stride to be used for the Y the data. ***Type: int***

Z**: pointer to the output vector. ***Type: float *

strideZ: stride to be used for the Z data. ***Type: int***

Nelements: Number of element to be computed. ***Type: int***

The function vaddlrr adds two float vectors: the first is stored in left memory while the second in right memory. It writes the output in right memory.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

Z must be in left memory

N umber of cycles:

31+ 2 × Nelements

Number of VLIW:

25

File: vaddlrr.mas

3.76 vaddrrl

Function: sum of 2 input float array stored in right memory and output in left memory

$$Z(k) = X(k) + Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vaddrrl(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

strideX: stride to be used for the X the data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float *

strideY: stride to be used for the Y the data. ***Type: int***

Z**: pointer to the output vector. ***Type: float *

strideZ: stride to be used for the Z data. ***Type: int***

Nelements: Number of element to be computed. ***Type: int***

The function vaddrrl adds two float vectors stored in right memory and writes the output in left memory.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

X must be in right memory

Y must be in right memory

Z must be in left memory

Number of cycles:

40 + 2 × Nelements

Number of VLIW:

36

File: vaddrrl.mas

3.77 vaddrrr

Function: sum of 2 input float array stored in right memory and output in right memory

$$Z(k) = X(k) + Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vaddrrr(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

strideX: stride to be used for the X the data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float *

strideY: stride to be used for the Y the data. ***Type: int***

Z**: pointer to the output vector. ***Type: float *

strideZ: stride to be used for the Z data. ***Type: int***

Nelements: Number of element to be computed. ***Type: int***

The function vaddrrr adds two float vectors stored in right memory and writes the output in right memory.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in right memory

Z must be in right memory

Number of cycles:

35 + 2 × Nelements

Number of VLIW:

25

File: vaddrrr.mas

3.78 vaddv

Function: sum of 2 vectorial float array

$$Z(k) = X(k) + Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vaddv(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: __vector__ float *

strideX: stride to be used for the X the data. ***Type: int***

Y**: pointer to the second input vector. ***Type: __vector__ float *

strideY: stride to be used for the Y the data. ***Type: int***

Z**: pointer to the output vector. ***Type: __vector__ float *

strideZ: stride to be used for the Z data. ***Type***: int

Nelements: Number of element to be computed. ***Type***: int

The function vaddv works on complex data arranged vectorially in memory; they can represent pair of complex vectors or two vectorial streams of real vectors that will be processed in parallel.

Restrictions:

Nelements must be multiple of 4

Number of cycles:

32 + 2.75 × Nelements

Number of VLIW:

27

File: vaddv.mas

3.79 varll

Function: variance of a float array

$$var = mean\{[X - mean(X)]^2\} = mean(X^2) - \{mean(X)\}^2$$

Synopsis: __vector__ int varll(*X, strideX, *Z, M, Nelements, InvNelements)

Include file: DSPlib.h.

X**: pointer to input vector X. ***Type: float*

strideX: stride to be used for the X the data. ***Type: int***

Z**: pointer to the output .Type: float****

M: mean value of the input. ***Type: float***

Nelements: Number of element to be computed. ***Type: int***

InvNelements: 1/Nelements. ***Type: float***

The function varll computes the variance of a float array X. The mean of X can be calculated by the multiplication between InvNelements and the output of the function vsum with input X, see “vsumv” on page 3-225.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in left memory

Z must be in left memory

Number of cycles:

53 + 1.75 × Nelements

Number of VLIW:

33

File: varll.mas

3.80 vasinhll

Function: inverse hyperbolic sine of a float input array and left to left move

$$Y(k) = \boldsymbol{asinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vasinhll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinhll computes the inverse hyperbolic sine of an input array stored in left memory space and writes the output to an array in left memory space.

Precision:

see Table 3-3 on page 84

Restrictions:

Nelements must be multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

400 + 27.75 × Nelements

Number of VLIW:

249

File: vasinhll.mas, vlogll.mas, vsqrtll.mas, lnCoeff.mas

3.81 vasinhlr

Function: inverse hyperbolic sine of a float input array and left to right move

$$Y(k) = \boldsymbol{asinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vasinhlr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinhlr computes the inverse hyperbolic sine of an input array stored in left memory space and writes the output to an array in right memory space.

Precision:

see Table 3-3 on page 84

Restrictions:

Nelements must be multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

389 + 27.75 × Nelements

Number of VLIW:

252

File: vasinhlr.mas, vlogrr.mas, vsqrtrr.mas, lnCoeff.mas

3.82 vasinhrl

Function: inverse hyperbolic sine of a float input array and right to left move

$$Y(k) = \boldsymbol{asinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vasinhrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinhrl computes the inverse hyperbolic sine of an input array stored in right memory space and writes the output to an array in left memory space.

Precision:

See Table 3-3 on page 84

Restrictions:

Nelements must be multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

400 + 27.75 × Nelements

Number of VLIW:

250

File: vasinhrl.mas, vlogll.mas, vsqrtll.mas, lnCoeff.mas

3.83 vasinhrr

Function: inverse hyperbolic sine of a float input array and right to right move

$$Y(k) = \boldsymbol{asinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vasinhrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X: pointer to the input array. ***Type: float****

strideX: stride to be used for the input array. ***Type: int***

Y: pointer to the output array into which the computed value is written. ***Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinhrr computes the inverse hyperbolic sine of an input array stored in right memory space and writes the output to an array in right memory space.

Precision:

see Table 3-3 on page 84

Restrictions:

Nelements must be multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

390 + 27.75 × Nelements

Number of VLIW:

252

File: vasinhrr.mas, vlogrr.mas, vsqrtrr.mas, lnCoeff.mas

3.84 vasinhv

Function: inverse hyperbolic sine of a vectorial input array

$$Y(k) = \boldsymbol{asinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vasinhv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinhv computes the inverse hyperbolic sine of a vectorial input array stored in vector space and writes the output to an array in vector space. For computing the Inverse hyperbolic sine, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: “vasinhll” on page 3-80, “vasinhlr” on page 3-81, “vasinhrl” on page 3-82 and “vasinhrr” on page 3-83.

Precision:

the following table provides the information about the precision for this function

Table 3-3.

Range of input values	Absolute error	Relative error
1 to 1.414	3.35331e-009	Inf
1 to 10^{18}	1.19466e-009	1.74353e-009
10^{18} to 10^{38}	1.03627e-009	1.38434e-009

Table 3-3.

Range of input values	Absolute error	Relative error
0 to 1	1.96387e-009	Inf
-1 to- 10^8	3.01231e-009	1.72469e-010
-10^8 to -10^{38}	163.56	60.8094

Restrictions:

Nelements must be multiple of 2

Number of cycles:

354 + 50.5 × Nelements

Number of VLIW:

219

File: vasinhv.mas, vlogv.mas, vsqrtv.mas, lnCoeff.mas

3.85 vasinll

Function: inverse sine of a float input array and left to left move

$$Y(k) = \boldsymbol{asin}(X(k)) \qquad k = 0\ldots Nelements-1$$

Synopsis: __vector__ int vasinll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array which the computed value is written.Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinll computes the arc sine of an input array stored in left memory space and writes the output to an array in left memory space.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before

RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

see Table 3-4 on page 90

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

310 + 26.25 × Nelements

Number of VLIW:

233

File: vasinll.mas, vsqrtll.mas, asinCoeff.mas

3.86 vasinlr

Function: inverse sine of a float input array and left to right move

$$Y(k) = \boldsymbol{asin}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vasinlr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written.Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinlr computes the arc sine of an input array stored in left memory space and writes the output to an array in right memory space.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

see Table 3-4 on page 90

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

299 + 26.75 × Nelements

Number of VLIW:

231

File: vasinlr.mas, vsqrtrr.mas, asinCoeff.mas

3.87 vasinrl

Function: inverse sine of a float input array and right to left move

$$Y(k) = \boldsymbol{asin}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vasinrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written.Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinrl computes the arc sine of an input array stored in right memory space and writes the output to an array in left memory space.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

see Table 3-4 on page 90

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

290 + 26 × Nelements

Number of VLIW:

232

File: vasinrl.mas, vsqrtll.mas, asinCoeff.mas

3.88 vasinrr

Function: inverse sine of a float input array and right to right move

$$Y(k) = \boldsymbol{asin}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vasinrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written.Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinrr computes the arc sine of an input array stored in right memory space and writes the output to an array in right memory space.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

see Table 3-4 on page 90

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

297 + 26.5 × Nelements

Number of VLIW:

236

File: vasinrr.mas, vsqrtrr.mas, asinCoeff.mas

3.89 vasinv

Function: inverse sine of a vectorial input array

$$Y(k) = \boldsymbol{asin}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vasinv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vasinv computes the arc sine of an input array stored in vector space and writes the output to an array in vector space. For computing the arc sine, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: “vasinll” on page 3-85, “vasinlr” on page 3-86, “vasinrl” on page 3-87 and “vasinrr” on page 3-88.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Precision:

the following table provides the information about the precision for this function

Table 3-4.

Description of input values	Absolute error	Relative error
0.01 to 0.5	5.64143e-009	5.27317e-008
0.5 to 0.9999	5.45383e-009	5.27317e-008
-0.9999 to -0.0001	5.64143e-009	5.27317e-008

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

Number of cycles:

290 + 51 × Nelements

Number of VLIW:

210

File: vasinv.mas, vsqrtv.mas, asinCoeff.mas

3.90 vatan2

Function: argument (arctan2) of a complex input array and result in a float array in left memory

$$Y(k) = \boldsymbol{atan}2(Re(X(k)), Im(X(k))) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vatan2 (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type __complex__float

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY**: stride to be used for the output array. ***Type: int

Nelements**: number of elements to be computed.Type: int***

The function vatan2 computes the arc tan2 of a complex array and writes the output to an array in left memory space.

Note: the function vatan2 uses 23 locations of the stack

Precision: the following table provides the information about the precision for this function.

Table 3-5.

Description of input values $X = \boldsymbol{cos}\theta + j\boldsymbol{sin}\theta$	**Absolute error**	**Relative error**
$0 < \theta < \pi/2$	4.87426e-010	1.15685e-009
$\pi/2$	0.570796	0.36338
$\pi/2 < \theta < \pi$	8.12197e-010	3.52541e-010
π	8.97931e-011	2.8582e-011
$\pi < \theta < (3\pi)/2$	8.12197e-010	3.52541e-010
$3(\pi/2)$	2.5708	1.63662
$3(\pi/2) < \theta < 2\pi$	4.87426e-010	1.15685e-009
2π	0	0

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Y must be in left memory

Real Part of X < = 10^38, to avoid overflow / underflow of the computed result

Imaginary Part of X < = 10^38, to avoid overflow / underflow of the computed result

Real Part of X not = 0 , to avoid invalid result

Number of cycles:

339 + 26.5 × Nelements

Number of VLIW:

224

File: vatan2.mas, atanCoeff.mas

3.91 vatanhll

Function: inverse hyperbolic tangent of a float input array and left to left move

$$Y(k) = \boldsymbol{atanh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vatanhll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vatanhll computes the inverse hyperbolic tan of an input array stored in left memory space and writes the output to an array in left memory space.

Note: the function vatanhll uses 3 locations of the stack

Precision:

see Table 3-6

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

323 +19.25 × Nelements

Number of VLIW:

184

File: vatanhll.mas, vlogll.mas, lnCoeff.mas

3.92 vatanhlr

Function: inverse hyperbolic tangent of a float input array and left to right move

$$Y(k) = \boldsymbol{atanh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vatanhlr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY**: stride to be used for the output array. ***Type: int

Nelements**: number of elements to be computed.Type: int***

The function vatanhlr computes the inverse hyperbolic tan of an input array stored in left memory space and writes the output to an array in right memory space.

Note: the function vatanhlr uses 3 locations of the stack

Precision:

see Table 3-6 on page 96

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

320 +19.25 × Nelements

Number of VLIW:

186

File: vatanhlr.mas, vlogrr.mas, lnCoeff.mas

3.93 vatanhrl

Function: inverse hyperbolic tangent of a float input array and right to left move

$$Y(k) = \boldsymbol{atanh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vatanhrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vatanhrl computes the inverse hyperbolic tan of an input array stored in right memory space and writes the output to an array in left memory space.

Note: the function vatanhlr uses 3 locations of the stack

Precision:

see Table 3-6 on page 96

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

321 +19.25 × Nelements

Number of VLIW:

182

File: vatanhrl.mas, vlogll.mas, lnCoeff.mas

3.94 vatanhrr

Function: inverse hyperbolic tangent of a float input array and right to right move

$$Y(k) = \boldsymbol{atanh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vatanhrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

****X***: pointer to the input array. ***Type: float****

strideX: stride to be used for the input array. ***Type: int***

****Y***: pointer to the output array into which the computed value is written. ***Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vatanhrr computes the inverse hyperbolic tan of an input array stored in right memory space and writes the output to an array in right memory space.

Note: the function vatanhrr uses 3 locations of the stack

Precision:

see Table 3-6 on page 96

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

318 +19.25 × Nelements

Number of VLIW:

184

File: vatanhrr.mas, vlogrr.mas, lnCoeff.mas

3.95 vatanhv

Function: inverse hyperbolic tangent of a vectorial input array

$$Y(k) = \boldsymbol{atanh}(X(k))$$

Synopsis: __vector__ int vatanhv (*X, strideX, *Y, strideY, Nelements)

Include File: Dsplib.h

X**: pointer to the input array. ***Type: __vector__ Float*

strideX**: stride to be used for the input array. ***Type: Int

Y**: pointer to the output array into which the computed calue is written. ***Type: __vector__ Float*

strideY**: stride to be used for the output array. ***Type: Int

Nelements**: number of elements to be computed.type: Int***

The function vatanhv computes the inverse hyperbolic tan of an input array stored in vector space and writes the output to an array in vector space. For computing the inverse hyperbolic tan, with the input stored in left/right memory space and to output the values into left/right memory mpace, see the functions: “vatanhll” on page 3-91, “vatanhlr” on page 3-92, “vatanhrl” on page 3-93 and “vatanhrr” on page 3-94.

Note: the function vatanhv uses 3 locations of the stack

Precision:

the following table provides the information about the precision for this function

Table 3-6.

Range of input values	Absolute error	Relative error
1 to 1.414	9.78877e-010	9.8742e-010
1 to 10^{18}	9.04127e-009	3.22854e-010
10^{18} to 10^{38}	42.6711	0.912852
-1 to -10^{8}	7.22321e-005	9.50309e-006

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

Number of cycles:

300 + 35 × Nelements

Number of VLIW:

161

File: vatanhv.mas, vlogv.mas, lncoeff.mas

3.96 vbyvmulv

Function: vectorial element by element multiplication

$$Z(k) = X(k) \times Y(k) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vbyvmulv(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

****X***: pointer to the first input vector. ***Type: __vector__ float****

strideX: stride to be used for the X data. ***Type: int***

****Y***: pointer to the second input vector. ***Type: __vector__ float****

strideY: stride to be used for the Y data.***Type: int***

****Z***: pointer to the output vector. ***Type: __vector__ float****

strideZ: stride to be used for the Z data. ***Type: int***

Nelements: Number of elements to be computed.***Type: int***

The function vbyvmulv performs vectorial element-by-element multiplication.

Restrictions:

Nelements must be multiple of 4

Number of cycles:

25 + 2 × Nelements

Number of VLIW:

19

File: vbyvmulv.mas

3.97 vclipll

Function: clipping of a float array in left memory between two float values ClipUp and ClipDown and left to left move

$$\begin{cases} Y(k) = ClipUp & X(k) > ClipUp \\ Y(k) = ClipDown & X(k) < ClipDown \\ Y(k) = X(k) & ClipDown < X(k) < ClipUp \end{cases} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vclipll (*X, strideX, *Y, strideY, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

****X***: pointer to the input vector. ***Type: float ****

StrideX: stride to be used for the X data. ***Type: int***

****Y***: pointer to the output vector. ***Type: float ****

StrideY: stride to be used for the Y data. ***Type: int***

ClipUp: value to be used as upper limit for the data. ***Type: float***

ClipDown: value to be used as lower limit for the data. ***Type: float***

Nelements: Number of elements to be computed. ***Type: int***

The function vclipll clips the float array X stored in left memory, between the float values: ClipUp and ClipDown, and writes the result in the float output Y stored in left memory.

Restrictions:

Nelements must be greater than 12 and multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

25 + 2 × Nelements

Number of VLIW:

26

File: vclipll.mas

3.98 vcliprr

Function: clipping of a float array in right memory between two float values ClipUp and ClipDown and right to right move

$$\begin{cases} Y(k) = ClipUp & X(k) > ClipUp \\ Y(k) = ClipDown & X(k) < ClipDown \\ Y(k) = X(k) & ClipDown < X(k) < ClipUp \end{cases} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcliprr (*X, strideX, *Y, strideY, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: float *

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the output vector. ***Type: float *

StrideY: stride to be used for the Y data. ***Type: int***

ClipUp: value to be used as upper limit for the data. ***Type: float***

ClipDown: value to be used as lower limit for the data. ***Type: float***

Nelements: Number of elements to be computed. ***Type: int***

The function vcliprr clips the float array X stored in right memory, between the float values: ClipUp and ClipDown, and writes the result in the float output Y stored in right memory.

Restrictions:

Nelements must be greater than 12 and multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

31 + 2 × Nelements

Number of VLIW:

27

File: vcliprr.mas

3.99 vclipv

Function: vectorial clipping between the two values ClipUp and ClipDown

$$\begin{cases} Y(k) = ClipUp & X(k) > ClipUp \\ Y(k) = ClipDown & X(k) < ClipDown \\ Y(k) = X(k) & ClipDown < X(k) < ClipUp \end{cases} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vclipv (*X, strideX, *Y, strideY, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ float *

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the output vector. ***Type: __vector__ float *

StrideY: stride to be used for the Y data. ***Type: int***

ClipUp: value to be used as upper limit for the data. ***Type: __vector__ float***

ClipDown: value to be used as lower limit for the data. ***Type: __vector__ float***

Nelements: Number of elements to be computed. ***Type: int***

The function vclipv clips the vectorial float array X , between the vectorial values: ClipUp and ClipDown and writes the result in the vectorial output Y.

Restrictions:

Nelements must be greater than 8 and multiple of 4

Number of cycles:

36 + 2 × Nelements

Number of VLIW:

30

File: vclipv.mas

3.100 vcoshll

Function: hyperbolic cosine of a float input array and left to left move

$$Y(k) = \boldsymbol{cosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcoshll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vcoshll computes the hyperbolic cosine of an input array stored in left memory space and writes the output to an array in left memory space.

Note: the function vcoshll uses 3 locations of the stack

Precision:

see Table 3-7 on page 104

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in left memory

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

307 + 19 × Nelements

Number of VLIW:

165

File: vcoshll.mas, vexpll.mas, expCoeff.mas

3.101 vcoshlr

Function: hyperbolic cosine of a float input array and left to right move

$$Y(k) = \boldsymbol{cosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcoshlr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vcoshlr computes the hyperbolic cosine of an input array stored in left memory space and writes the output to an array in right memory space.

Note: the function vcoshlr uses 3 locations of the stack

Precision:

see Table 3-7 on page 104

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

306 +18.5 × Nelements

Number of VLIW:

159

File: vcoshlr.mas, vexplr.mas, expCoeff.mas

3.102 vcoshrl

Function: hyperbolic cosine of a float input array and right to left move

$$Y(k) = \boldsymbol{cosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcoshrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vcoshrl computes the hyperbolic cosine of an input array stored in right memory space and writes the output to an array in left memory space.

Note: the function vcoshrl uses 3 locations of the stack

Precision:

see Table 3-7 on page 104

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

304 +19 × Nelements

Number of VLIW:

166

File: vcoshrl.mas, vexprl.mas, expCoeff.mas

3.103 vcoshrr

Function: hyperbolic cosine of a float input array and right to right move

$$Y(k) = \boldsymbol{cosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcoshrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY**: stride to be used for the output array. ***Type: int

Nelements**: number of elements to be computed.Type: int***

The function vcoshrr computes the hyperbolic cosine of an input array stored in right memory space and writes the output to an array in right memory space.

Note: the function vcoshrr uses 3 locations of the stack

Precision:

see Table 3-7 on page 104

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in right memory

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

306 + 18.5 × Nelements

Number of VLIW:

161

File: vcoshrr.mas, vexprr.mas, expCoeff.mas

3.104 vcoshv

Function: hyperbolic cosine of a vectorial input array

$$Y(k) = \mathbf{cosh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcoshv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vcoshv computes the hyperbolic cosine of an input array stored in vector space and writes the output to an array in vector space. For computing the hyperbolic cosine, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: “vcoshll” on page 3-100, “vcoshlr” on page 3-101, “vcoshrl” on page 3-102 and “vcoshrr” on page 3-103.

Note: the function vcoshv uses 3 locations of the stack

Precision:

the following table provides the information about the precision for this function

Table 3-7.

Range of input values	Absolute error	Relative error
-0.1505 to 0.1505	1.05541e-009	1.05043e-009
0 to 10	8.42592e-006	8.28671e-010
10 to 86	1.32643e+027	5.28016e-010
-10 to 0	2.15741e-005	8.34309e-010
-86 to -10	1.32643e+027	5.28016e-010

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2
| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

320 + 31× Nelements

Number of VLIW:

156

File: vcoshv.mas, vexpv.mas, expCoeff.mas

3.105 vcosll

Function: cosine of a float input array and left to left move

$$Y(k) = \boldsymbol{cos}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcosll (*X, strideX, *Y, strideY, Nelements)

X:** pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y:** pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed. ***Type: int***

The function vcosll computes the cosine of an input array stored in left memory space and writes the output to an array in left memory space.

Precision:

see Table 3-8 on page 109

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

125 + 13.25 × Nelements

Number of VLIW:

65

File: vcosll.mas, cosCoeff.mas

3.106 vcoslr

Function: cosine of a float input array and left to right move

$$Y(k) = \boldsymbol{cos}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcoslr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X: pointer to the input array. ***Type: float****

strideX: stride to be used for the input array. ***Type: int***

Y: pointer to the output array into which the computed value is written.***Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vcoslr computes the cosine of an input array stored in left memory space and writes the output to an array in right memory space.

Precision:

see Table 3-8 on page 109

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

124 + 13 × Nelements

Number of VLIW:

66

File: vcoslr.mas, cosCoeff.mas

3.107 vcosrl

Function: cosine of a float input array and right to left move

$$Y(k) = \boldsymbol{cos}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcosrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written.Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vcosrl computes the cosine of an input array stored in right memory space and writes the output to an array in left memory space.

Precision:

see Table 3-8 on page 109

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

125 + 13 × Nelements

Number of VLIW:

67

File: vcosrl.mas, cosCoeff.mas

3.108 vcosrr

Function: cosine of a float input array and right to right move

$$Y(k) = \boldsymbol{cos}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcosrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written.Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vcosrr computes the cosine of an input array stored in right memory space and writes the output to an array in right memory space.

Precision:

see Table 3-8 on page 109

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

123 + 13 × Nelements

Number of VLIW:

66

File: vcosrr.mas, cosCoeff.mas

3.109 vcosv

Function: cosine of a vectorial input array

$$Y(k) = \boldsymbol{cos}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vcosv (*X, strideX, *Y, strideY, ***Nelements***)

Include file: DSPlib.h

X:** pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y:** pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vcosv computes the cosine of an input array stored in vector space and writes the output to an array in vector space. For computing the cosine, with the input stored in left/right memory space and to output the values into left/right memory space, see functions vcosll105,vcoslr106,vcosrl107,vcosrr108

Precision:

the following table provides the information about the precision for this function.

Table 3-8.

Description of input values	Absolute error	Relative error
0 to $\pi / 3$	3.25466e-009	4.39848e-009
$-\pi$ *to* π	3.25466e-009	2.41711e-008
2π, 6π	3.25466e-009	2.41711e-008
$\cdot 2\pi$, -6π	3.25466e-009	2.41711e-008

restrictions:

Nelements must be greater or equal to 2 and multiple of 2

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

107 + 20.5 × Nelements

Number of VLIW:

58

File: vcosv.mas, cosCoeff.mas

3.110 vdist

Function: euclidean distance between two input complex arrays

$$Z(k) = sqrt[(Re(X(k)) - Re(Y(k)))^2 + (Im(X(k)) - Im(Y(k)))^2]\ k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vdist (*X, strideX, *Y, strideY, * Z, Nelements)

Include file: DSPlib.h

X**: pointer to the input array . ***Type: __complex__ float *****

strideX: stride to be used for input array A. ***Type: int***

Y**: pointer to the input array . ***Type: __complex__ float *****

strideY: stride to be used for input array B. ***Type: int***

Z**: pointer to the output array . ***Type: __complex__ float *****

Nelements: number of elements to be computed. ***Type: int***

The function vdist computes the euclidean distance between two complex arrays.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

Number of cycles:

173 + 10.5 × Nelements

Number of VLIW:

109

File: vdist.mas, vsqrtll.mas

3.111 vdiv0rll

Function: float array division element by element (equivalent to Matlab: Y./X)

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vdiv0rll(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

Y: pointer to the Y input vector. ***Type: float ****

strideY: stride to be applied on Y vector. ***Type: int***

X: pointer to the X input vector. ***Type: float ****

strideX: stride to be applied on X vector. ***Type: int***

Z: pointer to the output vector (Z) . ***Type: float ****

strideZ: stride to be applied on Z vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The function vdiv0rll performs the division between inputs data vectors X and Y ordered as specified in Restrictions. Y and X are float array. In order to divide Y by a scalar x, simply set *X equal &x and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0. For a pipelined version see the function “vdivrll” on page 3-118.

Restrictions:

Nelements can be any number greater or equal to 1

Y must be on the right memory

X must be on the left memory

Z must be on the left memory

Result precision: 23 bits of mantissa

Number of cycles:

32 + 25 × Nelements

Number of VLIW:

27

File: vdiv0rll.mas

3.112 vdiv40lll

Function: float array division element by element (equivalent to Matlab: Y./ X), with Y and X il left memory and precision equal to 31 bit of mantissa

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vdiv40lll(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

Y**: pointer to the Y input vector. ***Type: float *

strideY: stride to be applied on Y vector. ***Type: int***

X**: pointer to the X input vector. ***Type: float *

strideX: stride to be applied on X vector. ***Type: int***

Z**: pointer to the output vector Z . ***Type: float *

strideZ: stride to be applied on Z vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The vdiv40lll performs the division with unroll 4 and precision equal to 31 bit of mantissa, between inputs data vectors X and Y ordered as specified in Restrictions. Y and X are float array, but after their moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. In order to divide Y by a scalar x, simply set *X equal &x and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

Y must be in the left memory

X must be in the left memory

Z must be in the left memory

Precision: 31 bit of mantissa

Number of cycles:

78 + 7.75 × Nelements

Number of VLIW:

64

File: vdiv40lll.mas

3.113 vdiv40lrl

Function: float array division element by element (equivalent to Matlab: Y./ X), with Y in left memory and X in right memory and precision equal to 31 bit of mantissa

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vdiv40lrl(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

Y**: pointer to the Y input vector. ***Type: float *

strideY: stride to be applied on Y vector. ***Type: int***

X**: pointer to the X input vector. ***Type: float *

strideX: stride to be applied on X vector. ***Type: int***

Z**: pointer to the output vector Z . ***Type: float *

strideZ: stride to be applied on Z vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The vdiv40lrl performs the division with unroll 4 and precision equal to 31 bit of mantissa, between inputs data vectors X and Y ordered as specified in Restrictions. Y and X are float array, but after their moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. In order to divide Y by a scalar x, simply set *X equal &x and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Y must be in the left memory

X must be in the right memory

Z must be in the left memory

Precision: 31 bit of mantissa

Number of cycles:

79 + 7.75 × Nelements

Number of VLIW:

68

File: vdiv40lrl.mas

3.114 vdiv40rll

Function: float array division element by element (equivalent to Matlab: Y./ X), with Y in right memory and X in left memory and precision equal to 31 bit of mantissa

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vdiv40rll(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

Y**: pointer to the Y input vector. ***Type: float *

strideY: stride to be applied on Y vector. ***Type: int***

X**: pointer to the X input vector. ***Type: float *

strideX: stride to be applied on X vector. ***Type: int***

Z**: pointer to the output vector Z . ***Type: float *

strideZ: stride to be applied on Z vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The vdiv40rll performs the division with unroll 4 and precision equal to 31 bit of mantissa, between inputs data vectors X and Y ordered as specified in Restrictions. Y and X are float array, but after their moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. In order to divide Y by a scalar x, simply set *X equal &x and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Y must be in the right memory

X must be in the left memory

Z must be in the left memory

Precision: 31 bit of mantissa

Number of cycles:

78 + 7.75 × Nelements

Number of VLIW:

66

File: vdiv40rll.mas

3.115 vdiv40rrl

Function: float array division element by element (equivalent to Matlab: Y./ X), with Y and X in right memory and precision equal to 31 bit of mantissa

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vdiv40rrl(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

Y**: pointer to the Y input vector. ***Type: float *

strideY**: stride to be applied on Y vector. ***Type: int

X**: pointer to the X input vector. ***Type: float *

strideX**: stride to be applied on X vector. ***Type: int

Z**: pointer to the output vector Z . ***Type: float *

strideZ**: stride to be applied on Z vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The vdiv40rrl performs the division with unroll 4 and precision equal to 31 bit of mantissa, between inputs data vectors X and Y ordered as specified in Restrictions. Y and X are float array, but after their moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. In order to divide Y by a scalar x, simply set *X equal &x and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Y must be in the right memory

X must be in the right memory

Z must be in the left memory

Precision: 31 bit of mantissa

Number of cycles:

80 + 7.75 × Nelements

Number of VLIW:

65

File: vdiv40rrl.mas

3.116 vdivlll

Function: float array division element by element (equivalent to Matlab: Y./ X) with Y and X in left memory and precision equal to 23 bit of mantissa

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vdivlll(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

Y**: pointer to the Y input vector. ***Type: float *

strideY: stride to be applied on Y vector. ***Type: int***

X**: pointer to the X input vector. ***Type: float *

strideX: stride to be applied on X vector. ***Type: int***

Z**: pointer to the output vector Z . ***Type: float *

strideZ: stride to be applied on Z vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The function vdivlll performs the division with unroll 4, between inputs data vectors X and Y ordered as specified in Restrictions. Y and X are float array, but after their moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. In order to divide Y by a scalar x, simply set *X equal &x and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Y must be in the left memory

X must be in the left memory

Z must be in the left memory

Precision: 23 bit of mantissa

Number of cycles:

96 + 3.75 × Nelements

Number of VLIW:

59

File: vdivlll.mas

3.117 vdivlrl

Function: float array division element by element (equivalent to Matlab: Y./ X), with Y in left memory and X in right memory and precision equal to 23 bit of mantissa

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vdivlrl(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

Y: pointer to the Y input vector. ***Type: float ****

strideY: stride to be applied on Y vector. ***Type: int***

X: pointer to the X input vector. ***Type: float ****

strideX: stride to be applied on X vector. ***Type: int***

Z: pointer to the output vector Z . ***Type: float ****

strideZ: stride to be applied on Z vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The vdivlrl performs the division with unroll 4, between inputs data vectors X and Y ordered as specified in Restrictions. Y and X are float array, but after their moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. In order to divide Y by a scalar x, simply set *X equal &x and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Y must be in the left memory

X must be in the right memory

Z must be in the left memory

Precision: 23 bit of mantissa

Number of cycles:

98 + 3.25 × Nelements

Number of VLIW:

61

File: vdivlrl.mas

3.118 vdivrll

Function: float array division element by element (equivalent to Matlab: Y./ X), with Y in right memory and X in left memory and precision equal to 23 bit of mantissa

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vdivrll(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

Y**: pointer to the Y input vector. ***Type: float *

strideY: stride to be applied on Y vector. ***Type: int***

X**: pointer to the X input vector. ***Type: float *

strideX: stride to be applied on X vector. ***Type: int***

Z**: pointer to the output vector Z . ***Type: float *

strideZ: stride to be applied on Z vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The vdivrll performs the division with unroll 4, between inputs data vectors X and Y ordered as specified in Restrictions. Y and X are float array, but after their moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. In order to divide Y by a scalar x, simply set *X equal &x

and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Y must be in the right memory

X must be in the left memory

Z must be in the left memory

Precision: 23 bit of mantissa

Number of cycles:

98 + 3.5 × Nelements

Number of VLIW:

59

File: vdivrll.mas

3.119 vdivrrl

Function: float array division element by element (equivalent to Matlab: Y./ X), with X and Y in right memory and precision equal to 23 bit of mantissa

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vdivrrl(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

***Y**: pointer to the Y input vector. ***Type: float ***

strideY: stride to be applied on Y vector. ***Type: int***

***X**: pointer to the X input vector. ***Type: float ***

strideX: stride to be applied on X vector. ***Type: int***

***Z**: pointer to the output vector Z . ***Type: float ***

strideZ: stride to be applied on Z vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The vdivrrl performs the division with unroll 4, between inputs data vectors X and Y ordered as specified in Restrictions. Y and X are float array, but after their moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. In order to divide Y by a scalar x, simply set *X equal &x and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Y must be in the right memory

X must be in the right memory

Z must be in the left memory

Precision: 23 bit of mantissa

Number of cycles:

93 + 3.75 × Nelements

Number of VLIW:

59

File: vdivrrl.mas

3.120 vdivv

Function: vectorial float division element by element (equivalent to Matlab: Y./X)

$$Z(k) = \frac{Y(k)}{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vdivv(*Y, strideY, *X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

***Y**: pointer to the Y input vector. ***Type: __vector__ float ***

strideY**: stride to be applied on Y vector. ***Type: int

***X**: pointer to the X input vector. ***Type: __vector__ float ***

strideX**: stride to be applied on X vector. ***Type: int

***Z**: pointer to the output vector Z. ***Type: __vector__ float ***

strideZ**: stride to be applied on Z vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The function vdivv performs the division between vectorial data inputs X and Y. The operation YL / XL and YR / XR are computed simultaneously. In order to divide Y by a vector float x, simply set *X equal &x and strideX = 0. In order to compute k / X, simply set *Y equal to &k, set k to the desired value and strideY = 0.

Restrictions:

Nelements must be greater than 4 and multiple of 4

Result precision: 23 bits of mantissa

Number of cycles:

90 + 6.75 × Nelements

Number of VLIW:

51

File: vdivv.mas

3.121 vexp10ll

Function: exponential to base 10 (10^x) of a float input array and left to left move

$$Y(k) = 10^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexp10ll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vexp10ll computes the exponential to base 10 of an input array stored in left memory space and writes the output to an array in left memory space.

Precision:

see Table 3-9 on page 125

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in left memory

| x | <= 38, to avoid overflow / underflow of the computed result

Number of cycles:

124 + 10 × Nelements

Number of VLIW:

69

File: vexp10ll.mas, exp10Coeff.mas

3.122 vexp10lr

Function: exponential to base 10 (10^x) of a float input array and left to left to right move

$$Y(k) = 10^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexp10lr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vexp10lr computes the exponential to base 10 of an input array stored in left memory space and writes the output to an array in right memory space.

Precision:

see Table 3-9 on page 125

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

| x | <= 38, to avoid overflow / underflow of the computed result

Number of cycles:

126 + 10 × Nelements

Number of VLIW:

69

File: vexp10lr.mas, exp10Coeff.mas

3.123 vexp10rl

Function: exponential to base 10 (10^x) of a float input array and right to left move

$$Y(k) = 10^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexp10rl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vexp10rl computes the exponential to base 10 of an input array stored in right memory space and writes the output to an array in left memory space.

Precision:

see Table 3-9 on page 125

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

| x | <= 38, to avoid overflow / underflow of the computed result

Number of cycles:

123 + 10 × Nelements

Number of VLIW:

69

File: vexp10rl.mas, exp10Coeff.mas

3.124 vexp10rr

Function: exponential to base 10 (10^x) of a float input array and right to right move

$$Y(k) = 10^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexp10rr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vexp10rr computes the exponential to base 10 of an input array stored in right memory space and writes the output to an array in right memory space.

Precision:

see Table 3-9 on page 125

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in right memory

| x | <= 38, to avoid overflow / underflow of the computed result

Number of cycles:

123 + 10 × Nelements

Number of VLIW:

69

File: vexp10rr.mas, exp10Coeff.mas

3.125 vexp10v

Function: exponential to base 10 (10^x) of a vectorial input array

$$Y(k) = 10^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexp10v (*X, strideX, *Y, strideY, Nelements)

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vexp10v computes the exponential to base 10 of an input array stored in vector space and writes the output to an array in vector space. For computing the base 10 exponential, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: See “vexp10ll” on page 3-121., “vexp10lr” on page 3-122, See “vexp10rl” on page 3-123. and See “vexp10rr” on page 3-124..

Precision:

the following table provides the information about the precision for this function

Table 3-9.

Range of input values	Absolute error	Relative error
-0.1505 to 0.1505	3.84841e-008	5.43603e-008
--38 to -1	1e-010	1
0 to 38	1.108e+028	4.997e-008

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

| x | <= 38, to avoid overflow / underflow of the computed result

Number of cycles:

115 + 18.5 × Nelements

Number of VLIW:

60

File: vexp10v.mas, exp10Coeff.mas

3.126 vexpll

Function: exponential to base **e** (e^x) of a float input array and left to left move

$$Y(k) = e^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexpll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY**: stride to be used for the output array. ***Type: int

Nelements**: number of elements to be computed.Type: int***

The function vexpll computes the exponential to base **e** of an input array stored in left memory space and writes the output to an array in in left memory space.

Precision:

see Figure 3-10 on page 130

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

X must be in left memory

| x | <= 85, to avoid overflow / underflow of the computed result

Number of cycles:

125 + 10 × Nelements

Number of VLIW:

70

File: vexpll.mas, expCoeff.mas

3.127 vexplr

Function: exponential to base **e** (e^x) of a float input array and left to right move

$$Y(k) = e^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexplr (*X, strideX, *Y, strideY, Nelements)

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vexplr computes the exponential to base **e** of an input array stored in left memory space and writes the output to an array in in right memory space.

Precision:

see Table 3-10 on page 130

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

| x | <= 85, to avoid overflow / underflow of the computed result

Number of cycles:

124 + 9.75 × Nelements

Number of VLIW:

66

File: vexplr.mas, expCoeff.mas

3.128 vexprl

Function: exponential to base **e** (e^x) of a float input array and right to left move

$$Y(k) = e^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexprl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vexprl computes the exponential to base **e** of an input array stored in right memory space and writes the output to an array in left memory space.

Precision:

see Table 3-10 on page 130

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

| x | <= 85, to avoid overflow / underflow of the computed result

Number of cycles:

124 + 10 × Nelements

Number of VLIW:

70

File: vexprl.mas, expCoeff.mas

3.129 vexprr

Function: exponential to base **e** (e^x) of a float input array and right to right move

$$Y(k) = e^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexprr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY**: Stride to be used for the output array. ***Type: int

Nelements**: Number of elements to be computed.Type: int***

The function vexprr computes the exponential to base **e** of an input array stored in right memory space and writes the output to an array in right memory space.

Precision:

see Table 3-10 on page 130

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in right memory

| x | <= 85, to avoid overflow / underflow of the computed result

Number of cycles:

123 + 9.75 × Nelements

Number of VLIW:

66

File: vexprr.mas, expCoeff.mas

3.130 vexpv

Function: exponential to base **e** (e^x) of a vectorial input array

$$Y(k) = e^{X(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vexpv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X: pointer to the input array. ***Type: __vector__ float****

strideX: stride to be used for the input array. ***Type: int***

Y: pointer to the output array into which the computed value is written. ***Type: __vector__ float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vexpv computes the exponential to base **e** of an input array stored in vector space and writes the output to an array in vector space. For computing the base **e** exponential, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: “vexpll” on page 3-126, “vexplr” on page 3-127, “vexprl” on page 3-128, “vexprr” on page 3-129.

Precision:

the following table provides the information about the precision for this function

Table 3-10.

Range of input values	Absolute error	Relative error
-0.1505 to 0.1505	3.68082e-010	3.35394e-010
--38 to -1	4.1744e-011	1
0 to 38	1.88624e+006	5.14375e-010

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

| x | <= 85, to avoid overflow / underflow of the computed result

Number of cycles:

116 + 18.5 × Nelements

Number of VLIW:

61

File: vexpv.mas, expCoeff.mas

3.131 vfillll

Function: filling of an array in left memory with a constant stored in left memory

$$Y(k) = X \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfillll (*X, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the input scalar whose value has to be filled. ***Type: float*

*__Y__: pointer to the output array into which the value has to be copied ***Type: float****

strideY: stride to be used for the output vector. ***Type: int***

Nelements: number of elements to be copied.***Type: int***

The function vfillll fills an array in left memory space with a value specified by the input scalar value stored in left memory space.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

20 + 1.5 × Nelements

Number of VLIW:

18

File: vfillll.mas

3.132 vfilllr

Function: filling of an array in right memory with a constant stored in left memory

$$Y(k) = X \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfilllr (*X, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the input scalar whose value has to be filled. ***Type: float*

*__*Y*__: pointer to the output array into which the value has to be copied ***Type: float****

strideY: stride to be used for the output vector. ***Type: int***

Nelements: number of elements to be copied.***Type: int***

The function vfilllr fills an array in right memory space with a value specified by the input scalar value stored in left memory space.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

20 + 1.5 × Nelements

Number of VLIW:

18

File: vfilllr.mas

3.133 vfillrl

Function: filling of an array in left memory with a constant stored in right memory

$$Y(k) = X \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfillrl (*X, *Y, strideY, Nelements)

X**: pointer to the input scalar whose value has to be filled. ***Type: float*

*__*Y*__: pointer to the output array into which the value has to be copied ***Type: float****

strideY: stride to be used for the output vector. ***Type: int***

Nelements: number of elements to be copied.***Type: int***

The function vfillrl fills an array in left memory space with a value specified by the input scalar value stored in right memory space.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

22 + 1.5 × Nelements

Number of VLIW:

19

File: vfillrl.mas

3.134 vfillrr

Function: filling of an array in right memory with a constant stored in right memory

$$Y(k) = X \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfillrr (*X, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the input scalar whose value has to be filled. ***Type: float*

*__*Y*__: pointer to the output array into which the value has to be copied ***Type: float****

strideY: stride to be used for the output vector. ***Type: int***

Nelements: number of elements to be copied.***Type: int***

The function vfillrr fills an array in right memory space with a value specified by the input scalar value stored in right memory space.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

22 + 1.5 × Nelements

Number of VLIW:

19

File: vfillrr.mas

3.135 vfillv

Function: filling of a vectorial array with a vectorial constant

$$Y(k) = X \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfillv (*X, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the input scalar whose value has to be filled. ***Type: __vector__ float*

*****Y**: pointer to the output array into which the value has to be copied ***Type: __vector__ float****

strideY: stride to be used for the output vector. ***Type: int***

Nelements: number of elements to be copied.***Type: int***

The function vfillv fills the vector memory space of the output array with a value specified by the input scalar value. For copying a scalar float value stored in left/right into memory space in left/right, see the fucntions: “vfillll” on page 3-131, “vfilllr” on page 3-132, “vfillrl” on page 3-132 and “vfillrr” on page 3-133.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Number of cycles:

22 + 1.5 × Nelements

Number of VLIW:

19

File: vfillv.mas

3.136 vfix1ll

Function: addition of a float offset, float to integer conversion and left to left move

$$Y(k) = round(X(k) + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix1ll(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X: pointer to float input vector X. ***Type: float ****

strideX: address stride for input vector X. ***Type int***

****Y***: pointer to integer output vector Y. ***Type: int ****

strideY: address stride for output vector Y. ***Type int***

Offset: offset to be applied. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix1ll adds a float offset (Offset) to a float vector input (X) stored in left memory and converts it to integer. The output (Y) is written in left memory. For vectorial data type see the function: “vfix1v” on page 3-139.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in the left memory

Y must be in the left memory

Offset can be either in left or right memory

Number of cycles:

42 + 1 × Nelements

Number of VLIW:

29

File: vfix1ll.mas

3.137 vfix1lr

Function: addition of a float offset, float to integer conversion and left to right move

$$Y(k) = round(X(k) + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix1lr(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to float input vector X. ***Type: float *

strideX**: address stride for input vector X. ***Type: int

Y**: pointer to integer output vector Y. ***Type: int *

strideY: address stride for output vector Y. ***Type: int***

Offset: offset to be applied. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix1lr adds a float offset (Offset) to a float vector input (X) stored in left memory and converts it to integer. The output (Y) is written in right memory. For vectorial data type see the function: “vfix1v” on page 3-139.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in the left memory

Y must be in the right memory

Offset can be either in left or right memory

Number of cycles:

42 + 1 × Nelements

Number of VLIW:

29

File: vfix1lr.mas

3.138 vfix1rl

Function: addition of a float offset, float to integer conversion and right to left move

$$Y(k) = round(X(k) + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix1rl(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to float input vector X. ***Type: float *

strideX: address stride for input vector X. ***Type: int***

Y**: pointer to integer output vector Y. ***Type: int *

strideY: address stride for output vector Y. ***Type: int***

Offset: offset to be applied. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix1rl adds a float offset (Offset) to a float vector input (X) stored in right memory and converts it to integer. The output (Y) is written in left memory. For vectorial data type see the function: “vfix1v” on page 3-139.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in the right memory

Y must be in the left memory

Offset can be either in left or right memory

Number of cycles:

43 + 1 × Nelements

Number of VLIW:

29

File: vfix1rl.mas

3.139 vfix1rr

Function: addition of a float offset, float to integer conversion and right to right move

$$Y(k) = round(X(k) + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix1rr(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to float input vector X. ***Type: float *

strideX**: address stride for input vector X. ***Type: int

Y**: pointer to integer output vector Y. ***Type: int *

strideY**: address stride for output vector Y. ***Type: int

Offset**: offset to be applied. ***Type: float

Nelements**: number of elements that will be moved. ***Type: int

The function vfix1rr adds a float offset (Offset) to a float vector input (X) stored in right memory and converts it to integer. The output (Y) is written in right memory. For vectorial data type see the function: “vfix1v” on page 3-139.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in the left memory

Y must be in the left memory

Offset can be either in left or right memory

Number of cycles:

43 + 1 × Nelements

Number of VLIW:

29

File: vfix1rr.mas

3.140 vfix1v

Function: addition of a vectorial float offset, float to integer conversion and vectorial move

$$Y(k) = round(X(k) + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix1v(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ float *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __vector__ int *

strideY: stride to be applied on output vector. ***Type: int***

Offset: vectorial scalar offset (i.e. pair of scalar offset) to be added to the input vector. ***Type: __vector__ float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfix1v adds a vectorial float offset (Offset) to a vectorial float input array (X) and converts it to integer. For non vectorial data types see the functions: “vfix1ll” on page 3-135, “vfix1lr” on page 3-136, “vfix1rl” on page 3-137 and “vfix1rr” on page 3-138.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

Number of cycles:

53 + 1× Nelements

Number of VLIW:

30

File: vfix1v.mas

3.141 vfix2ll

Function: multiplication by a float value, addition of a float offset, float to integer conversion and left to left move

$$Y(k) = round(X(k) \times Scale + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix2ll(*X, strideX, *Y, strideY, Scale, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to float input vector X. ***Type: float *

strideX: address stride for input vector X. ***Type: int***

Y**: pointer to integer output vector Y. ***Type: int *

strideY: address stride for output vector Y. ***Type: int***

Scale: scalar multiply factor to scale the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix2ll scales a float input array (X) stored in left memory by a float value (Scale), adds a float value (Offset) and converts the values computed into integer. The result (Y) is written in left memory. For vectorial data type see the function: “vfix2v” on page 3-144.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in the left memory

Y must be in the left memory

Offset and Scale can be either in left or right memory

Number of cycles:

34 + 2 × Nelements

Number of VLIW:

36

File: vfix2ll.mas

3.142 vfix2lr

Function: multiplication by a float value, addition of a float offset, float to integer conversion and left to right move

$$Y(k) = round(X(k) \times Scale + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix2lr (*X, strideX, *Y, strideY, Scale, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to float input vector X. ***Type: float*

strideX: address stride for input vector X. ***Type: int***

Y**: pointer to integer output vector Y. ***Type: int*

strideY: address stride for output vector Y. ***Type: int***

Scale: scalar multiply factor to scale the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix2lr scales a float input array (X) stored in left memory by a float value (Scale), adds a float value (Offset) and converts the values computed into integer. The result (Y) is written in right memory. For vectorial data type see the function: “vfix2v” on page 3-144

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in the left memory

Y must be in the right memory

Offset and Scale can be either in left or right memory

Number of cycles:

34 + 2 × Nelements

Number of VLIW:

36

File: vfix2lr.mas

3.143 vfix2rl

Function: multiplication by a float value, addition of a float offset, float to integer conversion and right to left move

$$Y(k) = round(X(k) \times Scale + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix2rl(*X, strideX, *Y, strideY, Scale, Offset, Nelements)

Include file: DSPlib.h.

*X: pointer to float input vector X. Type: ***float ****

strideX: address stride for input vector X. ***Type: int***

****Y***: pointer to integer output vector Y. ***Type: int ****

strideY: address stride for output vector Y. ***Type: int***

Scale: scalar multiply factor to scale the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix2rl scales a float input array (X) stored in right memory by a float value (Scale), adds a float value (Offset) and converts the values computed into integer. The result (Y) is written in left memory. For vectorial data type see the function: “vfix2v” on page 3-144

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in the right memory

Y must be in the left memory

Offset and Scale can be either in left or right memory

Number of cycles:

36 + 2 × Nelements

Number of VLIW:

35

File: vfix2rl.mas

3.144 vfix2rr

Function: multiplication by a float value, addition of a float offset, float to integer conversion and right to right move

$$Y(k) = round(X(k) \times Scale + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix2rr(*X, strideX, *Y, strideY, Scale, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to float input vector X. ***Type: float*

strideX: address stride for input vector X. ***Type: int***

Y**: pointer to integer output vector Y. ***Type: int*

strideY: address stride for output vector Y. ***Type: int***

Scale: scalar multiply factor to scale the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix2rr scales a float input array (X) stored in right memory by a float value (Scale), adds a float value (Offset) and converts the values computed into integer. The result (Y) is written in right memory. For vectorial data type see the function: “vfix2v” on page 3-144

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in the right memory

Y must be in the right memory

Offset and Scale can be either in left or right memory

Number of cycles:

36 + 2 × Nelements

Number of VLIW:

35

File: vfix2rr.mas

3.145 vfix2v

Function: multiplication by a vectorial float value, addition of a vectorial float offset and float to integer conversion

$$Y(k) = round(X(k) \times Scale + Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix2v(*X, strideX, *Y, strideY, Scale, Offset, Nelements)

Include file: DSPlib.h.

***X**: pointer to the input vector. ***Type: __vector__ float ***

strideX**: stride to be applied on input vector. ***Type: int

***Y**: pointer to the output vector. ***Type: __vector__ int ***

strideY**: stride to be applied on output vector. ***Type: int

Scale**: vectorial scalar multiply factor (i.e. pair of scalar multiplier) to scale the input vector. ***Type: __vector__ float

Offset**: vectorial scalar offset (i.e. pair of scalar offset) to be added to the input vector. ***Type: __vector__ float

Nelements**: Number of elements to be computed. ***Type: int

The function vfix2v scales a vectorial float input array (X) by a vectorial float value (Scale), adds a vectorial float value (Offset) and converts the values computed into integer. For non vectorial data type see the functions: “vfix2ll” on page 3-140, “vfix2lr” on page 3-141, “vfix2rl” on page 3-142 and “vfix2rr” on page 3-143.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

Number of cycles:

36 + 2 × Nelements

Number of VLIW:

35

File: vfix2v.mas

3.146 vfix3ll

Function: multiplication by a float value, addition of a float offset, clipping in a float range, float to integer conversion and left to left move

$$Y(k) = round(clip(X(k) \times Scale + Offset)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix3ll (*X, strideX, *Y, strideY, Scale, Offset, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

X**: pointer to float input vector X. ***Type: float *

strideX**: address stride for input vector X. ***Type: int

Y**: pointer to integer output vector Y. ***Type: int *

strideY**: address stride for output vector Y. ***Type: int

***Scal**e: scalar multiply factor to scale the input vector. ***Type: floa**t*

Offset**: scalar offset to be added to the input vector. ***Type: float

ClipUp**: value to be used as upper limit for the data. ***Type: float

ClipDown**: value to be used as lower limit for the data. ***Type: float

Nelements**: number of elements that will be moved. ***Type: int

The function vfix3ll scales a float input array (X) stored in left memory by a float value (Scale), adds a float value (Offset), clips the values computed in a float range (ClipDown, ClipUp) and converts them to integer. The result (Y) is written in left memory. For vectorial data type see the function: “vfix3v” on page 3-149. To clipping the vector float X in the range (ClipUp, ClipDown) means:

if(X > ClipUp) X = ClipUp;

if(X < ClipDown) X = ClipDown;

Restrictions:

Nelements must be greater or equal to 24 and multiple of 4

X must be in the left memory

Y must be in the left memory

Offset, Scale, ClipDown and ClipUp can be either in left or right memory

Number of cycles:

24 + 3.75 × Nelements

Number of VLIW:

55

File: vfix3ll.mas

3.147 vfix3lr

Function: multiplication by a float value, addition of a float offset, clipping in a float range, float to integer conversion and left to right move

$$Y(k) = round(clip(X(k) \times Scale + Offset)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix3lr(*X, strideX, *Y, strideY, Scale, Offset, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

X**: pointer to float input vector X. ***Type: float *

strideX**: address stride for input vector X. ***Type: int

Y**: pointer to integer output vector Y. ***Type: int *

strideY: address stride for output vector Y. ***Type: int***

Scale: scalar multiply factor to scale the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

ClipUp: value to be used as upper limit for the data. ***Type: float***

ClipDown: value to be used as lower limit for the data. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix3lr scales a float input array (X) stored in left memory by a float value (Scale), adds a float value (Offset), clips the values computed in a float range (ClipDown, ClipUp) and converts them to integer. The result (Y) is written in right memory. For vectorial data type see the function: “vfix3v” on page 3-149. To clipping the vector float X in the range (ClipUp, ClipDown) means:

if(X > ClipUp) X = ClipUp;

if(X < ClipDown) X = ClipDown;

Restrictions:

Nelements must be greater or equal to 24 and multiple of 4

X must be in the left memory

Y must be in the right memory

Offset, Scale, ClipDown and ClipUp can be either in left or right memory

Number of cycles:

24 + 3.75 × Nelements

Number of VLIW:

57

File: vfix3lr.mas

3.148 vfix3rl

Function: multiplication by a float value, addition of a float offset, clipping in a float range, float to integer conversion and right to left move

$$Y(k) = round(clip(X(k) \times Scale + Offset)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix3rl(*X, strideX, *Y, strideY, Scale, Offset, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

X**: pointer to float input vector X. ***Type: float *

strideX: address stride for input vector X. ***Type int***

Y**: pointer to integer output vector Y. ***Type: int *

strideY: address stride for output vector Y. ***Type: int***

Scale: scalar multiply factor to scale the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

ClipUp: value to be used as upper limit for the data. ***Type: float***

ClipDown: value to be used as lower limit for the data. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix3rl scales a float input array (X) stored in right memory by a float value (Scale), adds a float value (Offset), clips the values computed in a float range (ClipDown, ClipUp) and converts them to integer. The result (Y) is written in left memory. For vectorial data type see the function: “vfix3v” on page 3-149. To clipping the vector float X in the range (ClipUp, ClipDown) means:

if(X > ClipUp) X = ClipUp;

if(X < ClipDown) X = ClipDown;

Restrictions:

Nelements must be greater or equal to 24 and multiple of 4

X must be in the right memory

Y must be in the left memory

Offset, Scale, ClipDown and ClipUp can be either in left or right memory

Number of cycles:

27 + 3.75 × Nelements

Number of VLIW:

55

File: vfix3rl.mas

3.149 vfix3rr

Function: multiplication by a float value, addition of a float offset, clipping in a float range, float to integer conversion and right to right move

$$Y(k) = round(clip(X(k) \times Scale + Offset)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix3rr(*X, strideX, *Y, strideY, Scale, Offset, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

***X**: Pointer to float input vector X. ***Type: float ***

strideX**: Address stride for input vector X. ***Type: int

Y: Pointer to integer output vector Y. ***Type: int ***

strideY: Address stride for output vector Y. ***Type: int***

Scale: scalar multiply factor to scale the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

ClipUp: value to be used as upper limit for the data. ***Type: float***

ClipDown: value to be used as lower limit for the data. ***Type: float***

Nelements: number of elements that will be moved. ***Type: int***

The function vfix3rr scales a float input array (X) stored in right memory by a float value (Scale), adds a float value (Offset), clips the values computed in a float range (ClipDown, ClipUp) and converts them to integer. The result (Y) is written in right memory. For vectorial data type see the function: “vfix3v” on page 3-149. To clipping the vector float X in the range (ClipUp, ClipDown) means:

if(X > ClipUp) X = ClipUp;

if(X < ClipDown) X = ClipDown;

Restrictions:

Nelements must be greater or equal to 24 and multiple of 4

X must be in the right memory

Y must be in the right memory

Offset, Scale, ClipDown and ClipUp can be either in left or right memory

Number of cycles:

27 + 3.75 × Nelements

Number of VLIW:

57

File: vfix3rr.mas

3.150 vfix3v

Function: multiplication by a vectorial float value, addition of a vectorial float offset, clipping in a vectorial float range and float to integer conversion

$$Y(k) = round(clip(X(k) \times Scale + Offset)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfix3v(*X, strideX, *Y, strideY, Scale, Offset, ClipUp, ClipDown, Nelements)

Include file: DSPlib.h.

***X**:	pointer to the input vector. ***Type: __vector__ float****
strideX:	stride to be applied on input vector. ***Type: int***
***Y**:	pointer to the output vector. ***Type: __vector__ int****
strideY:	stride to be applied on output vector. ***Type: int***
Scale:	vectorial scalar multiply factor (i.e. pair of scalar multiplier) to scale the input vector. T***ype: __vector__ float***
Offset:	vectorial scalar offset (i.e. pair of scalar offset) to be added to the input vector.***Type: __vector__ floa***t
ClipUp:	value to be used as upper limit for the data. ***Type: __vector__ float***
ClipDown:	value to be used as lower limit for the data. ***Type: __vector__ float***
Nelements:	number of elements to be computed. ***Type: int***

The function vfix3v scales a vectorial float input array (X) by a vectorial float value (Scale), adds a vectorial float value (Offset), clips the values computed in a vectorial float range (ClipDown, ClipUp) and converts them to integer. For non vectorial data type see the functions: “vfix3ll” on page 3-145, “vfix3lr” on page 3-146, “vfix3rl” on page 3-147 and “vfix3rr” on page 3-148. To clipping the vector float X in the range (ClipUp, ClipDown) means:

if(Re(X) > Re(ClipUp)) Re(X) = Re(ClipUp);

if(Im(X) > Im(ClipUp)) Im(X) = Im(ClipUp);

if(Re(X) > Re(ClipDown)) Re(X) = Re(ClipDown);

if(Im(X) > Im(ClipDown)) Im(X) = Im(ClipDown);

Restrictions:

Nelements must be greater or equal to 24 and multiple of 4

Number of cycles:

44 + 3 × Nelements

Number of VLIW:

61

File: vfix3v.mas

3.151 vfloat1ll

Function: integer to float conversion, addition of a float offset and left to left move

$$Y(k) = float(X(k)) + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfloat1ll(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: int *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: float *

strideY: stride to be applied on output vector. ***Type: int***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfloat1ll executes the float conversion of the integer data input (X) and adds to it a float scalar offset (Offset). For function operating on vectorial types see the function “vfloat1v” on page 3-155.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in left memory

Y must be in left memory

Offset can be either in left or right memory

Number of cycles:

36 + 1 × Nelements

Number of VLIW:

28

File: vfloat1ll.mas

3.152 vfloat1lr

Function: integer to float conversion, addition of a float offset and left to right move

$$Y(k) = float(X(k)) + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfloat1lr(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

***X**: pointer to the input vector. ***Type: int ***

strideX**: stride to be applied on input vector. ***Type: int

***Y**: pointer to the output vector. ***Type: float ***

strideY**: stride to be applied on output vector. ***Type: int

Offset**: scalar offset to be added to the input vector. ***Type: float

Nelements**: Number of elements to be computed. ***Type: int

The function vfloat1lr executes the float conversion of the integer data input (X) and adds to it a float scalar offset (Offset). For function operating on vectorial types see the function “vfloat1v” on page 3-155.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in left memory

Y must be in right memory

Offset can be either in left or right memory

Number of cycles:

36 + 1 × Nelements

Number of VLIW:

28

File: vfloat1lr.mas

3.153 vfloat1rl

Function: integer to float conversion, addition of a float offset and right to left move

$$Y(k) = float(X(k)) + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfloat1rl(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: int *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: float *

strideY: stride to be applied on output vector. ***Type: int***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfloat1rl executes the float conversion of the integer data input (X) and adds to it a float scalar offset (Offset). For function operating on vectorial types see the function “vfloat1v” on page 3-155.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in right memory

Y must be in left memory

Offset can be either in left or right memory

Number of cycles:

39 + 1 × Nelements

Number of VLIW:

29

File: vfloat1rl.mas

3.154 vfloat1rr

Function: integer to float conversion, addition of a float offset and right to right move

$$Y(k) = float(X(k)) + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfloat1rr(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: int *

strideX**: stride to be applied on input vector. ***Type: int

Y**: pointer to the output vector. ***Type: float *

strideY: stride to be applied on output vector. ***Type: int***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfloat1rr executes the float conversion of the integer data input (X) and adds to it a float scalar offset (Offset). For function operating on vectorial types see the function “vfloat1v” on page 3-155.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

X must be in right memory

Y must be in left memory

Offset can be either in left or right memory

Number of cycles:

39 + 1 × Nelements

Number of VLIW:

29

File: vfloat1rr.mas

3.155 vfloat1v

Function: vectorial integer to float conversion and addition of a vectorial float offset

$$Re(Y(k)) = float(Re(X(k))) + Re(Offset) \qquad k = 0 \ldots Nelements - 1$$
$$Im(Y(k)) = float(Im(X(k))) + Im(Offset) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfloat1v(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ int *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __vector__ float *

strideY: stride to be applied on output vector. ***Type: int***

Offset: vectorial scalar offset to be added to the input vector. ***Type: __vector__ float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfloat1v works on vectorial (or complex) data type. It returns the float conversion of the input data vector and the addition of a vectorial float scalar offset to it. For function operating on not vectorial types see the functions: “vfloat1ll” on page 3-151, “vfloat1lr” on page 3-152, “vfloat1rl” on page 3-153, “vfloat1rr” on page 3-154.

Restrictions:

Nelements must be greater or equal to 16 and multiple of 4

Number of cycles:

39 + 1 × Nelements

Number of VLIW:

29

File: vfloat1v.mas

3.156 vfloat2ll

Function: integer to float conversion, multiplication by a float scale factor, addition of a float offset and left to left move

$$Y(k) = float(X(k)) \times Scale + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfloat2ll(*X, strideX, *Y, strideY, Scale, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: int *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: float *

strideY: stride to be applied on output vector. ***Type: int***

Scale: scalar factor to multiply the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfloat2ll executes the float conversion of the integer data input (X), multiplies it by a float scale factor (Scale) and adds to it a float scalar offset (Offset). For function operating on vectorial types see the function “vfloat2v” on page 3-160.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

X must be in left memory

Y must be in left memory

Scale can be either in left or right memory

Offset can be either in left or right memory

Number of cycles:

37 + 2 × Nelements

Number of VLIW:

33

File: vfloat2ll.mas

3.157 vfloat2lr

Function: integer to float conversion, multiplication by a float scale factor, addition of a float offset and left to right move

$$Y(k) = float(X(k)) \times Scale + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfloat2lr(*X, strideX, *Y, strideY, Scale, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: int *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: float *

strideY: stride to be applied on output vector. ***Type: int***

Scale: scalar factor to multiply the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfloat2lr executes the float conversion of the integer data input (X), multiply it by a float scale factor (Scale) and adds to it a float scalar offset (Offset). For function operating on vectorial types see the function “vfloat2v” on page 3-160.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

X must be in left memory

Y must be in right memory

Scale can be either in left or right memory

Offset can be either in left or right memory

Number of cycles:

37 + 2 × Nelements

Number of VLIW:

33

File: vfloat2lr.mas

3.158 vfloat2rl

Function: integer to float conversion, multiplication by a float scale factor, addition of a float offset and right to left move

$$Y(k) = float(X(k)) \times Scale + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfloat2rl(*X, strideX, *Y, strideY, Scale, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: int *

strideX**: stride to be applied on input vector. ***Type: int

Y**: pointer to the output vector. ***Type: float *

strideY**: stride to be applied on output vector. ***Type: int

Scale: scalar factor to multiply the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfloat2rl executes the float conversion of the integer data input (X), multiply it by a float scale factor (Scale) and adds to it a float scalar offset (Offset). For function operating on vectorial types see the function “vfloat2v” on page 3-160.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

X must be in right memory

Y must be in left memory

Scale can be either in left or right memory

Offset can be either in left or right memory

Number of cycles:

39 + 2 × Nelements

Number of VLIW:

34

File: vfloat2rl.mas

3.159 vfloat2rr

Function: integer to float conversion, multiplication by a float scale factor, addition of a float offset and right to right move

$$Y(k) = float(X(k)) \times Scale + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vfloat2rr(*X, strideX, *Y, strideY, Scale, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: int *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: float *

strideY: stride to be applied on output vector. ***Type: int***

Scale: scalar factor to multiply the input vector. ***Type: float***

Offset: scalar offset to be added to the input vector. ***Type: float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfloat2rr executes the float conversion of the integer data input (X), multiply it by a float scale factor (Scale) and adds to it a float scalar offset (Offset). For function operating on vectorial types see the function “vfloat2v” on page 3-160.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

X must be in right memory

Y must be in right memory

Scale can be either in left or right memory

Offset can be either in left or right memory

Number of cycles:

39 + 2 × Nelements

Number of VLIW:

34

File: vfloat2rr.mas

3.160 vfloat2v

Function: vectorial integer to vectorial float conversion, multiplication by a vectorial float scale factor and addition of a vectorial float offset

$$Re(Y(k)) = float(Re(X(k))) + Re(Offset) \qquad k = 0\ldots Nelements - 1$$
$$Im(Y(k)) = float(Im(X(k))) + Im(Offset) \qquad k = 0\ldots Nelements - 1$$

Synopsis: __vector__ int vfloat2v(*X, strideX, *Y, strideY, Offset, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ int *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __vector__ float *

strideY: stride to be applied on output vector. ***Type: int***

Scale: vectorial scale factor to multiply the input vector. ***Type: __vector__ float***

Offset: vectorial scalar offset to be added to the input vector. ***Type: __vector__ float***

Nelements: Number of elements to be computed. ***Type: int***

The function vfloat2v works on vectorial (or complex) data type. It returns the float conversion of the input data vector, multiplies it by a vectorial scale factor (Scale) and addits to it a vectorial float scalar offset. For function operating on not vectorial types see the functions: “vfloat2ll” on page 3-156, “vfloat2lr” on page 3-157, “vfloat2rl” on page 3-158, “vfloat2rr” on page 3-159.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

39 + 2 × Nelements

Number of VLIW:

34

File: vfloat2v.mas

3.161 vlog10ll

Function: logarithm to base **10** of a float input array and left to left move

$$Y(k) = \boldsymbol{log}10(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vlog10ll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY**: stride to be used for the output array. ***Type: int

Nelements**: number of elements to be computed.Type: int***

The function vlog10ll computes the logarithm to base 10 of an input array stored in left memory space and writes the output to an array in left memory space.

Precision:

see Table 3-11 on page 165

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

156 + 13 × Nelements

Number of VLIW:

85

File: vlog10ll.mas

3.162 vlog10lr

Function: logarithm to base **10** of a float input array and left to right move

$$Y(k) = \boldsymbol{log}10(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int File: vlog10ll.mas (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written.Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function File: vlog10ll.mas computes the logarithm to base 10 of an input array stored in left memory space and writes the output to an array in RIGHT memory space.

Precision:

see Table 3-11 on page 165

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

156 + 13 × Nelements

Number of VLIW:

85

File: vlog10lr.mas, log10Coeff.mas

3.163 vlog10rl

Function: logarithm to base **10** of a float input array and right to left move

$$Y(k) = \boldsymbol{log}10(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vlog10rl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written.Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vlog10rl computes the logarithm to base 10 of an input array stored in right memory space and writes the output to an array in left memory space.

Precision:

see Table 3-11 on page 165

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

156 + 13 × Nelements

Number of VLIW:

85

File: vlog10rl.mas, log10Coeff.mas

3.164 vlog10rr

Function: logarithm to base **10** of a float input array and right to right move

$$Y(k) = \boldsymbol{log}10(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vlog10rr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written.Type: float****

strideY**: stride to be used for the output array. ***Type: int

Nelements**: number of elements to be computed.Type: int***

The function vlog10rr computes the logarithm to base 10 of an input array stored in right memory space and writes the output to an array in right memory space.

Precision:

see Table 3-11 on page 165

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

154 + 13 × Nelements

Number of VLIW:

86

File: vlog10rr.mas, log10Coeff.mas

3.165 vlog10v

Function: logarithm to base **10** of a vectorial input array

$$Y(k) = \boldsymbol{log}10(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vlog10v (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vlog10v computes the natural logarithm of an input array stored in vector space and writes the output to an array in vector memory space. For computing the natural logarithm, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: “vlog10ll” on page 3-161, “File: vlog10ll.mas” on page 3-161, “vlog10rl” on page 3-163 and “vlog10rr” on page 3-164.

Precision:

the following table provides the information about the precision for this function

Table 3-11.

Range of input values	Absolute error	Relative error
1 to 1.414	3.78428e-010	NA
10 to 10^{38}	2e-010	2e-010
10^{-1} to 10^{-38}	NA	2.22045e-016

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

Number of cycles:

143 + 24.5 × Nelements

Number of VLIW:

74

File: vlog10v.mas, log10Coeff.mas

3.166 vlogll

Function: natural logarithm of a float input array and left to left move

$$Y(k) = \boldsymbol{log}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vlogll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vlogll computes the natural logarithm of an input array stored in left memory space and writes the output to an array in left memory space.

Precision:

see Table 3-12 on page 170

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

157 + 13 × Nelements

Number of VLIW:

85

File: vlogll.mas, lnCoeff.mas

3.167 vloglr

Function: natural logarithm of a float input array and left to right move

$$Y(k) = \boldsymbol{log}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vloglr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written.Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vloglr computes the natural logarithm of an input array stored in left memory space and writes the output to an array in right memory space.

Precision:

see Table 3-12 on page 170

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

156 + 13 × Nelements

Number of VLIW:

82

File: vloglr.mas, lnCoeff.mas

3.168 vlogrl

Function: natural logarithm of a float input array and right to left move

$$Y(k) = \boldsymbol{log}(X(k)) \qquad k = 0\ldots Nelements - 1$$

Synopsis: __vector__ int vlogrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written.Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vlogrl computes the natural logarithm of an input array stored in right memory space and writes the output to an array in left memory space.

Precision:

see Table 3-12 on page 170

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

157 + 13 × Nelements

Number of VLIW:

86

File: vlogrl.mas, lnCoeff.mas

3.169 vlogrr

Function: natural logarithm of a float input array and right to right move

$$Y(k) = \boldsymbol{log}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vlogrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written.Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vlogrr computes the natural logarithm of an input array stored in right memory space and writes the output to an array in right memory space.

Precision:

see Table 3-12 on page 170

Restrictions:

Nelements must be multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

154 + 13 × Nelements

Number of VLIW:

86

File: vlogrr.mas, lnCoeff.mas

3.170 vlogv

Function: natural logarithm of a vectorial input array

$$Y(k) = \boldsymbol{log}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vlogv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vlogv computes the natural logarithm of an input array stored in vector space and writes the output to an array in vector memory space. For computing the natural logarithm, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: “vlogll” on page 3-166, See “vloglr” on page 3-167., “vlogrl” on page 3-168 and “vlogrr” on page 3-169.

Precision:

the following table provides the information about the precision for this function

Table 3-12.

Range of input values	Absolute error	Relative error
1 to 1.414	7.43022e-010	NA
10 to 10^{38}	2.20154e-008	3.08425e-010
10^{-1} to 10^{-38}	NA	2.97906e-010

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

Number of cycles:

143 + 24.5 × Nelements

Number of VLIW:

74

File: vlogv.mas, lnCoeff.mas

3.171 vmagnlrl

Function: vector magnitude

$$Z(k) = \sqrt{X(k) + Y(k)} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmagnlrl(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the float input vector X. ***Type: float *

StrideX: stride to be applied on X input vector. ***Type: int***

Y**: pointer to the float input vector X. Type: ***float *

StrideY: stride to be applied on Y input vector. ***Type: int***

Z**: pointer to the result vector Z. ***Type: float *

strideZ: stride to be applied on Z output vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The function vmagnlrl computes the magnitude of a pair of float array: X and Y. The first must be stored in left memory, the second in right memory. The result is written in left memory.

Restrictions:

Nelements can be any number greater or equal to 1

X must be in left data memory

Y must be in right data memory

Z must be in left data memory

Precision: 23 bit of mantissa. If higher precision is required it is possible to perform on more Newton iteration by modifying the source code (simply uncomment the last iteration).

Number of cycles:

30 + 41 × Nelements

Number of VLIW:

31

File: vmagnlrl.mas

3.172 vmagnv

Function: complex magnitude

$$Z(k) = \sqrt{(ReX(k))^2 + (ImX(k))^2} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmagnv(*X, strideX, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __complex__ float *

strideX**: stride to be applied on input vector. ***Type: int

Z**: pointer to the result vector Z. ***Type: float *

strideZ**: stride to be applied on Z vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The function vmagnv ia a fully pipelined vectorial version of vmagnlrl.

Restrictions:

Nelements must be grater or equal to 4 and multiple of 4

Z must be in left data memory

Result precision: 23 bits of mantissa. If higher precision is required it is possible to perform on more Newton iteration by modifying the source code (simply uncomment the last iteration)

Number of cycles:

115 + 8.75 × Nelements

Number of VLIW:

84

File: vmagnv.mas

3.173 vmaxv

Function: vectorial maximum

$$max(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmaxv(*X, strideX, *Max, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ float *

strideX: stride to be applied on input vector. ***Type: int***

Max**: pointer to the vectorial float locations containing left and right maxims. ***Type: __vector__ float *

Nelements: number elements to be compared. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses. ***Type: int***

The function vmaxv performs a vectorial maxims search.

Restrictions:

Nelements can be any number greater or equal to 8 and multiple of 4

Number of cycles:

43 + 1 × Nelements

Number of VLIW:

29

File: vmaxv.mas

3.174 vmax1v

Function: pipelined vectorial maximum with indexes extraction

$$\begin{cases} Max = max(X(k)) \\ Idx\_Max = index(X(k)) \end{cases} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmax1v(*X, strideX, *Max, *Idx_Max, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ float*

strideX**: stride to be applied on input vector. ***Type: int

***Max**: pointer to the vectorial float location containing left and right maxims.

Type: __vector__ float*

Idx_Max**: pointer to the vectorial int location containing left and right indexes of maxims. ***Type: __vector__ int*

Nelements**: number of elements to be compared. Set this parameter to the length of the vector divided by the stride. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses. ***Type: int

The function vmax1v performs the vectorial maxims and index of maxims search. For a non pipelined version see the function: “vmax2v” on page 3-174.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4.

Number of cycles:

54 + 7.25 × Nelements

Number of VLIW:

63

File: vmax1v.mas

3.175 vmax2v

Function: vectorial maximum with indexes extraction

$$\begin{cases} Max = max(X(k)) \\ Idx\_Max = index(X(k)) \end{cases} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmax2v(*X, stride, *Max, *Idx_Max, Vsize)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ float*

strideX**: stride to be applied on input vector. ***Type: int

***Max**: pointer to the vectorial float location containing left and right maxims.

Type: __vector__ float*

Idx_Max**: pointer to the vectorial int location containing left and right indexes of maxims. ***Type: __vector__ int *

Nelements**: number elements to be compared. Set this parameter to the length of the vector divided by the stride. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses. ***Type: int

The function vmax2v performs the vectorial maxims and index of maxims search. For a pipelined version see the function: “vmax1v” on page 3-173.

Restrictions:

Nelements can be any number greater or equal to 3

Number of cycles:

33 + 8 × Nelements

Number of VLIW:

35

File: vmax2v.mas

3.176 vmmul

Function: product of a complex vector with a complex matrix

$$C(k) = \sum_{i=0}^{M-1} A(i) \times B(i,k) \qquad k = 0 \ldots N-1$$

Synopsis: __vector__ int vmmul (*A, *B, M, N, *C)

Include file: DSPlib.h

A**: pointer to the input vector . ***Type: __complex__ float *****

B**: pointer to the input matrix . ***Type: __complex__ float *****

M**: number of coloumns of matrix A and rows of matrix B. ***Type: int

N**: number of coloumns of matrix B. ***Type: int

C**: pointer to the output matrix . ***Type: __complex__ float *****

The function vmmul computes the product of a complex vector of length M (order $1 \times M$) with a complex matrix of order $M \times N$.

Restrictions:

M should be > 1.

Number of cycles:

50 + ((6 × (M - 1)) + 18) × N

umber of VLIW:

42

File: vmmul.mas

3.177 vmove2cx

Function: complex conjugate vector move with scale factor and offset

$$Y(k) = conj(X(k)) \times Scale + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmove2cx(*X, strideX, *Y, strideY, Scale, Offset, Nelements);

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __complex__ float *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __complex__ float *

strideY: stride to be applied on input vector. ***Type: int***

Scale: is the scale factor. ***Type: __complex__ float***

Offset: is the offset to be added . ***Type: __complex__ float***

Nelements: number elements to be moved. Set this parameter to the length of the vector divided by the stride. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses.

The function vmove2cx moves complex conjugate data with scale and offset. Note that simple move is obtained by multiply with the complex unity (1.0 + 0.0i) and addition with complex zero (0.0 + 0.0i).

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

30 + 1× Nelements

Number of VLIW:

26

File: vmove2cx.mas

3.178 vmove2cxint

Function: complex conjugate vector integer move with scale factor and offset

$$Y(k) = conj(X(k)) \times Scale + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmove2cxint(*X, strideX, *Y, strideY, Scale, Offset, Nelements);

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __complex__ int*

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __complex__ int*

strideY: stride to be applied on input vector. ***Type: int***

Scale: is the scale factor. ***Type: __complex__ int***

Offset: is the offset to be added . ***Type: __complex__ int***

Nelements: number elements to be moved. Set this parameter to the length of the vector divided by the stride. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses. ***Type: int***

The function vmove2cxint moves complex conjugate integer data with scale and offset. Note that simple move is obtained by multiply with the complex unity (1 + 0i) and addition with complex zero (0 + 0i).

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

32 + 2.25 × Nelements

Number of VLIW:

31

File: vmove2cxint.mas

3.179 vmove2v

Function: vectorial move with scale factor and offset

$$\begin{cases} ReY(k) = ReX(k) \times Re(Scale) + Re(Offset) \\ ImY(k) = ReX(k) + Im(Scale) + Im(Offset) \end{cases} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmove2v(*X, strideX, *Y, strideY, Scale, Offset, Nelements);

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ float *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __vector__ float *

strideY: stride to be applied on input vector. ***Type: int***

Scale: is the scale factor. ***Type: __vector__ float***

Offset: is the offset to be added . ***Type: __vector__ float***

Nelements: number elements to be moved. Set this parameter to the length of the vector divided by the stride. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses.

The function vmove2v moves vectorial data with scale and offset. Note that simple move is obtained by multiply with the complex unity (1.0 + 1.0i) and addition with complex zero (0.0 + 0.0i).

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

28 + 1× Nelements

Number of VLIW:

25

File: vmove2v.mas

3.180 vmove2vint

Function: vectorial integer move with scale factor and offset

$$\begin{cases} ReY(k) = ReX(k) \times Re(Scale) + Re(Offset) \\ ImY(k) = ImX(k) + Im(Scale) + Im(Offset) \end{cases} \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmove2vint(*X, strideX, *Y, strideY, Scale, Offset, Nelements);

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ int*

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __vector__ int*

strideY: stride to be applied on input vector. ***Type: int***

Scale: is the scale factor. ***Type: __vector__ int***

Offset: is the offset to be added . ***Type: __vector__ int***

Nelements: number elements to be moved. Set this parameter to the length of the vector divided by the stride. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses. ***Type: int***

The function vmove2vint moves vector integer data with scale and offset. Note that simple move is obtained by multiply with the complex unity (1.0 + 1.0i) and addition with complex zero (0.0 + 0.0i).

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

30 + 2 × Nelements

Number of VLIW:

30

File: vmove2vint.mas

3.181 vmove2x

Function: complex vector move with scale factor and offset

$$Y(k) = X(k) \times Scale + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmove2x(*X, strideX, *Y, strideY, Scale, Offset, Nelements);

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __complex__ float *

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __complex__ float *

strideY: stride to be applied on input vector. ***Type: int***

Scale: is the scale factor. ***Type: __complex__ float***

Offset: is the offset to be added . ***Type: __complex__ float***

Nelements: number elements to be moved. Set this parameter to the length of the vector divided by the stride. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses.

The function vmove2x moves complex data with scale and offset. Note that simple move is obtained by multiply with the complex unity (1.0 + 0.0i) and addition with complex zero (0.0 + 0.0i).

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

30 + 1× Nelements

Number of VLIW:

27

File: vmove2x.mas

3.182 vmove2xint

Function: complex integer vector move with scale factor and offset

$$Y(k) = X(k) \times Scale + Offset \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmove2xint(*X, strideX, *Y, strideY, Scale, Offset, Nelements);

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __complex__ int*

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __complex__ int*

strideY: stride to be applied on input vector. ***Type: int***

Scale: is the scale factor. ***Type: __complex__ int***

Offset: is the offset to be added . ***Type: __complex__ int***

Nelements: number elements to be moved. Set this parameter to the length of the vector divided by the stride. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses. ***Type: int***

The function vmove2xint moves complex integer data with scale and offset. Note that simple move is obtained by multiply with the complex unity (1 + 0i) and addition with complex zero (0 + 0i).

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

32 + 2.25 × Nelements

Number of VLIW:

31

File: vmove2xint.mas

3.183 vmovell

Function: left to left float array move

$$Y(k) = X(k) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vmovell(*X, StrideX, *Y, StrideY, Nelements)

Include file: DSPlib.h.

X: pointer to the input vector. The ***type*** can be: ***float*** * or ***int****

strideX: stride to be applied on input vector. ***Type: int***

****Y***: pointer to the output vector. The ***type*** can be: ***float*** * or ***int****

strideY: stride to be applied on input vector. ***Type: int***

Nelements: number of elements to be moved

The function vmovell moves data from left to left memory banks .

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

20 + 1× Nelements

Number of VLIW:

18

File: vmovell.mas

3.184 vmovelr

Function: left to right float array move

$$Y(k) = X(k) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vmovelr(*X, StrideX, *Y, StrideY, Nelements)

Include file: DSPlib.h.

X: pointer to the input vector. The ***type*** can be: ***float*** * or ***int****

strideX: stride to be applied on input vector. ***Type: int***

****Y***: pointer to the output vector. The ***type*** can be: ***float*** * or ***int****

strideY: stride to be applied on input vector. ***Type: int***

Nelements: number of elements to be moved

The function vmovelr moves data from left to right memory banks .

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

20 + 1 × Nelements

Number of VLIW:

18

File: vmovelr.mas

3.185 vmoverl

Function: right to left float array move

$$Y(k) = X(k) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vmoverl(*X, StrideX, *Y, StrideY, Nelements)

Include file: DSPlib.h.

X: pointer to the input vector. The ***type*** can be: ***float*** * or ***int****

strideX: stride to be applied on input vector. ***Type: int***

****Y***: pointer to the output vector. The ***type*** can be: ***float*** * or ***int****

strideY: stride to be applied on input vector. ***Type: int***

Nelements: number of elements to be moved

The function vmoverl moves data from right to left memory banks .

Restrictions:

Nelements must be multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

24 + 1 × Nelements

Number of VLIW:

18

File: vmoverl.mas

3.186 vmoverr

Function: right to right float array move

$$Y(k) = X(k) \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vmoverr(*X, StrideX, *Y, StrideY, Nelements)

Include file: DSPlib.h.

X: pointer to the input vector. The ***type*** can be: ***float*** * or ***int****

strideX: stride to be applied on input vector. ***Type: int***

****Y***: pointer to the output vector. The ***type*** can be: ***float*** * or ***int****

strideY: stride to be applied on input vector. ***Type: int***

Nelements: number of elements to be moved

The function vmoverr moves data from right to right memory banks .

Restrictions:

Nelements must be multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

23 + 1× Nelements

Number of VLIW:

19

File: vmoverr.mas

3.187 vmovev

Function: vector move

$$Y(k) = X(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmovev(*X, strideX, *Y, strideY, Nelements);

Include file: DSPlib.h.

X**: pointer to the input vector. The ***type can be: ***__vector__ float*** * or ***__vector__ int****

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. The ***type can be: ***__vector__ float*** * or ***__vector__ int****

strideY: stride to be applied on input vector. ***Type: int***

Nelements: number elements to be moved. Set this parameter to the length of the vector divided by the stride. Note: since the parameters aren't checked by the function the user has to properly set this parameter to avoid incorrect results and out of vector accesses.

The function vmovev moves vectorial data.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

Number of cycles:

19 + 1× Nelements

Number of VLIW:

18

File: vmovev.mas

3.188 vmvell

Function: mean stored in left memory of a float input array stored in left memory

$$Y = \frac{1}{Nelements} \sum_{k=0}^{Nelements-1} X(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmvell (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output scalar into which the computed value is written. ***Type: float*

Nelements: number of elements to be computed.***Type: int***

The function vmvell computes the mean of a float input array stored in left memory space and writes the computed value to an output location in left memory space. To computing the mean on a vectorail float array see the function: “vmvev” on page 3-189.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4,

X must be in left memory

Y must be in left memory

Number of cycles:

54 + 1 × Nelements

Number of VLIW:

29

File: vmvell.mas

3.189 vmvelr

Function: mean stored in right memory of a float input array stored in left memory

$$Y = \frac{1}{Nelements}\sum_{k=0}^{Nelements-1} X(k) \qquad k = 0 \ldots Nelements-1$$

Synopsis: __vector__ int vmvelr (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output scalar into which the computed value is written.Type: float****

Nelements: number of elements to be computed.***Type: int***

The function vmvelr computes the mean of an input array stored in left memory space and writes the computed value to an output location in right memory space. To computing the mean on a vectorail float array see the function: “vmvev” on page 3-189.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

54 + 1 × Nelements

Number of VLIW:

29

File: vmvelr.mas

3.190 vmverl

Function: mean stored in left memory of a float input array stored in right memory

$$Y = \frac{1}{Nelements}\sum_{k=0}^{Nelements-1} X(k) \qquad k = 0 \ldots Nelements-1$$

Synopsis: __vector__ int vmverl (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output scalar into which the computed value is written ***Type: float*

Nelements: number of elements to be computed.***Type: int***

The function vmverl computes the mean of an input array stored in right memory space and writes the computed value to an output location in left memory space. To computing the mean on a vectorail float array see the function: “vmvev” on page 3-189.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

54 + 1 × Nelements

Number of VLIW:

30

File: vmverl.mas

3.191 vmverr

Function: mean stored in right memory of a float input array stored in right memory

$$Y = \frac{1}{Nelements} \sum_{k=0}^{Nelements-1} X(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vmverr (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output scalar into which the computed value is written. ***Type: float*

Nelements: number of elements to be computed.***Type: int***

The function vmverr computes the mean of an input array stored in right memory space and writes the computed value to an output location in right memory space. To computing the mean on a vectorail float array see the function: “vmvev” on page 3-189.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

55 + 1 × Nelements

Number of VLIW:

30

File: vmverr.mas

3.192 vmvev

Function: mean of a vectorial input array

$$Y = \frac{1}{Nelements} \sum_{k=0}^{Nelements-1} X(k) \qquad k = 0 \ldots Nelements-1$$

Synopsis: __vector__ int vmvev (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output scalar in the vector space into which the computed value is written. ***Type: __vector__ float*

Nelements: number of elements to be computed.***Type: int***

The function vmvev computes the mean of a vectorial input arrays (X). To computing mean on non vectorial data see the functions: “vmvell” on page 3-186, “vmvelr” on page 3-187, “vmverl” on page 3-187 and “vmverr” on page 3-188.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Number of cycles:

55 + 1 × Nelements

Number of VLIW:

31

File: vmvev.mas

3.193 vq2vq

Function: copy of vectorial (left - right) data from the vector q1 to the vector q2

Synopsis: int vq2vq(*q1, *q2, Nelements);

q1**: pointer to a vector queue structure defined using the vqdef macro. ***Type: void *

q2**: pointer to a vector queue structure defined using the vqdef macro. ***Type: void *

Nelements: number of elements copied. ***Type: int***

The function vq2vq copies data from the vector queue q1 to vector queue q2. If the number of elements available in the vector queue1 is lower than Nelements a -1 is returned (q1 underrun), but the copy is anyway done. If the number of elements available in the vector q2 is lower than Nelements a -2 is returned (q2 overrun), but the copy is anyway done.This allows using the vq2vq also in a non-strictly queued structure, but in structures where circular addressing is used over a vector. If the number of elements available in the vector q1 and in the vector q2 are both lower than Nelements a -3 is returned (q1 underrun and q2 overrun), but the copy is anyway done. A vector queue is a structure defined using the macro “vqdef” and explicitly declared using that macro see the function: “initvq” on page 3-40. If the return code is not checked the structure is simply a circular buffer and consistency must be guaranteed by the user.

Return code:

0	no error
-1	queue1 underrun
-2	queue2 overrun
-3	queue1 underrun and queue2 overrun

Recall:

Nelements can be 2047 elements max

Restrictions:

Number of element must be greater than 12 and multiple of 4.

Number of cycles:

132 + 1 × Nelements

Number of VLIW:

56

File: vq2vq.mas

3.194 vrandl

Function: random numbers generator in left memory

$$\begin{cases} Y(k) = SEED_k \times Norm + Offset & k = 0 \ldots Nelements \\ SEED_k = (A \times SEED_{k-1} + C) Mod 2^{32} & A = 69069 \quad C = 1 \end{cases}$$

Synopsis: vrandl(*Y, strideY, Norm, Offset, Nelements)

Include file: DSPlib.h.

Y**: pointer to the output vector. ***Type: float*

strideY**: stride to be applied on output vector. ***Type: int

Norm: normalization factor. Norm must be equal to 2^(-32) if a random number with module in the range [0, 1) is needed. ***Type: float***

Offset: is the offset to be added. ***Type: float***

Nelements: number of elements to be computed. ***Type:int***

The function vrandl generates a float array in left memory, of random numbers using a linear congruential method, described above, multiplies for a float normalization factor and adds a float offset. For the float version with output in right memory, see the function “vrandr” on page 3-192. For the vectorial version see the function “vrandv” on page 3-193. All the 3 functions uses the same 2 vectorials SEED variables. This variables are updated by the 3 functions coherently in order to generate random non-correlated subsequences independently from the order of usage of the different functions. The initial 2 values of the SEED variables: SEED1 and SEED2, have been choosen in order to compute 4 independent pseudorandom values at each algorithm execution and to maintain the maximum repetition period (must be 2^32):

$$\begin{cases} SEED1 & (SEED_0, SEED_{2^{32}/4}) \\ SEED2 & (SEED_{2^{32}/2}, SEED_{3 \times 2^{32}/4}) \\ with & SEED_0 = 0 \end{cases}$$

They are stored in Internal Memory at the address of the ***LABEL ___ATMlib__SEED.*** The call to the randl or the randr function can be mixed with the call to the vrand function still generating a maximum length pseudorandom sequence. For this reason the vrandl and vrandr functions are built with unroll 4 while the vrand functios is built with unroll 2. The real and the imaginary part of two pseudorandom vectorials numbers generated at each iteration of the algorithm, are arranged in left memory in a float array.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Y must be in left memory

Number of cycles:

37 + 2.5 × Nelements

Number of VLIW:

41

File: vrandl.mas

3.195 vrandr

Function: random numbers generator in right memory

$$\begin{cases} Y(k) = SEED_k \times Norm + Offset & k = 0 \ldots Nelements \\ SEED_k = (A \times SEED_{k-1} + C) Mod 2^{32} & A = 69069 \qquad C = 1 \end{cases}$$

Synopsis: vrandr(*Y, strideY, Norm, Offset, Nelements)

Include file: DSPlib.h.

****Y***: pointer to the output vector. ***Type: float****

strideY: stride to be applied on output vector. ***Type: int***

Norm: normalization factor. Norm must be equal to 2^(-32) if a random number with module in the range [0, 1) is needed. ***Type: float***

Offset: is the offset to be added. ***Type: float***

Nelements: number of elements to be computed. ***Type:int***

The function vrandr generates a float array in left memory, of random numbers using a linear congruential method, described above, multiplies for a float normalization factor and adds a float offset. For the float version with output in left memory, see the function “vrandl” on page 3-191. For the vectorial version see the function “vrandv” on page 3-193. All the 3 functions uses the same 2 vectorials SEED variables. This variables are updated by the 3 functions coherently in order to generate random non-correlated sub-sequences independently from the order of usage of the different functions. The initial 2 values of the SEED variables: SEED1 and SEED2, have been choosen in order to compute 4 independent pseudorandom values at each algorithm execution and to maintain the maximum repetition period (must be 2^32):

$$\begin{cases} SEED1 \qquad (SEED_0, SEED_{2^{32}/4}) \\ SEED2 \qquad (SEED_{2^{32}/2}, SEED_{3 \times 2^{32}/4}) \\ with \qquad SEED_0 = 0 \end{cases}$$

They are stored in Internal Memory at the address of the ***LABEL ___ATMlib__SEED.*** The call to the randl or the randr function can be mixed with the call to the vrand function still generating a maximum length pseudorandom sequence. For this reason the vrandl and vrandr functions are built with unroll 4 while the vrand functios is built with unroll 2. The real and the imaginary part of two pseudorandom vectorials numbers generated at each iteration of the algorithm, are arranged in right memory in a float array.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Y must be in right memory

Number of cycles:

41 + 2.25 × Nelements

Number of VLIW:

41

File: vrandr.mas

3.196 vrandv

Function: vectorial float array random numbers generator

$$\begin{cases} Y(k) = SEED_k \times Norm + Offset & k = 0 \ldots Nelements \\ SEED_k = (A \times SEED_{k-1} + C) Mod 2^{32} & A = 69069 \qquad C = 1 \end{cases}$$

Synopsis: vrandv(*Y, strideY, Norm, Offset, Nelements)

Include file: DSPlib.h.

Y**: pointer to the output vector. ***Type: __vector__ float*

strideY: stride to be applied on output vector. ***Type: int***

Norm: normalization factor. Norm must be equal to 2^(-32) if a random number with module in the range [0, 1) is needed. ***Type: __vector__ float***

Offset: is the offset to be added. ***Type: __vector__ float***

Nelements: number of elements to be computed. ***Type:int***

The function vrandv generates a vectorial float array of random numbers using a linear congruential method, described above, multiplies for a vectorial float normalization factor and adds a vectorial float offset. For the float version , see the functions: “vrandl” on page 3-191 and “vrandr” on page 3-192. All the 3 functions uses the same 2 vectorials SEED variables. This variables are updated by the 3 functions coherently in order to generate random non-correlated subsequences independently from the order of usage of the different functions. The initial 2 values of the SEED variables: SEED1 and SEED2, have been choosen in order to compute 4 independent pseudorandom values at each algorithm execution and to maintain the maximum repetition period (must be 2^32):

$$\begin{cases} SEED1 & (SEED_0, SEED_{2^{32}/4}) \\ SEED2 & (SEED_{2^{32}/2}, SEED_{3 \times 2^{32}/4}) \\ with & SEED_0 = 0 \end{cases}$$

They are stored in Internal Memory at the address of the ***LABEL ___ATMlib__SEED.*** The call to the randl or the randr function can be mixed with the call to the vrand function still generating a maximum length pseudorandom sequence. For this reason the vrandl and vrandr functions are built with unroll 4 while the vrand functios is built with unroll 2.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be greater or equal to 6 and multiple of 2

Number of cycles:

35 + 4.5 × Nelements

Number of VLIW:

37

File: vrandv.mas

3.197 vrmvesqll

Function: root mean square stored in left memory of an input array stored in left memory

$$Y = \sqrt{\frac{\sum_{k=0}^{Nelements-1} (X(k))^2}{Nelements}}$$

Synopsis: __vector__ int vrmvesqll (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output scalar into which the computed value is written.Type: float****

Nelements: number of elements to be computed.***Type: int***

The function vrmvesqll computes the root mean square of an input array stored in left memory space and writes the computed value to an output location in left memory space.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in left memory

Y must be in left memory

Number of cycles:

104 + 1 × Nelements

Number of VLIW:

46

File: vrmvesqll.mas

3.198 vrmvesqlr

Function: root mean square stored in right memory of an input array stored in left memory

$$Y = \sqrt{\frac{\sum_{k=0}^{Nelements-1} (X(k))^2}{Nelements}}$$

Synopsis: __vector__ int vrmvesqlr (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output scalar into which the computed value is written. ***Type: float*

Nelements**: number of elements to be computed.Type: int***

The function vrmvesqlr computes the root mean square of an input array stored in left memory space and writes the computed value to an output location in right memory space.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in left memory

Y must be in right memory

Number of cycles:

104 + 1 × Nelements

Number of VLIW:

46

File: vrmvesqlr.mas

3.199 vrmvesqrl

Function: root mean square stored in left memory of an input array stored in right memory

$$Y = \sqrt{\frac{\sum_{k=0}^{Nelements-1} (X(k))^2}{Nelements}}$$

Synopsis: __vector__ int vrmvesqrl (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output scalar which the computed value is written. ***Type: float*

Nelements: number of elements to be computed.***Type: int***

The function vrmvesqrl computes the root mean square of an input array stored in right memory space and writes the computed value to an output location in left memory space.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in right memory

Y must be in left memory

Number of cycles:

104 + 1 × Nelements

Number of VLIW:

47

File: vrmvesqrl.mas

3.200 vrmvesqrr

Function: root mean square stored in left memory of an input array stored in right memory

$$Y = \sqrt{\frac{\sum_{k=0}^{Nelements-1} (X(k))^2}{Nelements}}$$

Synopsis: __vector__ int vrmvesqrr (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output scalar into which the computed value is written. ***Type: float*

Nelements: number of elements to be computed.***Type: int***

The function vrmvesqrr computes the root mean square of an input array stored in right memory space and writes the computed value to an output location in right memory space.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

X must be in right memory

Y must be in right memory

Number of cycles:

105 + 1 × Nelements

Number of VLIW:

47

File: vrmvesqrr.mas

3.201 vrmvesqv

Function: root mean square of a vectorial input array

$$Y = \sqrt{\frac{\sum_{k=0}^{Nelements-1} (X(k))^2}{Nelements}}$$

Synopsis: __vector__ int vrmvesqv (*X, strideX, *Y, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output scalar in the vector space into which the computed value is written. ***Type: __vector__ float*

Nelements: Number of elements to be computed.***Type: int***

The function vrmvesqv computes the root mean square of the input arrays stored in vectors space and writes the computed value to the output locations in vector space. For computing the root mean square, with the input stored in left/right memory space and to output the values into left/right memory space, see functions: “vrmvesqll” on page 3-195, “vrmvesqlr” on page 3-196, “vrmvesqrl” on page 3-197 and “vrmvesqrr” on page 3-198.

Restrictions:

Nelements must be greater or equal to 8 and multiple of 4

Number of cycles:

109 + 1× Nelements

Number of VLIW:

47

File: vrmvesqv.mas

3.202 vrotate32v

Function: vectorial integer left or right shift mod.32 with number of shifts (0 to 31)

Synopsis: __vector__ int vrotate32v(*X, strideX, *Y, strideY, LShift, RShif, Nelements)

$$Y(k) = vshift(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ int*

strideX**: stride to be applied on input vector. ***Type: int

Y**: pointer to the output vector. ***Type: __vector__ int*

strideY: stride to be applied on output vector. ***Type: int***

LShift: number of the shifts for the real part of the vector. ***Type: int***

RShift: number of the shifts for the imaginary part of the vector. ***Type: int***

Nelements: number of elements to be computed. ***Type:int***

The function vrotate32v performs a left or right shift mod.32 of the integer vector X. The number of shifts is respectively equal to LShift for the real part and RShift for the imaginary part of X. LShift and RShift can be positive o negative. If they are positive the function performs a left shift otherwise a right shift.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

47 + 1 × Nelements

Number of VLIW:

31

File: vrotate32v.mas

3.203 vshandv

Function: vectorial integer left or right shift with number of shifts (0 to 31) and logical AND

$$Y(k) = vshand(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vshandv(*X, strideX, *Y, strideY, LShift, RShift, LMask, RMask, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ int*

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __vector__ int*

strideY: stride to be applied on output vector. ***Type: int***

LShift: number of the shifts for the real part of the vector. ***Type: int***

RShift: number of the shifts for the imaginary part of the vector. ***Type: int***

LMask: mask for the logical AND of the real part of the vector. ***Type: int***

RMask: mask for the logical AND of the imaginary part of the vector. ***Type:int***

Nelements: number of elements to be computed. ***Type:int***

The function vshandv performs a left or right shift and a logical AND of the integer vector X. The number of shifts and the mask for the logical AND are respectively equal to LShift and LMask for the real part and RShift and RMask for the imaginary part of X. LShift and RShift can be positive o negative. If they are positive the function performs a left shift otherwise a right shift.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

57 + 1 × Nelements

Number of VLIW:

33

File: vshandv.mas

3.204 vshiftv

Function: vectorial integer left or right shift with number of shifts (0 to 31)

$$Y(k) = vshift(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vshiftv(*X, strideX, *Y, strideY, LShift, RShift,LMask, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ int*

strideX: stride to be applied on input vector. ***Type: int***

Y**: pointer to the output vector. ***Type: __vector__ int*

strideY: stride to be applied on output vector. ***Type: int***

LShift: number of the shifts for the real part of the vector. ***Type: int***

RShift: number of the shifts for the imaginary part of the vector. ***Type: int***

Nelements: number of elements to be computed. ***Type:int***

The function vshiftv performs a left or right shift of the integer vector X. The number of shifts is respectively equal to LShift for the real part and RShift for the imaginary part of X. LShift and RShift can be positive o negative. If they are positive the function performs a left shift otherwise a right shift.

Restrictions:

Nelements must be greater or equal to 12 and multiple of 4

Number of cycles:

44 + 1 × Nelements

Number of VLIW:

30

File: vshiftv.mas

3.205 vsinhll

Function: hyperbolic sine of a float input array and left to left move

$$Y(k) = \boldsymbol{sinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinhll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinhll computes the hyperbolic sine of an input array stored in left memory space and writes the output to an array in left memory space.

Note: the function vsinhll uses 3 locations of the stack

Precision:

see Table 3-13 on page 207

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

| x | <= 87, to avoid overflow / underflow of the computed result

X must be in left memory

Y must be in left memory

Number of cycles:

307 + 19 × Nelements

Number of VLIW:

164

File: vsinhll.mas, vexpll.mas, expCoeff.mas

3.206 vsinhlr

Function: hyperbolic sine of a float input array and left to right move

$$Y(k) = \boldsymbol{sinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinhlr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

****X***: pointer to the input array. ***Type: float****

strideX: stride to be used for the input array. ***Type: int***

****Y***: pointer to the output array into which the computed value is written. Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinhlr computes the hyperbolic sine of an input array stored in left memory space and writes the output to an array in right memory space.

Note: the function vsinhlr uses 3 locations of the stack

Precision:

see Table 3-13 on page 207

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

| x | <= 87, to avoid overflow / underflow of the computed result

X must be in left memory

Y must be in right memory

Number of cycles:

303 + 18.5 × Nelements

Number of VLIW:

161

File: vsinhlr.mas, vexplr.mas. expCoeff.mas

3.207 vsinhrl

Function: hyperbolic sine of a float input array and right memory to left move

$$Y(k) = \boldsymbol{sinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinhrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinhrl computes the hyperbolic sine of an input array stored in right memory space and writes the output to an array in left memory space.

Note: the function vsinhrl uses 3 locations of the stack

Precision:

see Table 3-13 on page 207

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

| x | <= 87, to avoid overflow / underflow of the computed result

X must be in right memory

Y must be in left memory

Number of cycles:

304 + 19 × Nelements

Number of VLIW:

165

File: vsinhrl.mas, vexprl.mas, expCoeff.mas

3.208 vsinhrr

Function: hyperbolic sine of a float input array and right to right move

$$Y(k) = \boldsymbol{sinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinhrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinhrr computes the hyperbolic sine of an input array stored in right memory space and writes the output to an array in right memory space.

Note: the function vsinhrr uses 3 locations of the stack

Precision:

see Table 3-13 on page 207

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

| x | <= 87, to avoid overflow / underflow of the computed result
X must be in right memory
Y must be in right memory

Number of cycles:
306 + 18.5 × Nelements

Number of VLIW:
161

File: vsinhrr.mas, vexprr.mas, expCoeff.mas

3.209 vsinhv

Function: hyperbolic sine of a vectorial input array

$$Y(k) = \boldsymbol{sinh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinhv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinhv computes the hyperbolic sine of an input array stored in vector space and writes the output to an array in vector space. For computing the hyperbolic sine, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: Table 3.205 on page 202, Table 3.206 on page 203, Table 3.207 on page 204 and Table 3.208 on page 205.

Note: the function vsinhv uses 3 locations of the stack

Precision:

the following table provides the information about the precision for this function

Table 3-13.

Range of input values	Absolute error	Relative error
-0.1505 to 0.1505	4.58297e-010	7.06714e-008
0 to 10	1.45701e-005	9.58188e-010
10 to 86	1.32643e+027	5.28016e-010
-10 to 0	1.54298e-005	4.09391e-010
-86 to -10	1.32643e+027	5.28016e-010

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

313 + 31 × Nelements

Number of VLIW:

167

File: vsinhv.mas, vexpv.mas, expCoeff.mas

3.210 vsinll

Function: sine of a float input array and left to left move

$$Y(k) = \boldsymbol{sin}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

****X:*** pointer to the input array. ***Type: float****

strideX: stride to be used for the input array. ***Type: int***

****Y:*** pointer to the output array into which the computed value is written.***Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinll computes the sine of an input array stored in left memory space and writes the output to an array in left memory space.

Precision:

see Table 3-14 on page 211

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

| x | <= 10^10, to avoid overflow / underflow of the computed result

X must be in left memory

Y must be in left memory

Number of cycles:

117 + 11.25 × Nelements

Number of VLIW:

63

File: vsinll.mas, sinCoeff.mas

3.211 vsinlr

Function: sine of a float input array and left to right move

$$Y(k) = \boldsymbol{sin}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinlr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X:** pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y:** pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinlr computes the sine of an input array stored in left memory space and writes the output to an array in right memory space.

Precision:

see Table 3-14 on page 211

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4
| x | <= 10^10, to avoid overflow / underflow of the computed result
X must be in left memory
Y must be in right memory

Number of cycles:

117 + 11.25 × Nelements

Number of VLIW:

63

File: vsinlr.mas, sinCoeff.mas

3.212 vsinrl

Function: sine of a float input array and right to left move

$$Y(k) = \boldsymbol{sin}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

*X: pointer to the input array. ***Type: float****

strideX: stride to be used for the input array. ***Type: int***

****Y:*** pointer to the output array into which the computed value is written. ***Type: float****

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinrl computes the sine of an input array stored in right memory space and writes the output to an array in left memory space.

Precision:

see Table 3-14 on page 211

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

| x | <= 10^10, to avoid overflow / underflow of the computed result

X must be in left memory

Y must be in right memory

Number of cycles:

119 + 11.25 × Nelements

Number of VLIW:

64

File: vsinrl.mas, sinCoeff.mas

3.213 vsinrr

Function: sine of a float input array and right to right move

$$Y(k) = \textbf{sin}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X:** pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

****Y:*** pointer to the output array into which the computed value is written. Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinrr computes the sine of an input array stored in right memory space and writes the output to an array in right memory space.

Precision:

see Table 3-14 on page 211

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

| x | <= 10^10, to avoid overflow / underflow of the computed result

X must be in right memory

Y must be in right memory

Number of cycles:

118 + 11.25 × Nelements

Number of VLIW:

64

File: vsinrr.mas, sinCoeff.mas

3.214 vsinv

Function: sine of a vectorial input array

$$Y(k) = \boldsymbol{sin}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsinv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X:** pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y:** pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vsinv computes the sine of an input array stored in vector space and writes the output to an array in vector space. For computing the sine, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: Table 3.210 on page 207, Table 3.211 on page 208, Table 3.212 on page 209 and Table 3.213 on page 210.

Precision:

the following table provides the information about the precision for this function

Table 3-14.

Description of input values	Absolute error	Relative error
0 to $\pi/3$	6.16753e-010	1.84526e-009
$-\pi$ *to* π	5.45383e-009	0.559979
2π, 6π	5.45383e-009	1.92443
-2π, -6π	5.45383e-009	1.92443

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

109 + 21.5 × Nelements

Number of VLIW:

58

File: vsinv.mas, sinCoeff.mas

3.215 vsqrt0ll

Function: single vector square root computation and left to left move

$$Y(k) = \sqrt{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vsqrt0ll(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the X input vector. ***Type: float *

strideX: stride to be applied on X vector. ***Type: int***

Y**: pointer to the Y output vector. ***Type: float *

strideY: stride to be applied on Y vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The function vsqrt0ll performs the square root of the input data vector X ordered as specified in Restrictions. X is a float array.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be any number greater or equal to 1

X must be in the left memory

Y must be in the left memory

Precision: 31 bit of mantissa

Number of cycles:

118 + 22 × Nelements

Number of VLIW:

55

File: vsqrt0ll.mas

3.216 vsqrt0lr

Function: single vector square root computation and left to right move

$$Y(k) = \sqrt{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vsqrt0lr(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the X input vector. ***Type: float *

strideX**: stride to be applied on X vector. ***Type: int

Y**: pointer to the Y output vector. ***Type: float *

strideY**: stride to be applied on Y vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The function vsqrt0lr performs the square root of the input data vector X ordered as specified in Restrictions. X is a float array.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be any number greater or equal to 1

X must be in the left memory

Y must be in the right memory

Precision: 31 bit of mantissa

Number of cycles:

118 + 22 × Nelements

Number of VLIW:

55

File: vsqrt0lr.mas

3.217 vsqrt0rl

Function: single vector square root computation and right to left move

$$Y(k) = \sqrt{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vsqrt0rl(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the X input vector. ***Type: float *

strideX: stride to be applied on X vector. ***Type: int***

Y**: pointer to the Y output vector. ***Type: float *

strideY: stride to be applied on Y vector. ***Type: int***

Nelements: number elements to be computed. ***Type: int***

The function vsqrt0rl performs the square root of the input data vector X ordered as specified in Restrictions. X is a float array.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be any number greater or equal to 1

X must be in the right memory

Y must be in the left memory

Precision: 31 bit of mantissa

Number of cycles:

118 + 22 × Nelements

Number of VLIW:

55

File: vsqrt0rl.mas

3.218 vsqrt0rr

Function: single vector square root computation and right to right move

$$Y(k) = \sqrt{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vsqrt0rr(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the X input vector. ***Type: float *

strideX**: stride to be applied on X vector. ***Type: int

Y**: pointer to the Y output vector. ***Type: float *

strideY**: stride to be applied on Y vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The function vsqrt0rr performs the square root of the input data vector X ordered as specified in Restrictions. X is a float array.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be any number greater or equal to 1

X must be in the right memory

Y must be in the right memory

Precision: 31 bit of mantissa

Number of cycles:

118 + 22 × Nelements

Number of VLIW:

55

File: vsqrt0rr.mas

3.219 vsqrt0v

Function: vectorial square root computation

$$Y(k) = \sqrt{X(k)} \qquad K = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsqrt0v(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __vector__ float *

strideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the output vector. ***Type: __vector__ float *

strideY: stride to be used for the Y data. ***Type: int***

Nelements: Number of elements to be computed. ***Type: int***

The function vsqrt0v performs the square root of the input data vector X . X is a vectorial data type. The operation are performed in vectorial mode i.e. pair of results are computed simultaneously:

YLeft = sqrt (XLeft)

YRight = sqrt (XRight)

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements can be any number greater than 0

Result precision: 31 bits of mantissa

Number of cycles:

118 + 22 × Nelements

Number of VLIW:

55

File: vsqrt0v.mas

3.220 vsqrtll

Function: pipelined single vector square root computation and left to left move

$$Y(k) = \sqrt{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vsqrtll(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the X input vector. ***Type: float *

strideX**: stride to be applied on X vector. ***Type: int

Y**: pointer to the Y output vector. ***Type: float *

strideY**: stride to be applied on Y vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The function vsqrtll performs the single square root of the input data vector X ordered as specified in Restrictions. X is a float array, but after its moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. For a not pipelined version see the function “vsqrt0ll” on page 3-212. For a vectorial version see the function “vsqrtv” on page 3-221.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in the left memory

Y must be in the left memory

Precision: 31 bit of mantissa

Number of cycles:

130 + 7.75 × Nelements

Number of VLIW:

74

File: vsqrtll.mas

3.221 vsqrtlr

Function: pipelined single vector square root computation and left to right move

$$Y(k) = \sqrt{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vsqrtlr(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the X input vector. ***Type: float *

strideX**: stride to be applied on X vector. ***Type: int

Y**: pointer to the Y output vector. ***Type: float *

strideY**: stride to be applied on Y vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The function vsqrtlr performs the square root of the input data vector X ordered as specified in Restrictions. X is a float array, but after its moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. For a not pipelined version see the function “vsqrt0lr” on page 3-213. For a vectorial version see the function “vsqrtv” on page 3-221.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in the left memory

Y must be in the right memory

Precision: 31 bit of mantissa

Number of cycles:

130 + 7.75 × Nelements

Number of VLIW:

74

File: vsqrtlr.mas

3.222 vsqrtrl

Function: pipelined single vector square root computation and right to left move

$$Y(k) = \sqrt{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vsqrtrl(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the X input vector. ***Type: float *

strideX**: stride to be applied on X vector. ***Type: int

Y**: pointer to the Y output vector. ***Type: float *

strideY**: stride to be applied on Y vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The function vsqrtrl performs the square root of the input data vector X ordered as specified in Restrictions. X is a float array, but after its moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. For a not pipelined version see the function “vsqrt0rl” on page 3-214. For a vectorial version see the “vsqrtv” on page 3-221.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before

RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in the right memory

Y must be in the left memory

Precision: 31 bit of mantissa

Number of cycles:

122 + 7.75 × Nelements

Number of VLIW:

74

File: vsqrtrl.mas

3.223 vsqrtrr

Function: pipelined single vector square root computation and right to right move

$$Y(k) = \sqrt{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vsqrtrr(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to the X input vector. ***Type: float *

strideX**: stride to be applied on X vector. ***Type: int

Y**: pointer to the Y output vector. ***Type: float *

strideY**: stride to be applied on Y vector. ***Type: int

Nelements**: number elements to be computed. ***Type: int

The function vsqrtrr performs the square root of the input data vector X ordered as specified in Restrictions. X is a float array, but after its moving from the Data Memory to the Register File, data are arranged in a vectorial way in order to perform vectorial operations. For a not pipelined version see the function “vsqrt0rr” on page 3-215. For a vectorial version see the function “vsqrtv” on page 3-221.

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

X must be in the right memory

Y must be in the right memory

Precision: 31 bit of mantissa

Number of cycles:

122 + 7.75 × Nelements

Number of VLIW:

74

File: vsqrtrr.mas

3.224 vsqrtv

Function: pipelined vectorial square root computation

$$Y(k) = \sqrt{X(k)} \qquad k = 0 \ldots Nelements$$

Synopsis: __vector__ int vsqrtv(*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h.

X**: pointer to input vector. ***Type: __vector__ float *

strideX**: stride to be used for the X data. ***Type: int

Y**: pointer to the output vector. ***Type: __vector__ float *

strideY**: stride to be used for the X the data. ***Type: int

Nelements**: Number of elements to be computed . ***Type: int

The function vsqrtv works on vectorial data. It performs the operations in vectorial mode i.e. pair of results are computed simultaneously in a code unrolled 4 times:

Y1Left = sqrt (X1Left) and Y1Right = sqrt (X1Right)

Y2Left = sqrt (X2Left) and Y2Right = sqrt (X2Right)

Y3Left = sqrt (X3Left) and Y3Right = sqrt (X3Right)

Y4Left = sqrt (X4Left) and Y4Right = sqrt (X4Right)

Note: to use this function correctly, some numerical exceptions must be masked. This can be done including the following instruction: MaarGSR_BASE->GSR_mask=0x7, in the ARM source C before RUNMAGIC. For more details on the Exception Mask Registers (GSR_mask) refer to the DIOPSIS 740 Data Sheet (doc7001.pdf).

Restrictions:

Nelements must be greater or equal to 2 and multiple of 2

Result precision: 31 bits of mantissa

Number of cycles:

115 + 15.5 × Nelements

Number of VLIW:

66

File: vsqrtv.mas

3.225 vsubll

Function: subtraction of 2 float array in left memory

$$Z(k) = X(k) - Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsubll(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: __vector__ float *

strideX**: stride to be used for the X data. ***Type: int

Y**: pointer to the second input vector. ***Type: __vector__ float *

strideY**: stride to be used for the Y the data. ***Type: int

Z**: pointer to the output vector. ***Type: __vector__ float *

strideZ**: stride to be used for the Y the data. ***Type: int

Nelements**: Number of element to be computed. ***Type: int

The function vsubll executes the difference between 2 float array in left memory: X and Y and writes the result in the float array Z stored in left memory.

Restrictions:

Nelements must be greater than 4 and multiple of 4

Number of cycles:

27 + 2 × Nelements

Number of VLIW:

22

File: vsubll.mas

3.226 vsubrr

Function: subtraction of 2 float array in right memory

$$Z(k) = X(k) - Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsubrr(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X: pointer to the first input vector. ***Type: __vector__ float ****

strideX: stride to be used for the X data. ***Type: int***

Y: pointer to the second input vector. ***Type: __vector__ float ****

strideY: stride to be used for the Y the data. ***Type: int***

Z: pointer to the output vector. ***Type: __vector__ float ****

strideZ: stride to be used for the Y the data. ***Type: int***

Nelements: Number of element to be computed. ***Type: int***

The function vsubrr executes the difference between 2 float array in right memory: X and Y and writes the result in the float array Z stored in right memory.

Restrictions:

Nelements must be greater than 4 and multiple of 4

Number of cycles:

32 + 2 × Nelements

Number of VLIW:

20

File: vsubrr.mas

3.227 vsubv

Function: subtraction of 2 vectorial float array

$$Z(k) = X(k) - Y(k) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vsubv(*X, strideX, *Y, strideY, *Z, strideZ, Nelements)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: __vector__ float *

strideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the second input vector. ***Type: __vector__ float *

strideY: stride to be used for the Y the data. ***Type: int***

Z**: pointer to the output vector. ***Type: __vector__ float *

strideZ: stride to be used for the Y the data. ***Type: int***

Nelements: Number of element to be computed. ***Type: int***

The function vsubv works on complex data arranged vectorially in memory; they can represent pair of complex vectors or two vectorial streams of real vectors that will be processed in parallel.

Restrictions:

Nelements must be greater or equal to 4 and multiple of 4

Number of cycles:

29 + 2.75 × Nelements

Number of VLIW:

24

File: vsubv.mas

3.228 vsumv

Function: sum of vector elements

$$Y(k) = \sum_{k=0}^{Nelements-1} X(k) \qquad k = 0 \ldots Nelements-1$$

Synopsis: __vector__ int vsumv(*X, strideX, *Y, Nelements)

Include file: DSPlib.h.

X**: pointer to the input vector. ***Type: __complex__ float *

strideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the sum memory location. ***Type: __ complex __ float *

Nelements: Number of element to be added. ***Type: int***

The function vsumv works on complex (or vectorial) data type returning the sum of the real (left) parts in the real (left) output location and the sum of the imaginary (right) parts in the imaginary (right) output location.

Restrictions:

Nelements must be greater than 8 and multiple of 4

Number of cycles:

44 + 1 × Nelements

Number of VLIW:

27

File: vsumv.mas

3.229 vtanhll

Function: hyperbolic tan of a float input array and left to left move

$$Y(k) = \boldsymbol{tanh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanhll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanhll computes the hyperbolic tan of an input array stored in left memory space and writes the output to an array in left memory space.

Note: the function vtanhll uses 3 locations of the stack

Precision:

see Table 3-15 on page 230

Restrictions:

Nelements must be greater than 4 and multiple of 4.

X must be in left memory

Y must be in left memory

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

309 +19.75 × Nelements

Number of VLIW:

165

File: vtanhll.mas, vexpll.mas, expCoeff.mas

3.230 vtanhlr

Function: hyperbolic tan of a float input array and left to right move

$$Y(k) = \boldsymbol{tanh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanhlr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX**: stride to be used for the input array. ***Type: int

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanhlr computes the hyperbolic tan of an input array stored in left memory space and writes the output to an array in right memory space.

Precision:

see Table 3-15 on page 230

Restrictions:

Nelements must be greater than 4 and multiple of 4.

X must be in left memory

Y must be in right memory

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

304 + 18.75 × Nelements

Number of VLIW:

161

File: vtanhlr.mas, vexplr.mas, expCoeff.mas

3.231 vtanhrl

Function: hyperbolic tan of a float input array and right to left move

$$Y(k) = \boldsymbol{tanh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanhrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanhrl computes the hyperbolic tan of an input array stored in right memory space and writes the output to an array in left memory space.

Note: the function vtanhrl uses 3 locations of the stack

Precision:

see Table 3-15 on page 230

Restrictions:

Nelements must be greater than 4 and multiple of 4.

X must be in right memory

Y must be in left memory

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

302 + 18.75 × Nelements

Number of VLIW:

165

File: vtanhrl.mas, vexprl.mas, expCoeff.mas

3.232 vtanhrr

Function: hyperbolic tan of a float input array and right to right move

$$Y(k) = \textbf{tanh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanhrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanhrr computes the hyperbolic tan of an input array stored in right memory space and writes the output to an array in right memory space.

Note: the function vtanhrr uses 3 locations of the stack

Precision:

see Table 3-15 on page 230

Restrictions:

Nelements must be greater than 4 and multiple of 4

X must be in right memory

Y must be in right memory

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

308 + 19 × Nelements

Number of VLIW:

162

File: vtanhrr.mas, vexprr.mas, expCoeff.mas

3.233 vtanhv

Function: hyperbolic tan of a vectorial input array

$$Y(k) = \boldsymbol{tanh}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanhv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanhv computes the hyperbolic tan of an input array stored in vector space and writes the output to an array in vector space. For computing the hyperbolic tan, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: Table 3.229 on page 226, Table 3.230 on page 227, Table 3.231 on page 228 and Table 3.232 on page 229.

Note: the function vtanhv uses 3 locations of the stack

Precision:

the following table provides the information about the precision for this function

Table 3-15.

Range of input values	Absolute error	Relative error
-0.1505 to 0.1505	5.13845e-010	8.45721e-008
0 to 10	5.29676e-010	5.29676e-010
10 to 90	1.38289e-012	1.38289e-012
-10 to 0	2.50796e-010	2.50799e-010

Restrictions:

Nelements must be greater than 2 and multiple of 2

| x | <= 87, to avoid overflow / underflow of the computed result

Number of cycles:

325 + 30 × Nelements

Number of VLIW:

178

File: vtanhv.mas, vexpv.mas, expCoeff.mas

3.234 vtanll

Function: tan of a float input array and left to left move

$$Y(k) = \boldsymbol{tan}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanll (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written. Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanll computes the tan of an input array stored in left memory space and writes the output to an array in left memory space.

Precision:

see Table 3-16 on page 235

Restrictions:

Nelements must be greater than 4 and multiple of 4.

X must be in left memory

Y must be in left memory

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

142 + 18 × Nelements

Number of VLIW:

79

File: vtanll.mas, tanCoeff.mas

3.235 vtanlr

Function: tan of a float input array and left to right move

$$Y(k) = \boldsymbol{tan}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanlr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written. Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanlr computes the tan of an input array stored in left memory space and writes the output to an array in right memory space.

Precision:

see Table 3-16 on page 235

Restrictions:

Nelements must be greater than 4 and multiple of 4.

X must be in left memory

Y must be in right memory

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

140 + 17.5 × Nelements

Number of VLIW:

79

File: vtanlr.mas, tanCoeff.mas

3.236 vtanrl

Function: tan of a float input array and right to left move

$$Y(k) = \boldsymbol{tan}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanrl (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written. Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanrl computes the tan of an input array stored in right memory space and writes the output to an array in left memory space.

Precision:

see Table 3-16 on page 235

Restrictions:

Nelements must be greater than 4 and multiple of 4.

X must be in right memory

Y must be in left memory

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

141+ 17.5 × Nelements

Number of VLIW:

79

File: vtanrl.mas, tanCoeff.mas

3.237 vtanrr

Function: tan of a float input array and right to right memory

$$Y(k) = \boldsymbol{tan}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanrr (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: float*

strideX: stride to be used for the input array. ***Type: int***

***Y**: pointer to the output array into which the computed value is written. Type: float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanrr computes the tan of an input array stored in right memory space and writes the output to an array in right memory space.

Precision:

see Table 3-16 on page 235

Restrictions:

Nelements must be greater than 4 and multiple of 4.

X must be in right memory

Y must be in right memory

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

143 + 18 × Nelements

Number of VLIW:

74

File: vtanrr.mas, tanCoeff.mas

3.238 vtanv

Function: tan of a vectorial input array

$$Y(k) = \boldsymbol{tan}(X(k)) \qquad k = 0 \ldots Nelements - 1$$

Synopsis: __vector__ int vtanv (*X, strideX, *Y, strideY, Nelements)

Include file: DSPlib.h

X**: pointer to the input array. ***Type: __vector__ float*

strideX: stride to be used for the input array. ***Type: int***

Y**: pointer to the output array into which the computed value is written. ***Type: __vector__ float*

strideY: stride to be used for the output array. ***Type: int***

Nelements: number of elements to be computed.***Type: int***

The function vtanv computes the tan of an input array stored in vector space and writes the output to an array in vector space. For computing the tan, with the input stored in left/right memory space and to output the values into left/right memory space, see the functions: Table 3.234 on page 231, Table 3.235 on page 232, Table 3.236 on page 233 and Table 3.237 on page 234.

Precision:

the following table provides the information about the precision for this function

Table 3-16.

Description of input values	Absolute error	Relative error
0 to $\pi/3$	2.7567e-009	1.64263e-009
-$-\pi$ to π except $-\pi/2$, $\pi/2$	2.79771e-007	0.504265
-1.5708	1.01281e+008	0.324616
1.5708	1.01281e+008	0.324616

Restrictions:

Nelements must be greater than 2 and multiple of 2

| x | <= 10^10, to avoid overflow / underflow of the computed result

Number of cycles:

134 + 34.5 × Nelements

Number of VLIW:

74

File: vtanv.mas, tanCoeff.mas

3.239 xcorrc

Function: cross-correlation between 2 complex float array or auto-correlation of a complex float array

$$R_{XY}(i) = \begin{cases} \sum_{k=0}^{N-i-1} X(k-i) \times Y^*(k) & i = 0 \ldots \frac{Ncorr}{2} \\ R^*_{YX}(-i) & i = -\frac{Ncorr}{2} \ldots -1 \end{cases}$$

$$Z(i) = R_{XY}\left(i - \frac{Ncorr}{2}\right) \qquad i = 1 \ldots Ncorr$$

Synopsis: __vector__ int xcorrc (*X, strideX, *Y, strideY, *Z, strideZ, N, NCorr)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: __complex__ float *

StrideX**: stride to be used for the X data. ***Type: int

Y**: pointer to the second input vector. ***Type: __complex__ float *

StrideY**: stride to be used for the Y data. ***Type: int

Z**: pointer to the output vector. ***Type: __complex__ float *

StrideZ**: stride to be used for the Z data. ***Type: int

N**: lenght of the input vectors. If X and Y aren't of the same length, N must be set to the length of the shorter vector. ***Type: int

NCorr**: number of coefficients to be computed. ***Type: int

The function xcorrc can perform the cross-correlation of 2 complex float array : X and Y or the autocorrelation of the complex float array X. In the second case the third parameter passed to the function must be equal to the first.

Restrictions:

NCorr must be greater than 4 and multiple of 4

Number of cycles:

80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))

Number of VLIW:

94

File: xcorrc.mas

3.240 xcorrlll

Function: cross-correlation between 2 float array stored in left memory or auto-correlation of a float array stored in left memory. The result is stored in left memory

$$R_{XY}(i) = \begin{cases} \sum_{k=0}^{N-i-1} X(k-i) \times Y(k) & i = 0 \ldots \frac{Ncorr}{2} \\ R_{YX}(-i) & i = -\frac{Ncorr}{2} \ldots -1 \end{cases}$$

$$Z(i) = R_{XY}\left(i - \frac{Ncorr}{2}\right) \qquad i = 1 \ldots Ncorr$$

Synopsis: __vector__ int xcorrlll (*X, strideX, *Y, strideY, *Z, strideZ, N, NCorr)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float *

StrideY: stride to be used for the Y data. ***Type: int***

Z**: pointer to the output vector. ***Type: float *

StrideZ: stride to be used for the Z data. ***Type: int***

N: lenght of the input vectors. If X and Y aren't of the same length, N must be set to the length of the shorter vector. ***Type: int***

NCorr: number of coefficients to be computed. ***Type: int***

The function xcorrlll can perform the cross-correlation of 2 real float array : X and Y or the autocorrelation of the float array X. In the second case the third parameter passed to the function must be equal to the first. The input and output vectors must be stored in left memory.

Restrictions:

NCorr must be greater than 4 and multiple of 4

X must be in left memory

Y must be in left memory

Z must be in left memory

Number of cycles:

80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))

Number of VLIW:

94

File: xcorrlll.mas

3.241 xcorrllr

Function: cross-correlation between 2 float array stored in left memory or auto-correlation of a float array stored in left memory. The result is stored in right memory

$$R_{XY}(i) = \begin{cases} \sum_{k=0}^{N-i-1} X(k-i) \times Y(k) & i = 0 \ldots \frac{Ncorr}{2} \\ R_{YX}(-i) & i = -\frac{Ncorr}{2} \ldots -1 \end{cases}$$

$$Z(i) = R_{XY}\left(i - \frac{Ncorr}{2}\right) \qquad i = 1 \ldots Ncorr$$

Synopsis: __vector__ int xcorrllr (*X, strideX, *Y, strideY, *Z, strideZ, N, NCorr)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float *

StrideY: stride to be used for the Y data. ***Type: int***

Z**: pointer to the output vector. ***Type: float *

StrideZ: stride to be used for the Z data. ***Type: int***

N: lenght of the input vectors. If X and Y aren't of the same length, N must be set to the length of the shorter vector. ***Type: int***

NCorr: number of coefficients to be computed. ***Type: int***

The function xcorrllr can perform the cross-correlation of 2 real float array : X and Y or the autocorrelation of the float array X. In the second case the third parameter passed to the function must be equal to the first. The input vectors must be stored in left memory while the output in right memory.

Restrictions:

NCorr must be greater than 4 and multiple of 4

X must be in left memory

Y must be in left memory

Z must be in right memory

Number of cycles:

80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))

Number of VLIW:

94

File: xcorrllr.mas

3.242 xcorrlrl

Function: cross-correlation between 2 float array: the first stored in left memory and the second in right memory . The result is stored in left memory

$$R_{XY}(i) = \begin{cases} \sum_{k=0}^{N-i-1} X(k-i) \times Y(k) & i = 0 \ldots \frac{Ncorr}{2} \\ R_{YX}(-i) & i = -\frac{Ncorr}{2} \ldots -1 \end{cases}$$

$$Z(i) = R_{XY}\left(i - \frac{Ncorr}{2}\right) \qquad i = 1 \ldots Ncorr$$

Synopsis: __vector__ int xcorrlrl (*X, strideX, *Y, strideY, *Z, strideZ, N, NCorr)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

StrideX**: stride to be used for the X data. ***Type: int

Y**: pointer to the second input vector. ***Type: float *

StrideY**: stride to be used for the Y data. ***Type: int

Z**: pointer to the output vector. ***Type: float *

StrideZ**: stride to be used for the Z data. ***Type: int

N**: lenght of the input vectors. If X and Y aren't of the same length, N must be set to the length of the shorter vector. ***Type: int

NCorr**: number of coefficients to be computed. ***Type: int

The function xcorrlrl can perform the cross-correlation of 2 real float array : X and Y. X must be stored in left memory while Y in right memory. The output must be stored in left memory.

Restrictions:

NCorr must be greater than 4 and multiple of 4

X must be in left memory

Y must be in right memory

Z must be in left memory

Number of cycles:

80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))

Number of VLIW:

94

File: xcorrlrl.mas

3.243 xcorrlrr

Function: cross-correlation between 2 float array: the first stored in left memory and the second in right memory . The result is stored in right memory

$$R_{XY}(i) = \begin{cases} \sum_{k=0}^{N-i-1} X(k-i) \times Y(k) & i = 0 \ldots \frac{Ncorr}{2} \\ R_{YX}(-i) & i = -\frac{Ncorr}{2} \ldots -1 \end{cases}$$

$$Z(i) = R_{XY}\left(i - \frac{Ncorr}{2}\right) \qquad i = 1 \ldots Ncorr$$

Synopsis: __vector__ int xcorrlrr (*X, strideX, *Y, strideY, *Z, strideZ, N, NCorr)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float *

StrideY: stride to be used for the Y data. ***Type: int***

Z**: pointer to the output vector. ***Type: float *

StrideZ: stride to be used for the Z data. ***Type: int***

N: lenght of the input vectors. If X and Y aren't of the same length, N must be set to the length of the shorter vector. ***Type: int***

NCorr: number of coefficients to be computed. ***Type: int***

The function xcorrlrr can perform the cross-correlation of 2 real float array : X and Y. X must be stored in left memory while Y in right memory. The output must be stored in right memory.

Restrictions:

NCorr must be greater than 4 and multiple of 4

X must be in left memory

Y must be in right memory

Z must be in right memory

Number of cycles:

80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))

Number of VLIW:

94

File: xcorrlrr.mas

3.244 xcorrrrl

Function: cross-correlation between 2 float array stored in right memory or auto-correlation of a float array stored in right memory. The result is stored in left memory

$$R_{XY}(i) = \begin{cases} \sum_{k=0}^{N-i-1} X(k-i) \times Y(k) & i = 0 \ldots \frac{Ncorr}{2} \\ R_{YX}(-i) & i = -\frac{Ncorr}{2} \ldots -1 \end{cases}$$

$$Z(i) = R_{XY}\left(i - \frac{Ncorr}{2}\right) \qquad i = 1 \ldots Ncorr$$

Synopsis: __vector__ int xcorrrrl (*X, strideX, *Y, strideY, *Z, strideZ, N, NCorr)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float *

StrideY: stride to be used for the Y data. ***Type: int***

Z**: pointer to the output vector. ***Type: float *

StrideZ: stride to be used for the Z data. ***Type: int***

N: lenght of the input vectors. If X and Y aren't of the same length, N must be set to the length of the shorter vector. ***Type: int***

NCorr: number of coefficients to be computed. ***Type: int***

The function xcorrrrl can perform the cross-correlation of 2 real float array : X and Y or the autocorrelation of the float array X. In the second case the third parameter passed to the function must be equal to the first. The input vectors must be stored in right memory while the output in left memory.

Restrictions:

NCorr must be greater than 4 and multiple of 4

X must be in right memory

Y must be in right memory

Z must be in left memory

Number of cycles:

80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))

Number of VLIW:

94

File: xcorrrrl.mas

3.245 xcorrrrr

Function: cross-correlation between 2 float array stored in right memory or auto-correlation of a float array stored in right memory. The result is stored in right memory

$$R_{XY}(i) = \begin{cases} \sum_{k=0}^{N-i-1} X(k-i) \times Y(k) & i = 0 \ldots \frac{Ncorr}{2} \\ R_{YX}(-i) & i = -\frac{Ncorr}{2} \ldots -1 \end{cases}$$

$$Z(i) = R_{XY}\left(i - \frac{Ncorr}{2}\right) \qquad i = 1 \ldots Ncorr$$

Synopsis: __vector__ int xcorrrrr (*X, strideX, *Y, strideY, *Z, strideZ, N, NCorr)

Include file: DSPlib.h.

X**: pointer to the first input vector. ***Type: float *

StrideX: stride to be used for the X data. ***Type: int***

Y**: pointer to the second input vector. ***Type: float *

StrideY: stride to be used for the Y data. ***Type: int***

Z**: pointer to the output vector. ***Type: float *

StrideZ: stride to be used for the Z data. ***Type: int***

N: lenght of the input vectors. If X and Y aren't of the same length, N must be set to the length of the shorter vector. ***Type: int***

NCorr: number of coefficients to be computed. ***Type: int***

The function xcorrrrr can perform the cross-correlation of 2 real float array : X and Y or the autocorrelation of the float array X. In the second case the third parameter passed to the function must be equal to the first. The input and output vectors must be stored in right memory.

Restrictions:

NCorr must be greater than 4 and multiple of 4

X must be in right memory

Y must be in right memory

Z must be in right memory

Number of cycles:

80 + (26 + 20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))

Number of VLIW:

94

File: xcorrrrr.mas

3.246 xcorrv

Function: cross-correlation between 2 vectorial float array or auto-correlation of a vectorial float array

$$R_{XY}(i) = \begin{cases} \sum_{k=0}^{N-i-1} X(k-i) \times Y^*(k) & i = 0 \ldots \frac{Ncorr}{2} \\ R^*_{YX}(-i) & i = -\frac{Ncorr}{2} \ldots -1 \end{cases}$$

$$Z(i) = R_{XY}\left(i - \frac{Ncorr}{2}\right) \qquad i = 1 \ldots Ncorr$$

Synopsis: __vector__ int xcorrv (*X, strideX, *Y, strideY, *Z, strideZ, N, NCorr)

Include file: DSPlib.h.

***X**: pointer to the first input vector. ***Type: __vector__ float ***

StrideX: stride to be used for the X data. ***Type: int***

***Y**: pointer to the second input vector. ***Type: __vector__ float ***

StrideY: stride to be used for the Y data. ***Type: int***

***Z**: pointer to the output vector. ***Type: __vector__ float ***

StrideZ: stride to be used for the Z data. ***Type: int***

N: lenght of the input vectors. If X and Y aren't of the same length, N must be set to the length of the shorter vector. ***Type: int***

NCorr: number of coefficients to be computed. ***Type: int***

The function xcorrv can perform the cross-correlation of 2 vectorial float array : X and Y or the autocorrelation of the vectorial float array X. In the second case the third parameter passed to the function must be equal to the first.

Restrictions:

NCorr must be greater than 4 and multiple of 4

Number of cycles:

80 + (26+20) × NCorr / 4 + 11 / 8 × sum(N ... (N-NCorr))

Number of VLIW:

94

File: xcorrv.mas

Section 4

Related Documents

1. ATMEL: mAgic DSP Reference Manual - Rev. 7002A (04/04)
2. ATMEL: DIOPSIS 740 Data Sheet - Rev. 7001A (05/04)
3. ATMEL: MADE User Guide - Rev. DRAFT (05/04)
4. ATMEL: MCC User Manual - Rev. DRAFT (05/04)

Atmel Corporation

2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

Literature Requests
www.atmel.com/literature

DRAFT–DPS–04/05 0M