MicroUnity

BroadMX™ C/C++ Functions

Version 1.1
September 2009
# Contents

1 Overview ............................................................................................................................1
  1.1 Introduction ....................................................................................................................1
  1.2 The BroadMX Architecture ............................................................................................1
  1.3 Vector and Matrix Functions ..........................................................................................3
    1.3.1 Group Functions .......................................................................................................3
    1.3.2 Crossbar Functions ..................................................................................................6
    1.3.3 Ensemble Functions ................................................................................................10
    1.3.4 Wide Functions .......................................................................................................15
  1.4 Support for Operating Systems ......................................................................................17
  1.5 Software Development Environment .............................................................................18
    1.5.1 Software Development Tools ................................................................................18
    1.5.2 Complete Test and Debug in Standard UNIX Environments ..................................18
    1.5.3 The GDB Debugger ...............................................................................................18

2 Programming Conventions ................................................................................................21
  2.1 Introduction ....................................................................................................................21
  2.2 Data Types .....................................................................................................................22
    2.2.1 Addresses .................................................................................................................22
    2.2.2 Fixed-Point Data Types ............................................................................................22
      2.2.2.1 Signed and Unsigned Operations ........................................................................27
      2.2.2.2 Rounding and Limiting ......................................................................................28
      2.2.2.3 Carry Bit .............................................................................................................28
      2.2.2.4 Fractional Representation ...................................................................................29
    2.2.3 Floating-Point Data Types .......................................................................................29
      2.2.3.1 Discretionary Aspects of ANSI/IEEE Standard 754 .........................................35
    2.2.4 Complex Numbers .................................................................................................35
    2.2.5 Programming with Vector Data Types ....................................................................36
      2.2.5.1 Loading and Storing Vectors ..............................................................................36
      2.2.5.2 Writing Procedures ............................................................................................37
      2.2.5.3 Representation of Vector Types .........................................................................38
    2.3 Exceptions, Rounding, and Limiting .........................................................................39
      2.3.1 Fixed-Point Modes ..................................................................................................39
      2.3.2 Floating-Point Modes ............................................................................................40
    2.4 Floating-Point Conventions .......................................................................................42
      2.4.1 Floating-Point Compare and Set, and Branch Conditional ....................................42
      2.4.2 Floating-Point Rounding and Exceptions ...............................................................44
        2.4.2.1 Invalid Operation Exception .............................................................................45
        2.4.2.2 Division By Zero Exception ...............................................................................45
        2.4.2.3 Overflow Exception ..........................................................................................45
        2.4.2.4 Underflow Exception .........................................................................................46
        2.4.2.5 Inexact Exception ..............................................................................................46
2.4.3 NaN Handling ............................................................................................................... 46
2.5 Procedures and Calling Conventions............................................................................. 49
2.6 Performance Optimization............................................................................................. 49

3 Vector and Matrix Functions .......................................................................................... 51
3.1 Introduction ........................................................................................................................ 51
3.1.1 Notation .......................................................................................................................... 51
3.2 Support Functions and Macros ....................................................................................... 59
3.2.1 Create Vector .................................................................................................................. 60
   NELEM Number of Elements............................................................................................... 61
   VCONST Create Vector Constant ......................................................................................... 62
   _vector Create Vector with Specified Values ........................................................................ 63
   _vall Create Vector with Identical Components ..................................................................... 64
   _velem Create Vector with First Component Specified ..................................................... 65
3.2.2 Load and Store Vector .................................................................................................... 66
   _lv Load Vector .................................................................................................................. 67
   _sv Store Vector ................................................................................................................. 69
3.2.3 Manipulate Vector .......................................................................................................... 71
   _vget Get Element from Vector ........................................................................................... 72
   _vput Put Element in Vector ................................................................................................ 73
   _reindex Reindex Elements in Vector .................................................................................. 74
   _reindexpair Reindex Elements in Pairs of Vectors ............................................................. 75
   _interleave Interleave Elements in Pairs of Vectors .......................................................... 76
   _even Get Even Elements of Vector .................................................................................... 78
   _odd Get Odd Elements of Vector ...................................................................................... 79
3.2.4 Complex Numbers .......................................................................................................... 80
   _makecplx Make Complex Vector ....................................................................................... 81
   _realpart Get Real Part of Complex Vector .......................................................................... 83
   _imagpart Get Imaginary Part of Complex Vector ................................................................ 84
3.3 Group Functions ................................................................................................................. 85
3.3.1 Group Add and Subtract ............................................................................................. 86
   _gadd Add ............................................................................................................................ 87
   _gaddi Add Immediate ........................................................................................................ 88
   _gaddh, _gaddhu Add and Halve ......................................................................................... 89
   _gaddl, _gaddlu Add with Limiting ...................................................................................... 90
   _gaddo, _gadduo Add with Overflow Trap ........................................................................... 91
   _gaaa Triple Add .................................................................................................................. 92
   _gasa Add-Subtract-Add ...................................................................................................... 93
   _gsub Subtract .................................................................................................................... 94
   _gsubi Subtract Immediate .................................................................................................. 95
   _gsubh, _gsubhu Subtract and Halve .................................................................................... 96
   _gsubl, _gsublu Subtract with Limiting ............................................................................... 98
   _gsubo, _gsubuo Subtract with Overflow Trap ................................................................. 99
   _gsubio, _gsubiuo Subtract Immediate with Overflow Trap ............................................... 100
3.3.2 Group Shift Arithmetic .................................................................................................. 101
   _gshliadd Shift Left Immediate and Add ............................................................................ 102
   _gshlisub Shift Left Immediate and Subtract ..................................................................... 103
3.3.3 Group Boolean or Multiplex .......................................................................................... 104
_gand  Bit-wise AND .................................................................105
_gandi Bit-wise AND Immediate..............................................106
_gaaand Bit-wise Triple-AND ....................................................107
_gnand Bit-wise NAND ............................................................108
_gnandi Bit-wise NAND Immediate .........................................109
_gnaaand Bit-wise Triple-NAND .............................................110
_gandn Bit-wise AND-NOT ......................................................111
_gor  Bit-wise OR ..................................................................112
_gori Bit-wise OR Immediate ................................................113
_goor  Bit-wise Triple-OR .......................................................114
_gnor Bit-wise NOR ..............................................................115
_gnori Bit-wise NOR Immediate ............................................116
_gnooor Bit-wise Triple-NOR ................................................117
_gorn Bit-wise OR-NOT ..........................................................118
_gxor Bit-wise XOR ..............................................................119
_gxori Bit-wise XOR Immediate ............................................120
_gxxxor Bit-wise Triple-XOR ................................................121
_gxnor Bit-wise XNOR ...........................................................122
_gnxxxor Bit-wise Triple-XNOR ..............................................123
_gmux Bit-wise Multiplex .......................................................124
 gboolean Bit-wise Triple Boolean .........................................125

3.3.4 Group Copy Immediate ..................................................................135
_gcopyi Copy Immediate ................................................................136

3.3.5 Group Compare and Set ................................................................137
_gset<cond> Set If Condition ......................................................138
_gset<cond> Set Immediate Condition ......................................139
_gset<cond> Set Floating Condition .........................................140
_gset<cond> Set If Complex Condition .....................................141
_gset<cond> Set Zero Condition .................................................142

3.3.6 Group Compare and Trap ................................................................143
_gcom<cond> Compare Condition ...............................................144
_gcom<cond> Compare Floating Condition ....................................145
_gcom<cond> Compare Complex Condition ....................................146
_gcom<cond> Compare Zero Condition .........................................147

3.4 Crossbar Functions ....................................................................153

3.4.1 Crossbar Shift and Rotate ..................................................154
_xrotl, _xrotl Rotate Left ..........................................................155
_xrotr, _xrotri Rotate Right ........................................................156
_xshl, _xshli Shift Left ..............................................................157
_xshlo, _xshlio Shift Left with Overflow Trap .............................158
_xshluo, _xshliuo Shift Left Unsigned with Overflow Trap ..........159
_xshlm, _xshlmi Shift Left and Merge ........................................160
_xshr, _xshri Shift Right ............................................................161
_xshru, _xshriu Shift Right Unsigned ........................................162
_xshrm, _xshrmi Shift Right and Merge .....................................163

3.4.2 Crossbar Compress and Expand ..........................................164
_xcompress, _xcompressi Compress ........................................165
_xcompressu, _xcompressiu Compress Unsigned ........................166
_xexpand, _xexpandi Expand .....................................................167
_xexpandu, _xexpandiu Expand Unsigned .............................................................. 168

3.4.3 Crossbar Deposit and Withdraw.............................................................. 169
_xdeposit Deposit ................................................................................... 170
_xdeposittt Deposit Unsigned .............................................................. 171
_xdepositm Deposit and Merge .............................................................. 172
_xwithdraw Withdraw ........................................................................... 173
_xwithdrawu Withdraw Unsigned ........................................................... 174

3.4.3 Crossbar Deposit and Withdraw.............................................................. 169
_xdepositt Deposit .............................................................................. 170
_xdeposittu Deposit Unsigned .............................................................. 171
_xdeposittm Deposit and Merge .............................................................. 172
_xwithdrawt Withdraw ........................................................................... 173
_xwithdrawtt Withdraw Unsigned ........................................................... 174

3.4.4 Crossbar Shuffle................................................................................... 175
_xshuffle Shuffle ...............................................................................
_xshufflepair 256-Bit Shuffle...................................................................

3.4.5 Crossbar Swizzle................................................................................... 179
_xcopy Swizzle Copy ............................................................................
_xreverse Swizzle Reverse ....................................................................... 181
_xfieldreverse Field Reverse .................................................................... 182
_xswizzle Swizzle ................................................................................... 183

3.4.6 Crossbar Select Byte........................................................................... 186
_xselect Byte Select .............................................................................

3.4.7 Crossbar Extract................................................................................... 188
_xextractt Extract from Single-Size Source ........................................ 189
_xextract Extract from Double-Size Source ...................................... 190
_xextractm Extract and Merge ............................................................. 191
_xextract Extract (Generic) ................................................................. 193

3.5 Ensemble Functions ........................................................................... 194
3.5.1 Ensemble Arithmetic ........................................................................ 195
_eabsf Absolute Value Floating-Point .............................................. 196
_eaddf Add Floating-Point .................................................................... 197
_enegf Negate Floating-Point .............................................................. 198
_esubf Subtract Floating-Point .............................................................. 199
_esum Extract Elements ....................................................................
_esumf Sum Elements Floating-Point ..............................................
_esump Sum Polynomial ....................................................................

3.5.2 Ensemble Multiply (Non-Extracting) ................................................ 203
_emul Multiply .....................................................................................
_emulu Multiply Unsigned .................................................................... 204
_emulm Multiply Mixed Sign ............................................................... 205
_emulc Multiply Complex .................................................................... 206
_emulf Multiply Floating-Point ............................................................. 207
_emulcf Multiply Complex Floating-Point .......................................... 208
_emulfg Multiply Galois Field ............................................................... 209
_emulp Multiply Polynomial ................................................................. 210
_emulpm Multiply Polynomial .............................................................. 211
_emuladd Multiply and Add ...................................................................
_emuladdu Multiply and Add Unsigned ................................................ 212
_emuladdm Multiply and Add Mixed Sign ............................................ 213
_emuladdd Multiply and Add Complex ................................................ 214
_emuladdc Multiply and Add Complex ................................................ 215
_emuladddf Multiply and Add Floating-Point ...................................... 216
_emuladddc Multiply and Add Complex Floating-Point ....................... 217
_emuladddcf Multiply and Add Complex Floating-Point ...................... 218
_emulsub Multiply and Subtract .......................................................... 219
_emulsubu Multiply and Subtract Unsigned ......................................... 220
_emulsubm Multiply and Subtract Mixed Sign ..................................... 221
3.5.3 **Ensemble Multiply (Extract Immediate)**

- `_emulxi` Multiply and Extract Immediate ........................................ 227
- `_emulxic` Multiply and Extract Immediate Complex .......................... 228
- `_emuladdxi` Multiply, Add, and Extract Immediate ............................ 230
- `_emuladdxic` Multiply, Add, and Extract Immediate Complex ............... 232

3.5.4 **Ensemble Multiply (Generalized Extract)**

- `_emulx` Multiply and Extract ........................................................... 228
- `_emulxx` Multiply and Extract/Expand .............................................. 229
- `_emulxc` Multiply and Extract Complex ........................................... 230
- `_emulxxc` Multiply and Extract/Expand Complex ................................ 231
- `_emulx` Multiply and Extract (Generic) ............................................ 232
- `_emuladdx` Multiply, Add, and Extract ............................................ 233
- `_emuladdxx` Multiply, Add, and Extract/Expand ................................ 234
- `_emuladdxc` Multiply, Add, and Extract Complex ............................. 235
- `_emuladdxxc` Multiply, Add, and Extract/Expand Complex ................. 237
- `_emuladdx` Multiply, Add, and Extract (Generic) ............................. 238

3.5.5 **Ensemble Multiply Sum (Dot Product)**

- `_emulsum` Multiply and Sum ............................................................. 239
- `_emulsumc` Multiply and Sum Complex ............................................. 240
- `_emulsumm` Multiply and Sum Mixed Sign ....................................... 241
- `_emulsumu` Multiply and Sum Unsigned ........................................... 242
- `_emulsumg` Multiply and Sum Galois Field ..................................... 243

3.5.6 **Ensemble Divide**

- `_ediv` Divide .................................................................................. 244
- `_edivf` Divide Floating-Point ............................................................ 245
- `_edivu` Divide Unsigned .................................................................. 246

3.5.7 **Ensemble Scale Add**

- `_escaladdf` Scale and Add Floating-Point .......................................... 247
- `_escaladdx` Scale, Add, and Extract .................................................. 248

3.5.8 **Ensemble Convolve**

- `_econ` Convolve.................................................................................. 249
- `_econu` Convolve Unsigned ................................................................. 250
- `_econm` Convolve Mixed Sign ............................................................. 251
- `_econc` Convolve Complex ................................................................. 252
- `_econf` Convolve Floating-Point ......................................................... 253
- `_econcf` Convolve Complex Floating-Point ........................................ 254
- `_econx` Convolve and Extract ............................................................ 255
- `_econxi` Convolve and Extract Immediate ......................................... 256
- `_econxic` Convolve and Extract Immediate Complex ........................ 257

3.5.9 **Ensemble Conversion and Scaling**

- `_edeflatef` Deflate Floating-Point ...................................................... 258
- `_einflatef` Inflate Floating-Point ....................................................... 259
- `_efloatf` Convert Integer to Floating-Point ......................................... 260
- `_esinkf` Convert Floating-Point to Integer .......................................... 261
- `_eextracti` Extract Immediate ........................................................... 262
- `_eextractiu` Extract Immediate Unsigned .......................................... 263

---

BroadMX Instruction Set C/C++ Intrinsic Functions
_eextractx  Extract from Single-Size Source ................................................................. 291
_eextract  Extract from Double-Size Source ............................................................... 292
_eextractm  Extract and Merge .................................................................................... 293
_eextract  Extract (Generic) .......................................................................................... 295

3.5.10  Ensemble Special Operations ..................................................................................... 296
_elogmost  Log of Most-Significant Bit ................................................................................. 297
_elogmostu  Log of Most-Significant Bit Unsigned .............................................................. 298
_erecestf  Reciprocal Estimate Floating-Point ........................................................................ 299
_ersqestf  Reciprocal Square-Root Estimate Floating-Point ................................................ 300
_esqrf  Square-Root Floating-Point ...................................................................................... 301

3.5.11  Extraction Control ...................................................................................................... 302
3.5.11.1  Extract-Merge Functions ....................................................................................... 302
3.5.11.2  Extract Functions Combined with Other Functions ................................................ 304
3.5.11.3  The Control Parameter (ctrl) .................................................................................. 305

3.6  Wide Functions ................................................................................................................. 310
3.6.1  Wide Table Lookup ..................................................................................................... 311
_wtranslate  Wide Table Lookup .......................................................................................... 312

3.6.2  Wide Crossbar ............................................................................................................. 314
_wswitch  Wide Switch ......................................................................................................... 315

3.6.3  Wide Ensemble ............................................................................................................ 317
_wmulmat  Wide Multiply Matrix .......................................................................................... 318
_wmulmatu  Wide Multiply Matrix Unsigned .......................................................................... 319
_wmulmatm  Wide Multiply Matrix Mixed-Sign ...................................................................... 320
_wmulmatc  Wide Multiply Matrix Complex .......................................................................... 321
_wmulmatf  Wide Multiply Matrix Floating-Point .................................................................... 323
_wmulmatcf  Wide Multiply Matrix Complex Floating-Point .................................................. 324
_wmulmatp  Wide Multiply Matrix Polynomial ...................................................................... 326
_wmulmatg  Wide Multiply Matrix Galois Field ...................................................................... 328
_wmulmatxi  Wide Multiply Matrix Extract Immediate .......................................................... 331
_wmulmatxic  Wide Multiply Matrix Extract Immediate Complex ........................................... 333
_wmulmatx  Wide Multiply Matrix Extract ............................................................................. 335

Appendix A.  Glossary .............................................................................................................. 339
Appendix B.  Index .................................................................................................................... 349
1 Overview

1.1 Introduction

This manual describes MicroUnity’s BroadMX vector functions from the perspective of C or C++ programmers. It contains overviews of the various types of C functions, graphic depictions of each function’s operation, equivalent equations, and examples of useful applications. The manual assumes only that you are familiar with C programming conventions, although some background in signal processing, image processing, or other broadband processing is helpful.

Here is some related documentation:

• BroadMX DSP Manual including the DSP Library, available from MicroUnity Systems Engineering, Inc.
• ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic, available from The Institute of Electrical and Electronics Engineers, Inc.

1.2 The BroadMX Architecture

The BroadMX architecture includes more than 200 Vector and Matrix operations that support the programming of high-performance broadband communications and other computationally intensive algorithms. The functions have been implemented as a library of C intrinsic functions that can be included in C or C++ programs, allowing programmers to develop and test applications on their existing host machine. These applications can then be accelerated with hardware that supports the BroadMX architecture as direct machine instructions.

The architecture includes the following sets of operations:

• Vector and Matrix Functions—These functions support digital signal and image processing, control, and other sophisticated algorithms with all common and many uncommon fixed- and floating-point operations. The functions include not only arithmetic and logic operations, but also bit-switching, convolution, matrix multiplication, polynomial multiplication, Galois multiplication, and table lookups. They typically operate on sets of 128-bit vectors. These vectors can be partitioned into elements ranging from 1 to 128 bits, although some operate on 256-bit vectors and on tables of data in memory as large as 32,768 bits. The functions are based
on power-of-two divisions of the data path and maintain the fullest possible use of 128-bit data paths when they operate on lower-precision vectors. Table 1 summarizes the characteristics of the vector and matrix function types.

- **Access (Scalar) Functions**—These functions support not only memory access (address arithmetic, branching, memory loads, and memory stores) but also a complete set of standard 64-bit RISC ALU operations. These latter operations are referred to as the BroadMX architecture’s general-purpose 64-bit front-end RISC core. The Access functions support full virtual-memory management, memory protection, and memory synchronization mechanisms for sharing memory and computational resources among many user-level and secure-kernel execution threads. The full set of Access functions is described in the *BroadMX Architecture Reference Manual*.

**Table 1. Characteristics of Vector and Matrix Functions**

<table>
<thead>
<tr>
<th>Type</th>
<th>Operand Width and Depth</th>
<th>Execution Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>64 address and/or 128 bits by 1 or 2 vectors</td>
<td></td>
<td>Load and store vectors, create vectors, manipulate vectors.</td>
</tr>
<tr>
<td>Group</td>
<td>128 bits by 1, 2, or 3 vectors</td>
<td>Group Logic Unit</td>
<td>ALU operations on corresponding elements of vectors.</td>
</tr>
<tr>
<td>Crossbar</td>
<td>128 bits by 1, 2, or 3 vectors, or 256 bits by 1 vector plus 1 control parameter</td>
<td>Crossbar Unit</td>
<td>Bit movement or bit logic on corresponding elements of vectors.</td>
</tr>
<tr>
<td>Ensemble</td>
<td>128 bits by 1, 2, 3, or 4 vectors</td>
<td>Ensemble Unit (multiplier, floating-point unit, and summing tree)</td>
<td>Matrix multiplies on corresponding elements of vectors, or matrix multiplies that combine vector elements into scalars.</td>
</tr>
<tr>
<td>Wide</td>
<td>16 to 128 bits by 4 to 256 table rows</td>
<td>Table Lookup Unit</td>
<td>Lookups on large tables of data in memory.</td>
</tr>
<tr>
<td></td>
<td>128 bytes by 8 table rows</td>
<td>Crossbar Unit</td>
<td>Crossbar operations on large tables of data in memory.</td>
</tr>
<tr>
<td></td>
<td>16x16 bytes/matrix up to 16 matrices</td>
<td>Ensemble Unit</td>
<td>Ensemble operations on large tables of data in memory.</td>
</tr>
</tbody>
</table>

1. Many functions have versions in which immediate operands take the place of a vector or control parameters.
1.3 **Vector and Matrix Functions**

The following sections illustrate examples of the functions listed in Table 1 on page 2.

1.3.1 **Group Functions**

Group functions perform simple ALU operations on corresponding elements of 1, 2, or 3 128-bit source vectors, or immediate operands.

Figure 1 shows two functions involving vector-element arithmetic and shifts. Figure 1 (top) is a function that adds elements of two vectors and subtracts elements of a third vector. Figure 1 (bottom) is a function that left-shifts elements of one vector and adds them to elements of another vector.

**Figure 1. Group Examples: Vector-Element Arithmetic or Shift and Arithmetic**

```
127
  x  0
  element element element element

127
  y  0
  element element element element

127
  z  0
  element element element element

result

127
  x  0
  element element element element

127
  y  0
  element element element element

immediate

127
  sh
  << << << <<

127
  r  0
  size size size size
  element element element element
```

---

**Note:** The diagram uses placeholders and symbols to represent elements and operations. The specific values and operations are not described in the text. The diagram illustrates the basic concept of how these operations are performed on vector data.
Figure 2 shows two functions that expand an immediate into vector elements. Figure 2 (top) is a function that simply replicates a sign-extended immediate into each element of a vector. Figure 2 (bottom) is a function that performs a Boolean operation on a sign-extended immediate and the elements of a vector.

**Figure 2. Group Examples: Expansion of Immediates Into Vector Elements**

![Diagram showing expansion of immediates into vector elements](image_url)
Group conditional arithmetic or logic operations are supported in the sense that set-on-condition operations can be used to construct bit masks, which can be used to select between alternate vector expressions using bit-wise Boolean operations.

Figure 3 shows two functions that set elements of a result vector, based on comparisons between elements of source vectors. Figure 3 (top) is a function that compares each element of two vectors and sets the result elements accordingly; for floating-point vectors, exceptions can optionally be trapped. Figure 3 (bottom) is a function that performs 128-bit-wide bit-wise multiplexing, selecting one of two bits based on the value of a third bit.
Crossbar Functions

Crossbar functions mix parts of fixed-point vector elements, or move bits around within elements, of up to three 128-bit source vectors or immediate operands, or one 256-bit vector and control parameters. They return their results as a 128-bit vector of elements. The Crossbar Logic Unit is essentially a full crossbar switch with 256 inputs and 128 outputs. The unit contains embedded memory that stores eight distinct switching configurations in which each output bit can come from any of the 256 input bits.

Figure 4 shows a function that left-shifts elements of a vector and fills the low bits with zero.

**Figure 4. Crossbar Example: Shift Left**

```
+-----------------+     +-----------------+     +-----------------+
| 127             |   =>  | 127             |   =>  | 127             |
| element element |         | element element |         | element element |
| <<              |         | <<              |         | <<              |
| 0 0 0 0         |        | 0 0             |        | 0               |
| 127             |   =>  | 127             |   =>  | 127             |
| result          |        | result          |        | result          |
```

Figure 5 shows a function that compresses elements of a vector by halving their width.

**Figure 5. Crossbar Example: Compress**

```
+-----------------+     +-----------------+     +-----------------+
| 127             |   =>  | 127             |   =>  | 127             |
| element element |         | element element |         | element element |
| >>              |         | >>              |         | >>              |
| element element |         | element element |         | element element |
| 0 0             |        | 0 0             |        | 0               |
| 127             |   =>  | 127             |   =>  | 127             |
| result          |        | result          |        | result          |
```
Figure 6 shows a function that shuffles fields within a 256-bit vector, and selects the low half as the result.

**Figure 6. Crossbar Example: Shuffle**

```
<table>
<thead>
<tr>
<th></th>
<th>xhi</th>
<th>128 127</th>
<th>xlo</th>
<th>0</th>
<th>immediates</th>
<th>npile</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Q</td>
<td>N</td>
<td>M</td>
<td>L</td>
<td>K</td>
<td>J</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>G</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>
```

Fields: pile

```
255
<table>
<thead>
<tr>
<th></th>
<th>128 127</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Q</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>J</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>G</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>
```

NPile

```
csize/2
```

```
r
```

Figure 6-299.png
Figure 7 shows a function that re-arranges fields within a vector according to a flexible copy-and-swap rule.

**Figure 7. Crossbar Examples: Swizzle**
Figure 8 shows a function that rearranges source bytes or performs table lookups.

**Figure 8. Crossbar Example: Select Byte**

Figure 9 shows a function that extracts fields in elements of one vector, and deposits (and optionally merges) them in elements of another vector.

**Figure 9. Crossbar Example: Extract**
1.3.3 **Ensemble Functions**

Ensemble functions perform fixed-point or floating-point matrix multiplies and other operations on elements of up to four 128-bit source vectors or immediate values. They return their results as a 128-bit vector of elements. The functions include fixed- and floating-point convolutions, matrix multiplication, polynomial multiplication, and Galois multiplication.

The Ensemble Unit combines a multiplier, floating-point unit, exponent logic, and summing tree. The unit also contains embedded memory, to supply a large number of operands to the multiplier array for vector-matrix operations. Because these operations involve intense computation, using the majority of the processor’s logic and embedded-memory resources, *the Ensemble functions constitute the heart of the BroadMX architecture*. The other types of functions—Group, Crossbar, and Wide—typically play the role of setting up data on which Ensemble functions can operate with maximum throughput.

Figure 10 shows a function that adds all elements of a single vector.

**Figure 10. Ensemble Example: Sum All Elements**

![Diagram showing a function that adds all elements of a single vector](image)
Figure 11 shows a function that multiplies elements of two complex-number vectors and extracts fields.

**Figure 11. Ensemble Example: Complex Multiply and Extract**

Key:
- `real`  Real part of complex number
- `imag`  Imaginary part of complex number
- `−`     Negate
- `∗`     Multiply row element by column element
- `Σ`     Sum of products
Figure 12 shows a function that multiplies elements of two vectors and sums them, to produce a dot-product result.

**Figure 12. Ensemble Example: Multiply and Sum (Dot Product)**

Key:
- \( \ast \) Multiply row element by column element
- \( \Sigma \) Sum of products
Figure 13 shows a function that multiplies elements of two vectors, each vector using a different scalar multiplier, adds the products, and extracts fields.

**Figure 13. Ensemble Example: Scale Add**

Figure 14 shows a function that convolves elements of two vectors.
Figure 14. Ensemble Example: Convolve

Figure 15 shows a function that computes reciprocal square roots of vector elements, with optional trapping to an exception handler.

Figure 15. Ensemble Example: Special Operations
1.3.4 **Wide Functions**

Wide functions perform lookups on large tables (matrices) in memory, or they perform memory-table-based Crossbar or Ensemble operations. Table lookups can be done on up to 128 bits by up to 256 table entries (rows). They load a table of source data from memory, perform a set of operations on partitions of bits in the data, and return the result as a 128-bit vector of elements.

Figure 16 shows a table lookup function that accesses multiple indexed entries in a set of tables in memory. The function can access entries in tables of up to 32,768 bits (128 x 256).

**Figure 16. Wide Example: Table Lookup**
Figure 17 shows a function that multiplies elements of a vector with a matrix in memory and extracts fields.

**Figure 17. Wide Example: Multiply Matrix**

Key:
- * Multiply row element by column element
- \( \sum \) Sum of products
Figure 18 shows a function that selects one of 256 source bits for each bit of a 128-bit result.

**Figure 18. Wide Example: Crossbar**

---

1.4 **Support for Operating Systems**

The BroadMX architecture supports robust operating systems, including 64-bit UNIX operating system and real-time kernels, with virtual memory management, protection, and synchronization mechanisms. These mechanisms support sharing memory and computational resources among user-level and secure-kernel execution threads. For more information about this support, see the *BroadMX System Architecture Reference Manual*, or contact MicroUnity.
1.5 **Software Development Environment**

1.5.1 **Software Development Tools**

The BroadMX architecture is supported in a UNIX environment by the following software development tools:

- BroadMX emulation library (`gops`).
- GNU-Based Optimizing C Compiler.
- GNU-Based Assembler.
- GNU-Based Linker.
- GNU-Based GDB Symbolic Debugger.
- Instruction-Set Simulator.

1.5.2 **Complete Test and Debug in Standard UNIX Environments**

The BroadMX emulation library expands the BroadMX functions described in this manual into C-function equivalents that are linked with the user’s compiled application, making the BroadMX functions available on a wide range of platforms. This allows broadband applications to be developed and tested on existing machines.

The GNU-based BroadMX C compiler generates object modules that conform to the BroadMX Instruction Set Architecture. These object modules can be executed directly on hardware implementing the BroadMX extensions such as the MicroUnity Broadband Microprocessor chip, or they can be executed by the BroadMX instruction-set simulator, running on a host platform.

Versions of the BroadMX instruction-set simulator are available for X-86 PC workstations running Linux. The simulator can precisely mimic the operation of BroadMX functions running on a MicroUnity Broadband co-processor core. The simulator is useful for many development tasks including:

- Starting code development before hardware is available.
- Examining internal register status.
- Estimating DSP algorithm times.
- Validating program flow and integrity.

1.5.3 **The GDB Debugger**

The BroadMX GNU-based GDB source- and assembly-language debugger augments the instruction-set simulator by making visible the events occurring inside a MicroUnity Broadband Microprocessor chip, thereby allowing program bugs to be caught in the act. The debugger provides the following features:
• Source-level debugging.
• Specify constraints that bound program behavior.
• Single-step instructions in both C source and assembly language.
• Halt program execution on specified conditions.
• Examine state at breakpoints.
• Change boundary conditions and variables to study bug interactions.
Chapter 2: Programming Conventions

2 Programming Conventions

2.1 Introduction

The BroadMX functions include over 200 fixed-point and floating-point operations for very diverse broadband applications. All functions take up to four 128-bit source parameters, produce a single 128-bit result, and are free of side effects such as condition-code and flag setting that can complicate hardware implementation and diminish performance. The functions are carefully structured to simplify their physical realization in a microprocessor allowing efficient hardware implementations.

From a software perspective, the entire machine state consists of a single bank of 128-bit general-purpose registers, a program counter, and a linear, byte-addressed, shared, virtual memory space with memory-mapped interface registers. All interrupts and exceptions are precise. The architecture’s virtual and physical addresses and its protection and synchronization mechanisms support robust operating systems.
2.2 **Data Types**

The architecture supports all common, and several uncommon, fixed- and floating-point data-structure types, plus a complete set of fixed- and floating-point vector data types. Figure 19 through Figure 21 illustrate the data types for fixed-point data and floating-point data.

BroadMX C compilers support strong data-type checking. The data types required by a function are denoted in the function name syntax. The type is designated by a letter and a number. The letter (u,f,c) specifies the type (unsigned, float, complex) while the number gives the size of an element in bits. If the letter is omitted, signed integer is implied. Two examples are _gadd8 and _eaddf32 for byte-add and floating point add.

Data structures in memory may be organized in either little- or big-endian byte order, but bit-ordering in registers is always little-endian. The selection of byte ordering is dynamically specified in load and store functions, so that little- and big-endian processes, and even data structures within a process, can be intermixed freely. Unless otherwise specified, data structures in memory have no alignment requirement, although data accesses aligned to 128-bit boundaries are an option in the load and store mnemonic syntax and will often require one fewer processor or memory clock cycle than unaligned accesses.

2.2.1 **Addresses**

The single data type used for all virtual and physical addresses is typed as `void*` in the syntax of load and store functions.

2.2.2 **Fixed-Point Data Types**

Figure 19 and Table 2 and Table 3 show the fixed-point scalar data types. These include signed integer and complex-number scalars. Fixed-point arithmetic is performed on twos-complement (signed) or unsigned binary-number or complex-number formats. Figure 20 and Table 4 and Table 5 show the fixed-point vector data types. These include vectors of signed integer or complex-number elements. The vector-element sizes are powers of two.
Figure 19. Fixed-Point Scalar Data Types

triclet, int256
256 bits, 32 bytes

hexlet, int128
128 bits, 16 bytes

octlet, int64
64 bits, 8 bytes

quadlet, int32
32 bits, 4 bytes

doublet, int16
16 bits, 2 bytes

byte, int8
8 bits

nibble
4 bits

peck
2 bits

bit
1 bit

cmplxi64
128 bits, 16 bytes

cmplxi32
64 bits, 8 bytes

cmplxi16
32 bits, 4 bytes

cmplxi8
16 bits, 2 bytes
### Table 2. Fixed-Point Signed Scalar Data Types and Representable Values

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Type in C</th>
<th>Size (Bits)</th>
<th>Description</th>
<th>Maximum Representable Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triclet</td>
<td>int256</td>
<td>256</td>
<td>The concatenation of 256 bits, and the concatenation of thirty-two bytes.</td>
<td>$-2^{255}$ to $+(2^{255} - 1)$</td>
</tr>
<tr>
<td>Hexlet</td>
<td>int128</td>
<td>128</td>
<td>The concatenation of 128 bits, and the concatenation of sixteen bytes.</td>
<td>$-2^{127}$ to $+(2^{127} - 1)$</td>
</tr>
<tr>
<td>Octlet</td>
<td>int64</td>
<td>64</td>
<td>The concatenation of 64 bits, and the concatenation of eight bytes.</td>
<td>$-2^{63}$ to $+(2^{63} - 1)$</td>
</tr>
<tr>
<td>Quadlet</td>
<td>int32</td>
<td>32</td>
<td>The concatenation of 32 bits, and the concatenation of four bytes.</td>
<td>$-2^{31}$ to $+(2^{31} - 1)$</td>
</tr>
<tr>
<td>Doublet</td>
<td>int16</td>
<td>16</td>
<td>The concatenation of 16 bits, and the concatenation of two bytes.</td>
<td>$-2^{15}$ to $+(2^{15} - 1)$</td>
</tr>
<tr>
<td>Byte</td>
<td>int8</td>
<td>8</td>
<td>The concatenation of eight bits, and a single addressable element of the memory array.</td>
<td>$-2^{7}$ to $+(2^{7} - 1)$</td>
</tr>
<tr>
<td>Nibble</td>
<td>—</td>
<td>4</td>
<td>The concatenation of four bits.</td>
<td>$-2^{3}$ to $+(2^{3} - 1)$</td>
</tr>
<tr>
<td>Peck</td>
<td>—</td>
<td>2</td>
<td>The concatenation of two bits.</td>
<td>$-2^{1}$ to $+(2^{1} - 1)$</td>
</tr>
<tr>
<td>Bit</td>
<td>—</td>
<td>1</td>
<td>A primitive data element.</td>
<td>+0 to +1, or +0 to -1</td>
</tr>
</tbody>
</table>

### Table 3. Fixed-Point Scalar Complex Data Types and Representable Values

<table>
<thead>
<tr>
<th>Data Type in C</th>
<th>Size (Bits)</th>
<th>Description</th>
<th>Maximum Representable Values Per Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>cplxi64</td>
<td>128</td>
<td>Complex number formed by two components (real and imaginary) of 64 bits each.</td>
<td>$-2^{63}$ to $+(2^{63} - 1)$</td>
</tr>
<tr>
<td>cplxi32</td>
<td>64</td>
<td>Complex number formed by two components (real and imaginary) of 32 bits each.</td>
<td>$-2^{31}$ to $+(2^{31} - 1)$</td>
</tr>
<tr>
<td>cplxi16</td>
<td>32</td>
<td>Complex number formed by two components (real and imaginary) of 16 bits each.</td>
<td>$-2^{15}$ to $+(2^{15} - 1)$</td>
</tr>
<tr>
<td>cplxi8</td>
<td>16</td>
<td>Complex number formed by two components (real and imaginary) of 8 bits each.</td>
<td>$-2^{7}$ to $+(2^{7} - 1)$</td>
</tr>
</tbody>
</table>
Figure 20. Fixed-Point Vector Data Types

- **v128_t**: 128-bit element
- **v64_t**: 2 64-bit elements
- **v32_t**: 4 32-bit elements
- **v16_t**: 8 16-bit elements
- **v8_t**: 16 8-bit elements
- **v4_t**: 32 4-bit elements
- **v2_t**: 64 2-bit elements
- **v1_t**: 128 1-bit elements

- **hexlet_t**: n k-bit elements, as specified by a control parameter
- **vc64_t**: 2 64-bit elements
- **vc32_t**: 4 32-bit elements
- **vc16_t**: 8 16-bit elements
- **vc8_t**: 16 8-bit elements
Table 4. Fixed-Point Signed Vector Data Types and Representable Values

<table>
<thead>
<tr>
<th>Data Type in C</th>
<th>Size (Bits)</th>
<th>Description</th>
<th>Maximum Representable Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>v128_t</td>
<td>128</td>
<td>A 128-bit vector of 1 128-bit element.</td>
<td>$-2^{127}$ to $+(2^{127} - 1)$</td>
</tr>
<tr>
<td>v64_t</td>
<td>128</td>
<td>A 128-bit vector of 2 64-bit elements.</td>
<td>$-2^{63}$ to $+(2^{63} - 1)$ in each element</td>
</tr>
<tr>
<td>v32_t</td>
<td>128</td>
<td>A 128-bit vector of 4 32-bit elements.</td>
<td>$-2^{31}$ to $+(2^{31} - 1)$ in each element</td>
</tr>
<tr>
<td>v16_t</td>
<td>128</td>
<td>A 128-bit vector of 8 16-bit elements.</td>
<td>$-2^{15}$ to $+(2^{15} - 1)$ in each element</td>
</tr>
<tr>
<td>v8_t</td>
<td>128</td>
<td>A 128-bit vector of 16 8-bit elements.</td>
<td>$-2^{7}$ to $+(2^{7} - 1)$ in each element</td>
</tr>
<tr>
<td>v4_t</td>
<td>128</td>
<td>A 128-bit vector of 32 4-bit elements.</td>
<td>$-2^{3}$ to $+(2^{3} - 1)$ in each element</td>
</tr>
<tr>
<td>v2_t</td>
<td>128</td>
<td>A 128-bit vector of 64 2-bit elements.</td>
<td>$-2^{1}$ to $+(2^{1} - 1)$ in each element</td>
</tr>
<tr>
<td>v1_t</td>
<td>128</td>
<td>A 128-bit vector of 128 1-bit elements.</td>
<td>+0 to +1, or +0 to -1 in each element</td>
</tr>
<tr>
<td>hexlet_t</td>
<td>128</td>
<td>A 128-bit vector of n k-bit elements, as specified by a control parameter.</td>
<td>As specified by the control parameter.</td>
</tr>
</tbody>
</table>

Table 5. Fixed-Point Vector Complex Data Types and Representable Values

<table>
<thead>
<tr>
<th>Data Type in C</th>
<th>Size (Bits)</th>
<th>Description</th>
<th>Maximum Representable Values Per Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>vc64_t</td>
<td>128</td>
<td>A 128-bit vector of 1 complex number formed by two components (real and imaginary) of 64 bits each. Same as cplxi64.</td>
<td>$-2^{63}$ to $+(2^{63} - 1)$</td>
</tr>
<tr>
<td>vc32_t</td>
<td>128</td>
<td>A 128-bit vector of 2 complex numbers formed by two components (real and imaginary) of 32 bits each.</td>
<td>$-2^{31}$ to $+(2^{31} - 1)$</td>
</tr>
<tr>
<td>vc16_t</td>
<td>128</td>
<td>A 128-bit vector of 4 complex numbers formed by two components (real and imaginary) of 16 bits each.</td>
<td>$-2^{15}$ to $+(2^{15} - 1)$</td>
</tr>
<tr>
<td>vc8_t</td>
<td>128</td>
<td>A 128-bit vector of 8 complex numbers formed by two components (real and imaginary) of 8 bits each.</td>
<td>$-2^{7}$ to $+(2^{7} - 1)$</td>
</tr>
</tbody>
</table>
### Table 6. Fixed-Point Unsigned Vector Data Types and Representable Values

<table>
<thead>
<tr>
<th>Data Type in C</th>
<th>Size (Bits)</th>
<th>Description</th>
<th>Maximum Representable Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>vu128_t</td>
<td>128</td>
<td>A 128-bit vector of 1 128-bit element.</td>
<td>0 to +(2^{128} - 1)</td>
</tr>
<tr>
<td>vu64_t</td>
<td>128</td>
<td>A 128-bit vector of 2 64-bit elements.</td>
<td>0 to +(2^{64} - 1) in each element</td>
</tr>
<tr>
<td>vu32_t</td>
<td>128</td>
<td>A 128-bit vector of 4 32-bit elements.</td>
<td>0 to +(2^{32} - 1) in each element</td>
</tr>
<tr>
<td>vu16_t</td>
<td>128</td>
<td>A 128-bit vector of 8 16-bit elements.</td>
<td>0 to +(2^{16} - 1) in each element</td>
</tr>
<tr>
<td>vu8_t</td>
<td>128</td>
<td>A 128-bit vector of 16 8-bit elements.</td>
<td>0 to +(2^{8} - 1) in each element</td>
</tr>
<tr>
<td>vu4_t</td>
<td>128</td>
<td>A 128-bit vector of 32 4-bit elements.</td>
<td>0 to +(2^{4} - 1) in each element</td>
</tr>
<tr>
<td>vu2_t</td>
<td>128</td>
<td>A 128-bit vector of 64 2-bit elements.</td>
<td>0 to +(2^{2} - 1) in each element</td>
</tr>
<tr>
<td>vu1_t</td>
<td>128</td>
<td>A 128-bit vector of 128 1-bit elements.</td>
<td>0 to 1 in each element</td>
</tr>
</tbody>
</table>

2.2.2.1 **Signed and Unsigned Operations**

Some operations (for example, add) internally use the same logic to perform their function on both signed and unsigned data types. Other operations (for example, add with overflow trap) need separate functions for signed and unsigned data types. Table 7 shows the effect on the result when using signed or unsigned fixed-point functions.

### Table 7. Effect of Signed vs. Unsigned Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect on Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Signed</strong></td>
</tr>
<tr>
<td>add (+)</td>
<td>add (carry bit ignored)</td>
</tr>
<tr>
<td>subtract (-)</td>
<td>subtract (carry bit ignored)</td>
</tr>
<tr>
<td>right shift (&gt;&gt;)</td>
<td>arithmetic shift (sign-fill)</td>
</tr>
</tbody>
</table>

1. This property is only significant for unsigned functions.
### Rounding and Limiting

Some fixed-point functions, such as `_gaddh` (vector add-and-halve), involve division by a power of 2. These support four rounding modes, described in detail in Section 2.3 on page 39. The default rounding mode is round-to-nearest. The other choices are round toward minus infinity (floor), round towards positive infinity (ceiling), and round toward zero (truncate).

Certain add and subtract functions perform limiting — specifically, `_gaddl`, `_gaddlu`, `_gsubl`, and `_gsublu`. Results that would otherwise have overflowed are clamped to the minimum or maximum representable values according to the sign of the result. The extreme representable fixed-point values are shown in Table 4 on page 26.

### Carry Bit

Sometimes the value of the carry bit must be explicitly known. Multi-word high-precision arithmetic is such a case. The following code fragment illustrates an efficient way of obtaining the carry bit:

```c
unsigned int x, y, sum, carry;
sum = x + y;       /* add the operands */
carry = sum < x;    /* determine the carry bit, c */
```

Using the BroadMX functions this would be expressed as follows:

```c
vu128_t x, y, sum, neg_carry;   /* do arithmetic on 128-bit quantities*/
sum = _gadd128(x, y);       /* add the operands */
neg_carry = _gsetgeu128(x, s);  /* determine the (negated) carry bit, c */
```
The last line above sets neg_carry to 0 if there is no carry from the addition of x and y, and -1 if there is a carry. This can be negated (using \_gsubil28(0,neg_carry)) if a 0 or 1 result is desired, or leave it as is and use a subtraction rather than addition when propagating the carry to subsequent operations.

2.2.2.4 Fractional Representation

The integer data types accommodate all types of fixed-point numeric representation. For example, the format can represent fractional numeration systems such as the S.15 (or Q1.15). In this format, the sign bit is bit 15 and there is an implied decimal point between bits 15 and 14. Bits 14 through 0 represent the fraction. The value represented has a range of -1/2 to almost 1/2 (precisely, 32767/65536) and is given by the formula:

\[
value = (-x_{15} + 2^{-1}x_{14} + 2^{-2}x_{13} + 2^{-3}x_{12} + 2^{-4}x_{11} + \ldots + 2^{-14}x_{1} + 2^{-15}x_{0})
\]

The integer add functions can be used to add fixed-point quantities, if the radix points of all operands occur at the same bit position in the operands. If the radix points occur at different positions in the operands, a shift operation must be used on one or more of the operands to bring their radix points in line with the those of the other operands. The Shift and Add functions (for example, \_gshliadd) can useful here.

For multiplication operations, the regular integer multiply functions produce double-width results in which the radix point is at the bit position that is the sum of the radix-point bit positions of the operands. Usually, the position of this resulting radix point needs to be adjusted—not only to bring it in line with the radix-point bit positions of the operands but also to reduce the width of the result to match the width of the operands. The Multiply Extract functions (for example, \_emulxi) are useful for this. A single function does the multiply and shift operations, and delivers the properly-rounded result.

2.2.3 Floating-Point Data Types

Figure 21 and Table 8 and Table 9 show the floating-point scalar data types. These include floating-point and complex floating-point scalars. They are a superset of ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic number formats.

Figure 22 and Table 10 and Table 11 show the fixed-point vector data types. These include vectors with elements of floating-point numbers or complex floating-point numbers. The vector-element sizes are powers of two. For more information about complex-number data types, see Section 2.2.4 on page 35.
Figure 21. Floating-Point Data Types

Table 8. Floating-Point Scalar Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Type in C</th>
<th>Size (Bits)</th>
<th>Bit Positions</th>
<th>IEEE 754 Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quad Precision</td>
<td>float128</td>
<td>128</td>
<td>127</td>
<td>126..112 (15 bits) 111..0 (112 bits)</td>
</tr>
<tr>
<td>Double Precision</td>
<td>float64</td>
<td>64</td>
<td>63</td>
<td>62..52 (11 bits) 51..0 (52 bits)</td>
</tr>
<tr>
<td>Single Precision</td>
<td>float32</td>
<td>32</td>
<td>31</td>
<td>30..23 (8 bits) 22..0 (23 bits)</td>
</tr>
<tr>
<td>Half Precision</td>
<td>float16</td>
<td>16</td>
<td>15</td>
<td>14..10 (5 bits) 9..0 (10 bits)</td>
</tr>
</tbody>
</table>
### Table 9. Floating-Point Scalar Complex Data Types

<table>
<thead>
<tr>
<th>Data Type in C</th>
<th>Size</th>
<th>Component</th>
<th>Sign (s)</th>
<th>Exponent (e)</th>
<th>Fraction (f)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cplx64</td>
<td>128</td>
<td>Imaginary</td>
<td>127</td>
<td>126..116</td>
<td>115..64</td>
<td>Complex floating-point number formed by two 64-bit double-precision components (real and imaginary).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>63</td>
<td>62..52</td>
<td>51..0</td>
<td></td>
</tr>
<tr>
<td>cplx32</td>
<td>64</td>
<td>Imaginary</td>
<td>63</td>
<td>62..55</td>
<td>54..32</td>
<td>Complex floating-point number formed by two 32-bit single-precision components (real and imaginary).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>31</td>
<td>30..23</td>
<td>22..0</td>
<td></td>
</tr>
<tr>
<td>cplx16</td>
<td>32</td>
<td>Imaginary</td>
<td>31</td>
<td>30..26</td>
<td>25..16</td>
<td>Complex floating-point number formed by two 16-bit half-precision components (real and imaginary).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>15</td>
<td>14..10</td>
<td>9..0</td>
<td></td>
</tr>
</tbody>
</table>
**Figure 22. Floating-Point Vector Data Types**

**Table 10. Floating-Point Vector Data Types**

<table>
<thead>
<tr>
<th>Data Type in C</th>
<th>Size (Bits)</th>
<th>Bit Positions</th>
<th>IEEE 754 Format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sign (s)</td>
<td>Exponent (e)</td>
</tr>
<tr>
<td>vf128_t</td>
<td>128</td>
<td>127</td>
<td>126..112 (15 bits)</td>
</tr>
<tr>
<td>vf64_t</td>
<td>128</td>
<td>127</td>
<td>126..116 (11 bits)</td>
</tr>
</tbody>
</table>
### Table 10. Floating-Point Vector Data Types (continued)

<table>
<thead>
<tr>
<th>Data Type in C</th>
<th>Size (Bits)</th>
<th>Bit Positions</th>
<th>IEEE 754 Format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sign(s)</td>
<td>Exponent (e)</td>
</tr>
<tr>
<td>vf32_t</td>
<td>128</td>
<td>127</td>
<td>126..119 (8 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>95</td>
<td>94..87 (8 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63</td>
<td>62..55 (8 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
<td>30..23 (8 bits)</td>
</tr>
<tr>
<td>vf16_t</td>
<td>128</td>
<td>127</td>
<td>126..122 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111</td>
<td>110..106 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>95</td>
<td>94..90 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>79</td>
<td>78..74 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63</td>
<td>62..58 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>47</td>
<td>46..42 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
<td>30..26 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>14..10 (5 bits)</td>
</tr>
</tbody>
</table>
### Table 11. Floating-Point Vector Complex Data Types

<table>
<thead>
<tr>
<th>Data Type in C</th>
<th>Size (Bits)</th>
<th>Component</th>
<th>Bit Positions Per Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sign (s)</td>
<td>Exponent (e)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Imaginary</td>
<td>127</td>
<td>126..116 (11 bits)</td>
</tr>
<tr>
<td>vcf64_t</td>
<td>128</td>
<td>Real</td>
<td>63</td>
<td>62..52 (11 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Imaginary</td>
<td>127</td>
<td>126..119 (8 bits)</td>
</tr>
<tr>
<td>vcf32_t</td>
<td>128</td>
<td>Imaginary</td>
<td>127</td>
<td>126..119 (8 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>95</td>
<td>94..87 (8 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Imaginary</td>
<td>63</td>
<td>62..55 (8 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>31</td>
<td>30..23 (8 bits)</td>
</tr>
<tr>
<td>vcf16_t</td>
<td>128</td>
<td>Imaginary</td>
<td>127</td>
<td>126..122 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>111</td>
<td>110..106 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Imaginary</td>
<td>95</td>
<td>94..90 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>79</td>
<td>78..74 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Imaginary</td>
<td>63</td>
<td>62..58 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>47</td>
<td>46..42 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Imaginary</td>
<td>31</td>
<td>30..26 (5 bits)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>15</td>
<td>14..10 (5 bits)</td>
</tr>
</tbody>
</table>
2.2.3.1 Discretionary Aspects of ANSI/IEEE Standard 754

The ANSI/IEEE standard 754-1985 leaves certain aspects of data format and operational details to the discretion of the implementation. These include production and propagation of quiet NaN (Not-a-Number) values. The BroadMX architecture handles the discretionary aspects of the floating-point formats as follows (the encoding is expressed in ANSI/IEEE standard 754-1985 notation):

- **Additional Precision Formats**—Adds Half Precision and Quad Precision formats.

- **Normalized Values**—Normalized number values are denoted by any sign-bit value, an exponent field that is not all 1 bits or all 0 bits, and any fraction field value. The numeric value encoded is \((-1)^s \times 2^{e-bias}(1.f)\). The bias is equal to the value resulting from setting to 1 all but the most-significant bit of the exponent field. The resulting biases are: half-precision: 15, single-precision: 127, double-precision: 1,023, and quad-precision: 16,383.

- **Denormalized Values**—Denormalized number values are denoted by any sign bit value, an exponent field that is all zero bits, and a non-zero fraction field value. The numeric value encoded is \((-1)^s \times 2^{1-bias}(0.f)\).

- **Zero Values**—Zero values are denoted by any sign bit value, an exponent field that is all zero bits, and a fraction field that is all zero bits. The numeric value encoded is \((-1)^s \times 0\). The distinction between +0 and -0 is significant in some operations.

- **Infinite Values**—Infinite values are denoted by any sign-bit value, an exponent field of all 1 bits, and a 0 fraction field. The numeric value encoded is \((-1)^s \times \infty\).

- **Quiet NaN (QNaN) and Signaling NaN (SNaN) Values**
  - Quiet NaN values are denoted by any sign-bit value, an exponent field of all one bits, and a non-zero fraction with the most-significant bit set. Quiet NaN values generated by default exception handling of standard operations have a 0 sign bit, an exponent field of all 1 bits, a fraction field with the most-significant bit set to 1, and all other bits cleared to 0.
  - Signaling NaN values are denoted by any sign-bit value, an exponent field of all 1 bits, and a non-zero fraction with the most-significant bit cleared to 0.

2.2.4 Complex Numbers

The complex-number data types are described in Section 2.2.2 on page 22 and Section 2.2.3 on page 29. Several functions (Section 1.3.3) are available to operate on complex numbers. Such numbers are assumed to be represented in rectangular coordinate format—data pairs that represent numerical values of the form \((a + i b)\).

There is no direct support for complex numbers that are represented in polar coordinate format—data pairs that represent numerical values of the form \((r, \phi)\), although the two coordinate formats can be converted to one another using the equations:
a = r \cos \varphi \\
b = r \sin \varphi \\

The paired rectangular-coordinate values are always arranged in registers with the real part in a less-significant location (to the right) and the imaginary part, in a more-significant location (to the left). When a vector represents a set of complex numbers, the real and imaginary parts of the numbers are considered two parts of a single vector element, as shown in Figure 23. Bit-ordering in registers is always little-endian.

**Figure 23. Complex Number Format in Registers**

<table>
<thead>
<tr>
<th>esize</th>
<th>esize</th>
<th>esize</th>
<th>esize</th>
</tr>
</thead>
<tbody>
<tr>
<td>imag.</td>
<td>real</td>
<td>imag.</td>
<td>real</td>
</tr>
<tr>
<td>complex number</td>
<td>complex number</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.2.5 Programming with Vector Data Types

The vector data types are illustrated in Figure 20 on page 25 and Figure 22 on page 32. A vector variable is declared using the vector data types. The fixed-point types are shown in Table 4 on page 26 and Table 5 on page 26. The floating-point types are shown in Table 10 on page 32 and Table 11 on page 34. The code below declares x to be a vector of 16-bit integers, and y to be a vector of complex integers with 16-bit real and 16-bit imaginary parts:

```c
v16_t x;
vc16_t y;
```

Vector variables always hold 128 bits of data. In the code above, x is a vector of 16-bit integers, which is composed of 128/16 = 8 separate 16-bit integers. The vector y has complex integer components, each having 16-bit real and imaginary parts, for a total width of 32 bits each. Therefore vector y contains 128/32 = 4 complex integers.

The macro `NELEM` simplifies the determination of how many elements are in a vector. It takes one parameter—the size in bits of the data type. For example `NELEM(16)` evaluates the number of elements in a vector of 16-bit integers or 16-bit floating-point numbers (namely, 8). `NELEMC(16)` evaluates to the number of elements in a vector of 16-bit complex numbers (namely, 4).

#### 2.2.5.1 Loading and Storing Vectors

The first step of programming using vector types to interface the common C language representation of vectors (usually as an array of elements) and the vector variables described above. BroadMX provides two utility functions that do this: `_lv` to load a vector variable from an array and `_sv` to store a vector variable into an array.
The _lv function takes an address as its only parameter, loads as many vector elements as will fit into a vector variable, and returns the vector as its result. The NELEM macro, described above, tells how many elements fit into the vector. An example of the use of _lv is shown here:

```c
v16_t x, y;
int16 array[24];
x = _lv16(array); /* load array[0] through array[7] into x */
y = _lv16(array+8); /* load array[8] through array[15] into y */
```

The _sv function takes an address as its first parameter and a vector variable as its second parameter. It stores the vector back to memory at the specified address. An example is shown below:

```c
v16_t x;
int16 array[24];
_svl6(array, x); /* store x into array[0] through array[7] */
```

As the first example suggests, arrays in C can be any size while vector variables are fixed size. This is handled by processing fixed-sized segments of the array using the vector functions, storing the results, and advancing to the next section of the array in a loop.

### 2.2.5.2 Writing Procedures

When writing procedures that operate on vector types, it is generally advisable to pass the vector parameters using a pointer to the corresponding scalar data type such as (int16 *). Use the vector load and store corresponding to the type, such as _lv16 and _svl6, inside the loop to process the data one vector at a time. This is as efficient as passing pointers to the vector types themselves, and has the added benefit that the code will work with other code that has not been modified to use BroadMX. The example below illustrates this point. It sums two arrays element-by-element and returns the result in a third array. Note the use of the NELEM macro to increment the loop-index by the number of elements processed in an iteration.

```c
array_add(int16 *result, int16 const *a1, int16 const *a2, int n) {
    int i;
    v16_t x, y, z;
    for(i = 0; i < n; i += NELEM(16)) {
        x = _lv16(&a1[i]);
        y = _lv16(&a2[i]);
        z = _gadd16(x, y);
        _svl6(&result[i], z);
    }
}
```
2.2.5.3  *Representation of Vector Types*

Array elements in memory are indexed starting with element 0, and higher numbered elements occupy higher addresses. This is the standard C convention.

Vector elements in registers are normally placed with element 0 occupying the low-order position (bit 0) in a register, with higher-numbered elements filling the register from right to left. The `lv` and `sv` functions fill the vector in this order.

Complex numbers in memory have the real part stored at the lower address and the imaginary part immediately following at the next higher address.

Complex vector elements in registers place the real part of element 0 in the low-order position (bit 0) and the imaginary part at the next higher position (starting at bit 16 for 16-bit complex types). The real part of element 1 is stored immediately following element 0 (starting at bit 32 for 16-bit complex types), and its imaginary part following that (at bit 48). The pattern repeats filling the register from right to left, real part first followed by imaginary.
exceptions, rounding, and limiting

Certain functions can signal an exception or modify their result by rounding or limiting when they encounter unexpected source parameters or produce an unexpected or non-useful result. In fixed-point functions, exceptions, rounding, and limiting are separate operations that can be separately specified by the choice of mnemonic syntax. In floating-point functions, however, exceptions and rounding are inter-related, and the signaling of exceptions may or may not cause the exception to be taken (trapped).

First, some definitions:

- **Exceptions**: An exception is said to be **signaled** or **occur** when the unexpected condition occurs; the exception is said to be **taken, raised or trapped** if it causes a jump to a service routine. All exceptions are precise. There are only two types of exception traps, **FixedPointArithmetic** and **FloatingPointArithmetic**:
  - **FixedPointArithmetic** traps occur when the result of certain fixed-point functions overflow.
  - **FloatingPointArithmetic** traps occur when the source or result of certain floating-point functions signals one of the IEEE-754 exceptions. These are: **invalid operation**, **division by zero**, **overflow**, **underflow**, or **inexact**.

- **Rounding**: Rounding occurs in one of four ways: round to **ceiling** (c), **floor** (f), **nearest** (n), or **zero** (z), as described in Table 12 on page 40. Certain fixed-point and floating-point functions can perform software-specified rounding, or they can perform default rounding, which is round-to-nearest.

- **Limiting and Overflow**: Limiting, also called saturation, is the mechanism by which fixed-point overflows cause the result to be clamped to the maximum or minimum integer value representable by the data type of the result, as shown in Table 4 on page 26. Floating-point functions have a default rounding mode that carries overflows to infinity with the sign of the intermediate result.

2.3.1 **Fixed-Point Modes**

Fixed-point functions have the following exception, rounding, and limiting modes:

- **Rounding**—Fixed-point functions with the words “and Halve” at the end of their descriptive name must have one mnemonic suffix {c, f, n, z} that specifies a rounding mode. Table 12 shows the fixed-point rounding modes.

  Fixed-point functions with the word “Extract” at the end of their descriptive name, and which use a register parameter for extract control (Section 3.5.11 on page 302), have one of these rounding modes specified in the control parameter. Immediate versions of the extract functions always use round-to-nearest by default.

- **Limiting**—Fixed-point functions with the words “with Limiting” at the end of their descriptive name limit the result on overflow, without trapping the exception.
• *Overflow Exceptions*—Fixed-point functions with the words “with Overflow Trap” at the end of their descriptive name (and an “o” suffix in their mnemonic) take a *FixedPointArithmetic* exception whenever there is an overflow. These are the *only* fixed-point functions that trap on an exception.

**Table 12. Rounding Modes**

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ceiling</td>
<td>c</td>
<td>Round toward positive infinity (+∞). For example, rounding the computed value 0.66 toward positive infinity equals 1. Rounding the computed value -1.5 toward positive infinity equals -1.</td>
</tr>
<tr>
<td>Floor</td>
<td>f</td>
<td>Round toward negative infinity (−∞). For example, rounding the computed value 0.66 toward negative infinity equals 0. Rounding the computed value -1.5 toward negative infinity equals -2.</td>
</tr>
<tr>
<td>Nearest (default)</td>
<td>n</td>
<td>Round to nearest representable value. Fractional remainders greater than 0.5 round up, remainders less than 0.5 round down, and remainders of exactly 0.5 round up or down, whichever provides an even result.</td>
</tr>
<tr>
<td>Zero</td>
<td>z</td>
<td>Round toward zero (0). For example, rounding the computed value 0.66 toward zero equals 0, as in floor rounding. Rounding the computed value -1.5 toward zero equals -1, as in ceiling rounding.</td>
</tr>
</tbody>
</table>

### 2.3.2 Floating-Point Modes

Certain floating-point functions may optionally have one suffix \{c, f, n, z, x\} in their mnemonic that specifies both rounding and IEEE exception-handling in an inter-related way:

- \{\}\: If no rounding-and-exception mode is specified for a floating-point function—indicated by the blank first element in the sequence \{c, f, n, z, x\}—round-to-nearest rounding is done, and all exception handling conforms to the IEEE default mode, in which the result is returned without a trap.

- \{c, f, n, z\}: These options cause the specified rounding to be done, and they also cause any floating-point exception—except the *inexact* exception—to be trapped. That is, the option rounds as specified, and it traps on an *invalid operation*, *division by zero*, *overflow*, or *underflow* exception.

- \{x\}: This option causes any floating-point exception—including the *inexact* exception—to be trapped. Functions that offer only this option, and not the rounding-mode options, always produce exact results and thus need no rounding.
The _esinkf function can use any of the modes described above or additional rounding modes that are specific to it:

- \{cd, fd, zd\}: These options cause the specified rounding to be done (c, f, or z), and all exception handling conforms to the IEEE default mode, in which the result is returned without a trap.

Floating-point exceptions are described in greater detail in Section 2.4.2 on page 44.
2.4  **Floating-Point Conventions**

The architecture and supporting software provides all floating-point facilities required and recommended by *ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic*. Floating-point data types are described in Section 2.2.3 on page 29. This section describes conventions used for floating-point arithmetic.

The vector and matrix floating-point functions support the common floating-point arithmetic operations, including add, subtract, multiply, and format conversions among the floating-point formats and between floating-point and binary integer formats. Divide and square root are provided either in hardware or by software libraries depending on chip implementation. Software libraries also provide the other operations required by the ANSI/IEEE standard, for example conversion between binary and decimal format.

The floating-point vector functions explicitly specify the precision of the operation (16, 32, 64, or 128 bits), and they round the result (or check that the result is exact) to the specified precision at the conclusion of each operation. Each function performs its operation corresponding elements of its vector parameters.

In addition the basic functions, BroadMX supplies floating-point vector functions that sum one or more products to each other and/or to an additional operand. These functions include fused multiply-add (_emuladd...), convolve (_econ...), scale-add (_escaladd...), and vector-matrix multiply (_wmulmat...). The results of these functions are computed as if the multiplies and adds have infinite precision and then rounded only once. There is no rounding of intermediate results that could limit the result accuracy.

2.4.1  **Floating-Point Compare and Set, and Branch Conditional**

The ANSI/IEEE standard specifies four mutually exclusive comparison relations between floating-point numbers of any format (even when the compared operands' formats differ): *less-than*, *equal*, *greater-than*, and *unordered*. A NaN (Not-a-Number) value compares as *unordered* with respect to any other value, even that of an identical NaN. The BroadMX floating-point compare-and-set functions augment these comparison types with special handling for NaN (not-a-number) values: these functions may optionally generate an exception on comparisons involving quiet or signaling NaNs.

Table 13 shows the compare-and-set relations. Table 14 shows the floating-point compare-branch relations.
The floating-point compare-and-branch functions do not generate exceptions on comparisons involving NaN values (either quiet or signaling). If such exceptions are desired, a floating-point compare-and-set function with exceptions enabled can be inserted in front of the branch.

The missing less-than-or-equal (le) condition for both compare-and-set and compare-and-branch can be obtained by using the supplied greater-than-or-equal condition and reversing the order of the parameters. That is, \( x \leq y \) is expressed as \( y \geq x \). Similarly the greater (g) condition is derived from the less (l) condition by reversing the parameters.
The equal condition (e) can be used to determine the *unordered* condition of a single operand by comparing the operand with itself. It will compare True if the parameter is a number (or infinity), and False if it is a NaN. No functions are provided that branch when values compare as *unordered*. To accomplish such an operation, use the reverse comparison relation to branch over an immediately following unconditional branch or, in the case of an if-then-else clause, reverse the clauses and use the reverse condition.

### 2.4.2 Floating-Point Rounding and Exceptions

Certain floating-point functions may optionally have one suffix \{, c, f, n, z, x\} in their mnemonic that specifies both rounding and ANSI/IEEE exception-handling in an interrelated way, as described in Section 2.3.2 on page 40. Rounding—to ceiling, floor, nearest, or zero—is explicitly specified within these floating-point functions in order to avoid the need for explicit rounding-mode state registers. Similarly, the handling of standard exceptions—*invalid operation, division by zero, overflow, underflow, and inexact*—is also explicitly specified within functions.

When no rounding is explicitly specified in the function, round-to-nearest rounding is performed by default, and all floating-point exceptions produce the ANSI/IEEE standard default result, which is to proceed without a trap. When rounding is explicitly specified with the \(c, f, n,\) or \(z\) suffix, the rounding is performed and floating-point exceptions (other than *inexact*) cause a floating-point exception trap. When the \(x\) suffix is specified, any floating-point exception signal, including the *inexact* exception, cause a floating-point exception trap.

This rounding and exception technique assists processors in executing BroadMX operations with greater parallelism. When default rounding and exception-handling is specified, the architecture may safely retire results out of order, because they are guaranteed not to cause data-dependent exceptions. Similarly, floating-point functions specified with nearest, zero, floor, or ceiling rounding can be guaranteed not to cause data-dependent exceptions after the operands have been examined to rule out invalid operations, division by zero, overflow, or underflow exceptions. Only floating-point functions specified with the \(x\) suffix or functions specified with \(c, f, n,\) or \(z\) suffix for which exceptions cannot be ruled out—need to avoid retiring subsequent functions until the final result is generated.

The ANSI/IEEE standard specifies information to be given to trap handlers for the five floating-point exceptions. The BroadMX instruction-set architecture produces precise exceptions: the program counter points to the machine instruction that caused the exception and the entire register state is preserved. From these precise exceptions, all of the information required for recovery is available to software, because all source-operand values and the specified operation are available.
The ANSI/IEEE standard specifies a set of five *status flags*, for recording the occurrence of exceptions that are handled by default. The BroadMX architecture does not produce status flags, but software can generate and maintain corresponding status flags from the exception information provided. The exception handler can execute the same function with default control to compute the default result without causing further exceptions.

The BroadMX architecture defines many compound floating-point operations that are not covered in the scope of the ANSI/IEEE standard. Floating-point convolve and vector-matrix multiply are examples of these. These compound operations provide round-to-nearest and default exception handling (i.e., proceed without a trap) as the only rounding and exception option.

### 2.4.2.1 Invalid Operation Exception

The ANSI/IEEE standard specifies that the invalid operation exception shall be signaled if an operand is invalid for the operation to be performed. An example of an invalid operation is subtracting infinity from itself. BroadMX functions that specify a non-default rounding mode will trap on invalid operation. BroadMX functions that use the default rounding mode (round-to-nearest) do not trap on invalid operation and produce a quiet NaN result as described in Section 2.4.3 on page 46..

### 2.4.2.2 Division By Zero Exception

The ANSI/IEEE standard specifies that the division by zero exception shall be signaled if the divisor is zero and the dividend is a finite non-zero number. BroadMX functions that specify a non-default rounding mode will trap on division by zero. BroadMX functions that use the default rounding mode (round-to-nearest) do not trap on division by zero and produce a signed infinity result.

### 2.4.2.3 Overflow Exception

The ANSI/IEEE standard specifies that the overflow exception shall be signaled whenever the destination format’s largest finite number (either positive or negative) is exceeded in magnitude by what would have been the rounded floating-point result, if the exponent range were unbounded.

BroadMX functions that specify a non-default rounding mode will trap on overflow. BroadMX functions that use the default rounding mode (round-to-nearest) do not trap on overflow and produce a result that carries all overflows to infinity with the sign of the intermediate result.
2.4.2.4  *Underflow Exception*

The ANSI/IEEE standard specifies that the underflow exception is dependent on two correlated events—tininess and loss of accuracy—but it allows some latitude in the definition of these conditions. For BroadMX functions, tininess is detected after rounding—that is, when a non-zero result computed as though the exponent range were unbounded would be less than the smallest normalized number for the format of the result. The BroadMX architecture does not produce floating-point status flags, so the notion of loss-of-accuracy does not apply.

BroadMX functions that specify a non-default rounding mode will trap on underflow, which is signaled whenever tininess occurs. BroadMX functions that use the default rounding mode (round-to-nearest) do not trap on underflow and produce a result that is zero or a denormalized number.

2.4.2.5  *Inexact Exception*

The ANSI/IEEE standard specifies that the inexact exception be signaled whenever the rounded result of an operation is not exact or if it overflows without an overflow trap. BroadMX functions that specify exact rounding (the optional “x” suffix in a function’s mnemonic) trap on inexact. BroadMX functions that use the default rounding mode (round-to-nearest) or specify a rounding mode do not trap on inexact and produce a rounded or overflowed result.

2.4.3  *NaN Handling*

There are a number of situations involving NaNs that the ANSI/IEEE standard leaves unspecifed. This section describes how BroadMX handles the operations left unspecified by the standard.

The ANSI/IEEE standard specifies that quiet NaN values should be propagated from operand to result by the basic operations (add, subtract, etc.). However, it fails to specify which of several quiet NaN values to propagate when more than one operand is a quiet NaN. It also does not specify how quiet and signaling NaN values are propagated though format conversions and the absolute-value, negate, and copy operations. Also unspecified is which specific quiet NaN value results from an invalid operation when exceptions are not taken.

Functions that produce a floating-point result and do not trap invalid operations propagate signaling NaN values from operands to results. The signaling NaN values are changed to quiet NaN values by setting the most-significant fraction bit and leaving the remaining bits unchanged. Other causes of invalid operations produce the default quiet NaN value, in which the sign bit is zero, the exponent field is all one bits, the most-significant fraction bit is set to 1, and the remaining fraction bits are all zero bits.
For floating-point functions that produce vector results, quiet and signaling NaN propagation is handled separately and independently for each element in the result. A quiet or signaling NaN value in a single element of an operand causes only those result elements that are dependent on that operand element’s value to be propagated as that quiet NaN. Multiple quiet or signaling NaN values in elements of an operand which influence separate elements of the result are propagated independently of each other. Any signaling NaN that is propagated is first converted to a quiet NaN by setting its high-order fraction bit to 1.

For results in which more than one operand is a quiet or signaling NaN, a priority rule determines which NaN is propagated. The rule is based on register fields of the instruction encoding. Priority is given to the operand residing in the register specified at a lower-numbered (little-endian) bit position within the instruction opcode. For operands which are concatenated from two registers, priority is assigned based on the register with highest priority. When there is a tie (as when an _escaladdf scaling operand has two corresponding NaN values, or when an _emulcf operand has NaN values for both real and imaginary components of a value), the value located at a lower-numbered (little-endian) bit position within the instruction opcode receives priority. The identification of a NaN as quiet or signaling does not confer any priority for selection. Only the operand’s position confers priority, although a signaling NaN will cause an invalid operand exception.

The sign bit of propagated NaN values is complemented if the function subtracts or negates the corresponding operand or multiplies it by, or divides it by, or divides it into, an operand that has the sign bit set, even if that operand is another NaN. If a NaN is both subtracted and multiplied by a negative value, the sign bit is propagated unchanged.

The _einflatef and _edeflatef functions convert between two floating-point formats. NaN values are propagated by preserving the sign and the most-significant fraction bits, and changing signaling NaN to quiet NaN by setting the most-significant bit to 1. For _edeflatef, the least-significant fraction bit preserved is combined, via a logical-OR of all fraction bits not preserved. For _einflatef, all additional fraction bits are cleared to zero.

The _esinkf function converts from a floating-point format to a fixed-point format. NaN values produce zero values (maximum-likelihood estimate). Infinity values produce the largest representable positive or negative fixed-point value that fits in the destination field. When a function is used that traps exceptions, NaN or infinity values produce a floating-point exception. Underflows do not occur in the _esinkf function; instead, the function produces –1, 0 or +1 for small operands, depending on rounding controls.
For absolute-value, negate, or copy functions, NaN values are propagated with the sign bit cleared to 0, complemented, or copied, respectively. Signalling NaN values are converted to quiet NaN values by setting the most significant fraction bit to 1, or causing the invalid operation exception if specified by the function.
2.5 **Procedures and Calling Conventions**

Procedure calls do not necessarily involve a memory access. Some leaf procedures (those that call no other procedure) can be compiled so that the saving and restoring of register values does not involve a memory access. This ability keeps call overhead low, assuming that the target address of the call branch hits in the cache.

BroadMX functions are not procedure calls, and BroadMX compilers do not turn leaf procedures into non-leaf procedures. If a procedure call is in-lined, by declaring it "INLINE", the compiler does not generate a procedure call, and what remains is a leaf procedure. The INLINE macro expands to the correct directive for the compiler being used.

2.6 **Performance Optimization**

BroadMX functions are designed to accelerate the performance of algorithms with the following characteristics:

- **Vectorizable Operations**—Performance is best for operations on sequentially ordered data items that are independent and identical in size. Such order data items are viewed as *vectors* of same-size elements, or *frames* of same-size data samples. Operations on such vector elements occur in parallel.

- **Low-Precision Arithmetic**—Low-precision arithmetic provides the opportunity for single functions to operate in parallel on many sequential data items. The architecture defines vector (or frame) data types with elements (or samples) of between 1 and 256 bits. Data vectors that contain smaller (lower-precision) elements offer greater parallelism, and thus, better performance.

- **Sequential Access To Data**—Performance is best for operations on sequentially ordered items in memory, which permit normal load and store operations. 64- and 128-bit loads and stores reference several sequential items of data, the number depending on the operand precision.

- **Fixed-Point Advantages**—Although the architecture supports both fixed-point and floating-point arithmetic, fixed-point operations have certain advantages over floating-point operations. Commutativity, associativity, and distribution identities can be used to reorder fixed-point operations, and fixed-point adds, subtracts, and shifts may be specified to trap on various conditions.

- **Unconditional Arithmetic and Logical Operations**—Conditional functions, such as `_gmux`, avoid branches. They do this by computing both alternatives for all elements in a vector, and then selecting once for each element. Since the selections are independent for all elements, they occur in parallel. A `_gmux` function that compares 128 bits, for example, has 128 elements and can avoid 128 branches.
3 Vector and Matrix Functions

3.1 Introduction

BroadMX functions fall into two sets: vector and matrix functions, which perform vector and matrix operations, and Scalar (or access) functions, that access memory and perform all standard scalar operations except multiply. The vector and matrix functions include Group, Crossbar, Ensemble, and Wide functions, which are described in the pages that follow. The scalar functions are automatically generated by the compiler from normal C expressions and do not need special functions to use them. They are described in the BroadMX System Architecture Reference Manual.

3.1.1 Notation

The notation used in this chapter is summarized in Tables 15 through 18, and Figure 24. For complete definitions of terms, see Appendix A "Glossary" on page 339.

Table 15. Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>A one-dimensional array of data elements, each of which can be located with an array index.</td>
</tr>
<tr>
<td>esize</td>
<td>The size, in bits, of one element in a vector of elements. The value of esize is defined for a given function. If an integer appears in a function’s mnemonic (e.g., _econf32), it is that function’s esize. A few functions, however, define esize in a control value. In most functions, esize is the size of source-vector elements. However, in the case of _xexpand, esize is the size of result-vector elements, and in the cases of some _emuladd.. and _emulsub.. functions, different source vectors have different element sizes (esize or 2*esize).</td>
</tr>
<tr>
<td>instruction</td>
<td>An assembly or machine instruction, as opposed to a C function. BroadMX instructions are described in the BroadMX System Architecture Reference Manual, available from MicroUnity.</td>
</tr>
</tbody>
</table>
Table 15. Terms (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>One of the following rounding modes:</td>
<td></td>
</tr>
<tr>
<td>• c—Ceiling rounding. Numbers are rounded up towards +∞.</td>
<td></td>
</tr>
<tr>
<td>• f—Floor rounding. Numbers are rounded down towards -∞.</td>
<td></td>
</tr>
<tr>
<td>• n—Nearest rounding. Numbers are rounded to the nearest representable number, rounding down when the fraction is less than 1/2 and rounding up when the fraction is greater than 1/2. When the fraction is exactly 1/2, the nearest even number is chosen.</td>
<td></td>
</tr>
<tr>
<td>• z—Zero rounding. Numbers are rounded toward zero: positive numbers round down while negative numbers round up.</td>
<td></td>
</tr>
<tr>
<td>See “rounding and/or exception handling”, immediately below, for further details.</td>
<td></td>
</tr>
<tr>
<td>A rounding and/or exception-handling mode can optionally be specified in the mnemonic of certain functions by including one or two characters as a suffix to the mnemonic. The following character choices are available to define rounding and exception-handling modes:</td>
<td></td>
</tr>
<tr>
<td>• {c, f, n, z}—Specifies a rounding and exception-handling mode. Specifying any one of these modes causes both rounding (ceiling, floor, nearest, or zero) and trapping on any exception, except the floating-point inexact exception. The first comma in the sequence indicates that this option may be empty (i.e., no option selected).</td>
<td></td>
</tr>
<tr>
<td>• {x}—Specifies trapping on any floating-point exception, including the floating-point inexact exception. This option appears either alone, as {x}, or with rounding modes, as {c, f, n, z, x}.</td>
<td></td>
</tr>
<tr>
<td>• {cd, fd, zd}—Specifies a rounding mode (ceiling, floor, or zero) and default floating-point exception handling (in which exceptions are not trapped). Valid only with the _esinkf function.</td>
<td></td>
</tr>
<tr>
<td>• o—Specifies trapping on overflow exceptions (valid only in certain fixed-point functions). The overflows may be positive or negative.</td>
<td></td>
</tr>
<tr>
<td>floor(x)</td>
<td>The largest integer less than or equal to x.</td>
</tr>
<tr>
<td>set</td>
<td>With respect to a bit, set to the value 1.</td>
</tr>
<tr>
<td>clear</td>
<td>With respect to a bit, clear to the value 0.</td>
</tr>
</tbody>
</table>
### Table 16. Scalar Data Types

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>int128, ... , int8</td>
<td>A scalar integer of the specified bit size.</td>
</tr>
<tr>
<td>int128*, ... , int8*</td>
<td>A pointer to an array of scalar integers of the specified bit size.</td>
</tr>
<tr>
<td>float128, ... , float16</td>
<td>A floating-point number of the specified bit size.</td>
</tr>
<tr>
<td>float128*, ... , float16*</td>
<td>A pointer to an array of floating-point numbers of the specified bit size.</td>
</tr>
<tr>
<td>cplxi64, ... , cplxi8</td>
<td>A complex integer containing a real part and an imaginary part, each of the specified size.</td>
</tr>
<tr>
<td>&lt;scalar-type&gt; const *</td>
<td>Same as the pointer type without const, but signifying that the memory pointed to will be read but never written.</td>
</tr>
</tbody>
</table>

### Table 17. Vector Data Types

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hexlet_t</td>
<td>A 128-bit fixed-point vector of ( n ) ( k )-bit elements, as specified by a control parameter.</td>
</tr>
<tr>
<td>v128_t, ... , v1_t</td>
<td>A 128-bit fixed-point vector data type containing elements of the specified bit size.</td>
</tr>
<tr>
<td>vu128_t, ... , vu1_t</td>
<td>A 128-bit fixed-point unsigned vector data type containing elements of the specified bit size.</td>
</tr>
<tr>
<td>vc64_t, ... , vc8_t</td>
<td>A 128-bit complex integer vector data type containing elements of the specified bit size.</td>
</tr>
<tr>
<td>vf128_t, ... , vf16_t</td>
<td>A 128-bit floating-point vector data type containing elements of the specified bit size.</td>
</tr>
<tr>
<td>vcf64_t, ... , vcf16_t</td>
<td>A 128-bit complex floating-point vector data type containing elements of the specified bit size.</td>
</tr>
</tbody>
</table>

### Table 18. Symbols

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>literals</td>
<td>Within function-syntax expressions, literal characters appear in bold fixed-pitch font—for example, <code>xextract</code>.</td>
</tr>
<tr>
<td>variables</td>
<td>Within function-syntax expressions, variables appear in italic fixed-pitch font—for example, <code>x, y, z, w, or m</code>.</td>
</tr>
<tr>
<td><code>{,c,f,n,z,x,cd,fd,zd}</code></td>
<td>See “rounding and/or exception handling”, above.</td>
</tr>
<tr>
<td><code>{,x}</code></td>
<td>See “rounding and/or exception handling”, above.</td>
</tr>
</tbody>
</table>
### Table 18. Symbols (continued)

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>A prefix indicating a hexadecimal value.</td>
</tr>
<tr>
<td>x</td>
<td>A scalar non-immediate constant value that is used as a source operand. Depending on the compiler, these operands are normally in registers.</td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
</tr>
<tr>
<td>ctrl</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>A scalar immediate constant value, determined by the compiler at compile time.</td>
</tr>
<tr>
<td>sh</td>
<td></td>
</tr>
<tr>
<td>fsize</td>
<td></td>
</tr>
<tr>
<td>x[i]</td>
<td>The (i^{th}) element of a vector, whose elements are (esize) bits wide, that is used as a source operand. Unless otherwise stated, the index value (i = 0..NELEM -1).</td>
</tr>
<tr>
<td>y[i]</td>
<td></td>
</tr>
<tr>
<td>z[i]</td>
<td></td>
</tr>
<tr>
<td>w[i]</td>
<td></td>
</tr>
<tr>
<td>m[i]</td>
<td></td>
</tr>
<tr>
<td>r[i]</td>
<td>The (i^{th}) element of a vector, whose elements are (esize) bits wide, that is the returned result of a function.</td>
</tr>
<tr>
<td>i..j</td>
<td>Bit range, bit i to bit j, inclusive. For example, 127..0 means bits 127 through 0, inclusive.</td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>x(^i)</td>
<td>Binary digit x repeated, concatenated (i) times. Size of result is (i).</td>
</tr>
<tr>
<td>x(_i)</td>
<td>Extraction of bit (i) (using little-endian bit numbering) from value (x). Result is a single bit.</td>
</tr>
<tr>
<td>x(_{i..j})</td>
<td>Extraction of bit field formed from bits (i) through (j) of value (x). Size of result is (i-j+1); if (j&gt;i), result is an empty string.</td>
</tr>
<tr>
<td>x + y</td>
<td>Signed or unsigned addition of (x) and (y). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>x +(^l) y</td>
<td>Signed addition of (x) and (y), with limiting ((l)). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>x +(^ul) y</td>
<td>Unsigned ((u)) addition of (x) and (y), with limiting ((l)). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>x +(^o) y</td>
<td>Signed addition of (x) and (y), with trap on overflow ((o)). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>x +(^uo) y</td>
<td>Unsigned ((u)) addition of (x) and (y), with trap on overflow ((o)). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
</tbody>
</table>
### Table 18. Symbols (continued)

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r = \sum_{j=0}^{n} x[j] )</td>
<td>Sum of ( x ) over the range ( j ). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x - y )</td>
<td>Signed or unsigned subtraction of ( y ) from ( x ). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x \cdot^o y )</td>
<td>Signed subtraction of ( y ) from ( x ), with trap on overflow (o). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x \cdot^{uo} y )</td>
<td>Unsigned (u) subtraction of ( y ) from ( x ), with trap on overflow (o). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x \cdot^l y )</td>
<td>Signed subtraction of ( x ) and ( y ), with limiting (l). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x \cdot^{ul} y )</td>
<td>Unsigned (u) subtraction of ( x ) and ( y ), with limiting (l). Result is the same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x \cdot^y )</td>
<td>Signed multiplication or floating-point multiplication of ( x ) and ( y ). Result is double the size of the source operands, and operands must be of equal size. Used to mean floating-point multiplication when the arguments are floating-point.</td>
</tr>
<tr>
<td>( x \cdot^m y )</td>
<td>Mixed-sign multiplication of unsigned ( x ) by signed ( y ). Result is double the size of the source operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x \cdot^p y )</td>
<td>Polynomial multiplication of ( x ) and ( y ). Result is double the size of the source operands, and operands must be of equal size. See page 211 for a description of polynomial multiplication.</td>
</tr>
<tr>
<td>( x \cdot^u y )</td>
<td>Unsigned multiplication of ( x ) and ( y ). Result is double the size of the source operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x / y )</td>
<td>Signed division or floating-point division of ( x ) by ( y ). Result is the same size as the operands, and operands must be of equal size. Used to mean floating-point division when the arguments are floating-point.</td>
</tr>
<tr>
<td>( x /^u y )</td>
<td>Unsigned integer division of ( x ) and ( y ). Result is the same size as the operands, and operands must be of equal size. The unsigned division of ( x ) by ( y ) is equal to floor(( x/y )).</td>
</tr>
<tr>
<td>( x &amp; y )</td>
<td>Bit-wise AND of ( x ) and ( y ). Result is same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x \mid y )</td>
<td>Bit-wise OR of ( x ) and ( y ). Result is same size as the operands, and operands must be of equal size.</td>
</tr>
</tbody>
</table>
### Table 18. Symbols (continued)

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ^ y )</td>
<td>Bit-wise exclusive-OR of ( x ) and ( y ). Result is same size as the operands, and operands must be of equal size.</td>
</tr>
<tr>
<td>( \sim x )</td>
<td>Bit-wise inversion of ( x ) (NOT ( x )). Result is same size as the operand.</td>
</tr>
<tr>
<td>( x = y )</td>
<td>Signed equality comparison between ( x ) and ( y ). Result is an element equal to -1 if true, else 0 if false, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x != y )</td>
<td>Signed inequality comparison between ( x ) and ( y ). Result is an element equal to -1 if true, else 0 if false, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x &lt; y )</td>
<td>Signed less-than comparison between ( x ) and ( y ). Result is an element equal to -1 if true, else 0 if false, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x &gt; y )</td>
<td>Signed greater-than comparison between ( x ) and ( y ). Result is an element equal to -1 if true, else 0 if false, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x &lt;&gt; y )</td>
<td>Signed greater-than or less-than comparison between ( x ) and ( y ). Result is an element equal to -1 if true, else 0 if false, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x &lt;0 )</td>
<td>Signed less-than comparison between ( x ) and 0. Result is an element equal to -1 if true, else 0 if false.</td>
</tr>
<tr>
<td>( x &gt;0 )</td>
<td>Signed greater-than comparison between ( x ) and 0. Result is an element equal to -1 if true, else 0 if false.</td>
</tr>
<tr>
<td>( x &gt;= y )</td>
<td>Signed greater-than-or-equal comparison between ( x ) and ( y ). Result is an element equal to -1 if true, else 0 if false, and operands must be of equal size.</td>
</tr>
<tr>
<td>( x &lt;=0 )</td>
<td>Signed less-than-or-equal comparison between ( x ) and 0. Result is an element equal to -1 if true, else 0 if false.</td>
</tr>
<tr>
<td>( x &gt;=0 )</td>
<td>Signed greater-than-or-equal comparison between ( x ) and 0. Result is an element equal to -1 if true, else 0 if false.</td>
</tr>
<tr>
<td>( x &lt;&lt; y )</td>
<td>Signed or unsigned left shift of ( x ) by ( y ) bit-positions. Result is the size of ( x ).</td>
</tr>
<tr>
<td>( x &lt;&lt;0 ) ( y )</td>
<td>Signed left shift of ( x ) by ( y ) bit-positions, with trap on overflow (( o )). Result is the size of ( x ).</td>
</tr>
<tr>
<td>( x &lt;&lt;uo ) ( y )</td>
<td>Unsigned (( u )) left shift of ( x ) by ( y ) bit-positions, with trap on overflow (( o )). Result is the size of ( x ).</td>
</tr>
<tr>
<td>( x &gt;&gt; y )</td>
<td>Signed right shift of ( x ) by ( y ) bit-positions. Result is the size of ( x ).</td>
</tr>
<tr>
<td>( x &gt;&gt;u ) ( y )</td>
<td>Unsigned (( u )) right shift of ( x ) by ( y ) bit-positions, with trap on overflow (( o )). Result is the size of ( x ).</td>
</tr>
</tbody>
</table>
Table 18. Symbols (continued)

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x mod y</td>
<td>For positive y, the positive remainder of x that is less than y.</td>
</tr>
<tr>
<td>x mod_p y</td>
<td>The polynomial remainder of the polynomial division of x by y.</td>
</tr>
<tr>
<td>(\sqrt{x})</td>
<td>Floating-point square root of x.</td>
</tr>
<tr>
<td>x?y:z</td>
<td>value of y, if x is true, otherwise value of z. Value of x is a single bit (true = 1, false = 0).</td>
</tr>
</tbody>
</table>
Figure 24. Syntax of Function Mnemonics

**Function Type**
- g = Group function
- x = Crossbar function
- e = Ensemble function
- w = Wide function

**Immediate**

**Data Type**
- u = unsigned integer
- c = complex integer
- f = floating-point
- cf = complex floating-point
- none = signed integer (see Notes, below)

**Vector-Element Size (esize)**

**Rounding, Exception-Handling, and/or High-Low Half Result Selector**

*Fixed-point optional modes (shown in curly braces) that can be added to certain mnemonics:*
- c = round to ceiling, and trap on any but inexact exceptions
- f = round to floor, and trap on any but inexact exceptions
- n = round to nearest, and trap on any but inexact exceptions
- z = round to zero, and trap on any but inexact exceptions
- o = trap on overflow exceptions

*Floating-point optional modes (shown in curly braces) that can be added to certain mnemonics:*
- c = round to ceiling, and trap on any but inexact exceptions
- f = round to floor, and trap on any but inexact exceptions
- n = round to nearest, and trap on any but inexact exceptions
- z = round to zero, and trap on any but inexact exceptions
- x = trap on any exception
- cd = round to ceiling, and default floating-point exception handling (valid only in esinkf function)
- fd = round to floor, and default floating-point exception handling (valid only in esinkf function)
- zd = round to zero, and default floating-point exception handling (valid only in esinkf function)
- hi = high-half result selector
- lo = low-half result selector

**Notes:**
If there is no data type designation, the data type is signed integer, if a sign matters. If a sign does not matter, the operation usually works for both signed and unsigned source operands.
3.2 **Support Functions and Macros**

Support functions and macros perform operations commonly needed to set up Broad-band vector and matrix operations. These include creating, loading, and storing vectors.

The Support functions and macros include the following types:

- Create Vector
- Load and Store Vector
- Manipulate Vector
- Complex Numbers
3.2.1  **Create Vector**

The Create Vector functions include:

- **NELEM**  Number of Elements
- **VCONST**  Create Vector Constant
- **_vector**  Create Vector with Specified Values
- **_vall**  Create Vector with Identical Components
- **_velem**  Create Vector with First Component Specified

These functions support creation of 128-bit vectors that have identical or non-identical elements, including creation of vectors representing sets of complex numbers.
NELEM \hspace{1cm} Number of Elements

Returns the number of elements in a vector.

\begin{verbatim}
int NELEM(esize)
\end{verbatim}

This macro returns the number of elements in a vector of \textit{esize}-bit elements, which is equal to \textit{128/esize}.

The function is useful in loops, such as:

\begin{verbatim}
for(p=data; p < end; p += NELEM(8))
\end{verbatim}
**VCONST**

Create Vector Constant

Initializes the elements of a statically-declared vector with scalar constant data.

- `v8_t VCONST8(int x0, int x1, ..., int x15)`
- `v16_t VCONST16(int x0, int x1, ..., int x7)`
- `v32_t VCONST32(int x0, int x1, int x2, int x3)`
- `v64_t VCONST64(int x0, int x1)`
- `v128_t VCONST128(int x0)`
- `vf16_t VCONSTF16(double x0, double x1, ..., double x7)`
- `vf32_t VCONSTF32(double x0, double x1, double x2, double x3)`
- `vf64_t VCONSTF64(double x0, double x1)`
- `vf128_t VCONSTF128(double x0)`
- `vc8_t VCONSTC8(int r0, int i0, ..., int r7, int i7)`
- `vc16_t VCONSTC16(int r0, int i0, ..., int r3, int i3)`
- `vc32_t VCONSTC32(int r0, int i0, int r1, int i1)`
- `vc64_t VCONSTC64(int r0, int i0)`
- `vcf16_t VCONSTCF16(double r0, double i0, ..., double r3, double i3)`
- `vcf32_t VCONSTCF32(double r0, double i0, double r1, double i1)`
- `vcf64_t VCONSTCF64(double r0, double i0)`

This macro takes NELEM scalar constant parameters, \(x_0\) through \(x_{<\text{NELEM}-1}\). The constants are concatenated, producing a 128-bit result vector of esize-bit elements.

This macro is used to initialize statically-declared or global vector data. The related function, `_vector`, creates vectors from any type of scalar, whether or not constant, but cannot be used to initialize static or global data.

\[
\begin{align*}
\mathbf{r}[i] &= x_i, & i &= 0..\text{NELEM} - 1
\end{align*}
\]
_vector

Create Vector with Specified Values

Make a vector from its scalar components.

\[
\begin{align*}
v8\_t &\quad _\text{vector}8(\text{int } x0, \text{int } x1, \ldots, \text{int } x15) \\
v16\_t &\quad _\text{vector}16(\text{int } x0, \text{int } x1, \ldots, \text{int } x7) \\
v32\_t &\quad _\text{vector}32(\text{int } x0, \text{int } x1, \ldots, \text{int } x3) \\
v64\_t &\quad _\text{vector}64(\text{int } x0, \text{int } x1) \\
v128\_t &\quad _\text{vector}128(\text{int } x0) \\
vf16\_t &\quad _\text{vectorf}16(\text{double } x0, \text{double } x1, \ldots, \text{double } x7) \\
vf32\_t &\quad _\text{vectorf}32(\text{double } x0, \text{double } x1, \ldots, \text{double } x3) \\
vf64\_t &\quad _\text{vectorf}64(\text{double } x0, \text{double } x1) \\
vf128\_t &\quad _\text{vectorf}128(\text{double } x0) \\
v8\_t &\quad _\text{vectorc}8(\text{int } re0, \text{int } im0, \ldots, \text{int } re7, \text{int } im7) \\
v16\_t &\quad _\text{vectorc}16(\text{int } re0, \text{int } im0, \ldots, \text{int } re3, \text{int } im3) \\
v32\_t &\quad _\text{vectorc}32(\text{int } re0, \text{int } im0, \text{int } re1, \text{int } im1) \\
v64\_t &\quad _\text{vectorc}64(\text{int } re0, \text{int } im0) \\
v16\_t &\quad _\text{vectorcf}16(\text{double } re0, \text{double } im0, \ldots, \text{double } re3, \text{double } im3) \\
v32\_t &\quad _\text{vectorcf}32(\text{double } re0, \text{double } im0, \text{double } re1, \text{double } im1) \\
v64\_t &\quad _\text{vectorcf}64(\text{double } re0, \text{double } im0)
\end{align*}
\]

This macro takes NELEM scalar parameters, \(x0 \text{ through } x^{<\text{NELEM}-1}\). The scalars can be of any type (constants, variables, etc.). The scalars are concatenated, producing a 128-bit result vector of esize-bit elements.

For a specialized version of this macro, see the VCONST macro which creates vectors from constants.

\[
x0, x1, \ldots, x^{<\text{NELEM}-1}
\]

\[
\begin{array}{c|c|c|c|c}
\hline
\text{element} & \text{element} & \text{element} & \text{element} \\
\hline
\text{esize} & \text{esize} & \text{esize} & \text{esize} \\
\hline
127 & r & 0
\end{array}
\]

\[
r[i] = x_i, \quad i = 0.. \text{NELEM}-1
\]
_vall

Create Vector with Identical Components

Make a vector with all components set to the same value.

\[
\begin{align*}
\text{v8_t } & \ _\text{val18}(\text{int } k) \\
\text{v16_t } & \ _\text{val16}(\text{int } k) \\
\text{v32_t } & \ _\text{val32}(\text{int } k) \\
\text{v64_t } & \ _\text{val64}(\text{int } k)
\end{align*}
\]

This macro takes a single scalar parameter, \( k \). The scalar can be of any type (constant, variable, etc.). The scalar is repeated \( esize \) times, producing a 128-bit result vector of \( esize \)-bit elements.

\[ r[i] = k, \quad i = 0..\text{NELEM}-1 \]
Create Vector with First Component Specified

Make a vector with the first component set to the specified value and all other components set to 0.

\[
\begin{align*}
\text{v8_t } & \_\text{velem8}(\text{int } k) \\
\text{v16_t } & \_\text{velem16}(\text{int } k) \\
\text{v32_t } & \_\text{velem32}(\text{int } k) \\
\text{v64_t } & \_\text{velem64}(\text{int } k) \\
\text{vc8_t } & \_\text{velemc8}(\text{int } \text{re}, \text{int } \text{im}) \\
\text{vc16_t } & \_\text{velemc16}(\text{int } \text{re}, \text{int } \text{im}) \\
\text{vc32_t } & \_\text{velemc32}(\text{int } \text{re}, \text{int } \text{im}) \\
\text{vc64_t } & \_\text{velemc64}(\text{int } \text{re}, \text{int } \text{im})
\end{align*}
\]

This macro takes a single scalar parameter, \( k \). The scalar can be of any type (constant, variable, etc.). The first component of the vector is set to \( k \). All other components of the vector are set to 0.

\[
r[0] = k; \ r[i] = 0, \quad i = 1..\text{NELEM} -1
\]
3.2.2 Load and Store Vector

The Load and Store Vector functions include:

- `_lv` Load Vector
- `_sv` Store Vector

These functions support loading and storing of either little- or big-endian data. Big-endian loads perform an automatic byte-swap after the load, thereby eliminating the need for this separate step.
Chapter 3: Vector and Matrix Functions

Load Vector

Loads a vector from memory.

\[
\begin{align*}
_v8_t & \ _lv8\{,a\}\{,1,b\}\(\text{int8} \ *\text{addr}\) \\
v16_t & \ _lv16\{,a\}\{,1,b\}\(\text{int16} \ *\text{addr}\) \\
v32_t & \ _lv32\{,a\}\{,1,b\}\(\text{int32} \ *\text{addr}\) \\
v64_t & \ _lv64\{,a\}\{,1,b\}\(\text{int64} \ *\text{addr}\) \\
v128_t & \ _lv128\{,a\}\{,1,b\}\(\text{int128} \ *\text{addr}\) \\
v8_t & \ _lvu8\{,a\}\{,1,b\}\(\text{uint8} \ *\text{addr}\) \\
v16_t & \ _lvu16\{,a\}\{,1,b\}\(\text{uint16} \ *\text{addr}\) \\
v32_t & \ _lvu32\{,a\}\{,1,b\}\(\text{uint32} \ *\text{addr}\) \\
v64_t & \ _lvu64\{,a\}\{,1,b\}\(\text{uint64} \ *\text{addr}\) \\
v128_t & \ _lvu128\{,a\}\{,1,b\}\(\text{uint128} \ *\text{addr}\) \\
v8_t & \ _lvc8\{,a\}\{,1,b\}\(\text{cplxi8} \ *\text{addr}\) \\
v16_t & \ _lvc16\{,a\}\{,1,b\}\(\text{cplxi16} \ *\text{addr}\) \\
v32_t & \ _lvc32\{,a\}\{,1,b\}\(\text{cplxi32} \ *\text{addr}\) \\
v64_t & \ _lvc64\{,a\}\{,1,b\}\(\text{cplxi64} \ *\text{addr}\) \\
v16_t & \ _lvf16\{,a\}\{,1,b\}\(\text{float16} \ *\text{addr}\) \\
v32_t & \ _lvf32\{,a\}\{,1,b\}\(\text{float32} \ *\text{addr}\) \\
v64_t & \ _lvf64\{,a\}\{,1,b\}\(\text{float64} \ *\text{addr}\) \\
v128_t & \ _lvf128\{,a\}\{,1,b\}\(\text{float128} \ *\text{addr}\) \\
v8_t & \ _lvcf16\{,a\}\{,1,b\}\(\text{cplxf16} \ *\text{addr}\) \\
v16_t & \ _lvcf32\{,a\}\{,1,b\}\(\text{cplxf32} \ *\text{addr}\) \\
v64_t & \ _lvcf64\{,a\}\{,1,b\}\(\text{cplxf64} \ *\text{addr}\)
\end{align*}
\]

This function takes an address, \texttt{addr}. A vector of \texttt{esize}-bit elements is loaded starting from the specified address in memory.

Specifying the “a” option signifies that address is aligned to 128-bit boundaries which can make the access faster. The operation raises the \texttt{UnalignedAddress} exception if the address is not aligned to 128 bits.

Specifying the “l” option signifies that the elements in memory are stored in little-endian order (lower-order byte at the lower address). The “b” option signifies that the elements in memory are stored in big-endian order (high-order byte at the lower address). If the endian mode is omitted, the default machine endian-ness is assumed.
r[i] = addr[i], \quad i = 0..NELEM -1
_sv

Stores a vector in memory.

void _sv8{,a}{,l,b}(int8 *addr, v8_t x)
void _sv16{,a}{,l,b}(int16 *addr, v16_t x)
void _sv32{,a}{,l,b}(int32 *addr, v32_t x)
void _sv64{,a}{,l,b}(int64 *addr, v64_t x)
void _sv128{,a}{,l,b}(int128 *addr, v128_t x)

void _svc8{,a}{,l,b}(cplxi8 *addr, vc8_t x)
void _svc16{,a}{,l,b}(cplxi16 *addr, vc16_t x)
void _svc32{,a}{,l,b}(cplxi32 *addr, vc32_t x)
void _svc64{,a}{,l,b}(cplxi64 *addr, vc64_t x)

void _svf16{,a}{,l,b}(float16 *addr, vf16_t x)
void _svf32{,a}{,l,b}(float32 *addr, vf32_t x)
void _svf64{,a}{,l,b}(float64 *addr, vf64_t x)
void _svf128{,a}{,l,b}(float128 *addr, vf128_t x)

void _svcf16{,a}{,l,b}(cplxf16 *addr, vcf16_t x)
void _svcf32{,a}{,l,b}(cplxf32 *addr, vcf32_t x)
void _svcf64{,a}{,l,b}(cplxf64 *addr, vcf64_t x)

This function takes an address, addr, and a 128-bit vector parameter, x. The vector is interpreted as containing elements of esize-bit values. The vector is stored at the specified address in memory.

Specifying the “a” option signifies that address is aligned to 128-bit boundaries which can make the access faster. The operation raises the UnalignedAddress exception if the address is not aligned to 128 bits.

Specifying the “l” option signifies that the elements in memory are stored in little-endian order (lower-order byte at the lower address). The “b” option signifies that the elements in memory are stored in big-endian order (high-order byte at the lower address). If the endian mode is omitted, the default machine endian-ness is assumed.
\texttt{addr[i] = r[i]}, \quad i = 0..\text{NELEM}-1
3.2.3 **Manipulate Vector**

The Manipulate Vector functions include:

- `_vget` Get Element from Vector
- `_vput` Put Element in Vector
- `_reindex` Reindex Elements of Vector
- `_reindexpair` Reindex Elements of a Pair of Vectors
- `_interleave` Interleave Elements of a Pair of Vectors
- `_even` Get Even Elements of Vector
- `_odd` Get Odd Elements of Vector
Returns the value of an element in a vector.

```
int _vget1(v1_t x, int index)
int _vget2(v2_t x, int index)
int _vget4(v4_t x, int index)
int _vget8(v8_t x, int index)
int _vget16(v16_t x, int index)
int _vget32(v32_t x, int index)
int _vget64(v64_t x, int index)

int _vgetu1(v1_t x, int index)
int _vgetu2(v2_t x, int index)
int _vgetu4(v4_t x, int index)
int _vgetu8(v8_t x, int index)
int _vgetu16(v16_t x, int index)
int _vgetu32(v32_t x, int index)
int _vgetu64(v64_t x, int index)
```

This function takes a 128-bit vector parameter, \( x \), and a scalar index, \( \text{index} \). The vector is interpreted as containing \( \text{esize} \)-bit elements. The element of the vector that is indexed by \( \text{index} \) is returned, producing an \( \text{esize} \)-bit result.

\[
r = x[\text{index}]
\]
_vput

Put Element in Vector

Replaces a value in one element of a vector, and returns the vector.

\[
\begin{align*}
\text{v1_t} & \quad \text{vput1(v1_t \ x, \ int \ index, \ int \ value)} \\
\text{v2_t} & \quad \text{vput2(v2_t \ x, \ int \ index, \ int \ value)} \\
\text{v4_t} & \quad \text{vput4(v4_t \ x, \ int \ index, \ int \ value)} \\
\text{v8_t} & \quad \text{vput8(v8_t \ x, \ int \ index, \ int \ value)} \\
\text{v16_t} & \quad \text{vput16(v16_t \ x, \ int \ index, \ int \ value)} \\
\text{v32_t} & \quad \text{vput32(v32_t \ x, \ int \ index, \ int \ value)} \\
\text{v64_t} & \quad \text{vput64(v64_t \ x, \ int \ index, \ int \ value)}
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), a scalar index, \( index \), and a scalar value, \( value \). The vector is interpreted as containing \( esize \)-bit elements. The value is placed into the element of the vector indexed by \( index \), and the complete vector is returned, producing a 128-bit result vector of \( esize \)-bit elements. The original vector parameter, \( x \), is left unchanged.

\[
\begin{align*}
\text{r[i]} & = \text{x[i]}, \quad i \neq \text{index}, \\
\text{r[i]} & = \text{value}, \quad i = \text{index}
\end{align*}
\]
**_reindex**  

Reindex Elements in Vector

Moves elements of a vector into a new position offset by the index.

\[
\begin{align*}
v1_t \_reindex1(v1_t x, \text{int index}) \\
v2_t \_reindex2(v2_t x, \text{int index}) \\
v4_t \_reindex4(v4_t x, \text{int index}) \\
v8_t \_reindex8(v8_t x, \text{int index}) \\
v16_t \_reindex8(v16_t x, \text{int index}) \\
v32_t \_reindex8(v32_t x, \text{int index}) \\
v64_t \_reindex8(v64_t x, \text{int index}) \\
v8_t \_reindexc8(vc8_t x, \text{int index}) \\
v16_t \_reindexc16(vc16_t x, \text{int index}) \\
v32_t \_reindexc32(vc32_t x, \text{int index}) \\
vf16_t \_reindexf16(vf16_t x, \text{int index}) \\
vf32_t \_reindexf32(vf32_t x, \text{int index}) \\
vf64_t \_reindexf64(vf64_t x, \text{int index}) \\
vcf16_t \_reindexcf16(vcf16_t x, \text{int index}) \\
vcf32_t \_reindexcf32(vcf32_t x, \text{int index})
\end{align*}
\]

This function takes a 128-bit vector parameter, \(x\), and scalar index, \(\text{index}\). The vector is interpreted as containing \(\text{esize}\)-bit elements. The result vector is formed by taking elements from this vector starting with element number \(\text{index}\) up to and including element number \(\text{index}+\text{NELEM}-1\).

\[
r[i] = x[i+\text{index}], \quad i < \text{NELEM-index}, \\
r[i] = 0, \quad i \geq \text{NELEM-index}
\]
_reindexpair  

Reindex Elements in Pairs of Vectors

Moves elements of a vector pair into a new position offset by the index.

\[
\begin{align*}
\text{v1_t } & \quad \_\text{reindexpair1}(\text{v1_t } xlo, \text{v1_t } xhi, \text{int } index) \\
\text{v2_t } & \quad \_\text{reindexpair2}(\text{v2_t } xlo, \text{v2_t } xhi, \text{int } index) \\
\text{v4_t } & \quad \_\text{reindexpair4}(\text{v4_t } xlo, \text{v4_t } xhi, \text{int } index) \\
\text{v8_t } & \quad \_\text{reindexpair8}(\text{v8_t } xlo, \text{v8_t } xhi, \text{int } index) \\
\text{v16_t } & \quad \_\text{reindexpair16}(\text{v16_t } xlo, \text{v16_t } xhi, \text{int } index) \\
\text{v32_t } & \quad \_\text{reindexpair32}(\text{v32_t } xlo, \text{v32_t } xhi, \text{int } index) \\
\text{v64_t } & \quad \_\text{reindexpair64}(\text{v64_t } xlo, \text{v64_t } xhi, \text{int } index) \\
\text{vc8_t } & \quad \_\text{reindexpairc8}(\text{vc8_t } xlo, \text{vc8_t } xhi, \text{int } index) \\
\text{vc16_t } & \quad \_\text{reindexpairc16}(\text{vc16_t } xlo, \text{vc16_t } xhi, \text{int } index) \\
\text{vc32_t } & \quad \_\text{reindexpairc32}(\text{vc32_t } xlo, \text{vc32_t } xhi, \text{int } index) \\
\text{vf16_t } & \quad \_\text{reindexpairf16}(\text{vf16_t } xlo, \text{vf16_t } xhi, \text{int } index) \\
\text{vf32_t } & \quad \_\text{reindexpairf32}(\text{vf32_t } xlo, \text{vf32_t } xhi, \text{int } index) \\
\text{vf64_t } & \quad \_\text{reindexpairf64}(\text{vf64_t } xlo, \text{vf64_t } xhi, \text{int } index) \\
\text{vcf16_t } & \quad \_\text{reindexpaircfc16}(\text{vcf16_t } xlo, \text{vcf16_t } xhi, \text{int } index) \\
\text{vcf32_t } & \quad \_\text{reindexpaircfc32}(\text{vcf32_t } xlo, \text{vcf32_t } xhi, \text{int } index)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( xlo \) and \( xhi \), and a scalar index, \( index \). The vectors are interpreted as containing \( \text{esize} \)-bit elements. The vectors \( xlo \) and \( xhi \) are concatenated forming one double-length vector. The result vector is formed by taking elements from this double length vector starting with element number \( index \) up to and including element number \( index + \text{NELEM}-1 \).

\[
\begin{align*}
r[i] &= xlo[i+index], \quad i < \text{NELEM}-index, \\
r[i] &= xhi[i+index-\text{NELEM}], \quad i \geq \text{NELEM}-index
\end{align*}
\]
_interleave Interleave Elements in Pairs of Vectors

Interleaves a vector pair.

\[
\begin{align*}
v_8_t \ _\text{interleave8lo}(v_8_t \ even, v_8_t \ odd) \\
v_{16_t} \ _\text{interleave16lo}(v_{16_t} \ even, v_{16_t} \ odd) \\
v_{32_t} \ _\text{interleave32lo}(v_{32_t} \ even, v_{32_t} \ odd) \\
v_8_t \ _\text{interleave8hi}(v_8_t \ even, v_8_t \ odd) \\
v_{16_t} \ _\text{interleave16hi}(v_{16_t} \ even, v_{16_t} \ odd) \\
v_{32_t} \ _\text{interleave32hi}(v_{32_t} \ even, v_{32_t} \ odd) \\
v_{c16_t} \ _\text{interleavec16lo}(v_{c16_t} \ even, v_{c16_t} \ odd) \\
v_{c16_t} \ _\text{interleavec16hi}(v_{c16_t} \ even, v_{c16_t} \ odd)
\end{align*}
\]

This function takes two 128-bit vector parameters, even and odd. The even vector is interpreted as containing the even-indexed esize-bit components of the vector to be formed. The odd vector is interpreted as containing the odd-indexed esize-bit components of the vector to be formed. The function mnemonic has two suffixes, lo and hi. The interleave<esize>lo version interleaves the elements in the low 64 bits of even with the elements in the low 64 bits of odd. The interleave<esize>hi version does the same with the high 64 bits of each vector, producing a 128-bit result vector in which all elements are esize bits wide.

_interleave<esize>lo is equivalent to this _xshufflepair function (page 177):

\[\_\text{xshufflepair<esize>lo}(even, odd, 1, 2)\]

_interleave<esize>hi is equivalent to this _xshufflepair function (page 177):

\[\_\text{xshufflepair<esize>hi}(even, odd, 1, 2)\]
Chapter 3: Vector and Matrix Functions

Support Functions and Macros

---

**_interleave<esize>lo**

\[
\begin{align*}
&\text{odd} & \text{even} \\
&0 & 0 \\
&\begin{array}{l}
\text{esize} \\
\text{esize} \\
\text{odd} \\
\text{odd} \\
\end{array} & \begin{array}{l}
\text{esize} \\
\text{esize} \\
\text{even} \\
\text{even} \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
&X_1 \\
&X_0 \\
&\begin{array}{l}
\text{odd} \\
\text{even} \\
\text{esize} \\
\text{esize} \\
\end{array} & \begin{array}{l}
\text{odd} \\
\text{even} \\
\text{esize} \\
\text{esize} \\
\end{array} \\
\end{align*}
\]

\[
r[2^i] = \text{even}[i], \\
r[2^i+1] = \text{odd}[i], \quad i = 0..\text{NELEM}/2 - 1
\]

---

**_interleave<esize>hi**

\[
\begin{align*}
&\text{odd} & \text{even} \\
&0 & 0 \\
&\begin{array}{l}
\text{esize} \\
\text{esize} \\
\text{odd} \\
\text{odd} \\
\end{array} & \begin{array}{l}
\text{esize} \\
\text{esize} \\
\text{even} \\
\text{even} \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
&X_1 \\
&X_0 \\
&\begin{array}{l}
\text{odd} \\
\text{even} \\
\text{esize} \\
\text{esize} \\
\end{array} & \begin{array}{l}
\text{odd} \\
\text{even} \\
\text{esize} \\
\text{esize} \\
\end{array} \\
\end{align*}
\]

\[
r[2^i] = \text{even}[i + \text{NELEM}/2], \\
r[2^i+1] = \text{imag}[i + \text{NELEM}/2], \quad i = 0..\text{NELEM}/2 - 1
\]
Get Even Elements of Vector

Returns the even-indexed elements of a vector.

\[ \begin{align*}
_\text{even8} & : \text{v8}_t \rightarrow \text{v8}_t \\
_\text{even16} & : \text{v16}_t \rightarrow \text{v16}_t \\
_\text{even32} & : \text{v32}_t \rightarrow \text{v32}_t \\
_\text{even64} & : \text{v64}_t \rightarrow \text{v64}_t \\
_\text{evenc8} & : \text{vc8}_t \rightarrow \text{vc8}_t \\
_\text{evenc16} & : \text{vc16}_t \rightarrow \text{vc16}_t \\
_\text{evenc32} & : \text{vc32}_t \rightarrow \text{vc32}_t \\
_\text{evenf16} & : \text{vf16}_t \rightarrow \text{vf16}_t \\
_\text{evenf32} & : \text{vf32}_t \rightarrow \text{vf32}_t \\
_\text{evenf64} & : \text{vf64}_t \rightarrow \text{vf64}_t \\
_\text{evencf16} & : \text{vcf16}_t \rightarrow \text{vcf16}_t \\
_\text{evencf32} & : \text{vcf32}_t \rightarrow \text{vcf32}_t 
\end{align*} \]

This function takes two 128-bit vector parameters, \( xlo \) and \( xhi \). The vectors are interpreted as containing sets of \( esize \)-bit numbers. The elements at even indices are extracted and concatenated, producing a 128-bit result vector of \( esize \)-bit elements.

\( _\text{even}<esize> \) is equivalent to this \( _\text{xshufflepair} \) function (page 177):

\[ _\text{xshufflepair}<esize>\text{lo}(xlo, xhi, 1, \text{NELEM}) \]

\[ r[i] = xlo[2i], \quad r[i + \text{NELEM}/2] = xhi[2i], \quad i = 0..\text{NELEM}/2 -1 \]
Returns the odd-indexed elements of a vector.

\[
\begin{align*}
&v8_t \_\text{odd8}(v8_t \ xlo, v8_t \ xhi) \\
v16_t \_\text{odd16}(v16_t \ xlo, v16_t \ xhi) \\
v32_t \_\text{odd32}(v32_t \ xlo, v32_t \ xhi) \\
v64_t \_\text{odd64}(v64_t \ xlo, v64_t \ xhi) \\
v8_t \_\text{oddc8}(vc8_t \ xlo, vc8_t \ xhi) \\
v16_t \_\text{oddc16}(vc16_t \ xlo, vc16_t \ xhi) \\
v32_t \_\text{oddc32}(vc32_t \ xlo, vc32_t \ xhi) \\
v64_t \_\text{oddf64}(vf64_t \ xlo, vf64_t \ xhi) \\
v16_t \_\text{oddf16}(vf16_t \ xlo, vf16_t \ xhi) \\
v32_t \_\text{oddf32}(vf32_t \ xlo, vf32_t \ xhi) \\
v64_t \_\text{oddf64}(vf64_t \ xlo, vf64_t \ xhi) \\
vc16_t \_\text{oddc16}(vcf16_t \ xlo, vcf16_t \ xhi) \\
vc32_t \_\text{oddc32}(vcf32_t \ xlo, vcf32_t \ xhi)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( xlo \) and \( xhi \). The vectors are interpreted as containing sets of \( \text{esize} \)-bit numbers. The elements at odd indices are extracted and concatenated, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

\( \_\text{odd}\langle\text{esize}\rangle \) is equivalent to this \( \_\text{xshufflepair} \) function (page 177):

\[
\_\text{xshufflepair}\langle\text{esize}\rangle\text{hi}(xlo, xhi, 1, \text{NELEM})
\]

\[
\begin{array}{cccccccc}
\text{xhi} & 255 & 128 & 127 & 0 \\
\text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{esize} \\
\text{element} & \text{element} & \text{element} & \text{element} & \text{element} & \text{element} & \text{element} & \text{element} \\
x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0
\end{array}
\]

\[
\begin{align*}
r[i] &= xlo[2i+1], \\
r[i] + \text{NELEM}/2 &= xhi[2i+1], \quad i = 0..\text{NELEM}/2 -1
\end{align*}
\]
3.2.4 **Complex Numbers**

The Complex Number functions include:

- **_makecplx** Make Complex Vector
- **_realpart** Get Real Part of Complex Vector
- **_imagpart** Get Imaginary Part of Complex Vector

The `realpart` and `imagpart` functions extract the real and imaginary parts of complex numbers. The `makecplx` function combines a vector of real and a vector of imaginary parts to make a vector of complex numbers.
_makecplx

**Make Complex Vector**

Creates a vector of complex numbers.

\[
\begin{align*}
\text{vc8_t} & \quad _\text{makecplx8lo}(\text{v8_t real, v8_t imag}) \\
\text{vc16_t} & \quad _\text{makecplx16lo}(\text{v16_t real, v16_t imag}) \\
\text{vc32_t} & \quad _\text{makecplx32lo}(\text{v32_t real, v32_t imag}) \\
\text{vc64_t} & \quad _\text{makecplx64lo}(\text{v64_t real, v64_t imag}) \\
\text{vc8_t} & \quad _\text{makecplx8hi}(\text{v8_t real, v8_t imag}) \\
\text{vc16_t} & \quad _\text{makecplx16hi}(\text{v16_t real, v16_t imag}) \\
\text{vc32_t} & \quad _\text{makecplx32hi}(\text{v32_t real, v32_t imag}) \\
\text{vc64_t} & \quad _\text{makecplx64hi}(\text{v64_t real, v64_t imag}) \\
\text{vcf16_t} & \quad _\text{makecplxf16lo}(\text{vf16_t real, vf16_t imag}) \\
\text{vcf32_t} & \quad _\text{makecplxf32lo}(\text{vf32_t real, vf32_t imag}) \\
\text{vcf64_t} & \quad _\text{makecplxf64lo}(\text{vf64_t real, vf64_t imag}) \\
\text{vcf16_t} & \quad _\text{makecplxf16hi}(\text{vf16_t real, vf16_t imag}) \\
\text{vcf32_t} & \quad _\text{makecplxf32hi}(\text{vf32_t real, vf32_t imag}) \\
\text{vcf64_t} & \quad _\text{makecplxf64hi}(\text{vf64_t real, vf64_t imag})
\end{align*}
\]

This function takes two 128-bit vector parameters, `real` and `imag`. The `real` vector is interpreted as containing elements of `esize`-bit real parts of complex numbers. The `imag` vector is interpreted as containing elements of `esize`-bit imaginary parts of complex numbers. The function mnemonic has two suffixes, `lo` and `hi`. The `makecplx<esize>lo` version interleaves the elements in the low 64 bits of `real` with the elements in the low 64 bits of `imag`. The `makecplx<esize>hi` version does the same with the high 64 bits of each vector, producing a 128-bit result vector of 2*`esize`-bit complex numbers, in which the real and imaginary elements are each `esize` bits wide.

_`makecplx<esize>lo` is equivalent to this `_xshufflepair` function (page 177):_

\[
_\text{xshufflepair}<\text{esize}>\text{lo}(\text{real, imag, 1, 2})
\]

_`makecplx<esize>hi` is equivalent to this `_xshufflepair` function (page 177):_

\[
_\text{xshufflepair}<\text{esize}>\text{hi}(\text{real, imag, 1, 2})
\]
**Chapter 3: Vector and Matrix Functions**

**Support Functions and Macros**

---

### _makecplx<esize>_lo

<table>
<thead>
<tr>
<th>imag</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>esize</td>
<td>esize</td>
</tr>
<tr>
<td>imag</td>
<td>imag</td>
</tr>
<tr>
<td>127</td>
<td>0</td>
</tr>
</tbody>
</table>

### _makecplx<esize>_hi

<table>
<thead>
<tr>
<th>imag</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>esize</td>
<td>esize</td>
</tr>
<tr>
<td>imag</td>
<td>imag</td>
</tr>
<tr>
<td>127</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**Code:**

- \( r[i].re = \text{real}[i], \)
- \( r[i].im = \text{imag}[i], \quad i = 0..\text{NELEM}/2 -1 \)

---

**Code:**

- \( r[i].re = \text{real}[i + \text{NELEM}/2], \)
- \( r[i].im = \text{imag}[i + \text{NELEM}/2], \quad i = 0..\text{NELEM}/2 -1 \)

---

82 BroadMX C/C++ Functions
_realpart Get Real Part of Complex Vector

Returns the real parts of complex numbers in a vector.

\[
\begin{align*}
\text{v8_t} & \ _\text{realpart8}(\text{vc8_t} \ xlo, \ \text{vc8_t} \ xhi) \\
\text{v16_t} & \ _\text{realpart16}(\text{vc16_t} \ xlo, \ \text{vc16_t} \ xhi) \\
\text{v32_t} & \ _\text{realpart32}(\text{vc32_t} \ xlo, \ \text{vc32_t} \ xhi) \\
\text{v64_t} & \ _\text{realpart64}(\text{vc64_t} \ xlo, \ \text{vc64_t} \ xhi) \\
\text{vf16_t} & \ _\text{realpartf16}(\text{vcf16_t} \ xlo, \ \text{vcf16_t} \ xhi) \\
\text{vf32_t} & \ _\text{realpartf32}(\text{vcf32_t} \ xlo, \ \text{vcf32_t} \ xhi) \\
\text{vf64_t} & \ _\text{realpartf64}(\text{vcf64_t} \ xlo, \ \text{vcf64_t} \ xhi)
\end{align*}
\]

This function takes two 128-bit vector parameters, \(xlo\) and \(xhi\). The vectors are interpreted as containing sets of \(2^{\text{esize}}\)-bit complex numbers, in which the real and imaginary elements are each \(\text{esize}\) bits wide, the real parts occupy the low element positions, and the imaginary parts occupy the high element positions. The real parts are extracted and concatenated, producing a 128-bit result vector of \(\text{esize}\)-bit elements.

_\_realpart<_\text{esize}> is equivalent to this _\_xshufflepair function (page 177):

_\_xshufflepair<_\text{esize}>lo(xlo, xhi, 1, NELEM)

\[
\begin{align*}
\text{r}[i] & = \text{xlo}[i].\text{re}, \\
\text{r}[i + \text{NELEM}/2] & = \text{xhi}[i].\text{re}, \quad i = 0.. \text{NELEM}/2 -1
\end{align*}
\]
_imagpart

Get Imaginary Part of Complex Vector

Returns the imaginary parts of complex numbers in a vector.

\[
\begin{align*}
\text{v8}_t & \text{ _imagpart8}(\text{vc8}_t \text{ xlo}, \text{vc8}_t \text{ xhi}) \\
\text{v16}_t & \text{ _imagpart16}(\text{vc16}_t \text{ xlo}, \text{vc16}_t \text{ xhi}) \\
\text{v32}_t & \text{ _imagpart32}(\text{vc32}_t \text{ xlo}, \text{vc32}_t \text{ xhi}) \\
\text{v64}_t & \text{ _imagpart64}(\text{vc64}_t \text{ xlo}, \text{vc64}_t \text{ xhi}) \\
\text{vf16}_t & \text{ _imagpartf16}(\text{vcf16}_t \text{ xlo}, \text{vcf16}_t \text{ xhi}) \\
\text{vf32}_t & \text{ _imagpartf32}(\text{vcf32}_t \text{ xlo}, \text{vcf32}_t \text{ xhi}) \\
\text{vf64}_t & \text{ _imagpartf64}(\text{vcf64}_t \text{ xlo}, \text{vcf64}_t \text{ xhi})
\end{align*}
\]

This function takes two 128-bit vector parameters, \text{xlo} and \text{xhi}. The vectors are interpreted as containing sets of \(2 \times \text{esize}\)-bit complex numbers, in which the real and imaginary elements are each \text{esize} bits wide, the real parts occupy the low element positions, and the imaginary parts occupy the high element positions. The imaginary parts are extracted and concatenated, producing a 128-bit result vector of \text{esize}-bit elements.

\text{ _imagpart<esize>} is equivalent to this \text{xshufflepair} function (page 177):

\text{xshufflepair<esize>hi}(\text{xlo}, \text{xhi}, 1, \text{NELEM})

\[
\begin{array}{c}
\text{xlo} \\
\text{xhi}
\end{array}
\]

\[
\begin{array}{cccccccc}
255 & 128 & 127 & 0 \\
\text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{esize} \\
\text{imag.} & \text{real} & \text{imag.} & \text{real} & \text{imag.} & \text{real} & \text{imag.} & \text{real} \\
X_3 & X_2 & X_1 & X_0 \\
\text{imag.} & \text{imag.} & \text{imag.} & \text{imag.} \\
127 & 0
\end{array}
\]

\[
\begin{align*}
r[i] & = \text{xlo}[i].\text{im}, \\
r[i + \text{NELEM}/2] & = \text{xhi}[i].\text{im}, \quad i = 0.. \text{NELEM}/2 -1
\end{align*}
\]
3.3 **Group Functions**

Group functions perform simple fixed-point or floating-point ALU operations on corresponding elements of 1, 2, or 3 128-bit source vectors, or immediate operands. Some of the functions return their results as a 128-bit vector of elements; others generate an exception under certain conditions.

The Group functions include the following types:

- Add and Subtract
- Shift Arithmetic
- Bit-wise Boolean
- Multiplex
- Copy Immediate
- Compare and Set
- Compare and Trap
3.3.1 **Group Add and Subtract**

The Group Add and Subtract functions include:

- `_gadd` Add
- `_gaddi` Add Immediate
- `_gaddh` Add and Halve
- `_gaddhu` Add and Halve Unsigned
- `_gaddl` Add with Limiting
- `_gaddlu` Add Unsigned with Limiting
- `_gaddo` Add with Overflow Trap
- `_gadduo` Add Unsigned with Overflow Trap
- `_gaddio` Add Immediate with Overflow Trap
- `_gaddiuno` Add Immediate Unsigned with Overflow Trap
- `_gaaa` Triple Add
- `_gasa` Add-Subtract-Add
- `_gsub` Subtract
- `_gsubi` Subtract Immediate
- `_gsubh` Subtract and Halve
- `_gsubhu` Subtract and Halve Unsigned
- `_gsbl` Subtract with Limiting
- `_gsublu` Subtract Unsigned with Limiting
- `_gsbtl` Subtract with Overflow Trap
- `_gsbuento` Subtract Unsigned with Overflow Trap
- `_gsbtiuo` Subtract Immediate with Overflow Trap
- `_gsbtiuno` Subtract Immediate Unsigned with Overflow Trap
Add

Adds elements of two vectors.

\[
\begin{align*}
\text{v8_t } & \_gadd8 (\text{v8_t } x, \text{v8_t } y) \\
\text{v16_t } & \_gadd16 (\text{v16_t } x, \text{v16_t } y) \\
\text{v32_t } & \_gadd32 (\text{v32_t } x, \text{v32_t } y) \\
\text{v64_t } & \_gadd64 (\text{v64_t } x, \text{v64_t } y) \\
\text{v128_t } & \_gadd128 (\text{v128_t } x, \text{v128_t } y) \\
\text{vc8_t } & \_gadd8 (\text{vc8_t } x, \text{vc8_t } y) \\
\text{vc16_t } & \_gadd16 (\text{vc16_t } x, \text{vc16_t } y) \\
\text{vc32_t } & \_gadd32 (\text{vc32_t } x, \text{vc32_t } y) \\
\text{vc64_t } & \_gadd64 (\text{vc64_t } x, \text{vc64_t } y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing elements of \( \text{esize} \)-bit integers. The vectors are added, element by element, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

\[
r[i] = x[i] + y[i], \quad i = 0..\text{NELEM}-1
\]
__gaddi

Add Immediate

Adds an immediate to vector elements.

\[
\text{r}[i] = \text{x}[i] + k, \quad i = 0..\text{NELEM} - 1
\]

This function takes one 128-bit vector parameter, \( \text{x} \), and a 10-bit signed immediate value, \( k \). The vector is interpreted as containing elements of \( \text{esize} \)-bit integers. The immediate value is sign-extended to \( \text{esize} \) bits and added, element by element, to the vector, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

```
v16_t _gaddi16(v16_t x, int k)
v32_t _gaddi32(v32_t x, int k)
v64_t _gaddi64(v64_t x, int k)
v128_t _gaddi128(v128_t x, int k)
```
__gaddh, __gaddhu

Add and Halve

Adds elements of two vectors, divides by two, and optionally rounds.

\[
\begin{align*}
\text{v8_t } & \_\text{gaddh8}\{c, f, n, z\}(\text{v8_t } x, \text{v8_t } y) \\
\text{v16_t } & \_\text{gaddh16}\{c, f, n, z\}(\text{v16_t } x, \text{v16_t } y) \\
\text{v32_t } & \_\text{gaddh32}\{c, f, n, z\}(\text{v32_t } x, \text{v32_t } y) \\
\text{v64_t } & \_\text{gaddh64}\{c, f, n, z\}(\text{v64_t } x, \text{v64_t } y) \\
\text{v128_t } & \_\text{gaddh128}\{c, f, n, z\}(\text{v128_t } x, \text{v128_t } y) \\
\text{vu8_t } & \_\text{gaddhu8}\{c, f, n, z\}(\text{vu8_t } x, \text{vu8_t } y) \\
\text{vu16_t } & \_\text{gaddhu16}\{c, f, n, z\}(\text{vu16_t } x, \text{vu16_t } y) \\
\text{vu32_t } & \_\text{gaddhu32}\{c, f, n, z\}(\text{vu32_t } x, \text{vu32_t } y) \\
\text{vu64_t } & \_\text{gaddhu64}\{c, f, n, z\}(\text{vu64_t } x, \text{vu64_t } y) \\
\text{vu128_t } & \_\text{gaddhu128}\{c, f, n, z\}(\text{vu128_t } x, \text{vu128_t } y) \\
\text{vc8_t } & \_\text{gaddh8}\{c, f, n, z\}(\text{vc8_t } x, \text{vc8_t } y) \\
\text{vc16_t } & \_\text{gaddh16}\{c, f, n, z\}(\text{vc16_t } x, \text{vc16_t } y) \\
\text{vc32_t } & \_\text{gaddh32}\{c, f, n, z\}(\text{vc32_t } x, \text{vc32_t } y) \\
\text{vc64_t } & \_\text{gaddh64}\{c, f, n, z\}(\text{vc64_t } x, \text{vc64_t } y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\). The vectors are interpreted as containing elements of esize-bit signed or unsigned integers. The vectors are added, element by element, and the result is divided by two and rounded, producing a 128-bit result vector of esize-bit elements. This operation never overflows.

See Table 12 on page 40 for details on optional rounding and exception modes.

\[
\begin{align*}
r[i] = (x[i] + y[i])/2, \text{ rounding as specified, } & \ i = 0..\text{NELEM} -1
\end{align*}
\]
Add with Limiting

Adds elements of two vectors, with limiting.

\[
\begin{align*}
\text{v8_t } & \_\text{gaddl8}(\text{v8_t } x, \text{v8_t } y) \\
\text{v16_t } & \_\text{gaddl16}(\text{v16_t } x, \text{v16_t } y) \\
\text{v32_t } & \_\text{gaddl32}(\text{v32_t } x, \text{v32_t } y) \\
\text{v64_t } & \_\text{gaddl64}(\text{v64_t } x, \text{v64_t } y) \\
\text{v128_t } & \_\text{gaddl128}(\text{v128_t } x, \text{v128_t } y) \\
\text{vu8_t } & \_\text{gaddlu8}(\text{vu8_t } x, \text{vu8_t } y) \\
\text{vu16_t } & \_\text{gaddlu16}(\text{vu16_t } x, \text{vu16_t } y) \\
\text{vu32_t } & \_\text{gaddlu32}(\text{vu32_t } x, \text{vu32_t } y) \\
\text{vu64_t } & \_\text{gaddlu64}(\text{vu64_t } x, \text{vu64_t } y) \\
\text{vu128_t } & \_\text{gaddlu128}(\text{vu128_t } x, \text{vu128_t } y) \\
\text{vc8_t } & \_\text{gaddl8}(\text{vc8_t } x, \text{vc8_t } y) \\
\text{vc16_t } & \_\text{gaddl16}(\text{vc16_t } x, \text{vc16_t } y) \\
\text{vc32_t } & \_\text{gaddl32}(\text{vc32_t } x, \text{vc32_t } y) \\
\text{vc64_t } & \_\text{gaddl64}(\text{vc64_t } x, \text{vc64_t } y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing elements of \( \text{esize} \)-bit signed or unsigned integers. The vectors are added, element by element, producing a 128-bit result vector of \( \text{esize} \)-bit elements. Sums that are outside the range representable in an \( \text{esize} \)-bit signed integer are clamped to the maximum (or minimum, if negative) representable integer.

\[
r[i] = x[i] + \text{limit} \cdot y[i], \quad i = 0..\text{NELEM}-1
\]
**_gaddo, _gadduo**  

Add with Overflow Trap

Adds elements of two vectors, and traps on overflow.

\[
\begin{align*}
\text{v8_t} & \quad \text{gadd8o}(\text{v8_t} \; x, \; \text{v8_t} \; y) \\
\text{v16_t} & \quad \text{gadd16o}(\text{v16_t} \; x, \; \text{v16_t} \; y) \\
\text{v32_t} & \quad \text{gadd32o}(\text{v32_t} \; x, \; \text{v32_t} \; y) \\
\text{v64_t} & \quad \text{gadd64o}(\text{v64_t} \; x, \; \text{v64_t} \; y) \\
\text{v128_t} & \quad \text{gadd128o}(\text{v128_t} \; x, \; \text{v128_t} \; y) \\
\text{vu8_t} & \quad \text{gaddu8o}(\text{vu8_t} \; x, \; \text{vu8_t} \; y) \\
\text{vu16_t} & \quad \text{gaddu16o}(\text{vu16_t} \; x, \; \text{vu16_t} \; y) \\
\text{vu32_t} & \quad \text{gaddu32o}(\text{vu32_t} \; x, \; \text{vu32_t} \; y) \\
\text{vu64_t} & \quad \text{gaddu64o}(\text{vu64_t} \; x, \; \text{vu64_t} \; y) \\
\text{vu128_t} & \quad \text{gaddu128o}(\text{vu128_t} \; x, \; \text{vu128_t} \; y) \\
\text{vc8_t} & \quad \text{gadd8o}(\text{vc8_t} \; x, \; \text{vc8_t} \; y) \\
\text{vc16_t} & \quad \text{gadd16o}(\text{vc16_t} \; x, \; \text{vc16_t} \; y) \\
\text{vc32_t} & \quad \text{gadd32o}(\text{vc32_t} \; x, \; \text{vc32_t} \; y) \\
\text{vc64_t} & \quad \text{gadd64o}(\text{vc64_t} \; x, \; \text{vc64_t} \; y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing elements of \( \text{esize}\)-bit signed or unsigned integers. The vectors are added, element by element, producing a 128-bit result vector of \( \text{esize}\)-bit elements. If any result exceeds the range representable in an \( \text{esize}\)-bit signed integer, the operation does not complete and the \text{FixedPointArithmetic} exception is taken.

\[
r[i] = x[i] +^o y[i], \quad i = 0..\text{NELEM}-1, \text{ or } \\
r[i] = x[i] +^u y[i], \quad i = 0..\text{NELEM}-1
\]
_gaddio, _gaddiuo  

Add Immediate with Overflow Trap

Adds an immediate to vector elements, and traps on overflow.

```
v16_t _gaddi16o(v16_t x, int k)
v32_t _gaddi32o(v32_t x, int k)
v64_t _gaddi64o(v64_t x, int k)
v128_t _gaddi128o(v128_t x, int k)
vu16_t _gaddiu16o(vu16_t x, int k)
vu32_t _gaddiu32o(vu32_t x, int k)
vu64_t _gaddiu64o(vu64_t x, int k)
vu128_t _gaddiu128o(vu128_t x, int k)
```

This function takes one 128-bit vector parameter, x, and a 10-bit signed immediate value, k. The vector is interpreted as containing elements of esize-bit signed or unsigned integers. The immediate value is sign-extended to esize bits and added, element by element, to the vector, producing a 128-bit result vector of esize-bit signed integers. If any element in the result exceeds the range representable in an esize-bit signed integer, the operation does not complete and the FixedPointArithmetic exception is taken.

```
r[i] = x[i] + k, i = 0..NELEM -1, or
r[i] = x[i] + ou k, i = 0..NELEM -1
```

![Diagram](image_url)
__gaaa

Adds elements of three vectors.

v8_t __gaaa8(v8_t x, v8_t y, v8_t z)
v16_t __gaaa16(v16_t x, v16_t y, v16_t z)
v32_t __gaaa32(v32_t x, v32_t y, v32_t z)
v64_t __gaaa64(v64_t x, v64_t y, v64_t z)
v128_t __gaaa128(v128_t x, v128_t y, v128_t z)

vc8_t __gaaa8(vc8_t x, vc8_t y, vc8_t z)
vc16_t __gaaa16(vc16_t x, vc16_t y, vc16_t z)
vc32_t __gaaa32(vc32_t x, vc32_t y, vc32_t z)
vc64_t __gaaa64(vc64_t x, vc64_t y, vc64_t z)

This function takes three 128-bit vector parameters, x, y, and z. The vectors are interpreted as containing elements of esize-bit integers. The vectors are added, element by element, producing a 128-bit result vector of esize-bit elements.

\[ r[i] = x[i] + y[i] + z[i], \quad i = 0..\text{NELEM}-1 \]
__gasa

Add-Subtract-Add

 Adds elements of two vectors and subtracts elements of a third vector.

\[ r[i] = x[i] - y[i] + z[i], \quad i = 0..\text{NELEM} -1 \]
Subtracts elements of two vectors.

\[
v8_t \ _gsub8(v8_t \ x, v8_t \ y) \\
v16_t \ _gsub16(v16_t \ x, v16_t \ y) \\
v32_t \ _gsub32(v32_t \ x, v32_t \ y) \\
v64_t \ _gsub64(v64_t \ x, v64_t \ y) \\
v128_t \ _gsub128(v128_t \ x, v128_t \ y) \\
vc8_t \ _gsub8(vc8_t \ x, vc8_t \ y) \\
vc16_t \ _gsub16(vc16_t \ x, vc16_t \ y) \\
vc32_t \ _gsub32(vc32_t \ x, vc32_t \ y) \\
vc64_t \ _gsub64(vc64_t \ x, vc64_t \ y)
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\). The vectors are interpreted as containing elements of \(esize\)-bit integers. The vectors are subtracted, element by element, producing a 128-bit result vector of \(esize\)-bit elements.

\[
r[i] = x[i] - y[i], \quad i = 0..\text{NELEM}-1
\]
_gsubi Subtract Immediate

Subtracts vector elements from an immediate value.

\[ r[i] = k - x[i], \quad i = 0..\text{NELEM} - 1 \]

This function takes one 128-bit vector parameter, \( x \), and a 10-bit signed immediate value, \( k \). The vector is interpreted as containing elements of \( \text{esize} \)-bit integers. The immediate value is sign-extended to \( \text{esize} \) bits and the vector is subtracted from this, element by element, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

Note that, for example, \(_gsubi(2, x)\) produces the scalar-vector difference \( 2 - x \). If you want \( x - 2 \), use \(_gaddi(x, -2)\).
**_gsubh, _gsubhu**

Subtracts elements of two vectors, divides by two, and optionally rounds.

\[
\begin{align*}
&v8_t \ _gsubh8\{c, f, n, z\}(v8_t \ x, v8_t \ y) \\
v16_t \ _gsubh16\{c, f, n, z\}(v16_t \ x, v16_t \ y) \\
v32_t \ _gsubh32\{c, f, n, z\}(v32_t \ x, v32_t \ y) \\
v64_t \ _gsubh64\{c, f, n, z\}(v64_t \ x, v64_t \ y) \\
v128_t \ _gsubh128\{c, f, n, z\}(v128_t \ x, v128_t \ y)
\end{align*}
\]

\[
\begin{align*}
&vu8_t \ _gsubhu8\{c, f, n, z\}(vu8_t \ x, vu8_t \ y) \\
vu16_t \ _gsubhu16\{c, f, n, z\}(vu16_t \ x, vu16_t \ y) \\
vu32_t \ _gsubhu32\{c, f, n, z\}(vu32_t \ x, vu32_t \ y) \\
vu64_t \ _gsubhu64\{c, f, n, z\}(vu64_t \ x, vu64_t \ y) \\
vu128_t \ _gsubhu128\{c, f, n, z\}(vu128_t \ x, vu128_t \ y)
\end{align*}
\]

\[
\begin{align*}
&vc8_t \ _gsubh8\{c, f, n, z\}(vc8_t \ x, vc8_t \ y) \\
vc16_t \ _gsubh16\{c, f, n, z\}(vc16_t \ x, vc16_t \ y) \\
vc32_t \ _gsubh32\{c, f, n, z\}(vc32_t \ x, vc32_t \ y) \\
vc64_t \ _gsubh64\{c, f, n, z\}(vc64_t \ x, vc64_t \ y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\). The vectors are interpreted as containing elements of \(esize\)-bit signed or unsigned integers. The vectors are subtracted, element by element, and the result is divided by two and rounded, producing a 128-bit result vector of \(esize\)-bit elements.

See Table 12 on page 40 for details on optional rounding and exception modes.

\[
r[i] = (x[i] - y[i])/2, \text{ rounding as specified, } \ i = 0..\text{NELEM} - 1
\]
Subtracts signed elements of two vectors, with limiting.

\[
\begin{align*}
\text{v8}_t & \quad \text{gsubl8}(\text{v8}_t \ x, \ \text{v8}_t \ y) \\
\text{v16}_t & \quad \text{gsubl16}(\text{v16}_t \ x, \ \text{v16}_t \ y) \\
\text{v32}_t & \quad \text{gsubl32}(\text{v32}_t \ x, \ \text{v32}_t \ y) \\
\text{v64}_t & \quad \text{gsubl64}(\text{v64}_t \ x, \ \text{v64}_t \ y) \\
\text{v128}_t & \quad \text{gsubl128}(\text{v128}_t \ x, \ \text{v128}_t \ y) \\
\text{vu8}_t & \quad \text{gsublu8}(\text{vu8}_t \ x, \ \text{vu8}_t \ y) \\
\text{vu16}_t & \quad \text{gsublu16}(\text{vu16}_t \ x, \ \text{vu16}_t \ y) \\
\text{vu32}_t & \quad \text{gsublu32}(\text{vu32}_t \ x, \ \text{vu32}_t \ y) \\
\text{vu64}_t & \quad \text{gsublu64}(\text{vu64}_t \ x, \ \text{vu64}_t \ y) \\
\text{vu128}_t & \quad \text{gsublu128}(\text{vu128}_t \ x, \ \text{vu128}_t \ y) \\
\text{vc8}_t & \quad \text{gsubl8}(\text{vc8}_t \ x, \ \text{vc8}_t \ y) \\
\text{vc16}_t & \quad \text{gsubl16}(\text{vc16}_t \ x, \ \text{vc16}_t \ y) \\
\text{vc32}_t & \quad \text{gsubl32}(\text{vc32}_t \ x, \ \text{vc32}_t \ y) \\
\text{vc64}_t & \quad \text{gsubl64}(\text{vc64}_t \ x, \ \text{vc64}_t \ y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing elements of \( \text{esize} \)-bit signed or unsigned integers. The vectors are subtracted, element by element, producing a 128-bit result vector of \( \text{esize} \)-bit elements. Differences that are outside the range representable in an \( \text{esize} \)-bit signed integer are clamped to the maximum (or minimum, if negative) representable integer.

\[
r[i] = x[i] - l y[i], \quad i = 0..\text{NELEM} - 1, \text{ or } \quad r[i] = x[i] - ul y[i], \quad i = 0..\text{NELEM} - 1
\]
Subtract with Overflow Trap

Subtracts elements of two vectors, and traps on overflow.

v8_t _gsub8o(v8_t x, v8_t y)
v16_t _gsub16o(v16_t x, v16_t y)
v32_t _gsub32o(v32_t x, v32_t y)
v64_t _gsub64o(v64_t x, v64_t y)
v128_t _gsub128o(v128_t x, v128_t y)

vu8_t _gsubu8o(vu8_t x, vu8_t y)
vu16_t _gsubu16o(vu16_t x, vu16_t y)
vu32_t _gsubu32o(vu32_t x, vu32_t y)
vu64_t _gsubu64o(vu64_t x, vu64_t y)
v128_t _gsubu128o(v128_t x, v128_t y)

vc8_t _gsub8o(vc8_t x, vc8_t y)
vc16_t _gsub16o(vc16_t x, vc16_t y)
vc32_t _gsub32o(vc32_t x, vc32_t y)
vc64_t _gsub64o(vc64_t x, vc64_t y)

This function takes two 128-bit vector parameters, x and y. The vectors are interpreted as containing elements of esize-bit signed or unsigned integers. The vectors are subtracted, element by element, producing a 128-bit result vector of esize-bit elements. If any result exceeds the range representable in an esize-bit signed integer, the operation does not complete and the FixedPointArithmetic exception is taken.

\[
\text{r[i]} = \text{x[i]} -^o \text{y[i]}, \quad i = 0..\text{NELEM} -1, \text{ or } \\
\text{r[i]} = \text{x[i]} -^uo \text{y[i]}, \quad i = 0..\text{NELEM} -1
\]
_gsubio, _gsubiuo       Subtract Immediate with Overflow Trap

Subtracts vector elements from an immediate value, and traps on overflow.

\[
\begin{align*}
\text{v16}_t & \quad \_gsubio16(\text{int} \ k, \text{v16}_t \ x) \\
\text{v32}_t & \quad \_gsubio32(\text{int} \ k, \text{v32}_t \ x) \\
\text{v64}_t & \quad \_gsubio64(\text{int} \ k, \text{v64}_t \ x) \\
\text{v128}_t & \quad \_gsubio128(\text{int} \ k, \text{v128}_t \ x) \\
\text{vu16}_t & \quad \_gsubiuo16(\text{int} \ k, \text{vu16}_t \ x) \\
\text{vu32}_t & \quad \_gsubiuo32(\text{int} \ k, \text{vu32}_t \ x) \\
\text{vu64}_t & \quad \_gsubiuo64(\text{int} \ k, \text{vu64}_t \ x) \\
\text{vu128}_t & \quad \_gsubiuo128(\text{int} \ k, \text{vu128}_t \ x)
\end{align*}
\]

This function takes one 128-bit vector parameter, \( x \), and a 10-bit signed immediate value, \( k \). The vector is interpreted as containing elements of \( esize \)-bit signed integers. The immediate value is sign-extended to \( esize \) bits and the vector is subtracted from this, element by element, producing a 128-bit result vector of \( esize \)-bit signed integers. If any element in the result exceeds the range representable in an \( esize \)-bit signed integer, the operation does not complete and the FixedPointArithmetic exception is taken.

Note that, for example, \( _gsubio(2, x) \) produces the scalar-vector difference \( 2 - x \). If you want \( x - 2 \), use \( _gaddio(x, -2) \).

\[
\begin{align*}
\text{r}[i] &= k -^o x[i], \quad i = 0..\text{NELEM} -1, \quad \text{or} \\
\text{r}[i] &= k -^uc x[i], \quad i = 0..\text{NELEM} -1
\end{align*}
\]
3.3.2 **Group Shift Arithmetic**

The Group Shift Arithmetic functions include:

- `_gshliadd` Shift Left Immediate and Add
- `_gshlisub` Shift Left Immediate and Subtract
_gshliadd Shift Left Immediate and Add

Left-shifts elements of one vector and adds them to elements of another vector.

v8_t _gshliadd8(v8_t x, int sh, v8_t y)
v16_t _gshliadd16(v16_t x, int sh, v16_t y)
v32_t _gshliadd32(v32_t x, int sh, v32_t y)
v64_t _gshliadd64(v64_t x, int sh, v64_t y)
v128_t _gshliadd128(v128_t x, int sh, v128_t y)

vc8_t _gshliadd8(vc8_t x, int sh, vc8_t y)
vc16_t _gshliadd16(vc16_t x, int sh, vc16_t y)
vc32_t _gshliadd32(vc32_t x, int sh, vc32_t y)
vc64_t _gshliadd64(vc64_t x, int sh, vc64_t y)

This function takes two 128-bit vector parameters, x and y, and a scalar immediate shift amount, sh, which must be 1,2,3, or 4. The vectors are interpreted as containing elements of esize-bit integers. The elements of vector x are left-shifted by sh. The corresponding elements of vector y are added to this, producing a 128-bit result vector of esize-bit elements.

\[
r[i] = (x[i] \ll sh) + y[i], \quad i = 0..\text{NELEM} -1
\]
Left-shifts elements of one vector and subtracts them from elements of another vector.

\[
r[i] = (x[i] \ll sh) - y[i], \quad i = 0..\text{NELEM} - 1
\]
3.3.3 **Group Boolean or Multiplex**

The Group Boolean functions include:

- `_gand`  Bit-wise AND
- `_gandi`  Bit-wise AND Immediate
- `_gaaand`  Bit-wise Triple-AND
- `_gnand`  Bit-wise NAND
- `_gnandi`  Bit-wise NAND Immediate
- `_gnaaand`  Bit-wise Triple-NAND
- `_gandn`  Bit-wise AND-NOT
- `_gor`  Bit-wise OR
- `_gori`  Bit-wise OR Immediate
- `_gooor`  Bit-wise Triple-OR
- `_gnor`  Bit-wise NOR
- `_gnori`  Bit-wise NOR Immediate
- `_gnooo`  Bit-wise Triple-NOR
- `_gorn`  Bit-wise OR-NOT
- `_gxor`  Bit-wise XOR
- `_gxori`  Bit-wise XOR Immediate
- `_gxxxor`  Bit-wise Triple-XOR
- `_gxnor`  Bit-wise XNOR
- `_gnxxxor`  Bit-wise Triple-XNOR
- `_gmux`  Bit-wise Multiplex
- `_gboolean`  Bit-wise Triple Boolean
_gand

Bit-wise AND

ANDs elements of two vectors.

\[ v_8\_t \ _{\text{gand8}}(v_8\_t \ x, v_8\_t \ y) \]
\[ v_{16}\_t \ _{\text{gand16}}(v_{16}\_t \ x, v_{16}\_t \ y) \]
\[ v_{32}\_t \ _{\text{gand32}}(v_{32}\_t \ x, v_{32}\_t \ y) \]
\[ v_{64}\_t \ _{\text{gand64}}(v_{64}\_t \ x, v_{64}\_t \ y) \]
\[ v_{128}\_t \ _{\text{gand128}}(v_{128}\_t \ x, v_{128}\_t \ y) \]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are ANDed, bit by bit, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

This is a special case of the \_gboolean operation (page 125):

\[ \_gand(x, y) = \_gboolean(x, x, y, 136) \]

\[
\begin{array}{cccccccccccc}
127 & \cdots & 0 & \cdots & 127 & \cdots & 0 \\
\mid \text{esize} & \text{esize} & \text{esize} & \text{esize} \\
\mid \text{element} & \text{element} & \text{element} & \text{element} \\

\downarrow & \downarrow & \downarrow & \downarrow \\

\text{element} & \text{element} & \text{element} & \text{element} \\
\mid \text{esize} & \text{esize} & \text{esize} & \text{esize} \\
127 & \cdots & 0 \\
\end{array}
\]

\[ r[i] = x[i] \& y[i], \quad i = 0..\text{NELEM} -1 \]
__gandi

Bit-wise AND Immediate

ANDs an immediate with vector elements.

\[
v16_t \text{ __gandi16}(v16_t x, \text{ int } k) \\
v32_t \text{ __gandi32}(v32_t x, \text{ int } k) \\
v64_t \text{ __gandi64}(v64_t x, \text{ int } k) \\
v128_t \text{ __gandi128}(v128_t x, \text{ int } k)
\]

This function takes one 128-bit vector parameter, \( x \), and a 10-bit signed immediate value, \( k \). The vector is interpreted as containing elements of \( esize \)-bit integers. The immediate value is sign-extended to \( esize \) bits and bit-wise ANDed, element by element, with the vector, producing a 128-bit result vector of \( esize \)-bit elements.

\[
r[i] = x[i] \& k, \quad i = 0..\text{NELEM} -1
\]
__gaaand

Bit-wise Triple-AND

ANDs elements of three vectors.

v8_t __gaaand8 (v8_t x, v8_t y, v8_t z)
v16_t __gaaand16 (v16_t x, v16_t y, v16_t z)
v32_t __gaaand32 (v32_t x, v32_t y, v32_t z)
v64_t __gaaand64 (v64_t x, v64_t y, v64_t z)
v128_t __gaaand128 (v128_t x, v128_t y, v128_t z)

This function takes three 128-bit vector parameters, x, y and z. The vectors are AND-ed, bit by bit, producing a 128-bit result vector of esize-bit elements.

This is a special case of the __gboolean operation (page 125):

__gaaand(x,y) = __gboolean(x, y, z, 128)

\[ r[i] = x[i] \& y[i] \& z[i], \quad i = 0..\text{NELEM} -1 \]
_gnand  

Bit-wise NAND

NANDs elements of two vectors.

```c
v8_t _gnand8(v8_t x, v8_t y)
v16_t _gnand16(v16_t x, v16_t y)
v32_t _gnand32(v32_t x, v32_t y)
v64_t _gnand64(v64_t x, v64_t y)
vl28_t _gnand128(vl28_t x, vl28_t y)
```

This function takes two 128-bit vector parameters, `x` and `y`. The vectors are NANDed, bit by bit, producing a 128-bit result vector of `esize`-bit elements.

This is a special case of the _gboolean operation (page 125):

\[ _{\text{gnand}}(x, y) = _{\text{gboolean}}(x, x, y, 119) \]

\[
\begin{array}{cccccccccc}
\text{x} & 127 & e & 126 & e & 125 & e & 124 & e & 123 & e & 122 & e & 121 & e & 120 & e & 0 \\
\text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element}
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{y} & 127 & e & 126 & e & 125 & e & 124 & e & 123 & e & 122 & e & 121 & e & 120 & e & 0 \\
\text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element}
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{~&} & \text{~&} & \text{~&} & \text{~&} & \text{~&}
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element}
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{r} & 127 & e & 126 & e & 125 & e & 124 & e & 123 & e & 122 & e & 121 & e & 120 & e & 0 \\
\text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element} & \text{esize} & \text{element}
\end{array}
\]

\[ r[i] = \sim(x[i] \& y[i]), \quad i = 0..\text{NELEM} -1 \]
_gnandi Bit-wise NAND Immediate

NANDs an immediate with vector elements.

\[
v16_t \ _{\text{gnandi16}}(v16_t \ x, \ int \ k) \\
v32_t \ _{\text{gnandi32}}(v32_t \ x, \ int \ k) \\
v64_t \ _{\text{gnandi64}}(v64_t \ x, \ int \ k) \\
v128_t \ _{\text{gnandi128}}(v128_t \ x, \ int \ k)
\]

This function takes one 128-bit vector parameter, \( x \), and a 10-bit signed immediate value, \( k \). The vector is interpreted as containing elements of \( esize \)-bit integers. The immediate value is sign-extended to \( esize \) bits and bit-wise NANDed, element by element, with the vector, producing a 128-bit result vector of \( esize \)-bit elements.

\[
r[i] = \neg(x[i] \ & \ k), \quad i = 0..\text{NELEM} -1
\]
Bit-wise Triple-NAND

NANDs elements of three vectors.

\[
\begin{align*}
\text{v8}_t & \ _\text{gnaaand8}(\text{v8}_t \ x, \ \text{v8}_t \ y, \ \text{v8}_t \ z) \\
\text{v16}_t & \ _\text{gnaaand16}(\text{v16}_t \ x, \ \text{v16}_t \ y, \ \text{v16}_t \ z) \\
\text{v32}_t & \ _\text{gnaaand32}(\text{v32}_t \ x, \ \text{v32}_t \ y, \ \text{v32}_t \ z) \\
\text{v64}_t & \ _\text{gnaaand64}(\text{v64}_t \ x, \ \text{v64}_t \ y, \ \text{v64}_t \ z) \\
\text{v128}_t & \ _\text{gnaaand128}(\text{v128}_t \ x, \ \text{v128}_t \ y, \ \text{v128}_t \ z)
\end{align*}
\]

This function takes three 128-bit vector parameters, \(x\), \(y\) and \(z\). The vectors are NANDed, bit by bit, producing a 128-bit result vector of \(esize\)-bit elements.

This is a special case of the \_gboolean operation (page 125): \_gnaaand\((x,y) = \_gboolean(x, y, z, 127)\)

\[
\begin{align*}
\text{r}[i] = \neg(x[i] \ \& \ y[i] \ \& \ z[i]), \quad i = 0..\text{NELEM} -1
\end{align*}
\]
\_gandn

**Bit-wise AND-NOT**

ANDs element of first vector with complement of second vector.

\[
v8\_t \ _gandn8(v8\_t \ x, v8\_t \ y) \\
v16\_t \ _gandn16(v16\_t \ x, v16\_t \ y) \\
v32\_t \ _gandn32(v32\_t \ x, v32\_t \ y) \\
v64\_t \ _gandn64(v64\_t \ x, v64\_t \ y) \\
v128\_t \ _gandn128(v128\_t \ x, v128\_t \ y)
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are ANDNed, bit by bit, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

This is a special case of the \_gboolean operation (page 125):

\[
\_gandn(x,y) = \_gboolean(x, x, y, 68)
\]

\[
r[i] = x[i] \& \sim y[i], \quad i = 0..\text{NELEM}-1
\]
_gor

Bit-wise OR

ORs elements of two vectors.

\( \text{v8_t \ _gor8(v8_t \ x, v8_t \ y)} \)
\( \text{v16_t \ _gor16(v16_t \ x, v16_t \ y)} \)
\( \text{v32_t \ _gor32(v32_t \ x, v32_t \ y)} \)
\( \text{v64_t \ _gor64(v64_t \ x, v64_t \ y)} \)
\( \text{v128_t \ _gor128(v128_t \ x, v128_t \ y)} \)

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are ORed, bit by bit, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

This is a special case of the _gboolean operation (page 125):

\[ _\text{gor}(x, y) = _\text{gboolean}(x, x, y, 238) \]

\[ r[i] = x[i] \ | \ y[i], \quad i = 0..\text{NELEM} -1 \]
_gori

Bit-wise OR Immediate

ORs an immediate with vector elements.

v16_t _gori16(v16_t x, int k)
v32_t _gori32(v32_t x, int k)
v64_t _gori64(v64_t x, int k)
v128_t _gori128(v128_t x, int k)

This function takes one 128-bit vector parameter, x, and a 10-bit signed immediate value, k. The vector is interpreted as containing elements of esize-bit integers. The immediate value is sign-extended to esize bits and bit-wise ORed, element by element, with the vector, producing a 128-bit result vector of esize-bit elements.

\[ r[i] = x[i] \mathbin{\|} k, \quad i = 0..\text{NELEM} - 1 \]
_goooor

Bit-wise Triple-OR

ORs elements of three vectors.

v8_t _goooor8(v8_t x, v8_t y, v8_t z)
v16_t _goooor16(v16_t x, v16_t y, v16_t z)
v32_t _goooor32(v32_t x, v32_t y, v32_t z)
v64_t _goooor64(v64_t x, v64_t y, v64_t z)
v128_t _goooor128(v128_t x, v128_t y, v128_t z)

This function takes three 128-bit vector parameters, x, y and z. The vectors are ORed, bit by bit, producing a 128-bit result vector of esize-bit elements.

This is a special case of the _gboolean operation (page 125):

_goooor(x,y) = _gboolean(x, y, z, 254)

\[ r[i] = x[i] \mid y[i] \mid z[i], \quad i = 0..\text{NELEM}-1 \]
NORs elements of two vectors.

\[
\begin{align*}
\text{v8_t } & \text{ _gnor8(v8_t } \ x, \ \text{v8_t } \ y) \\
\text{v16_t } & \text{ _gnor16(v16_t } \ x, \ \text{v16_t } \ y) \\
\text{v32_t } & \text{ _gnor32(v32_t } \ x, \ \text{v32_t } \ y) \\
\text{v64_t } & \text{ _gnor64(v64_t } \ x, \ \text{v64_t } \ y) \\
\text{v128_t } & \text{ _gnor128(v128_t } \ x, \ \text{v128_t } \ y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\). The vectors are NORed, bit by bit, producing a 128-bit result vector of \(\text{esize}\)-bit elements.

This is a special case of the \_gboolean operation (page 125):

\[
\text{gnor(x,y) = gboolean(x, x, y, 17)}
\]

\[
\begin{align*}
\text{r}[i] & = \sim(x[i] \mid y[i]), \quad i = 0..\text{NELEM} -1
\end{align*}
\]
_gnori

Bit-wise NOR Immediate

NORs an immediate with vector elements.

\[
v16_t \ _\text{gnori16}(v16_t \ x, \ \text{int} \ k) \\
v32_t \ _\text{gnori32}(v32_t \ x, \ \text{int} \ k) \\
v64_t \ _\text{gnori64}(v64_t \ x, \ \text{int} \ k) \\
v128_t \ _\text{gnori128}(v128_t \ x, \ \text{int} \ k)
\]

This function takes one 128-bit vector parameter, \(x\), and a 10-bit signed immediate value, \(k\). The vector is interpreted as containing elements of \(esize\)-bit integers. The immediate value is sign-extended to \(esize\) bits and bit-wise NORred, element by element, with the vector, producing a 128-bit result vector of \(esize\)-bit elements.

\[
r[i] = \neg(x[i] \ | \ k), \quad i = 0..\text{NELEM} -1
\]

496-044.aps
**_gnooor**

Bit-wise Triple-NOR

NORs elements of three vectors.

| v8_t  _gnoo8r8(v8_t x, v8_t y, v8_t z) |
| v16_t _gnoo16r16(v16_t x, v16_t y, v16_t z) |
| v32_t _gnoo32r32(v32_t x, v32_t y, v32_t z) |
| v64_t _gnoo64r64(v64_t x, v64_t y, v64_t z) |
| v128_t _gnoo128r128(v128_t x, v128_t y, v128_t z) |

This function takes three 128-bit vector parameters, x, y and z. The vectors are NORed, bit by bit, producing a 128-bit result vector of esize-bit elements.

This is a special case of the _gboolean operation (page 125):

\[ _\text{gnoo8r}(x, y) = _\text{gboolean}(x, y, z, 254) \]

\[
r[i] = \neg(x[i] | y[i] | z[i]), \quad i = 0..\text{NELEM} - 1
\]
_gorn

ORs elements of first vector with complement of second vector.

\[ \text{v8}_t \_gorn8 (\text{v8}_t \ x, \ \text{v8}_t \ y) \]
\[ \text{v16}_t \_gorn16 (\text{v16}_t \ x, \ \text{v16}_t \ y) \]
\[ \text{v32}_t \_gorn32 (\text{v32}_t \ x, \ \text{v32}_t \ y) \]
\[ \text{v64}_t \_gorn64 (\text{v64}_t \ x, \ \text{v64}_t \ y) \]
\[ \text{v128}_t \_gorn128 (\text{v128}_t \ x, \ \text{v128}_t \ y) \]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The complement of the second vector is ORed bit by bit with the first vector, producing a 128-bit result vector of esize-bit elements.

This is a special case of the _gboolean operation (page 125):

\[ _\text{gorn}(x,y) = _\text{gboolean}(x, \ x, \ y, \ 221) \]

\[ r[i] = x[i] | \sim y[i], \quad i = 0..\text{NELEM} -1 \]
_gxor Bit-wise XOR

XORs elements of two vectors.

\[
\begin{align*}
\text{v8}_t \ _\text{gxor8} & (\text{v8}_t \ x, \ \text{v8}_t \ y) \\
\text{v16}_t \ _\text{gxor16} & (\text{v16}_t \ x, \ \text{v16}_t \ y) \\
\text{v32}_t \ _\text{gxor32} & (\text{v32}_t \ x, \ \text{v32}_t \ y) \\
\text{v64}_t \ _\text{gxor64} & (\text{v64}_t \ x, \ \text{v64}_t \ y) \\
\text{v128}_t \ _\text{gxor128} & (\text{v128}_t \ x, \ \text{v128}_t \ y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are XORed, bit by bit, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

This is a special case of the _gboolean operation (page 125):

\[ \_\text{gxor}(x,y) = \_\text{gboolean}(x, x, y, 102) \]

\[
\begin{array}{c}
\text{x} \\
\hline
\text{element} & \text{element} & \text{element} & \text{element} \\
\hline
\text{esize} & \text{esize} & \text{esize} & \text{esize} \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{y} \\
\hline
\text{element} & \text{element} & \text{element} & \text{element} \\
\hline
\text{esize} & \text{esize} & \text{esize} & \text{esize} \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{r} \\
\hline
\text{element} & \text{element} & \text{element} & \text{element} \\
\hline
\text{esize} & \text{esize} & \text{esize} & \text{esize} \\
\hline
\text{esize} & \text{esize} & \text{esize} & \text{esize} \\
\hline
\end{array}
\]

\[
r[i] = x[i] \ ^{\wedge} y[i], \quad i = 0..\text{NELEM} -1
\]
_gxori

Bit-wise XOR Immediate

XORs an immediate with vector elements.

v8_t _gxori8(v8_t x, int k)
v16_t _gxori16(v16_t x, int k)
v32_t _gxori32(v32_t x, int k)
v64_t _gxori64(v64_t x, int k)
v128_t _gxori128(v128_t x, int k)

This function takes one 128-bit vector parameter, x, and a 10-bit signed immediate value, k. The vector is interpreted as containing elements of esize-bit integers. The immediate value is sign-extended to esize bits and bit-wise XORed, element by element, with the vector, producing a 128-bit result vector of esize-bit elements.

\[ r[i] = x[i] \oplus k, \quad i = 0..\text{NELEM}-1 \]
**_gxxxor**  

Bit-wise Triple-XOR

XORs elements of three vectors.

\[
\begin{align*}
\text{v8_t } _\text{gxxxor8} & (\text{v8_t } x, \text{v8_t } y, \text{v8_t } z) \\
\text{v16_t } _\text{gxxxor16} & (\text{v16_t } x, \text{v16_t } y, \text{v16_t } z) \\
\text{v32_t } _\text{gxxxor32} & (\text{v32_t } x, \text{v32_t } y, \text{v32_t } z) \\
\text{v64_t } _\text{gxxxor64} & (\text{v64_t } x, \text{v64_t } y, \text{v64_t } z) \\
\text{v128_t } _\text{gxxxor128} & (\text{v128_t } x, \text{v128_t } y, \text{v128_t } z)
\end{align*}
\]

This function takes three 128-bit vector parameters, \( x \), \( y \) and \( z \). The vectors are XORed, bit by bit, producing a 128-bit result vector of esize-bit elements.

This is a special case of the _gboolean operation (page 125):

\[
_\text{gxxxor}(x, y) = _\text{gboolean}(x, y, z, 150)
\]

\[
\begin{align*}
x & | 127 & \text{esize} & \text{esize} & \text{esize} & \text{esize} \\
& \text{element} & \text{element} & \text{element} & \text{element} \\

y & | 127 & \text{esize} & \text{esize} & \text{esize} & \text{esize} \\
& \text{element} & \text{element} & \text{element} & \text{element} \\

z & | 127 & \text{esize} & \text{esize} & \text{esize} & \text{esize} \\
& \text{element} & \text{element} & \text{element} & \text{element} \\

r & | 127 & 0 & \text{esize} & \text{esize} & \text{esize} & \text{esize} \\
& \text{element} & \text{element} & \text{element} & \text{element} \\
& 0 & \text{esize} \\

\end{align*}
\]

\[
r[i] = x[i] \ ^\land \ y[i] \ ^\land \ z[i], \quad i = 0 \ldots \text{NELEM} - 1
\]
_gxnor

Bit-wise XNOR

XNORs elements of two vectors.

\[ \text{v8}_t \ _\text{gxnor8}(\text{v8}_t \ x, \ \text{v8}_t \ y) \]
\[ \text{v16}_t \ _\text{gxnor16}(\text{v16}_t \ x, \ \text{v16}_t \ y) \]
\[ \text{v32}_t \ _\text{gxnor32}(\text{v32}_t \ x, \ \text{v32}_t \ y) \]
\[ \text{v64}_t \ _\text{gxnor64}(\text{v64}_t \ x, \ \text{v64}_t \ y) \]
\[ \text{v128}_t \ _\text{gxnor128}(\text{v128}_t \ x, \ \text{v128}_t \ y) \]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are XNORed, bit by bit, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

This is a special case of the \( _\text{gboolean} \) operation (page 125):

\[ _\text{gxnor}(x, y) = _\text{gboolean}(x, x, y, 153) \]

\[ r[i] = \sim(x[i] \ ^ \ sim y[i]), \ \ i = 0..\text{NELEM} -1 \]
**_gnxxxor**  

**Bit-wise Triple-XNOR**

XNORs elements of three vectors.

\[
\begin{align*}
\text{v8_t } & \text{ _gnxxxor8 }(\text{v8_t } x, \text{v8_t } y, \text{v8_t } z) \\
\text{v16_t } & \text{ _gnxxxor16 }(\text{v16_t } x, \text{v16_t } y, \text{v16_t } z) \\
\text{v32_t } & \text{ _gnxxxor32 }(\text{v32_t } x, \text{v32_t } y, \text{v32_t } z) \\
\text{v64_t } & \text{ _gnxxxor64 }(\text{v64_t } x, \text{v64_t } y, \text{v64_t } z) \\
\text{v128_t } & \text{ _gnxxxor128 }(\text{v128_t } x, \text{v128_t } y, \text{v128_t } z)
\end{align*}
\]

This function takes three 128-bit vector parameters, \(x\), \(y\) and \(z\). The vectors are XNORed, bit by bit, producing a 128-bit result vector of \(\text{esize}\)-bit elements.

This is a special case of the \_gboolean operation (page 125):

\[
\text{ _gnxxxor(x,y) } = \text{ _gboolean(x, y, z, 105) }
\]

\[
\begin{array}{cccccccc}
\text{x} & 0 & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{element} & \text{element} & \text{element} & \text{element} \\
\text{y} & 0 & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{element} & \text{element} & \text{element} & \text{element} \\
\text{z} & 0 & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{element} & \text{element} & \text{element} & \text{element} \\
\hline
\text{r} & 0 & \text{esize} & \text{esize} & \text{esize} & \text{esize} & \text{element} & \text{element} & \text{element} & \text{element} \\
\end{array}
\]

\[
r[i] = \neg(x[i] \oplus y[i] \oplus z[i]), \quad i = 0..\text{NELEM} -1
\]
_gmux Bit-wise Multiplex

Selects bits in one of two parameters, based on third parameter’s corresponding bits.

\[ v8_t \ _gmux8(v8_t \ x, v8_t \ y, v8_t \ z) \]
\[ v16_t \ _gmux16(v16_t \ x, v16_t \ y, v16_t \ z) \]
\[ v32_t \ _gmux32(v32_t \ x, v32_t \ y, v32_t \ z) \]
\[ v64_t \ _gmux64(v64_t \ x, v64_t \ y, v64_t \ z) \]
\[ v128_t \ _gmux128(v128_t \ x, v128_t \ y, v128_t \ z) \]

This function takes three 128-bit parameters, \( x, y, \) and \( z \). The bits in \( x \) select, bit-by-bit, the bit in \( y \) or \( z \) that is copied to the result. If the \( x \) bit is set to 1, the corresponding bit in \( y \) is copied. If the \( x \) bit is cleared to 0, the corresponding bit in \( z \) is copied.

\[
\begin{align*}
\text{r}_i &= x_i \ ? \ y_i : z_i, \quad i = 0..127 \\
\end{align*}
\]
_gboolean Bit-wise Triple Boolean

Performs one of 256 boolean operations on elements of three vectors.

v1_t gboolean1(v1_t x, v1_t y, v1_t z, int k)
v2_t gboolean2(v2_t x, v2_t y, v2_t z, int k)
v4_t gboolean4(v4_t x, v4_t y, v4_t z, int k)
v8_t gboolean8(v8_t x, v8_t y, v8_t z, int k)
v16_t gboolean16(v16_t x, v16_t y, v16_t z, int k)
v32_t gboolean32(v32_t x, v32_t y, v32_t z, int k)
v64_t gboolean64(v64_t x, v64_t y, v64_t z, int k)
v128_t gboolean128(v128_t x, v128_t y, v128_t z, int k)

This function takes three 128-bit parameters, x, y, and z, and an 8-bit immediate value, k. The boolean function specified by k is performed on each bit of x, y, and z, producing a 128-bit result vector of 1-bit elements.

The 256 immediate values that specify the operation are shown in Table 19. The rightmost column of this table lists equivalent mnemonics, which are listed in Table 20 on page 134 with their cross-references to the k values in Table 19. Depending on the value of k chosen, software may interpret the v1_t data type, shown in the syntax above, in any of several ways.

\[
r_i = (k >> (4x_i + 2y_i + z_i)) \& 1, \quad i = 0..127
\]
<table>
<thead>
<tr>
<th>Value of k</th>
<th>Boolean Operation</th>
<th>Equivalent Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>_gzero</td>
</tr>
<tr>
<td>1</td>
<td>~(y[i]</td>
<td>z[i]</td>
</tr>
<tr>
<td>2</td>
<td>~(y[i]</td>
<td>x[i]) &amp; z[i]</td>
</tr>
<tr>
<td>3</td>
<td>~(y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>4</td>
<td>~(z[i]</td>
<td>x[i]) &amp; y[i]</td>
</tr>
<tr>
<td>5</td>
<td>~(z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>6</td>
<td>~x[i] &amp; (z[i] ^ y[i])</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>((z[i] &amp; y[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>8</td>
<td>~x[i] &amp; z[i] &amp; y[i]</td>
<td>_gnaaand(x,y,z)</td>
</tr>
<tr>
<td>9</td>
<td>((z[i] ^ y[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>10</td>
<td>~x[i] &amp; z[i]</td>
<td>_gandn(z,x)</td>
</tr>
<tr>
<td>11</td>
<td>((~z[i] &amp; y[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>12</td>
<td>~x[i] &amp; y[i]</td>
<td>_gandn(y,x)</td>
</tr>
<tr>
<td>13</td>
<td>(~y[i] &amp; z[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>14</td>
<td>~x[i] &amp; (z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>15</td>
<td>~x[i]</td>
<td>_gnot(x)</td>
</tr>
<tr>
<td>16</td>
<td>~(z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>17</td>
<td>~(z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>18</td>
<td>~y[i] &amp; (z[i] ^ x[i])</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>((z[i] &amp; x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>20</td>
<td>(y[i] ^ x[i]) &amp; ~z[i]</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>~(z[i]</td>
<td>(y[i] &amp; x[i]))</td>
</tr>
<tr>
<td>22</td>
<td>(z[i]</td>
<td>(y[i] &amp; x[i]))</td>
</tr>
<tr>
<td>23</td>
<td>((z[i]</td>
<td>(y[i] &amp; x[i]))</td>
</tr>
<tr>
<td>24</td>
<td>(y[i] ^ x[i]) &amp; (z[i] ^ x[i])</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>z[i]</td>
<td>(~y[i]</td>
</tr>
<tr>
<td>26</td>
<td>(z[i]</td>
<td>(y[i] &amp; x[i]))</td>
</tr>
<tr>
<td>27</td>
<td>(z[i] &amp; x[i])</td>
<td>(~y[i]</td>
</tr>
<tr>
<td>28</td>
<td>((z[i] &amp; x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>29</td>
<td>(y[i] &amp; x[i])</td>
<td>(~z[i]</td>
</tr>
</tbody>
</table>
Table 19. Bit-wise Triple Boolean Operations (continued)

<table>
<thead>
<tr>
<th>Value of k</th>
<th>Boolean Operation</th>
<th>Equivalent Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>(z[i]</td>
<td>y[i]) ^ x[i]</td>
</tr>
<tr>
<td>31</td>
<td>~((z[i]</td>
<td>y[i]) &amp; x[i])</td>
</tr>
<tr>
<td>32</td>
<td>~y[i] &amp; z[i] &amp; x[i]</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>~((z[i] ^ x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>34</td>
<td>~y[i] &amp; z[i]</td>
<td>_gandn(z,y)</td>
</tr>
<tr>
<td>35</td>
<td>~((~z[i] &amp; x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>36</td>
<td>(y[i] ^ x[i]) &amp; (z[i] ^ y[i])</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>(~x[i]</td>
<td>(z[i] &amp; y[i])) ^ z[i]</td>
</tr>
<tr>
<td>38</td>
<td>y[i] ^ (z[i]</td>
<td>(y[i] &amp; x[i]))</td>
</tr>
<tr>
<td>39</td>
<td>(z[i] &amp; y[i]) ^ (~x[i]</td>
<td>z[i])</td>
</tr>
<tr>
<td>40</td>
<td>(y[i] ^ x[i]) &amp; z[i]</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>(~z[i]</td>
<td>(y[i] &amp; x[i])) ^ (y[i]</td>
</tr>
<tr>
<td>42</td>
<td>(~y[i] &amp; x[i]) &amp; z[i]</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>~y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>44</td>
<td>(y[i] ^ x[i]) &amp; (z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>45</td>
<td>(~z[i]</td>
<td>y[i]) ^ x[i]</td>
</tr>
<tr>
<td>46</td>
<td>(y[i] &amp; x[i]) ^ (z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>47</td>
<td>~x[i]</td>
<td>(~y[i] &amp; z[i])</td>
</tr>
<tr>
<td>48</td>
<td>~y[i] &amp; x[i]</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>~((~x[i] &amp; z[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>50</td>
<td>~y[i] &amp; (z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>51</td>
<td>~y[i]</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>y[i] ^ ((z[i] &amp; y[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>53</td>
<td>(y[i] &amp; x[i]) ^ (~z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>54</td>
<td>y[i] ^ (z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>55</td>
<td>~(y[i] &amp; (z[i]</td>
<td>x[i]))</td>
</tr>
<tr>
<td>56</td>
<td>(y[i] ^ x[i]) &amp; (z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>57</td>
<td>y[i] ^ (~z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>58</td>
<td>(y[i] &amp; x[i]) ^ (z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>59</td>
<td>(~x[i] &amp; z[i])</td>
<td>~y[i]</td>
</tr>
<tr>
<td>60</td>
<td>y[i] ^ x[i]</td>
<td>_gxor(y,x)</td>
</tr>
</tbody>
</table>
Table 19. Bit-wise Triple Boolean Operations (continued)

<table>
<thead>
<tr>
<th>Value of k</th>
<th>Boolean Operation</th>
<th>Equivalent Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>(~(z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>62</td>
<td>y[i] ^ ((~y[i] &amp; z[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>63</td>
<td>(~(y[i] &amp; x[i]))</td>
<td>_gnand(x, y)</td>
</tr>
<tr>
<td>64</td>
<td>(~z[i] &amp; y[i] &amp; x[i])</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>(~((y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>66</td>
<td>(z[i] ^ x[i]) &amp; (z[i] ^ y[i])</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>((~x[i]</td>
<td>(z[i] &amp; y[i])) ^ y[i])</td>
</tr>
<tr>
<td>68</td>
<td>(~z[i] &amp; y[i])</td>
<td>_gandn(y, z)</td>
</tr>
<tr>
<td>69</td>
<td>(~((~y[i] &amp; x[i])</td>
<td>z[i]))</td>
</tr>
<tr>
<td>70</td>
<td>z[i] ^ ((z[i] &amp; x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>71</td>
<td>(z[i] &amp; y[i]) ^ (~x[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>72</td>
<td>(z[i] ^ x[i]) &amp; y[i]</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>((~y[i]</td>
<td>(z[i] &amp; x[i])) ^ (z[i]</td>
</tr>
<tr>
<td>74</td>
<td>(z[i] ^ x[i]) &amp; (z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>75</td>
<td>((~y[i]</td>
<td>z[i])</td>
</tr>
<tr>
<td>76</td>
<td>(~(z[i] &amp; x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>77</td>
<td>(~(z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>78</td>
<td>(z[i] &amp; x[i]) ^ (z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>79</td>
<td>(~x[i]</td>
<td>(~z[i] &amp; y[i]))</td>
</tr>
<tr>
<td>80</td>
<td>(~z[i] &amp; x[i])</td>
<td>_gandn(x, z)</td>
</tr>
<tr>
<td>81</td>
<td>(~((~x[i] &amp; y[i])</td>
<td>z[i]))</td>
</tr>
<tr>
<td>82</td>
<td>z[i] ^ ((z[i] &amp; y[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>83</td>
<td>(z[i] &amp; x[i]) ^ (~y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>84</td>
<td>(~z[i] &amp; (y[i]</td>
<td>x[i]))</td>
</tr>
<tr>
<td>85</td>
<td>(~z[i])</td>
<td>_gnot(z)</td>
</tr>
<tr>
<td>86</td>
<td>z[i] ^ (y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>87</td>
<td>(~(z[i] &amp; (y[i]</td>
<td>x[i])))</td>
</tr>
<tr>
<td>88</td>
<td>(z[i] ^ x[i]) &amp; (y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>89</td>
<td>z[i] ^ (~y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>90</td>
<td>z[i] ^ x[i]</td>
<td>_gxor(z, x)</td>
</tr>
<tr>
<td>91</td>
<td>(~(y[i]</td>
<td>x[i])</td>
</tr>
</tbody>
</table>
### Table 19. Bit-wise Triple Boolean Operations (continued)

<table>
<thead>
<tr>
<th>Value of k</th>
<th>Boolean Operation</th>
<th>Equivalent Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>92</td>
<td>((z[i] &amp; x[i]) \oplus (y[i] \mid x[i]))</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>((\neg x[i] &amp; y[i]) \mid \neg z[i])</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>(z[i] \oplus ((\neg z[i] &amp; y[i]) \mid x[i]))</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>(\neg (z[i] &amp; x[i]))</td>
<td>_gnor(x,z)</td>
</tr>
<tr>
<td>96</td>
<td>((z[i] \oplus y[i]) &amp; x[i])</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>((\neg x[i] \mid (z[i] &amp; y[i])) \oplus (z[i] \mid y[i]))</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>((z[i] \oplus y[i]) &amp; (z[i] \mid x[i]))</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>((\neg x[i] \mid z[i]) \oplus y[i])</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>((z[i] \oplus y[i]) &amp; (y[i] \mid x[i]))</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>(z[i] \oplus (\neg x[i] \mid y[i]))</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>(z[i] \oplus y[i])</td>
<td>_gxor(y,z)</td>
</tr>
<tr>
<td>103</td>
<td>(\neg (y[i] \mid x[i]) \mid (z[i] \oplus y[i]))</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>((z[i] \oplus (y[i] &amp; x[i])) &amp; (y[i] \mid x[i]))</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>(\neg y[i] \oplus z[i] \oplus x[i])</td>
<td>_gnxxxor(x,y,z)</td>
</tr>
<tr>
<td>106</td>
<td>(z[i] \oplus (y[i] &amp; x[i]))</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>(\neg (y[i] \mid x[i]) \mid (z[i] \oplus (y[i] &amp; x[i])))</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>((z[i] &amp; x[i]) \oplus y[i])</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>(\neg (z[i] \mid x[i]) \mid ((z[i] &amp; x[i]) \oplus y[i]))</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>((\neg x[i] &amp; z[i]) \mid (z[i] \oplus y[i]))</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>(\neg x[i] \mid (z[i] \oplus y[i]))</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>(\neg (z[i] &amp; y[i]) &amp; x[i])</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>(\neg (z[i] \mid y[i]) \mid ((z[i] &amp; y[i]) &amp; x[i]))</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>((z[i] &amp; y[i]) \oplus (z[i] \mid x[i]))</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>(\neg y[i] \mid (\neg z[i] &amp; x[i]))</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>((z[i] &amp; y[i]) \oplus (y[i] \mid x[i]))</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>((\neg y[i] &amp; x[i]) \mid \neg z[i])</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>(z[i] \oplus ((\neg z[i] &amp; x[i]) \mid y[i]))</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>(\neg (z[i] &amp; y[i]))</td>
<td>_gnand(y,z)</td>
</tr>
<tr>
<td>120</td>
<td>((z[i] &amp; y[i]) \oplus x[i])</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>(\neg (z[i] \mid y[i]) \mid ((z[i] &amp; y[i]) &amp; x[i]))</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>((\neg y[i] &amp; z[i]) \mid (z[i] \oplus x[i]))</td>
<td></td>
</tr>
</tbody>
</table>
Table 19. Bit-wise Triple Boolean Operations (continued)

<table>
<thead>
<tr>
<th>Value of k</th>
<th>Boolean Operation</th>
<th>Equivalent Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>~y[i]</td>
<td>(z[i] ^ x[i])</td>
</tr>
<tr>
<td>124</td>
<td>(y[i] ^ x[i])</td>
<td>( ~z[i] &amp; y[i])</td>
</tr>
<tr>
<td>125</td>
<td>(y[i] ^ x[i])</td>
<td>~z[i]</td>
</tr>
<tr>
<td>126</td>
<td>(y[i] ^ x[i])</td>
<td>(z[i] ^ x[i])</td>
</tr>
<tr>
<td>127</td>
<td>~(z[i] &amp; y[i] &amp; x[i])</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>z[i] &amp; y[i] &amp; x[i]</td>
<td>_gaaand(x,y,z)</td>
</tr>
<tr>
<td>129</td>
<td>((y[i] ^ x[i])</td>
<td>(z[i] ^ x[i]))</td>
</tr>
<tr>
<td>130</td>
<td>z[i] &amp; (~x[i] ^ y[i])</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>~(y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>132</td>
<td>(~x[i] ^ z[i]) &amp; y[i]</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>~(z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>134</td>
<td>((y[i] ^ x[i]) &amp; (z[i]</td>
<td>y[i]))</td>
</tr>
<tr>
<td>135</td>
<td>(~x[i] ^ (z[i] &amp; y[i]))</td>
<td></td>
</tr>
<tr>
<td>136</td>
<td>z[i] &amp; y[i]</td>
<td>_gand(y,z)</td>
</tr>
<tr>
<td>137</td>
<td>~y[i] ^ ((~y[i] &amp; x[i])</td>
<td>z[i])</td>
</tr>
<tr>
<td>138</td>
<td>z[i] &amp; (~x[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>139</td>
<td>~(y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>140</td>
<td>(~x[i]</td>
<td>z[i]) &amp; y[i]</td>
</tr>
<tr>
<td>141</td>
<td>~(z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>142</td>
<td>((z[i] ^ y[i]) &amp; x[i]) ^ (z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>143</td>
<td>~x[i]</td>
<td>(z[i] &amp; y[i])</td>
</tr>
<tr>
<td>144</td>
<td>(~y[i] ^ z[i]) &amp; x[i]</td>
<td></td>
</tr>
<tr>
<td>145</td>
<td>~(z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>146</td>
<td>((y[i] ^ x[i]) &amp; (z[i]</td>
<td>x[i]))</td>
</tr>
<tr>
<td>147</td>
<td>~y[i] ^ (z[i] &amp; x[i])</td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>y[i] ^ x[i]</td>
<td>(z[i] &amp; (y[i]</td>
</tr>
<tr>
<td>149</td>
<td>~z[i]</td>
<td>(y[i] &amp; x[i])</td>
</tr>
<tr>
<td>150</td>
<td>y[i] ^ x[i]</td>
<td>z[i]</td>
</tr>
<tr>
<td>151</td>
<td>~(y[i] &amp; x[i])</td>
<td>(z[i] &amp; (y[i]</td>
</tr>
<tr>
<td>152</td>
<td>(~y[i] ^ z[i]) &amp; (z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>153</td>
<td>~y[i] ^ z[i]</td>
<td>_gxnor(y,z)</td>
</tr>
</tbody>
</table>
Table 19. Bit-wise Triple Boolean Operations (continued)

<table>
<thead>
<tr>
<th>Value of k</th>
<th>Boolean Operation</th>
<th>Equivalent Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>154</td>
<td>$\neg y[i] &amp; x[i] \oplus z[i]$</td>
<td></td>
</tr>
<tr>
<td>155</td>
<td>$y[i] \oplus (z[i] &amp; (y[i]</td>
<td>x[i]))$</td>
</tr>
<tr>
<td>156</td>
<td>$(\neg z[i] &amp; x[i]) \oplus y[i]$</td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>$\neg z[i] \oplus (y[i] &amp; (z[i]</td>
<td>x[i]))$</td>
</tr>
<tr>
<td>158</td>
<td>$((z[i]</td>
<td>y[i]) &amp; x[i])</td>
</tr>
<tr>
<td>159</td>
<td>$((z[i]</td>
<td>y[i]) &amp; x[i])$</td>
</tr>
<tr>
<td>160</td>
<td>$z[i] &amp; x[i]$</td>
<td>_gand(x,z)</td>
</tr>
<tr>
<td>161</td>
<td>$\neg x[i] \oplus ((\neg x[i] &amp; y[i])</td>
<td>z[i])$</td>
</tr>
<tr>
<td>162</td>
<td>$z[i] &amp; (\neg y[i]</td>
<td>x[i])$</td>
</tr>
<tr>
<td>163</td>
<td>$\neg (y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>164</td>
<td>$((x[i]</td>
<td>z[i]) &amp; (z[i]</td>
</tr>
<tr>
<td>165</td>
<td>$\neg x[i] \oplus z[i]$</td>
<td>_gandn(z,x)</td>
</tr>
<tr>
<td>166</td>
<td>$(\neg x[i] &amp; y[i]) \oplus z[i]$</td>
<td></td>
</tr>
<tr>
<td>167</td>
<td>$\neg x[i] \oplus (z[i] &amp; (y[i]</td>
<td>x[i])$</td>
</tr>
<tr>
<td>168</td>
<td>$z[i] &amp; (y[i]</td>
<td>x[i])$</td>
</tr>
<tr>
<td>169</td>
<td>$\neg z[i] \oplus (y[i]</td>
<td>x[i])$</td>
</tr>
<tr>
<td>170</td>
<td>$z[i]$</td>
<td>z</td>
</tr>
<tr>
<td>171</td>
<td>$\neg (y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>172</td>
<td>$((z[i]</td>
<td>y[i]) &amp; x[i]) \oplus y[i]$</td>
</tr>
<tr>
<td>173</td>
<td>$\neg z[i] \oplus ((z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>174</td>
<td>$(\neg x[i] &amp; y[i])</td>
<td>z[i]$</td>
</tr>
<tr>
<td>175</td>
<td>$\neg x[i]</td>
<td>z[i]$</td>
</tr>
<tr>
<td>176</td>
<td>$(\neg y[i]</td>
<td>z[i]) &amp; x[i]$</td>
</tr>
<tr>
<td>177</td>
<td>$\neg (z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>178</td>
<td>$((z[i]</td>
<td>x[i]) &amp; y[i]) \oplus (z[i]</td>
</tr>
<tr>
<td>179</td>
<td>$y[i]</td>
<td>(z[i] &amp; x[i])$</td>
</tr>
<tr>
<td>180</td>
<td>$(\neg z[i] &amp; y[i]) \oplus x[i]$</td>
<td></td>
</tr>
<tr>
<td>181</td>
<td>$\neg z[i] \oplus ((z[i]</td>
<td>y[i]) &amp; x[i])$</td>
</tr>
<tr>
<td>182</td>
<td>$(y[i] \oplus (z[i]</td>
<td>x[i]))</td>
</tr>
<tr>
<td>183</td>
<td>$\neg ((z[i]</td>
<td>x[i]) &amp; y[i])$</td>
</tr>
<tr>
<td>184</td>
<td>$((z[i]</td>
<td>x[i]) &amp; y[i]) \oplus x[i]$</td>
</tr>
</tbody>
</table>
Table 19. Bit-wise Triple Boolean Operations (continued)

<table>
<thead>
<tr>
<th>Value of k</th>
<th>Boolean Operation</th>
<th>Equivalent Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>185</td>
<td>~z[i] ^ ((z[i] &amp; x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>186</td>
<td>(~y[i] &amp; x[i])</td>
<td>z[i]</td>
</tr>
<tr>
<td>187</td>
<td>~y[i]</td>
<td>z[i]</td>
</tr>
<tr>
<td>188</td>
<td>(y[i] ^ x[i])</td>
<td>(z[i] &amp; y[i])</td>
</tr>
<tr>
<td>189</td>
<td>~(z[i] ^ x[i]) &amp; (z[i] ^ y[i])</td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>(y[i] ^ x[i])</td>
<td>z[i]</td>
</tr>
<tr>
<td>191</td>
<td>~(y[i] &amp; x[i])</td>
<td>z[i]</td>
</tr>
<tr>
<td>192</td>
<td>y[i] &amp; x[i]</td>
<td>_gand(x,y)</td>
</tr>
<tr>
<td>193</td>
<td>~x[i] ^ ((~x[i] &amp; z[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>194</td>
<td>(~x[i] ^ y[i]) &amp; (z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>195</td>
<td>~x[i] ^ y[i]</td>
<td>_gxnor(x,y)</td>
</tr>
<tr>
<td>196</td>
<td>y[i] &amp; (~z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>197</td>
<td>~(z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>198</td>
<td>(~x[i] &amp; z[i]) ^ y[i]</td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>~x[i] ^ (y[i] &amp; (z[i]</td>
<td>x[i]))</td>
</tr>
<tr>
<td>200</td>
<td>y[i] &amp; (z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>201</td>
<td>~y[i] ^ (z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>202</td>
<td>((z[i] ^ y[i]) &amp; x[i]) ^ z[i]</td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>~y[i] ^ ((z[i] &amp; y[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>204</td>
<td>y[i]</td>
<td>y</td>
</tr>
<tr>
<td>205</td>
<td>~(z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>206</td>
<td>(~x[i] &amp; z[i])</td>
<td>y[i]</td>
</tr>
<tr>
<td>207</td>
<td>~x[i]</td>
<td>y[i]</td>
</tr>
<tr>
<td>208</td>
<td>(~z[i]</td>
<td>y[i]) &amp; x[i]</td>
</tr>
<tr>
<td>209</td>
<td>~(z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>210</td>
<td>(~y[i] &amp; z[i]) ^ x[i]</td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>~y[i] ^ ((z[i]</td>
<td>y[i]) &amp; x[i])</td>
</tr>
<tr>
<td>212</td>
<td>((y[i] ^ x[i]) &amp; z[i]) ^ (y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>213</td>
<td>~z[i]</td>
<td>(y[i] &amp; x[i])</td>
</tr>
<tr>
<td>214</td>
<td>(z[i] ^ (y[i]</td>
<td>x[i]))</td>
</tr>
<tr>
<td>215</td>
<td>~((y[i] ^ x[i]) &amp; z[i])</td>
<td></td>
</tr>
</tbody>
</table>
Table 19. Bit-wise Triple Boolean Operations (continued)

<table>
<thead>
<tr>
<th>Value of ( k )</th>
<th>Boolean Operation</th>
<th>Equivalent Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>216</td>
<td>(((y[i] ^ x[i]) &amp; z[i]) ^ x[i])</td>
<td></td>
</tr>
<tr>
<td>217</td>
<td>(~y[i] ^ (z[i]</td>
<td>(y[i] &amp; x[i])))</td>
</tr>
<tr>
<td>218</td>
<td>((z[i] ^ x[i])</td>
<td>(z[i] &amp; y[i]))</td>
</tr>
<tr>
<td>219</td>
<td>(~((y[i] ^ x[i]) &amp; (z[i] ^ y[i])))</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>(~z[i] &amp; x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>221</td>
<td>(~z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>222</td>
<td>((z[i] ^ x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>223</td>
<td>(~(z[i] &amp; x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>224</td>
<td>((z[i]</td>
<td>y[i]) &amp; x[i])</td>
</tr>
<tr>
<td>225</td>
<td>(~x[i] ^ (z[i]</td>
<td>y[i]))</td>
</tr>
<tr>
<td>226</td>
<td>(((z[i] ^ x[i]) &amp; y[i]) ^ z[i])</td>
<td></td>
</tr>
<tr>
<td>227</td>
<td>(~x[i] ^ ((z[i] &amp; x[i])</td>
<td>y[i]))</td>
</tr>
<tr>
<td>228</td>
<td>(((y[i] ^ x[i]) &amp; z[i]) ^ y[i])</td>
<td></td>
</tr>
<tr>
<td>229</td>
<td>(~x[i] ^ (z[i]</td>
<td>(y[i] &amp; x[i])))</td>
</tr>
<tr>
<td>230</td>
<td>((z[i] ^ y[i])</td>
<td>(z[i] &amp; x[i]))</td>
</tr>
<tr>
<td>231</td>
<td>(~((y[i] ^ x[i]) &amp; (z[i] ^ x[i])))</td>
<td></td>
</tr>
<tr>
<td>232</td>
<td>((z[i]</td>
<td>(y[i] &amp; x[i])) &amp; (y[i]</td>
</tr>
<tr>
<td>233</td>
<td>(~((z[i]</td>
<td>(y[i] &amp; x[i])) ^ (y[i]</td>
</tr>
<tr>
<td>234</td>
<td>(z[i]</td>
<td>(y[i] &amp; x[i]))</td>
</tr>
<tr>
<td>235</td>
<td>(z[i]</td>
<td>(~x[i] ^ y[i]))</td>
</tr>
<tr>
<td>236</td>
<td>((z[i] &amp; x[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>237</td>
<td>(~x[i] ^ z[i])</td>
<td>y[i])</td>
</tr>
<tr>
<td>238</td>
<td>(z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>239</td>
<td>(z[i]</td>
<td>~x[i]</td>
</tr>
<tr>
<td>240</td>
<td>(x[i])</td>
<td>(x)</td>
</tr>
<tr>
<td>241</td>
<td>(~(z[i]</td>
<td>y[i])</td>
</tr>
<tr>
<td>242</td>
<td>(~y[i] &amp; z[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>243</td>
<td>(~y[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>244</td>
<td>(~z[i] &amp; y[i])</td>
<td>x[i])</td>
</tr>
<tr>
<td>245</td>
<td>(~z[i]</td>
<td>x[i])</td>
</tr>
<tr>
<td>246</td>
<td>((z[i] ^ y[i])</td>
<td>x[i])</td>
</tr>
</tbody>
</table>
Table 19. Bit-wise Triple Boolean Operations (continued)

<table>
<thead>
<tr>
<th>Value of k</th>
<th>Boolean Operation</th>
<th>Equivalent Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>247</td>
<td>~ (z[i] &amp; y[i])</td>
<td>x[i]</td>
</tr>
<tr>
<td>248</td>
<td>(z[i] &amp; y[i])</td>
<td>x[i]</td>
</tr>
<tr>
<td>249</td>
<td>(~y[i] ^ z[i])</td>
<td>x[i]</td>
</tr>
<tr>
<td>250</td>
<td>z[i]</td>
<td>x[i]</td>
</tr>
<tr>
<td>251</td>
<td>z[i]</td>
<td>~y[i]</td>
</tr>
<tr>
<td>252</td>
<td>y[i]</td>
<td>x[i]</td>
</tr>
<tr>
<td>253</td>
<td>y[i]</td>
<td>~z[i]</td>
</tr>
<tr>
<td>254</td>
<td>y[i]</td>
<td>z[i]</td>
</tr>
<tr>
<td>255</td>
<td>~0</td>
<td>_gset</td>
</tr>
</tbody>
</table>

Table 20. gboolean Equivalents

<table>
<thead>
<tr>
<th>Equivalent Mnemonic</th>
<th>Boolean Operation</th>
<th>Value of k</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaaand</td>
<td>Group three-way and</td>
<td>128</td>
</tr>
<tr>
<td>gand</td>
<td>Group and</td>
<td>136</td>
</tr>
<tr>
<td>gandn</td>
<td>Group and not</td>
<td>12, 34, 68</td>
</tr>
<tr>
<td>gcopy</td>
<td>Group copy</td>
<td>136</td>
</tr>
<tr>
<td>gnaaand</td>
<td>Group three-way nand</td>
<td>8, 127</td>
</tr>
<tr>
<td>gnand</td>
<td>Group nand</td>
<td>119</td>
</tr>
<tr>
<td>gnooor</td>
<td>Group three-way nor</td>
<td>1</td>
</tr>
<tr>
<td>gnor</td>
<td>Group nor</td>
<td>3</td>
</tr>
<tr>
<td>gnot</td>
<td>Group not</td>
<td>15</td>
</tr>
<tr>
<td>gnxxxxor</td>
<td>Group three-way exclusive-nor</td>
<td>105</td>
</tr>
<tr>
<td>gooor</td>
<td>Group three-way or</td>
<td>254</td>
</tr>
<tr>
<td>gor</td>
<td>Group or</td>
<td>238</td>
</tr>
<tr>
<td>gorn</td>
<td>Group or not</td>
<td>221</td>
</tr>
<tr>
<td>gset</td>
<td>Group set</td>
<td>255</td>
</tr>
<tr>
<td>gxnor</td>
<td>Group exclusive-nor</td>
<td>153</td>
</tr>
<tr>
<td>gxor</td>
<td>Group exclusive-or</td>
<td>60, 102</td>
</tr>
<tr>
<td>gxxxor</td>
<td>Group three-way exclusive-or</td>
<td>150</td>
</tr>
<tr>
<td>gzero</td>
<td>Group zero</td>
<td>0</td>
</tr>
</tbody>
</table>
3.3.4 **Group Copy Immediate**

The Group Copy Immediate functions include:

- `__gcopyi` Copy Immediate
Copies a sign-extended immediate to all elements of a vector.

\[
\begin{align*}
\text{v8_t } &\text{ _gcopyi8(int k)} \\
\text{v16_t } &\text{ _gcopyi16(int k)} \\
\text{v32_t } &\text{ _gcopyi32(int k)} \\
\text{v64_t } &\text{ _gcopyi64(int k)} \\
\text{v128_t } &\text{ _gcopyi128(int k)}
\end{align*}
\]

This function takes a 17-bit signed immediate value, \( k \). It sign-extends \( k \) to \( esize \) bits and copies it \( NELEM \) times, producing a 128-bit result vector of \( esize \)-bit elements. In the case of \(_{gcopyi16} \) and \(_{gcopyi8} \), only the lower bits of \( k \) are used.

This function is useful for creating vectors that have constant values without having to load from memory. See the function _xcopy on page 180 to take an arbitrary scalar and copy it into each vector element.

\[
r[i] = k, \quad i = 0..NELEM -1
\]
3.3.5 **Group Compare and Set**

The Group Compare and Set functions, collectively called the \_gset<cond> functions, set a result depending on a condition. They include the following variants:

- \_gsetande Set if AND Equal Zero
- \_gsetandei Set if AND Equal Zero Immediate
- \_gsetandne Set if AND Not Equal Zero
- \_gsetandnei Set if AND Not Equal Zero Immediate
- \_gsete Set if Equal
- \_gsetef Set if Equal Floating-Point
- \_gsetei Set if Equal Immediate
- \_gsetge Set if Greater or Equal
- \_gsetgeu Set if Greater or Equal Unsigned
- \_gsetgef Set if Greater or Equal Floating-Point
- \_gsetgei Set if Greater or Equal Immediate
- \_gsetgeiu Set if Greater or Equal Immediate Unsigned
- \_gsetgez Set if Greater or Equal Zero
- \_gsetgz Set if Greater than Zero
- \_gsetl Set if Less
- \_gsetlu Set if Less Unsigned
- \_gsetlf Set if Less Floating-Point
- \_gsetli Set if Less Immediate
- \_gsetliu Set if Less Immediate Unsigned
- \_gsetlez Set if Less or Equal Zero
- \_gsetlz Set if Less than Zero
- \_gsetne Set if Not Equal
- \_gsetnei Set if Not Equal Immediate
- \_gsetlgf Set if Less or Greater Floating-Point

A typical application of \_gset, together with a \_gmux function, is finding the element-by-element minimum of two vectors without branching:

```
sel = \_gsetl(a,b)
min = \_gmux(sel,a,b)
```

Similarly, one can find the element-by-element maximum of two vectors:

```
max(x,y) = \_gmux(gsetge(x,y),x,y)
```
_gset<cond>

Sets element of result vector to 1s if corresponding elements of the source vectors meet <condition>.

\[
\begin{align*}
  v_8 \_gset<\text{cond}> & : (v_8 \_x, v_8 \_y) \\
v_{16} \_gset<\text{cond}> & : (v_{16} \_x, v_{16} \_y) \\
v_{32} \_gset<\text{cond}> & : (v_{32} \_x, v_{32} \_y) \\
v_{64} \_gset<\text{cond}> & : (v_{64} \_x, v_{64} \_y) \\
v_{128} \_gset<\text{cond}> & : (v_{128} \_x, v_{128} \_y)
\end{align*}
\]

This function set has 8 variations. The functions take two 128-bit vectors, \( x \) and \( y \). The vectors are interpreted as containing elements of \( \text{esize} \)-bit values. A condition is applied bit-wise to each element of the vector(s), producing a 128-bit result vector of \( \text{esize} \)-bit elements. The result elements are set to all 1 bits if the condition is met, or cleared to all 0 bits if the condition is not met.

The list on page 137 gives the mnemonic and English description of each function. Table 21 on page 139 shows the mnemonics, operands, \( \text{esizes} \), and conditions.

\[
r[i] = (x[i] <\text{cond}> y[i]) \ ? -1 : 0, \quad i = 0..\text{NELEM} -1
\]
### Table 21. Conditions for `set if` Functions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Data Type¹</th>
<th><code>esize</code></th>
<th>Operands and Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gsetande</td>
<td>i</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] &amp; y[i] == 0</code></td>
</tr>
<tr>
<td>gsetandne</td>
<td>i</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] &amp; y[i] != 0</code></td>
</tr>
<tr>
<td>gsete</td>
<td>i</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] == y[i]</code></td>
</tr>
<tr>
<td>gsetge</td>
<td>i</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] &gt;= y[i]</code></td>
</tr>
<tr>
<td>gsetgeu</td>
<td>ui</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] &gt;= y[i]</code></td>
</tr>
<tr>
<td>gsetl</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] &lt; y[i]</code></td>
</tr>
<tr>
<td>gsetlu</td>
<td>ui</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] &lt; y[i]</code></td>
</tr>
<tr>
<td>gsetne</td>
<td>i</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] != y[i]</code></td>
</tr>
</tbody>
</table>

1. i = signed or unsigned integer. si = signed integer. ui = unsigned integer. fp = floating-point.
Sets element of result vector to 1s based on comparison between elements of the source vector and an immediate constant.

\[ r[i] = (k <\text{cond}> x[i]) \ ? -1 : 0, \quad i = 0..\text{NELEM} -1 \]
### Table 22. Conditions for `set immediate` Functions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Data Type¹</th>
<th>size</th>
<th>Operands and Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gsetandei</td>
<td>i</td>
<td>16, 32, 64, 128</td>
<td>x[i] &amp; k == 0</td>
</tr>
<tr>
<td>gsetandnei</td>
<td>i</td>
<td>16, 32, 64, 128</td>
<td>x[i] &amp; k != 0</td>
</tr>
<tr>
<td>gsetei</td>
<td>i</td>
<td>16, 32, 64, 128</td>
<td>x[i] == k</td>
</tr>
<tr>
<td>gsetgei</td>
<td>si</td>
<td>16, 32, 64, 128</td>
<td>x[i] &gt;= k</td>
</tr>
<tr>
<td>gsetgeiu</td>
<td>ui</td>
<td>16, 32, 64, 128</td>
<td>x[i] &gt;= k</td>
</tr>
<tr>
<td>gsetli</td>
<td>si</td>
<td>16, 32, 64, 128</td>
<td>x[i] &lt; k</td>
</tr>
<tr>
<td>gsetliu</td>
<td>ui</td>
<td>16, 32, 64, 128</td>
<td>x[i] &lt; k</td>
</tr>
<tr>
<td>gsetnei</td>
<td>i</td>
<td>16, 32, 64, 128</td>
<td>x[i] != k</td>
</tr>
</tbody>
</table>

¹ i = signed or unsigned integer. si = signed integer. ui = unsigned integer. fp = floating-point.
Sets element of result vector to 1s if elements of the source vectors meet \(<\text{condition}\>\).

\[
\begin{align*}
\text{v16}_t \_gset<\text{cond}>f16\{,x\} (\text{vf16}_t x, \text{vf16}_t y) \\
\text{v32}_t \_gset<\text{cond}>f32\{,x\} (\text{vf32}_t x, \text{vf32}_t y) \\
\text{v64}_t \_gset<\text{cond}>f64\{,x\} (\text{vf64}_t x, \text{vf64}_t y) \\
\text{v128}_t \_gset<\text{cond}>f128\{,x\} (\text{vf128}_t x, \text{vf128}_t y)
\end{align*}
\]

This function set has 4 variations. The functions take two 128-bit vectors, \(x\) and \(y\). Vectors are interpreted as containing elements of \(esize\)-bit floating-point values. A condition is applied bit-wise to each element of the vectors, producing a 128-bit result vector of \(esize\)-bit elements. The result elements are set to all 1 bits if the condition is met, or cleared to all 0 bits if the condition is not met.

The list on page 137 gives the mnemonic and English description of each function. Table 23 on page 143 shows the mnemonics, operands, \(esizes\), and conditions.

\[
r[i] = (x[i] <\text{cond}> y[i]) ? -1 : 0, \quad i = 0..\text{NELEM}-1
\]
### Table 23. Conditions for set floating Functions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Data Type¹</th>
<th>esize</th>
<th>Operands and Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gsetef</td>
<td>fp²</td>
<td>16, 32, 64, 128</td>
<td>x[i] == y[i]</td>
</tr>
<tr>
<td>gsetgef</td>
<td>fp²</td>
<td>16, 32, 64, 128</td>
<td>x[i] &gt;= y[i]</td>
</tr>
<tr>
<td>gsetlf</td>
<td>fp²</td>
<td>16, 32, 64, 128</td>
<td>x[i] &lt; y[i]</td>
</tr>
<tr>
<td>gsetlgf</td>
<td>fp²</td>
<td>16, 32, 64, 128</td>
<td>(x[i] &lt; y[i])</td>
</tr>
</tbody>
</table>

1. i = signed or unsigned integer. si = signed integer. ui = unsigned integer. fp = floating-point.
2. Floating-point functions support optional FloatingPointArithmetic exception trapping. See Section 2.3 on page 39 for details on this optional trap mode.
Sets element of result vector to 1s if corresponding elements of the source vectors meet <condition>.

vc8_t _gset<cond>c8(vc8_t x, vc8_t y)
vc16_t _gset<cond>c16(vc16_t x, vc16_t y)
vc32_t _gset<cond>c32(vc32_t x, vc32_t y)
vc64_t _gset<cond>c64(vc64_t x, vc64_t y)

This function set has 2 variations. The functions take two 128-bit vectors, x and y. Vectors are interpreted as containing elements of esize-bit complex integers. A condition is applied bit-wise to each element of the vector(s), producing a 128-bit result vector of esize-bit complex integers. The result elements are set to all 1 bits if the condition is met, or cleared to all 0 bits if the condition is not met.

The _gset list on page 137 gives the mnemonic and English description of each function. Table 21 on page 139 shows the mnemonics, operands, esizes, and conditions for real functions and Table 24 on page 145 for complex functions.

\[
r[i] = (x[i] <cond> y[i]) \ ? -1 - 1 : 0, \quad i = 0..\text{NELEM} - 1
\]
### Table 24. Conditions for `set if complex` Functions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Data Type</th>
<th><code>esize</code></th>
<th>Operands and Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gsete</td>
<td>c</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] == y[i]</code></td>
</tr>
<tr>
<td>gsetne</td>
<td>c</td>
<td>8, 16, 32, 64, 128</td>
<td><code>x[i] != y[i]</code></td>
</tr>
</tbody>
</table>

1. i = signed or unsigned integer. si = signed integer. ui = unsigned integer. fp = floating-point. c = complex integer
Sets element of result vector to 1s based on comparison between source vector and zero.

\[
\text{v8_t } \_\text{gset<cond>8(v8_t x)} \\
\text{v16_t } \_\text{gset<cond>16(v16_t x)} \\
\text{v32_t } \_\text{gset<cond>32(v32_t x)} \\
\text{v64_t } \_\text{gset<cond>64(v64_t x)} \\
\text{v128_t } \_\text{gset<cond>128(v128_t x)}
\]

This function set has 4 variations. They take one 128-bit vector, \( x \), which is interpreted as containing elements of \( \text{esize} \)-bit values. A condition is applied bit-wise to each element of the vector, producing a 128-bit result vector of \( \text{esize} \)-bit elements. The result elements are set to all 1 bits if the condition is met, or cleared to all 0 bits if the condition is not met.

The list on page 137 gives the mnemonic and English description of each function. Table 25 on page 147 shows the mnemonics, operands, \( \text{esizes} \), and conditions.

\[
r[i] = (x[i] <\text{cond}> 0) \ ? -1 : 0, \quad i = 0..\text{NELEM} -1
\]
## Table 25. Conditions for set zero Functions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Data Type&lt;sup&gt;1&lt;/sup&gt;</th>
<th>esize</th>
<th>Operands and Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gsetgez</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>x[i] &gt;= 0</td>
</tr>
<tr>
<td>gsetgz</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>x[i] &gt; 0</td>
</tr>
<tr>
<td>gsetlez</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>x[i] &lt;= 0</td>
</tr>
<tr>
<td>gsetlz</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>x[i] &lt; 0</td>
</tr>
</tbody>
</table>

1. i = signed or unsigned integer. si = signed integer. ui = unsigned integer. fp = floating-point.
3.3.6 **Group Compare and Trap**

The Group Compare and Trap functions include:

- `__gcomande`: Compare AND Equal Zero
- `__gcomandne`: Compare AND Not Equal Zero
- `__gcome`: Compare Equal
- `__gcomef`: Compare Equal Floating-Point
- `__gcomge`: Compare Greater or Equal
- `__gcomgeu`: Compare Greater or Equal Unsigned
- `__gcomgef`: Compare Greater or Equal Floating-Point
- `__gcomgez`: Compare Greater or Equal Zero
- `__gcomlz`: Compare Less than Zero
- `__gcoml`: Compare Less
- `__gcomlu`: Compare Less Unsigned
- `__gcomlf`: Compare Less Floating-Point
- `__gcomlez`: Compare Less or Equal Zero
- `__gcomlzh`: Compare Less than Zero
- `__gcomne`: Compare Not Equal
- `__gcomlgf`: Compare Less or Greater Floating-Point
_gcom<cond> Compare Condition

Traps if comparison of source elements meets <condition>.

void _gcom<cond>8(v8_t x, v8_t y)
void _gcom<cond>16(v16_t x, v16_t y)
void _gcom<cond>32(v32_t x, v32_t y)
void _gcom<cond>64(v64_t x, v64_t y)
void _gcom<cond>128(v128_t x, v128_t y)

This function set has 8 variations. The functions take two 128-bit vector parameters, \(x\) and \(y\). Vectors are interpreted as containing elements of \(esize\)-bit values. A condition is compared bit-wise with each element of the vectors, causing a trap if the condition is met.

The list on page 148 gives the mnemonic and English description of each function. Table 26 shows the mnemonics, operands, \(esizes\), and conditions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Data Type</th>
<th>(esize)</th>
<th>Operands and Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcomande</td>
<td>i</td>
<td>8, 16, 32, 64, 128</td>
<td>(x[i] &amp; y[i] == 0)</td>
</tr>
<tr>
<td>gcomandne</td>
<td>i</td>
<td>8, 16, 32, 64, 128</td>
<td>(x[i] &amp; y[i] != 0)</td>
</tr>
<tr>
<td>gcome</td>
<td>i</td>
<td>8, 16, 32, 64, 128</td>
<td>(x[i] == y[i])</td>
</tr>
<tr>
<td>gcomge</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>(x[i] &gt;= y[i])</td>
</tr>
<tr>
<td>gcomgeu</td>
<td>ui</td>
<td>8, 16, 32, 64, 128</td>
<td>(x[i] &gt;= y[i])</td>
</tr>
<tr>
<td>gcoml</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>(x[i] &lt; y[i])</td>
</tr>
<tr>
<td>gcomlu</td>
<td>ui</td>
<td>8, 16, 32, 64, 128</td>
<td>(x[i] &lt; y[i])</td>
</tr>
<tr>
<td>gcomne</td>
<td>i</td>
<td>8, 16, 32, 64, 128</td>
<td>(x[i] != y[i])</td>
</tr>
</tbody>
</table>

1. \(i = \text{signed or unsigned integer. si = signed integer. ui = unsigned integer. fp = floating-point.}\)
_gcom<cond>  Compare Floating Condition

Traps if comparison of source elements meets <condition>.

```c
void _gcom<cond>f16{,x} (vf16_t x, vf16_t y)
void _gcom<cond>f32{,x} (vf32_t x, vf32_t y)
void _gcom<cond>f64{,x} (vf64_t x, vf64_t y)
void _gcom<cond>f128{,x} (vf128_t x, vf128_t y)
```

This function set has 4 variations. The functions take one or two 128-bit vector parameters, x and y. Vectors are interpreted as containing elements of esize-bit values. A condition is compared bit-wise with each element of the vector(s), causing a trap if the condition is met.

The list on page 148 gives the mnemonic and English description of each function. Table 27 shows the mnemonics, operands, esizes, and conditions.

![Diagram](image_url)

trap \((x[i] < \text{cond}> y[i])\), \(i = 0..\text{NELEM} -1\)

**Table 27. Conditions for gcom<cond> Functions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Data Type(^1)</th>
<th>esize</th>
<th>Operands and Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcomef</td>
<td>fp</td>
<td>16, 32, 64, 128</td>
<td>(x[i] == y[i])</td>
</tr>
<tr>
<td>gcomgeff</td>
<td>fp</td>
<td>16, 32, 64, 128</td>
<td>(x[i] &gt;= y[i])</td>
</tr>
<tr>
<td>gcomlf</td>
<td>fp</td>
<td>16, 32, 64, 128</td>
<td>(x[i] &lt; y[i])</td>
</tr>
<tr>
<td>gcomlgyf</td>
<td>fp</td>
<td>16, 32, 64, 128</td>
<td>((x[i] &lt; y[i]) \text{ or } (x[i] &gt; y[i]))</td>
</tr>
</tbody>
</table>

1. \(i = \text{signed or unsigned integer. } si = \text{signed integer. } ui = \text{unsigned integer. } fp = \text{floating-point.}\)
_gcom<cond>  

Compare Complex Condition

Traps if comparison of source elements meets <condition>.

```c
void _gcom<cond>c8(vc8_t x, vc8_t y)
void _gcom<cond>c16(vc16_t x, vc16_t y)
void _gcom<cond>c32(vc32_t x, vc32_t y)
void _gcom<cond>c64(vc64_t x, vc64_t y)
```

This function set has 2 variations. The functions take two 128-bit vector parameters, \( x \) and \( y \). Vectors are interpreted as containing elements of \( esize \)-bit complex integers. A condition is compared bit-wise with each element of the vectors, causing a trap if the condition is met.

The list on page 148 gives the mnemonic and English description of each function. Table 28 shows the mnemonics, operands, \( esizes \), and conditions.

![Diagram](image.png)

trap \((x[i] <cond> y[i]), \quad i = 0..NELEM -1\)

**Table 28. Conditions for gcom<cond> Functions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Data Type</th>
<th>( esize )</th>
<th>Operands and Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcome</td>
<td>c</td>
<td>8, 16, 32, 64</td>
<td>( x[i] == y[i] )</td>
</tr>
<tr>
<td>gcomne</td>
<td>c</td>
<td>8, 16, 32, 64</td>
<td>( x[i] != y[i] )</td>
</tr>
</tbody>
</table>

1. \( i = \) signed or unsigned integer. \( si = \) signed integer. \( ui = \) unsigned integer. \( fp = \) floating-point. \( c = \) complex integer
**_gcom<cond>_**

**Compare Zero Condition**

Traps if comparison of source elements with zero meets `<condition>`.

```c
void _gcom<cond>8(v8_t x)
void _gcom<cond>16(v16_t x)
void _gcom<cond>32(v32_t x)
void _gcom<cond>64(v64_t x)
void _gcom<cond>128(v128_t x)
```

This function set has 4 variations. The functions take one 128-bit vector parameter, x. Vectors are interpreted as containing elements of esize-bit values. A condition is compared bit-wise with each element of the vector(s), causing a trap if the condition is met.

The list on page 148 gives the mnemonic and English description of each function. Table 29 shows the mnemonics, operands, esizes, and conditions.

![Diagram of vector comparison](496-256.png)

**trap (x[i] <cond> 0), i = 0..NELEM -1**

**Table 29. Conditions for gcom<cond> Functions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Data Type&lt;sup&gt;1&lt;/sup&gt;</th>
<th>esize</th>
<th>Operands and Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcomgez</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>x[i] &gt;= 0</td>
</tr>
<tr>
<td>gcomgz</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>x[i] &gt; 0</td>
</tr>
<tr>
<td>gcomlez</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>x[i] &lt;= 0</td>
</tr>
<tr>
<td>gcomlz</td>
<td>si</td>
<td>8, 16, 32, 64, 128</td>
<td>x[i] &lt; 0</td>
</tr>
</tbody>
</table>

<sup>1. i = signed or unsigned integer. si = signed integer. ui = unsigned integer. fp = floating-point.</sup>
3.4 Crossbar Functions

The Crossbar Logic Unit is essentially a multiplexer with two 128-bit inputs and one 128-bit output. The unit contains embedded memory that stores up to eight distinct switching configurations in which each output bit can come from any of the 256 input bits. The unit also contains logic that programs the switch with common switching patterns, for example, shifts, rotates, field-extraction, and shuffles.

The Crossbar functions include the following types:

- Crossbar Shift and Rotate
- Crossbar Expand and Compress
- Crossbar Deposit and Withdraw
- Crossbar Shuffle
- Crossbar Swizzle
- Crossbar Select Byte
- Crossbar Extract
3.4.1 **Crossbar Shift and Rotate**

The Crossbar Shift and Rotate functions include:

- `_xrotl` Rotate Left
- `_xrotli` Rotate Left Immediate
- `_xrotr` Rotate Right
- `_xrotri` Rotate Right Immediate
- `_xshl` Shift Left
- `_xshli` Shift Left Immediate
- `_xshlo` Shift Left with Overflow Trap
- `_xshlio` Shift Left Immediate with Overflow Trap
- `_xshluo` Shift Left Unsigned with Overflow Trap
- `_xshliuo` Shift Left Immediate Unsigned with Overflow Trap
- `_xshlm` Shift Left and Merge
- `_xshlmi` Shift Left and Merge Immediate
- `_xshri` Shift Right
- `_xshrii` Shift Right Immediate
- `_xshru` Shift Right Unsigned
- `_xshriu` Shift Right Immediate Unsigned
- `_xshrm` Shift Right and Merge
- `_xshrmii` Shift Right and Merge Immediate
_xrotl, _xrotli

Left-rotates vector elements.

\[
\begin{align*}
\text{v2}_t & \quad \_\text{xrotl2}(\text{v2}_t \ x, \ \text{int} \ sh) \\
\text{v4}_t & \quad \_\text{xrotl4}(\text{v4}_t \ x, \ \text{int} \ sh) \\
\text{v8}_t & \quad \_\text{xrotl8}(\text{v8}_t \ x, \ \text{int} \ sh) \\
\text{v16}_t & \quad \_\text{xrotl16}(\text{v16}_t \ x, \ \text{int} \ sh) \\
\text{v32}_t & \quad \_\text{xrotl32}(\text{v32}_t \ x, \ \text{int} \ sh) \\
\text{v64}_t & \quad \_\text{xrotl64}(\text{v64}_t \ x, \ \text{int} \ sh) \\
\text{v128}_t & \quad \_\text{xrotl128}(\text{v128}_t \ x, \ \text{int} \ sh)
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), and a scalar shift-amount parameter, \( sh \). The vector is interpreted as containing elements of \( esize \)-bit unsigned integers. Each vector element is left-rotated by \( sh \), modulo \( esize \), producing a 128-bit result vector of \( esize \)-bit elements. The immediate version is more efficient when \( sh \) is a constant.

\[
r[i] = x[i] \ll \text{shift} | x[i] \gg \text{(esize-shift)}, \quad i = 0..\text{NELEM} -1
\]

where: \( \text{shift} = sh \& (\text{esize}-1) \)
Chapter 3: Vector and Matrix Functions

Crossbar Functions

_xrotr, _xrotri

Rotate Right

Right-rotates vector elements.

v2_t _xrotr2(v2_t x, int sh)
v4_t _xrotr4(v4_t x, int sh)
v8_t _xrotr8(v8_t x, int sh)
v16_t _xrotr16(v16_t x, int sh)
v32_t _xrotr32(v32_t x, int sh)
v64_t _xrotr64(v64_t x, int sh)
v128_t _xrotr128(v128_t x, int sh)

This function takes a 128-bit vector parameter, x, and a scalar shift-amount parameter, sh. The vector is interpreted as containing elements of esize-bit unsigned integers. Each vector element is right-rotated by sh, modulo esize, producing a 128-bit result vector of esize-bit elements. The immediate version is more efficient when sh is a constant.

\[ r[i] = x[i] \gg^u \text{shift} | x[i] \ll (\text{esize-shift}), \quad i = 0..\text{NELEM}-1 \]

where: \( \text{shift} = \text{sh} \& (\text{esize}-1) \)
_xshl, _xshli

Shift Left

Left-shifts vector elements.

\[
\begin{align*}
&v2_t \ _xshl2(v2_t \ x, \ int \ sh) \\
v4_t \ _xshl4(v4_t \ x, \ int \ sh) \\
v8_t \ _xshl8(v8_t \ x, \ int \ sh) \\
v16_t \ _xshl16(v16_t \ x, \ int \ sh) \\
v32_t \ _xshl32(v32_t \ x, \ int \ sh) \\
v64_t \ _xshl64(v64_t \ x, \ int \ sh) \\
v128_t \ _xshl128(v128_t \ x, \ int \ sh)
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), and a scalar shift-amount parameter, \( sh \). The vector is interpreted as containing elements of \( esize \)-bit integers. Each vector element is left-shifted by \( sh \), modulo \( esize \), and zero-filled at the low end, producing a 128-bit result vector of \( esize \)-bit elements. The immediate version is more efficient when \( sh \) is a constant.

\[
r[i] = x[i] \ll shift, \quad i = 0..NELEM-1
\]

where: \( shift = sh \& (esize-1) \)
_xshlo, _xshlio

Shift Left with Overflow Trap

Left-shifts vector elements, and traps on overflow.

\[
\begin{align*}
\text{v2_t} & \quad _\text{xshlo2}(\text{v2_t}\ x, \ \text{int} \ sh) \\
\text{v4_t} & \quad _\text{xshlo4}(\text{v4_t}\ x, \ \text{int} \ sh) \\
\text{v8_t} & \quad _\text{xshlo8}(\text{v8_t}\ x, \ \text{int} \ sh) \\
\text{v16_t} & \quad _\text{xshlo16}(\text{v16_t}\ x, \ \text{int} \ sh) \\
\text{v32_t} & \quad _\text{xshlo32}(\text{v32_t}\ x, \ \text{int} \ sh) \\
\text{v64_t} & \quad _\text{xshlo64}(\text{v64_t}\ x, \ \text{int} \ sh) \\
\text{v128_t} & \quad _\text{xshlo128}(\text{v128_t}\ x, \ \text{int} \ sh)
\end{align*}
\]

This function takes a 128-bit vector parameter, \(x\), and a scalar shift-amount parameter, \(sh\). The vector is interpreted as containing elements of \(esize\)-bit integers. Each vector element is left-shifted by \(sh\), modulo \(esize\), and zero-filled at the low end, producing a 128-bit result vector of \(esize\)-bit elements. If any result exceeds the range representable in an \(esize\)-bit signed integer, the operation does not complete and the \textit{FixedPointArithmetic} exception is taken. The immediate version is more efficient when \(sh\) is a constant.

\[
\begin{array}{c}
\text{r}[i] = x[i] \ll^0 \text{shift}, \quad i = 0..\text{NELEM} - 1 \\
\text{where:} \quad \text{shift} = sh \& (\text{size}-1)
\end{array}
\]
_xshluo, _xshliuo

Shift Left Unsigned with Overflow Trap

Left-shifts unsigned vector elements, and traps on overflow.

\[
vu2_t \ xshlu2o(vu2_t \ x, \ int \ sh) \\
vu4_t \ xshlu4o(vu4_t \ x, \ int \ sh) \\
vu8_t \ xshlu8o(vu8_t \ x, \ int \ sh) \\
vul6_t \ xshlu16o(vul6_t \ x, \ int \ sh) \\
vu32_t \ xshlu32o(vu32_t \ x, \ int \ sh) \\
vu64_t \ xshlu64o(vu64_t \ x, \ int \ sh) \\
vul28_t \ xshlu128o(vul28_t \ x, \ int \ sh)
\]

This function takes a 128-bit vector parameter, \(x\), and a scalar shift-amount parameter, \(sh\). The vector is interpreted as containing elements of \(esize\)-bit unsigned integers. Each vector element is left-shifted by \(sh\), modulo \(esize\), and zero-filled at the low end, producing a 128-bit result vector of \(esize\)-bit elements. If any result exceeds the range representable in an \(esize\)-bit signed integer, the operation does not complete and the \texttt{FixedPointArithmetic} exception is taken. The immediate version is more efficient when \(sh\) is a constant.

\[
r[i] = x[i] \ll_{uo} shift, \quad i = 0..\text{NELEM} - 1
\]

where: \(shift = sh \& (\text{size}-1)\)
_xshlm, _xshlmi

Shift Left and Merge

Left-shifts elements of one vector and merges elements of another vector.

\[
\begin{align*}
\text{v2_t } & \quad \text{_xshlm2(v2_t } x, \text{ v2_t } y, \text{ int } sh) \\
\text{v4_t } & \quad \text{_xshlm4(v4_t } x, \text{ v4_t } y, \text{ int } sh) \\
\text{v8_t } & \quad \text{_xshlm8(v8_t } x, \text{ v8_t } y, \text{ int } sh) \\
\text{v16_t } & \quad \text{_xshlm16(v16_t } x, \text{ v16_t } y, \text{ int } sh) \\
\text{v32_t } & \quad \text{_xshlm32(v32_t } x, \text{ v32_t } y, \text{ int } sh) \\
\text{v64_t } & \quad \text{_xshlm64(v64_t } x, \text{ v64_t } y, \text{ int } sh) \\
\text{v128_t } & \quad \text{_xshlm128(v128_t } x, \text{ v128_t } y, \text{ int } sh)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \), and a scalar shift-amount parameter, \( sh \). The vectors are interpreted as containing elements of \( esize \)-bit integers. Each of the \( x \) vector’s elements is left-shifted by \( sh \), modulo \( esize \). Vacated bits at the low end are filled with bits from the corresponding element and bit-positions of the \( y \) vector, producing a 128-bit result vector of \( esize \)-bit elements. The immediate version is more efficient when \( sh \) is a constant.

\[
r[i] = (x[i] << sh) | (y[i] & mask)
\]

where mask = \((1 << sh) - 1\)
**_xshr, _xshri**

Right-shifts vector elements.

\[
\begin{align*}
&v2_t \_xshr2(v2_t x, \text{int } sh) \\
v4_t \_xshr4(v4_t x, \text{int } sh) \\
v8_t \_xshr8(v8_t x, \text{int } sh) \\
v16_t \_xshr16(v16_t x, \text{int } sh) \\
v32_t \_xshr32(v32_t x, \text{int } sh) \\
v64_t \_xshr64(v64_t x, \text{int } sh) \\
v128_t \_xshr128(v128_t x, \text{int } sh)
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), and a scalar shift-amount parameter, \( sh \). The vector is interpreted as containing elements of \( esize \)-bit signed integers. Each vector element is right-shifted by \( sh \), modulo \( esize \), and sign-extended at the high end (arithmetic shift), producing a 128-bit result vector of \( esize \)-bit elements. The immediate version is more efficient when \( sh \) is a constant.

\[
r[i] = x[i] >> \text{shift}, \quad i = 0..\text{NELEM -1}
\]

where: \( \text{shift} = sh \& (esize-1) \)
_xshru, _xshriu

Shift Right Unsigned

Right-shifts unsigned vector elements.

\[ r[i] = x[i] \gg^u \text{shift}, \quad i = 0..\text{NELEM} - 1 \]

where: \( \text{shift} = \text{sh} \& (\text{esize}-1) \)
_xshrm, _xshrmi  Shift Right and Merge

Right-shifts elements of one vector and merges elements of another vector.

\[
\begin{align*}
v2_t & \quad _xshrm2(v2_t \ x, v2_t \ y, int \ sh) \\
v4_t & \quad _xshrm4(v4_t \ x, v4_t \ y, int \ sh) \\
v8_t & \quad _xshrm8(v8_t \ x, v8_t \ y, int \ sh) \\
v16_t & \quad _xshrm16(v16_t \ x, v16_t \ y, int \ sh) \\
v32_t & \quad _xshrm32(v32_t \ x, v32_t \ y, int \ sh) \\
v64_t & \quad _xshrm64(v64_t \ x, v64_t \ y, int \ sh) \\
v128_t & \quad _xshrm128(v128_t \ x, v128_t \ y, int \ sh)
\end{align*}
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\), and a scalar shift-amount parameter, \(sh\). The vectors are interpreted as containing elements of \(esize\)-bit integers. Each of the \(x\) vector's elements is right-shifted by \(sh\), modulo \(esize\). Vacated bits at the high end are filled with the low-order bits from the corresponding element of the \(y\) vector, producing a 128-bit result vector of \(esize\)-bit elements. The immediate version is more efficient when \(sh\) is a constant.

\[
r[i] = (x[i] \gg u \ shift) | (y[i] << (esize - shift))
\]

where: \(\text{shift} = sh \& \ (esize -1)\)
3.4.2 **Crossbar Compress and Expand**

The Crossbar Compress and Expand functions include:

- `_xcompress` Compress
- `_xcompressi` Compress Immediate
- `_xcompressu` Compress Unsigned
- `_xcompressiu` Compress Immediate Unsigned
- `_xexpand` Expand
- `_xexpandi` Expand Immediate
- `_xexpandu` Expand Unsigned
- `_xexpandiu` Expand Immediate Unsigned

Compare the `edeflatef` and `einflatef` functions in Section 3.5.9 on page 284.
_xcompress, _xcompressi

Compress

Halves the width of vector elements.

```
v1_t _xcompress2(v2_t x, int sh)
v2_t _xcompress4(v4_t x, int sh)
v4_t _xcompress8(v8_t x, int sh)
v8_t _xcompress16(v16_t x, int sh)
v16_t _xcompress32(v32_t x, int sh)
v32_t _xcompress64(v64_t x, int sh)
v64_t _xcompress128(v128_t x, int sh)
```

This function takes a 128-bit vector parameter, `x`, and a scalar shift-amount parameter, `sh`. The vector is interpreted as containing elements of `esize`-bit signed integers. The vector elements are right-shifted by `sh`, modulo `esize`, and sign-extended at the high end (arithmetic shift). The low `esize/2` bits of each element are compressed into the low 64 bits of the result. The high 64 bits of the result are cleared to zero.

```
r[i] = x[i] >> shift,  i = 0..NELEM -1

r[i] = 0,  i = NELEM..2*NELEM -1
```

where:  
shift = sh & (esize -1)
_xcompressu, _xcompressiu

Compress Unsigned

Halves the width of unsigned vector elements.

\[
\begin{align*}
\text{vu1}_t & \quad \text{_xcompressu2}(\text{vu2}_t \ x, \ \text{int} \ y) \\
\text{vu2}_t & \quad \text{_xcompressu4}(\text{vu4}_t \ x, \ \text{int} \ y) \\
\text{vu4}_t & \quad \text{_xcompressu8}(\text{vu8}_t \ x, \ \text{int} \ y) \\
\text{vu8}_t & \quad \text{_xcompressu16}(\text{vu16}_t \ x, \ \text{int} \ y) \\
\text{vu16}_t & \quad \text{_xcompressu32}(\text{vu32}_t \ x, \ \text{int} \ y) \\
\text{vu32}_t & \quad \text{_xcompressu64}(\text{vu64}_t \ x, \ \text{int} \ y) \\
\text{vu64}_t & \quad \text{_xcompressu128}(\text{vu128}_t \ x, \ \text{int} \ y)
\end{align*}
\]

This function takes a 128-bit vector parameter, \(x\), and a scalar shift-amount parameter, \(y\). The vector is interpreted as containing elements of \(esize\)-bit unsigned integers. The vector elements are right-shifted by \(y\), modulo \(esize\), and zero-extended at the high end (logical shift). The low \(esize/2\) bits of each element are compressed into the low 64 bits of the result. The high 64 bits of the result are cleared to zero.

\[
\begin{align*}
\text{r}[i] &= x[i] >> u \text{ shift}, \quad i = 0..\text{NELEM} -1 \\
\text{r}[i] &= 0, \quad i = \text{NELEM}..2*\text{NELEM} -1
\end{align*}
\]

where: \(\text{shift} = \text{sh} \& (\text{esize} -1)\)
_xexpand, _xexpandi

Doubles the width of vector elements.

\[ \begin{align*}
    \text{v2}_t & \quad \text{v}_2 \_\text{xexpand}  \quad (v1_t \ x, \ \text{int} \ \text{sh}) \\
    \text{v4}_t & \quad \text{v}_4 \_\text{xexpand}  \quad (v2_t \ x, \ \text{int} \ \text{sh}) \\
    \text{v8}_t & \quad \text{v}_8 \_\text{xexpand}  \quad (v4_t \ x, \ \text{int} \ \text{sh}) \\
    \text{v16}_t & \quad \text{v}_{16} \_\text{xexpand}  \quad (v8_t \ x, \ \text{int} \ \text{sh}) \\
    \text{v32}_t & \quad \text{v}_{32} \_\text{xexpand}  \quad (v16_t \ x, \ \text{int} \ \text{sh}) \\
    \text{v64}_t & \quad \text{v}_{64} \_\text{xexpand}  \quad (v32_t \ x, \ \text{int} \ \text{sh}) \\
    \text{v128}_t & \quad \text{v}_{128} \_\text{xexpand} \quad (v64_t \ x, \ \text{int} \ \text{sh})
\end{align*} \]

This function takes a 128-bit vector parameter, \( x \), and an immediate scalar shift amount, \( \text{sh} \). The vector is interpreted as containing elements of \( \text{esize}/2 \)-bit signed integers. The elements in the low 64 bits of vector \( x \) are expanded from \( \text{esize}/2 \) bits to \( \text{esize} \) bits by sign-extending at the high end, left-shifted by \( \text{sh} \), modulo \( \text{esize} \), with zero-filling at the low end, producing a 128-bit result vector of \( \text{esize} \)-bit elements. The immediate version is more efficient when \( \text{sh} \) is a constant.

\[
\begin{align*}
    \text{r}[i] = \text{x}[i] \ll \text{shift}, \\
    \text{where:} \quad \text{shift} = \text{sh} \& (\text{esize} - 1), \quad i = 0..\text{NELEM} - 1
\end{align*}
\]
_xexpandu, _xexpandiu

Expand Unsigned

Doubles the width of unsigned vector elements.

vu2_t _xexpandiu2(vu1_t x, int sh)
vu4_t _xexpandiu4(vu2_t x, int sh)
vu8_t _xexpandiu8(vu4_t x, int sh)
vu16_t _xexpandiu16(vu8_t x, int sh)
vu32_t _xexpandiu32(vu16_t x, int sh)
vu64_t _xexpandiu64(vu32_t x, int sh)
vu128_t _xexpandiu128(vu64_t x, int sh)

This function takes a 128-bit vector parameter, x, and an immediate scalar shift amount, sh. The vector is interpreted as containing elements of esize/2-bit unsigned integers. The elements in the low 64 bits of vector x are expanded from esize/2 bits to esize bits by zero-filling at the high end, left-shifted by sh, modulo esize, with zero-filling at the low end, producing a 128-bit result vector of esize-bit elements. The immediate version is more efficient when sh is a constant.

\[
r[i] = x[i] \ll shift, \\
\text{where: \hspace{1em} shift = sh \& (esize -1), \hspace{1em} i = 0..NELEM/2 -1}
\]
3.4.3 **Crossbar Deposit and Withdraw**

The Crossbar Deposit and Withdraw functions include:

- `_xdeposit` Deposit
- `_xdepositu` Deposit Unsigned
- `_xdepositm` Deposit and Merge
- `_xwithdraw` Withdraw
- `_xwithdrawu` Withdraw Unsigned
_xdeposit

Left-shifts fields in vector elements.

\[
\begin{align*}
v2_t &\quad _{\text{xdeposit}}2(v2_t \ x, \ \text{int} \ \text{fsize}, \ \text{int} \ \text{sh}) \\
v4_t &\quad _{\text{xdeposit}}4(v4_t \ x, \ \text{int} \ \text{fsize}, \ \text{int} \ \text{sh}) \\
v8_t &\quad _{\text{xdeposit}}8(v8_t \ x, \ \text{int} \ \text{fsize}, \ \text{int} \ \text{sh}) \\
v16_t &\quad _{\text{xdeposit}}16(v16_t \ x, \ \text{int} \ \text{fsize}, \ \text{int} \ \text{sh}) \\
v32_t &\quad _{\text{xdeposit}}32(v32_t \ x, \ \text{int} \ \text{fsize}, \ \text{int} \ \text{sh}) \\
v64_t &\quad _{\text{xdeposit}}64(v64_t \ x, \ \text{int} \ \text{fsize}, \ \text{int} \ \text{sh}) \\
v128_t &\quad _{\text{xdeposit}}128(v128_t \ x, \ \text{int} \ \text{fsize}, \ \text{int} \ \text{sh})
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), and scalar immediate values for field size, \( \text{fsize} \), and shift amount, \( \text{sh} \). The vector is interpreted as containing elements of \( \text{esize} \)-bit signed integers. The low-order \( \text{fsize} \) bits are extracted from each of the vector's elements. The extracted bits are left-shifted by \( \text{sh} \), zero-filled on the low end, and sign-extended on the high end to pad out the corresponding \( \text{esize} \)-bit elements of the result. \( \text{fsize} \) and \( \text{sh} \) are subject to the constraints \( \text{fsize} + \text{sh} \leq \text{esize} \) and \( \text{fsize} \) lies in the range 1..\( \text{esize} \).

\[
r[i] = (x[i] << (\text{esize} - \text{fsize})) >> (\text{esize} - \text{fsize} - \text{shift}) \quad i = 0..\text{NELEM} -1
\]

where:
\[
\text{shift} = \text{sh} \& (\text{esize} -1) \\
1 \leq \text{fsize} < \text{esize}
\]
_xdepositu

Deposit Unsigned

Left-shifts unsigned fields in vector elements.

\[
\text{v2_t } \_\text{xdepositu2}(\text{v2_t } x, \text{int } fsize, \text{int } sh) \\
\text{v4_t } \_\text{xdepositu4}(\text{v4_t } x, \text{int } fsize, \text{int } sh) \\
\text{v8_t } \_\text{xdepositu8}(\text{v8_t } x, \text{int } fsize, \text{int } sh) \\
\text{v16_t } \_\text{xdepositu16}(\text{v16_t } x, \text{int } fsize, \text{int } sh) \\
\text{v32_t } \_\text{xdepositu32}(\text{v32_t } x, \text{int } fsize, \text{int } sh) \\
\text{v64_t } \_\text{xdepositu64}(\text{v64_t } x, \text{int } fsize, \text{int } sh) \\
\text{v128_t } \_\text{xdepositu128}(\text{v128_t } x, \text{int } fsize, \text{int } sh)
\]

This function takes a 128-bit vector parameter, \( x \), and scalar immediate values for field size, \( fsize \), and shift amount, \( sh \). The vector is interpreted as containing elements of \( esize \)-bit unsigned integers. The low-order \( fsize \) bits are extracted from each of the vector’s elements. The extracted bits are left-shifted by \( sh \), zero-filled on the low and high ends to pad out the corresponding \( esize \)-bit elements of the result. \( fsize \) and \( sh \) are subject to the constraints \( fsize + sh \leq esize \) and \( fsize \) lies in the range 1..\( esize \).

\[
r[i] = ((x[i] \ll \text{shift}) \& \text{mask}), \quad i = 0..\text{NELEM} -1
\]

where:

\begin{align*}
\text{shift} &= \text{sh} \& (\text{esize} -1), \quad \text{and} \quad \text{mask} = ((1 \ll fsize) - 1) \ll \text{shift} \\
1 \leq fsize < \text{esize}
\end{align*}
Crossbar Functions

Chapter 3: Vector and Matrix Functions

_xdepositm Deposit and Merge

Left-shifts fields in elements of one vector and merges them with fields in elements of another vector.

\[
\begin{align*}
\text{v2}_t & \text{ _xdepositm2(v2}_t m, \text{ v2}_t x, \text{ int fsize, int sh)} \\
\text{v4}_t & \text{ _xdepositm4(v4}_t m, \text{ v4}_t x, \text{ int fsize, int sh)} \\
\text{v8}_t & \text{ _xdepositm8(v8}_t m, \text{ v8}_t x, \text{ int fsize, int sh)} \\
\text{v16}_t & \text{ _xdepositm16(v16}_t m, \text{ v16}_t x, \text{ int fsize, int sh)} \\
\text{v32}_t & \text{ _xdepositm32(v32}_t m, \text{ v32}_t x, \text{ int fsize, int sh)} \\
\text{v64}_t & \text{ _xdepositm64(v64}_t m, \text{ v64}_t x, \text{ int fsize, int sh)} \\
\text{v128}_t & \text{ _xdepositm128(v128}_t m, \text{ v128}_t x, \text{ int fsize, int sh)}
\end{align*}
\]

This function takes two 128-bit vector parameters, \( m \) and \( x \), and scalar immediate values for field size, \( fsize \), and shift amount, \( sh \). The vectors are interpreted as containing elements of \( esize \)-bit integers. The low-order \( fsize \) bits from each of the \( x \) vector’s elements is left-shifted by \( sh \). Bits to the right and left of the shifted bit-field are filled with bits from the corresponding element and bit-position of the \( m \) vector. \( fsize \) and \( sh \) are subject to the constraints \( fsize + sh \leq esize \) and \( fsize \) lies in the range 1..\( esize \).

\[
r[i] = ((x[i] \ll shift) \& mask) | (m[i] \& \sim mask), \quad i = 0..\text{NELEM -1}
\]

where: \( \text{shift} = sh \& (esize -1) \) and \( \text{mask} = ((1 \ll fsize) -1) \ll \text{shift} \)

\( 1 \leq fsize < esize \)
_xwithdraw

Right-shifts fields in vector elements.

\[ r[i] = (x[i] \ll (\text{esize} - \text{fsize} - \text{shift})) \gg (\text{esize} - \text{fsize}), \quad i = 0..\text{NELEM} - 1 \]

where:

\[ \text{shift} = \text{sh} \& (\text{esize} - 1) \]
\[ 1 \leq \text{fsize} < \text{esize} \]
Right-shifts fields in unsigned vector elements.

\[ r[i] = (x[i] \gg \text{shift}) \& \text{mask}, \quad i = 0..\text{NELEM} -1 \]

where:

\[ \text{shift} = \text{sh} \& (\text{esize} -1), \quad \text{and mask} = (1 \ll \text{fsize}) -1 \]

\[ 1 \leq \text{fsize} < \text{esize} \]
3.4.4 **Crossbar Shuffle**

The Crossbar Shuffle functions include:

- `_xshuffle` Shuffle
- `_xshufflepair` 256-Bit Shuffle
_xshuffle

Shuffles vector-elements within a vector.

\[
v1_t \_xshuffle1(v1_t x, \text{int ssize, int gsize, int nseg})
\]
\[
v2_t \_xshuffle2(v2_t x, \text{int ssize, int gsize, int nseg})
\]
\[
v4_t \_xshuffle4(v4_t x, \text{int ssize, int gsize, int nseg})
\]
\[
v8_t \_xshuffle8(v8_t x, \text{int ssize, int gsize, int nseg})
\]
\[
v16_t \_xshuffle16(v16_t x, \text{int ssize, int gsize, int nseg})
\]
\[
v32_t \_xshuffle32(v32_t x, \text{int ssize, int gsize, int nseg})
\]

This function takes a 128-bit vector parameter, \(x\), and scalar immediate values for set size, \(\text{ssize}\), group size, \(\text{gsize}\), and number of segments, \(\text{nseg}\). The vector is interpreted as containing elements of \(\text{esize}\)-bit integers. The function works on sub-vectors of \(\text{ssize}\) elements at a time. Within each \(\text{ssize}\)-element subvector, elements are collected into groups of \(\text{gsize}\) elements each. The grouped elements are partitioned into \(\text{nseg}\) segments. The segments are shuffled together, taking the first group of elements from each of the piles in turn, then taking the second group of elements from each of the piles, and so on until all groups have been shuffled. This process is then repeated over the elements in each of the remaining sub-vectors.

For this function, the \(\text{ssize}\), \(\text{gsize}\), and \(\text{nseg}\) parameters must be powers of 2, \(\text{npile}\) must be \(\geq 2\) and \(\leq \text{ssize/gsize}/2\), and \(\text{ssize}\) must be \(\geq 4/\text{gsize}\).

An \_xshuffle16(source,4,1,2) is illustrated in the figure below.

\[
r[i] = x[s]. \quad i = 0..\text{NELEM} -1
\]

where: \(s = (i \& \text{mask}) \mid ((i \& \text{mask}) \gg \text{l2npile}) \mid ((i \ll ((\text{l2esize} - \text{l2npile})) \& \text{mask})
\]
\[
\text{mask} = \text{esize} -1
\]
\[
\text{l2esize} = \log_2(\text{esize}), \text{l2npile} = \log_2(\text{npile})
\]
Crossbar Functions

Chapter 3: Vector and Matrix Functions

_\text{xshufflepair} 256-\text{Bit Shuffle}

Re-arranges fields into \( n \) piles within a single 256-bit vector element, and selects the low or high half.

\[
\begin{align*}
v1_\text{t} & \quad \text{xshufflepair1lo}(v1_\text{t} \ xlo, v1_\text{t} \ xhi, \text{int nseg}) \\
v2_\text{t} & \quad \text{xshufflepair2lo}(v2_\text{t} \ xlo, v2_\text{t} \ xhi, \text{int nseg}) \\
v4_\text{t} & \quad \text{xshufflepair4lo}(v4_\text{t} \ xlo, v4_\text{t} \ xhi, \text{int nseg}) \\
v8_\text{t} & \quad \text{xshufflepair8lo}(v8_\text{t} \ xlo, v8_\text{t} \ xhi, \text{int nseg}) \\
v16_\text{t} & \quad \text{xshufflepair16lo}(v16_\text{t} \ xlo, v16_\text{t} \ xhi, \text{int nseg}) \\
v32_\text{t} & \quad \text{xshufflepair32lo}(v32_\text{t} \ xlo, v32_\text{t} \ xhi, \text{int nseg}) \\
v64_\text{t} & \quad \text{xshufflepair64lo}(v64_\text{t} \ xlo, v64_\text{t} \ xhi, \text{int nseg}) \\
v1_\text{t} & \quad \text{xshufflepair1hi}(v1_\text{t} \ xlo, v1_\text{t} \ xhi, \text{int nseg}) \\
v2_\text{t} & \quad \text{xshufflepair2hi}(v2_\text{t} \ xlo, v2_\text{t} \ xhi, \text{int nseg}) \\
v4_\text{t} & \quad \text{xshufflepair4hi}(v4_\text{t} \ xlo, v4_\text{t} \ xhi, \text{int nseg}) \\
v8_\text{t} & \quad \text{xshufflepair8hi}(v8_\text{t} \ xlo, v8_\text{t} \ xhi, \text{int nseg}) \\
v16_\text{t} & \quad \text{xshufflepair16hi}(v16_\text{t} \ xlo, v16_\text{t} \ xhi, \text{int nseg}) \\
v32_\text{t} & \quad \text{xshufflepair32hi}(v32_\text{t} \ xlo, v32_\text{t} \ xhi, \text{int nseg}) \\
v64_\text{t} & \quad \text{xshufflepair64hi}(v64_\text{t} \ xlo, v64_\text{t} \ xhi, \text{int nseg})
\end{align*}
\]

This function takes two 128-bit vector parameter, \( xlo \) and \( xhi \), and scalar immediate value for number of segments, \( nseg \). The vector is interpreted as containing elements of \( esize \)-bit integers. The two vectors are concatenated, \( xlo \) low and \( xhi \) high, forming one double-length vector. This vector is partitioned into \( npile \) piles. The piles are shuffled together, taking the first element from each of the piles in turn, then taking the second element from each of the piles, and so on until all elements have been shuffled. This process is then repeated over the elements in each of the remaining sets, resulting in a 256-bit vector. The high or low half of this vector is specified by the \( lo \) or \( hi \) string in the function name.

For this function, the \( ssize \) and \( npile \) parameters must be powers of 2, \( npile \) must be \( \geq 2 \) and \( \leq 256/esize \), and \( ssize \) must be \( \geq 4*esize \).

An \text{xshufflepair16lo}(xlo, xhi, 8) is illustrated in the figure below.
\[ r[i] = x[s], \quad i = 0..\text{NELEM} -1 \]

where:
\[
\begin{align*}
x &= xhi || xlo \\
s &= (i \& \text{mask}) | ((i \& \text{mask}) \gg \text{l2npile}) | ((i \ll (\text{l2esize} - \text{l2npile})) \& \text{mask}) \\
\text{mask} &= \text{esize} -1 \\
\text{l2esize} &= \log_2(\text{esize}) \quad \text{and} \quad \text{l2npile} = \log_2(\text{npile})
\end{align*}
\]
3.4.5 **Crossbar Swizzle**

The Crossbar Swizzle functions include:

- `_xcopy`  Swizzle Copy
- `_xreverse`  Swizzle Reverse
- `_xfieldreverse`  Field Swap
- `_xswizzle`  Swizzle
Replicates an element of a vector.

\[
v1_t \_xcopy1(v1_t \ x, \ int \ index) \\
v2_t \_xcopy2(v2_t \ x, \ int \ index) \\
v4_t \_xcopy4(v4_t \ x, \ int \ index) \\
v8_t \_xcopy8(v8_t \ x, \ int \ index) \\
v16_t \_xcopy16(v16_t \ x, \ int \ index) \\
v32_t \_xcopy32(v32_t \ x, \ int \ index) \\
v64_t \_xcopy64(v64_t \ x, \ int \ index)
\]

This function takes a 128-bit vector parameter, \( x \), and vector-element index, \( \text{index} \). The vector is interpreted as containing elements of \( \text{esize} \)-bit integers. \( \text{index} \) specifies an \( \text{esize} \)-bit field within each element to be copied, counting from the low end of the element. The indexed field is replicated, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

\( \_xcopy \) is a special case of the \( \_xswizzle \) function (page 183). The equivalent \( \_xswizzle \) syntax is:

\[
v128_t \_xswizzle(x, \ \text{esize}-1, \ \text{index} \times \text{esize})
\]

\[
r[i] = x[\text{index}], \quad i = 0..\text{NELEM} -1
\]
_xreverse

Reverses elements within a vector.

\[ \begin{align*}
&v_{1\_t} \_xreverse1(v_{1\_t} \_x) \\
v_{2\_t} \_xreverse2(v_{2\_t} \_x) \\
v_{4\_t} \_xreverse4(v_{4\_t} \_x) \\
v_{8\_t} \_xreverse8(v_{8\_t} \_x) \\
v_{16\_t} \_xreverse16(v_{16\_t} \_x) \\
v_{32\_t} \_xreverse32(v_{32\_t} \_x) \\
v_{64\_t} \_xreverse64(v_{64\_t} \_x)
\end{align*} \]

This function takes a 128-bit vector parameter, \( x \). The vector is interpreted as containing elements of \( esize \)-bit integers. The elements are reversed, producing a 128-bit result vector of \( esize \)-bit elements.

_\_xreverse is a special case of the _xswizzle function (page 183). The equivalent _xswizzle syntax is:

\[ v_{128\_t} \_xswizzle(x, 127, 128-esize) \]

![Diagram of _xreverse16(x)](496-242.ops)

\[ r[i] = x[NELEM-1-i], \quad i = 0..NELEM -1 \]
Crossbar Functions

Chapter 3: Vector and Matrix Functions

Field Reverse

`_xfieldreverse` Field Reverse

Reverses fields within elements of a vector.

```
v2_t _xfieldreverse2(v2_t x, int fsize)
v4_t _xfieldreverse4(v4_t x, int fsize)
v8_t _xfieldreverse8(v8_t x, int fsize)
v16_t _xfieldreverse16(v16_t x, int fsize)
v32_t _xfieldreverse32(v32_t x, int fsize)
v64_t _xfieldreverse64(v64_t x, int fsize)
v128_t _xfieldreverse128(v128_t x, int fsize)
```

This function takes a 128-bit vector parameter, `x`, and a field-size parameter `fsize`. The vector is interpreted as containing elements of `esize`-bit integers. Fields of `fsize` bits with each element are reversed, producing a 128-bit result vector of `esize`-bit elements.

`_xreverse` is a special case of the `_xswizzle` function on page 183. The equivalent `_xswizzle` syntax is:

```
_xswizzle(x, 127, esize-fsize)
```

```
x
    | esize | esize | esize | esize |
    | 127  |       |       |       |

_r_
    | esize | esize | esize | esize |
    | 127  |       |       |       |
```

```
r[i] = \sum_{j=0}^{nfield-1} ((x[i] >> j*fsize) & mask) << (esize-fsize-j*fsize), \quad i = 0..\text{NELEM} -1
```

where:

```
mask = (1 << fsize) - 1
nfield = esize / fsize
```
Copies and/or reverses fields within a vector.

\[ v1_t \_xswizzle1(v1_t x, \text{int copy, int swap}) \]
\[ v2_t \_xswizzle2(v2_t x, \text{int copy, int swap}) \]
\[ v4_t \_xswizzle4(v4_t x, \text{int copy, int swap}) \]
\[ v8_t \_xswizzle8(v8_t x, \text{int copy, int swap}) \]
\[ v16_t \_xswizzle16(v16_t x, \text{int copy, int swap}) \]
\[ v32_t \_xswizzle32(v32_t x, \text{int copy, int swap}) \]
\[ v64_t \_xswizzle64(v64_t x, \text{int copy, int swap}) \]
\[ v128_t \_xswizzle128(v128_t x, \text{int copy, int swap}) \]

This function takes a 128-bit vector parameter, \( x \), and scalar immediate values for copy size, \( \text{copy} \), and swap size, \( \text{swap} \). The two immediates determine how bit-fields in the source vector are re-arranged in the result. The come-from rule for how a result bit \( r \) comes from source bit \( s \) is:

\[ \text{sourceBit} = (\text{resultBit} \& \text{copy}) \oplus \text{swap} \]

The go-to rule is:

\[ r[i] = x[(i \& \text{copy}) \oplus \text{swap}], \quad i = 0..127 \]

The size of fields that are re-arranged can be determined from the binary values of \( \text{copy} \) and \( \text{swap} \): if \( \text{copy} \) ends in \( n \) 1s, and \( \text{swap} \) ends in \( n \) 0s, the field size is \( 2^n \) bits. Thus, to move bytes, \( \text{copy} \) must end in three 1s, and \( \text{swap} \) must end in three 0s. Or, to state it in modulo terms, \( \text{copy} \) mod 8 must be 7, and \( \text{swap} \) mod 8 must be 0.

\_xswizzle functions can perform many types of bit-field re-distribution, including:

- Scalar-to-vector conversion, in which a field is replicated across the entire result (see \_xcopy on page 180).
- Reversing the order of fields in a source vector, as in endian-swapping (see \_xreverse on page 181).
- The figures that follow show a small sampling of swizzles that are possible with 16-bit fields. Equivalent syntaxes are shown for several of the swizzles.
Chapter 3: Vector and Matrix Functions

Crossbar Functions

- \_xswizzle(x, 15, 0)
- \_xcopy16(x, 0)

- \_xswizzle(x, 127, 112)
- \_xreverse16(x)

- \_xswizzle(x, 127, 16)

- \_xswizzle(x, 31, 16)

- \_xswizzle(x, 63, 16)
3.4.6 **Crossbar Select Byte**

The Crossbar Select Byte functions include:

- `_xselect` Byte Select
Re-arranges source bytes or performs table lookups.

\[ v8_t \_xselect8(v8_t \text{xlo}, v8_t \text{xhi}, v8_t \text{ctrl}) \]

This function takes three 128-bit vector parameters, \( \text{xlo} \), \( \text{xhi} \), and \( \text{ctrl} \). The parameters are interpreted as vectors of bytes. The vectors are concatenated, \( \text{xlo} \) low and \( \text{xhi} \) high, forming an indexed sequence of 32 bytes. Bits 0-7 of \( \text{xlo} \) correspond to the byte at index location 0, and bits 120-127 of \( \text{xhi} \) (bits 248-255 of the concatenated vector) correspond to the byte at index location 31. The low 5 bits of each \( \text{ctrl} \) byte indexes the 32 bytes of \( \text{xlo} \) and \( \text{xhi} \). The value obtained from indexing fills the corresponding byte in the 128-bit result vector.

The function can be thought of as either byte re-arrangement or table lookup. In the first case, the concatenated \( \text{xlo} \) and \( \text{xhi} \) are the data and \( \text{ctrl} \) is a constant. In the second case, \( \text{ctrl} \) is the data and the concatenated \( \text{xlo} \) and \( \text{xhi} \) form a constant.

\[
\text{r}[i] = \text{x}[\text{ctrl}[i]]
\]

where:
- \( \text{x} \) is the concatenation of vectors \( \text{xhi}, \text{xlo} \)
- \( \text{x}[0..\text{NELEM}-1] = \text{xlo}[0..\text{NELEM}-1] \)
- \( \text{x}[\text{NELEM}..2\text{NELEM}-1] = \text{xhi}[0..\text{NELEM}-1] \)
3.4.7 **Crossbar Extract**

The Crossbar Extract functions include:

- `_xextractx`   Extract from Single-Size Source
- `_xextract`    Extract from Double-Size Source
- `_xextractm`   Extract and Merge
- `_xextract`    Extract (Generic)
**_xextractx**  
Extract from Single-Size Source

Extracts fields in elements of two vectors, and deposits them in elements of another vector.

```c
v8_t _xextractx8(v8_t x, int ctrl)
v16_t _xextractx16(v16_t x, int ctrl)
v32_t _xextractx32(v32_t x, int ctrl)
v64_t _xextractx64(v64_t x, int ctrl)
v128_t _xextractx128(v128_t x, int ctrl)
```

This function takes a 128-bit vector parameter, `x`, and a scalar control parameter, `ctrl`. The vector is interpreted as containing elements of `esize`-bit integers. The low 32 bits of `ctrl` specify control options, including `esize`, `fsize`, starting position (`spos`), and destination position (`dpos`) for the extraction. The `x` bit of `ctrl` must be set to 1 and the `m` bit of `ctrl` must be set to 0. See Section 3.5.11 on page 302 for the control parameter, `ctrl`, and extraction options.

A field of `fsize` bits is extracted from each source element. The fields are right-shifted by `spos`, and left-shifted by `dpos`, producing a 128-bit result vector of `esize`-bit elements.

The _xextractx function is very similar to the _eextractx function on page 291, except that _xextractx results cannot be rounded or limited.
_xextract  Extract from Double-Size Source

Extracts fields in elements of two vectors, and deposits them in elements of another vector.

\[
v8_t \_xextract8(v16_t x, v16_t y, int ctrl) \\
v16_t \_xextract16(v32_t x, v32_t y, int ctrl) \\
v32_t \_xextract32(v64_t x, v64_t y, int ctrl) \\
v64_t \_xextract64(v128_t x, v64_t y, int ctrl) \\
v128_t \_xextract128(v128_t x, v128_t y, int ctrl)
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\), and a scalar control parameter, \(ctrl\). The vectors are interpreted as containing elements of \(2^{*esize}\)-bit integers. The low 32 bits of \(ctrl\) specify control options, including \(esize\), \(fsize\), starting position \((spos)\), and destination position \((dpos)\) for the extraction. The \(x\) bit of \(ctrl\) must be set to 0 and the \(m\) bit of \(ctrl\) must be set to 0. See Section 3.5.11 on page 302 for the control parameter, \(ctrl\), and extraction options.

A field of \(fsize\) bits is extracted from each source element. The fields are right-shifted by \(spos\), and left-shifted by \(dpos\), producing a 128-bit result vector of \(esize\)-bit elements.

The _xextract function is very similar to the _eextract function on page 292, except that _xextract results cannot be rounded or limited.
**_xextractm**

Extracts fields in elements of two vectors, and deposits and merges them in elements of another vector.

```c
v8_t _xextractm8(v8_t x, v8_t y, int ctrl)
v16_t _xextractm16(v16_t x, v16_t y, int ctrl)
v32_t _xextractm32(v32_t x, v32_t y, int ctrl)
v64_t _xextractm64(v64_t x, v64_t y, int ctrl)
v128_t _xextractm128(v128_t x, v128_t y, int ctrl)
```

This function takes two 128-bit vector parameters, `x` and `y`, and a scalar control parameter, `ctrl`. The vectors are interpreted as containing elements of `esize`-bit integers. The low 32 bits of `ctrl` specify control options, including `esize`, `fsize`, starting position (`spos`), and destination position (`dpos`) for the extraction. The `m` bit of `ctrl` must be set to 1. See Section 3.5.11 on page 302 for the control parameter, `ctrl`, and extraction options.

A field of `fsize` bits is extracted from each source element. The fields are right-shifted by `spos`, rounded as specified, left-shifted by `dpos`, and merged with the bit positions from vector `y` that are outside the extracted fields from vector `x`, producing a 128-bit result vector of `esize`-bit elements.

The _xextractm function is very similar to the _eextractm function on page 293, except that _xextractm results cannot be rounded or limited.
Chapter 3: Vector and Matrix Functions

Crossbar Functions

1. **Vector and Matrix Functions**

   **Crossbar Functions**

   - **192 BroadMX C/C++ Functions**
   - **/K34/K39/K36/K2D/K30/K30/K37/K2E/K65/K70/K73**
   - **/K30/K33/K31/K31/K32/K37**
   - **/K66/K69/K65/K6C/K64**
   - **/K64/K70/K6F/K73/K66/K73/K69/K7A/K65/K64/K70/K6F/K73/K66/K73/K69/K7A/K65**
   - **/K65/K73/K69/K7A/K65**
   - **/K66/K69/K65/K6C/K64**
   - **/K73/K70/K6F/K73/K66/K73/K69/K7A/K65/K73/K70/K6F/K73/K66/K73/K69/K7A/K65**
   - **/K65/K73/K69/K7A/K65**
   - **/K43/K6F/K6E/K74/K72/K6F/K6C**
   - **/K41/K41**
   - **/K42/K42**
   - **/K72/K69/K67/K68/K74/K2D/K73/K68/K69/K66/K74/K62/K79**
   - **/K73/K70/K6F/K73**
   - **/K72/K6E/K64/K61/K73/K73/K70/K65/K63/K69/K66/K69/K65/K64**
   - **/K72/K6E/K64**
   - **/K72/K6E/K64**
   - **/K61/K6E/K64/K78/K62/K69/K74/K69/K67/K6E/K6F/K72/K65/K64**
   - **/K72/K6E/K64**
   - **/K72/K6E/K64**
   - **/K61/K6E/K64/K78/K62/K69/K74/K69/K67/K6E/K6F/K72/K65/K64**

   **Control Field**

   - **31**
   - **24 23**
   - **16 15 14 13 12 11 10 9 8**
   - **0**

   - **fsizes**
   - **dpos**
   - **x s n m l md gsp**

   **Control Field with m bit = 1 and x bit ignored**

   - **/K66/K73/K69/K7A/K65/K64/K70/K6F/K73/K67/K73/K73/K70/K78/K73/K6E/K6D/K6C/K72/K6E/K64**

   **Diagram**

   - **x**
   - **y**
   - **ctrl**

   - **/K34/K39/K36/K2D/K30/K30/K37/K2E/K65/K70/K73**
   - **/K30/K33/K31/K31/K32/K37**
   - **/K66/K69/K65/K6C/K64**
   - **/K64/K70/K6F/K73/K66/K73/K69/K7A/K65/K64/K70/K6F/K73/K66/K73/K69/K7A/K65**
   - **/K65/K73/K69/K7A/K65**
   - **/K66/K69/K65/K6C/K64**
   - **/K73/K70/K6F/K73/K66/K73/K69/K7A/K65/K73/K70/K6F/K73/K66/K73/K69/K7A/K65**
   - **/K65/K73/K69/K7A/K65**
   - **/K43/K6F/K6E/K74/K72/K6F/K6C**
   - **/K41/K41**
   - **/K42/K42**
   - **/K72/K69/K67/K68/K74/K2D/K73/K68/K69/K66/K74/K62/K79**
   - **/K73/K70/K6F/K73**
   - **/K72/K6E/K64/K61/K73/K73/K70/K65/K63/K69/K66/K69/K65/K64**
   - **/K72/K6E/K64**
   - **/K72/K6E/K64**
   - **/K61/K6E/K64/K78/K62/K69/K74/K69/K67/K6E/K6F/K72/K65/K64**
   - **/K72/K6E/K64**
   - **/K72/K6E/K64**
   - **/K61/K6E/K64/K78/K62/K69/K74/K69/K67/K6E/K6F/K72/K65/K64**

   **Diagram Legend**

   - **esize**
   - **fsize**
   - **spos**
   - **field**
   - **A**
   - **B**
   - **esize**
   - **ctrlx**
   - **/K66/K69/K65/K6C/K64**
   - **/K64/K70/K6F/K73/K66/K73/K69/K7A/K65/K64/K70/K6F/K73/K66/K73/K69/K7A/K65**
   - **/K65/K73/K69/K7A/K65**
   - **/K66/K69/K65/K6C/K64**
   - **/K73/K70/K6F/K73/K66/K73/K69/K7A/K65/K73/K70/K6F/K73/K66/K73/K69/K7A/K65**
   - **/K65/K73/K69/K7A/K65**
   - **/K43/K6F/K6E/K74/K72/K6F/K6C**
   - **/K41/K41**
   - **/K42/K42**
   - **/K72/K69/K67/K68/K74/K2D/K73/K68/K69/K66/K74/K62/K79**
   - **/K73/K70/K6F/K73**
   - **/K72/K6E/K64/K61/K73/K73/K70/K65/K63/K69/K66/K69/K65/K64**
   - **/K72/K6E/K64**
   - **/K72/K6E/K64**
   - **/K61/K6E/K64/K78/K62/K69/K74/K69/K67/K6E/K6F/K72/K65/K64**
   - **/K72/K6E/K64**
   - **/K72/K6E/K64**
   - **/K61/K6E/K64/K78/K62/K69/K74/K69/K67/K6E/K6F/K72/K65/K64**

   **Diagram Annotations**

   - **right-shift by spos**
   - **round as specified**
   - **left-shift by dpos**

   **Control Field with m bit = 1 and x bit ignored**

   - **/K66/K73/K69/K7A/K65/K64/K70/K6F/K73/K67/K73/K73/K70/K78/K73/K6E/K6D/K6C/K72/K6E/K64**
_xextract

Extracts fields in elements of two vectors, and deposits (and optionally merges) them in elements of another vector.

**hexlet_t _xextract(hexlet_t x, hexlet_t y, int ctrl)**

This function takes two 128-bit vector parameters, x and y, and a scalar control parameter, ctrl. The vectors are interpreted as containing elements of esize-bit or 2*esize-bit integers. The low 32 bits of ctrl specify control options, including esize, fsize, starting position (spos), and destination position (dpos) for the extraction. All three parameters are required, even though fewer may actually be used depending on the control mode selected. For modes requiring a single vector input, that input is provided via argument x, while argument y is ignored.

The following extraction modes are available:

- **Extract from Single-Size Source**: Extracts fields from esize-bit elements in vector x, and places them in esize-bit elements in the result. Vector y is not used, although its reference must be present in the function syntax. This mode is selected by setting the x bit to 1 in ctrl.
- **Extract from Double-Size Source**: Extracts fields from 2*esize-bit elements in the concatenated vectors x and y, and places them in esize-bit elements in the result. This mode is selected by clearing the x bit to 0 in ctrl.
- **Extract-Merge**—Extracts fields from esize-bit elements in vector x, merges them with the bit positions from vector y that are outside the extracted fields from vector x, and places them into esize-bit elements in the result. This mode is selected by setting the m bit to 1 in ctrl, in which case the x bit is ignored.

In all modes, a field of fsize bits is extracted from each source element. The fields are right-shifted by spos, rounded as specified, and left-shifted by dpos, producing a 128-bit result vector of esize-bit elements.

See Section 3.5.11 on page 302 for the control parameter, ctrl, and extraction options.

The generic _eextract function encompasses all the capabilities of the _eextractx, _eextract, and _eextractm functions which were described on the preceding pages. These three functions are typically used for ease of use with specific requirements.

The _xextract function is also very similar to the _eextract function on page 295, except that _xextract results cannot be rounded or limited.
3.5 **Ensemble Functions**

Ensemble functions perform fixed-point or floating-point matrix multiplies and other operations on elements of up to four 128-bit source vectors or immediate values into scalars. They return their results as a 128-bit vector of elements. The functions include fixed- and floating-point convolutions, matrix multiplication, polynomial multiplication (with the polynomial specified as a single operand), and Galois multiplication.

The Ensemble Unit combines a multiplier, floating-point unit, and summing tree. The unit contains embedded memory that supplies the large number of operands needed for the vector-matrix multiplies. Because these operations involve intense computation, using the majority of the processor’s logic and embedded-memory resources, the Ensemble functions constitute the heart of the BroadMX architecture. The other types of functions—Group, Crossbar, and Wide—typically play the role of setting up data on which Ensemble functions can operate with maximum throughput.

The Ensemble functions include the following types:

- Ensemble Arithmetic
- Ensemble Multiply
- Ensemble Multiply Sum (Dot Product)
- Ensemble Divide
- Ensemble Scale Add
- Ensemble Convolve
- Ensemble Conversion and Scaling
- Ensemble Special Operations
3.5.1 **Ensemble Arithmetic**

The Ensemble Arithmetic functions include:

- `_eabsf` Absolute Value Floating-Point
- `_eaddf` Add Floating-Point
- `_enegf` Negate Floating-Point
- `_esubf` Subtract Floating-Point
- `_esum` Sum Lateral
- `_esumf` Sum Lateral Floating-Point
- `_esump` Sum Lateral Polynomial
_eabsf  

Absolute Value Floating-Point

Computes the absolute value of vector elements, and traps on invalid operation.

\[
\begin{align*}
\text{vf16_t} & \quad \text{eabsf16}(,\mathbf{x})(\text{vf16_t} \ \mathbf{x}) \\
\text{vf32_t} & \quad \text{eabsf32}(,\mathbf{x})(\text{vf32_t} \ \mathbf{x}) \\
\text{vf64_t} & \quad \text{eabsf64}(,\mathbf{x})(\text{vf64_t} \ \mathbf{x}) \\
\text{vf128_t} & \quad \text{eabsf128}(,\mathbf{x})(\text{vf128_t} \ \mathbf{x})
\end{align*}
\]

This function takes one 128-bit vector parameter, \( \mathbf{x} \). The vector is interpreted as containing elements of \( \text{esize} \)-bit floating-point numbers. The absolute value of each element is computed, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = |x[i]|, \quad i = 0..\text{NELEM}-1
\]
_eaddf  

Add Floating-Point

Adds elements of two vectors.

```
vfloat16_t _eaddf16(c, f, n, z, x)(vfloat16_t x, vfloat16_t y)
vfloat32_t _eaddf32(c, f, n, z, x)(vfloat32_t x, vfloat32_t y)
vfloat64_t _eaddf64(c, f, n, z, x)(vfloat64_t x, vfloat64_t y)
vfloat128_t _eaddf128(c, f, n, z, x)(vfloat128_t x, vfloat128_t y)
```

This function takes two 128-bit vector parameters, \(x\) and \(y\). The vectors are interpreted as containing elements of \(\text{esize}\)-bit floating-point numbers. The vectors are added, element by element, and each resulting elements is rounded using the specified rounding mode, or round-to-nearest if not specified, producing a 128-bit result vector of \(\text{esize}\)-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = x[i] + y[i], \quad i = 0..\text{NELEM} -1
\]
Negates vector elements.

```c
vf16_t _enegf16(x) (vf16_t x)
vf32_t _enegf32(x) (vf32_t x)
vf64_t _enegf64(x) (vf64_t x)
vf128_t _enegf128(x) (vf128_t x)
```

This function takes a 128-bit vector parameter, `x`. The vector is interpreted as containing elements of `esize`-bit floating-point numbers. Each element is negated, producing a 128-bit result vector of `esize`-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = -x[i], \quad i = 0..\text{NELEM}-1
\]
_esubf

Subtract Floating-Point

Subtracts elements of two vectors.

\[ _{esubf} \] (vf16_t x, vf16_t y)
\[ _{esubf} \] (vf32_t x, vf32_t y)
\[ _{esubf} \] (vf64_t x, vf64_t y)
\[ _{esubf} \] (vf128_t x, vf128_t y)

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing elements of \( esize \)-bit floating-point numbers. Each element of \( y \) is subtracted from the corresponding element of \( x \), and each resulting element is rounded using the specified rounding mode, or round-to-nearest if not specified, producing a 128-bit result vector of \( esize \)-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[ r[i] = x[i] - y[i], \quad i = 0..NELEM -1 \]
**_esum**

**Sum Elements**

Adds all elements of a single vector.

\[
\text{v128_t } \_\text{esum8(v8_t } x) \\
\text{v128_t } \_\text{esum16(v16_t } x) \\
\text{v128_t } \_\text{esum32(v32_t } x) \\
\text{v128_t } \_\text{esum64(v64_t } x)
\]

This function takes one 128-bit vector parameter, \( x \). The vector is interpreted as containing elements of \( \text{esize} \)-bit signed integers. All the vector's elements are summed, producing a 128-bit integer result.

\[
r = \sum_{i=0}^{\text{NELEM} - 1} x[i]
\]
_esumf

Sum Elements Floating-Point

Adds all elements of a single vector.

\[
\begin{align*}
\text{vf16_t} & \quad \_\text{esumf16}(\text{vf16_t} \ x) \\
\text{vf32_t} & \quad \_\text{esumf32}(\text{vf32_t} \ x) \\
\text{vf64_t} & \quad \_\text{esumf64}(\text{vf64_t} \ x) \\
\text{vf128_t} & \quad \_\text{esumf128}(\text{vf128_t} \ x)
\end{align*}
\]

This function takes one 128-bit vector parameter, \( x \). The vector is interpreted as containing elements of \( esize \)-bit floating-point numbers. All the vector's elements are summed, producing an \( esize \)-bit floating-point result which is zero-filled at the high end.

\[
\sum_{i=0}^{\text{NELEM}-1} x[i]
\]
**_esump**

**Sum Polynomial**

Adds all elements of a single vector.

```c
uint8 _esump1 (vu1_t x)
uint8 _esump8 (vu8_t x)
uint16 _esump16 (vu16_t x)
uint32 _esump32 (vu32_t x)
uint64 _esump64 (vu64_t x)
```

This function takes a 128-bit vector parameter, `x`. The vector is interpreted as containing elements of 8 bits, representing 7th degree polynomials with binary coefficients. The elements are x-or’ed, producing a 8-bit scalar in the 128-bit result vector representing a 7th degree polynomial. Higher order bits (8 and above) are zeroed.

![Diagram of vector addition](496-202.op.s)

\[
r = \sum_{i=0}^{\text{NELEM} - 1} x[i]
\]
3.5.2  **Ensemble Multiply (Non-Extracting)**

The Ensemble Multiply (Non-Extracting) functions include:

- `_emul` Multiply
- `_emulu` Multiply Unsigned
- `_emulm` Multiply Mixed Sign
- `_emulc` Multiply Complex
- `_emulf` Multiply Floating-Point
- `_emulcf` Multiply Complex Floating-Point
- `_emulg` Multiply Galois Field
- `_emulp` Multiply Polynomial
- `_emuladd` Multiply and Add
- `_emuladdu` Multiply and Add Unsigned
- `_emuladdm` Multiply and Add Mixed Sign
- `_emuladdc` Multiply and Add Complex
- `_emuladdf` Multiply and Add Floating-Point
- `_emuladdcf` Multiply and Add Complex Floating-Point
- `_emulsub` Multiply and Subtract
- `_emulsubu` Multiply and Subtract Unsigned
- `_emulsubm` Multiply and Subtract Mixed Sign
- `_emulsubc` Multiply and Subtract Complex
- `_emulsubf` Multiply and Subtract Floating-Point
- `_emulsubcf` Multiply and Subtract Complex Floating-Point
_emul

Multiplies elements of two vectors.

\[
\begin{align*}
&v16_t \_emul8(v8_t \ x, v8_t \ y) \\
v32_t \_emul16(v16_t \ x, v16_t \ y) \\
v64_t \_emul32(v32_t \ x, v32_t \ y) \\
v128_t \_emul64(v64_t \ x, v64_t \ y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \(\mathbf{x}\) and \(\mathbf{y}\). The vectors are interpreted as containing elements of \(esize\)-bit signed integers. The elements in the low 64 bits of \(\mathbf{x}\) and \(\mathbf{y}\) are multiplied, element by element, producing a 128-bit result vector of \(2 \times esize\)-bit elements.

\[
r[i] = x[i] \times y[i], \quad i = 0..NELEM/2 - 1
\]
_emulu

Multiplying elements of two vectors.

vu16_t _emulu8 (vu8_t x, vu8_t y)
vu32_t _emulu16 (vu16_t x, vu16_t y)
vu64_t _emulu32 (vu32_t x, vu32_t y)
vu128_t _emulu64 (vu64_t x, vu64_t y)

This function takes two 128-bit vector parameters, x and y. The vectors are interpreted as containing elements of esize-bit unsigned integers. The elements in the low 64 bits of x and y are multiplied, element by element, producing a 128-bit result vector of 2*esize-bit elements.

\[ r[i] = x[i] \times u y[i], \quad i = 0..\text{NELEM}/2 -1 \]
_emulm

Multiply Mixed Sign

Multiplies elements of two vectors.

\[
v16_t \quad _{emulm8}(vu8_t \ x, v8_t \ y) \\
v32_t \quad _{emulm16}(vu16_t \ x, v16_t \ y) \\
v64_t \quad _{emulm32}(vu32_t \ x, v32_t \ y) \\
v128_t \quad _{emulm64}(vu64_t \ x, v64_t \ y)
\]

This function takes two 128-bit vector parameters, \( x \) (unsigned) and \( y \) (signed). Vector \( x \) is interpreted as containing elements of \( esize \)-bit unsigned integers, and vector \( y \) is interpreted as containing elements of \( esize \)-bit signed integers. The elements in the low 64 bits of \( x \) and \( y \) are multiplied, element by element, producing a 128-bit signed result vector of \( 2 \times esize \)-bit elements.

\[
r[i] = x[i] \times m \times y[i]. \quad i = 0..NELEM -1
\]
_emulc

Multiply Complex

Multiplies elements of two complex-number vectors.

\[
\begin{align*}
\text{vc16}_t & \quad \text{emulc8}(\text{vc8}_t \ x, \ \text{vc8}_t \ y) \\
\text{vc32}_t & \quad \text{emulc16}(\text{vc16}_t \ x, \ \text{vc16}_t \ y) \\
\text{vc64}_t & \quad \text{emulc32}(\text{vc32}_t \ x, \ \text{vc32}_t \ y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing sets of \( 2^{*}\text{esize} \)-bit complex integers, in which the real and imaginary elements are each \( \text{esize} \) bits wide. The elements in the low 64 bits of \( x \) and \( y \) are complex-multiplied, element by element, producing a 128-bit result vector of \( 2^{*}\text{esize} \)-bit elements.

\[
r[i] = x[i] \ast y[i], \quad i = 0..\text{NELEM}/2 -1
\]
_emulf

Multiply Floating-Point

Multiplies elements of two vectors.

\[ \text{vf16_t } _{\text{emulf16}}(\text{c}, \text{f}, \text{n}, \text{z}, \text{x})(\text{vf16_t } \text{x}, \text{vf16_t } \text{y}) \]
\[ \text{vf32_t } _{\text{emulf32}}(\text{c}, \text{f}, \text{n}, \text{z}, \text{x})(\text{vf32_t } \text{x}, \text{vf32_t } \text{y}) \]
\[ \text{vf64_t } _{\text{emulf64}}(\text{c}, \text{f}, \text{n}, \text{z}, \text{x})(\text{vf64_t } \text{x}, \text{vf64_t } \text{y}) \]
\[ \text{vf128_t } _{\text{emulf128}}(\text{c}, \text{f}, \text{n}, \text{z}, \text{x})(\text{vf128_t } \text{x}, \text{vf128_t } \text{y}) \]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing elements of \( \text{esize} \)-bit floating-point numbers. The elements are multiplied, element by element. The results are rounded, using the specified rounding mode, or round-to-nearest if not specified, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

See Section 12 on page 40 for details on optional rounding and exception modes.

\[ r[i] = x[i] \times y[i], \quad i = 0..\text{NELEM -1} \]
_emulcf  Multiply Complex Floating-Point

Multiplies elements of two complex-number vectors.

\[
\begin{align*}
\text{vcf16_t} & \quad \text{emulcf16}(\text{vcf16_t} \ x, \ \text{vcf16_t} \ y) \\
\text{vcf32_t} & \quad \text{emulcf32}(\text{vcf32_t} \ x, \ \text{vcf32_t} \ y) \\
\text{vcf64_t} & \quad \text{emulcf64}(\text{vcf64_t} \ x, \ \text{vcf64_t} \ y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\). The vectors are interpreted as containing sets of \(2^{\text{esize}}\)-bit complex floating-point numbers, in which the real and imaginary elements are each \(\text{esize}\) bits wide. The vectors are complex-multiplied, element by element. The results are rounded using round-to-nearest, producing a 128-bit result vector of \(2^{\text{esize}}\)-bit complex floating-point numbers.

\[
r[i] = x[i] \times y[i], \quad i = 0..\text{NELEM}-1
\]
_emulg

Multiply Galois Field

Multiplies 7\textsuperscript{th} degree polynomial vectors.

\texttt{v8\_t \_emulg8(v8\_t x, v8\_t y, int z)}

This function takes two 128-bit vector parameters, \textit{x} and \textit{y}, and a \textit{basis polynomial} \textit{z}. The \textit{x} and \textit{y} vectors are interpreted as containing elements of 8 bits, representing 7\textsuperscript{th} degree polynomials with binary coefficients. The integer \textit{z} is interpreted as the basis polynomial, with an implied 1 bit in bit-position 8. Corresponding elements of \textit{x} and \textit{y} are polynomial-multiplied (see page 211 for a description of polynomial multiplication), producing an intermediate product of double-size elements. Each intermediate product is then taken, modulo the basis polynomial (with its implied 1 bit), producing a 128-bit result vector with 8-bit elements, each representing a 7\textsuperscript{th} degree polynomial. The operation never overflows.

The basis polynomial is like a prime number, in that it cannot be factored into other polynomials. It is typically chosen so that the set of binary polynomials, together with this modulo-product operation, form a group-theoretic field known as a Galois field. The operation is used in error-correcting codes.

\[
r[i] = (x[i] \times^p y[i]) \mod_p z[i]
\]
_emulp

Multiply Polynomial

Multiplies polynomial vectors.

\[ \text{v16}_t \_\text{emulp8}(v8\_t \ x, v8\_t \ y) \]
\[ \text{v32}_t \_\text{emulp16}(v16\_t \ x, v16\_t \ y) \]
\[ \text{v64}_t \_\text{emulp32}(v32\_t \ x, v32\_t \ y) \]
\[ \text{v128}_t \_\text{emulp64}(v64\_t \ x, v64\_t \ y) \]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The low order 64 bits of the vectors are interpreted as containing \( 64/\text{esize} \) elements, each being a polynomial of degree \( \text{esize}-1 \) with binary coefficients. Corresponding elements of \( x \) and \( y \) are polynomial-multiplied, producing a 128-bit result vector of \( 2*\text{esize}-1 \)-degree polynomial elements.

The polynomial multiply is like an unsigned multiply, except there are no carries. For example, if an element of the \( x \) vector contains the binary coefficients, \( \text{hgfedcba} \), its polynomial is:

\[ h x^7 + gx^6 + fx^5 + ex^4 + dx^3 + cx^2 + bx + a \]

and if an element of the \( y \) vector contains the binary coefficients, \( \text{ponmlkji} \), its polynomial is:

\[ px^7 + ox^6 + nx^5 + mx^4 + lx^3 + kx^2 + jx + i \]

then the product of the \( x \) are \( y \) elements is:

\[ ... + (ak + bj + ci)x^2 + (aj + bi)x + (ai) \]

where the multiplications are actually bit-wise ANDs, the additions are actually bit-wise XORs, and the result is packed into 16 bits.

Another way to view the polynomial multiply is:

\[ z = p \star p q \]

where:

\[ z_i = (p \star p q)_i = \left( \sum_{j=0}^{i} p_j \cdot q_{i-j} \right) \]
\[ r[i] = x[i] \times y[i], \quad i = 0..NELEM/2 - 1 \]
_emuladd

Multiply and Add

Multiplies elements of two vectors and adds elements of a third vector.

\[
\begin{align*}
\text{v16_t } & \text{ _emuladd8(v16_t } \; x, \; \text{v8_t } \; y, \; \text{v8_t } \; z) \\
\text{v32_t } & \text{ _emuladd16(v32_t } \; x, \; \text{v16_t } \; y, \; \text{v16_t } \; z) \\
\text{v64_t } & \text{ _emuladd32(v64_t } \; x, \; \text{v32_t } \; y, \; \text{v32_t } \; z) \\
\text{v128_t } & \text{ _emuladd64(v128_t } \; x, \; \text{v64_t } \; y, \; \text{v64_t } \; z) \\
\end{align*}
\]

This function takes three 128-bit vector parameters, \( x, y, \) and \( z \). Vectors \( y \) and \( z \) are interpreted as containing elements of \( \text{esize} \)-bit signed integers. Vector \( x \) is interpreted as containing elements of \( 2^\text{esize} \)-bit signed integers. The elements in the low 64 bits of \( y \) and \( z \) are multiplied, element by element, producing a 128-bit vector of \( 2^\text{esize} \)-bit elements. The corresponding \( 2^\text{esize} \)-bit elements of \( x \) are added to each product, producing a 128-bit result vector of \( 2^\text{esize} \)-bit elements. Overflow is ignored.

\[
r[i] = x[i] + (y[i] \times z[i]), \quad i = 0..\text{NELEM -1}
\]
_emuladdu Multiply and Add Unsigned

Multiplies unsigned elements of two vectors and adds elements of a third vector.

```c
vu16_t _emuladdu8(vu16_t x, vu8_t y, vu8_t z)
vu32_t _emuladdu16(vu32_t x, vu16_t y, vu16_t z)
vu64_t _emuladdu32(vu64_t x, vu32_t y, vu32_t z)
vul28_t _emuladdu64(vul28_t x, vu64_t y, vu64_t z)
```

This function takes three 128-bit vector parameters, \( x \), \( y \), and \( z \). Vectors \( y \) and \( z \) are interpreted as containing elements of \( \text{esize} \)-bit unsigned integers. Vector \( x \) is interpreted as containing elements of \( 2 \times \text{esize} \)-bit unsigned integers. The elements in the low 64 bits of \( y \) and \( z \) are multiplied, element by element, producing a 128-bit vector of \( 2 \times \text{esize} \)-bit elements. The corresponding elements in the low 64 bits of \( x \) are added to each product, producing a 128-bit result vector of \( 2 \times \text{esize} \)-bit elements. Overflow is ignored.

\[
r[i] = x[i] + (y[i] \times u z[i]), \quad i = 0..\text{NELEM} - 1
\]
_emuladdm  Multiply and Add Mixed Sign

Multiplies elements of mixed-signed vectors and adds elements of a third vector.

\[
\begin{align*}
\text{v16}_t \_\text{emuladdm8}(\text{v16}_t \ x, \ \text{vu8}_t \ y, \ \text{v8}_t \ z) \\
\text{v32}_t \_\text{emuladdm16}(\text{v32}_t \ x, \ \text{vu6}_t \ y, \ \text{v16}_t \ z) \\
\text{v64}_t \_\text{emuladdm32}(\text{v64}_t \ x, \ \text{vu32}_t \ y, \ \text{v32}_t \ z) \\
\text{v128}_t \_\text{emuladdm64}(\text{v128}_t \ x, \ \text{vu64}_t \ y, \ \text{v64}_t \ z)
\end{align*}
\]

This function takes three 128-bit vector parameters, \(x\) (signed), \(y\) (unsigned), and \(z\) (signed). Vector \(y\) is interpreted as containing elements of \(\text{esize}\)-bit unsigned integers, and vector \(z\) is interpreted as containing elements of \(\text{esize}\)-bit signed integers. Vector \(x\) is interpreted as containing elements of \(2^{\text{esize}}\)-bit integers. The elements in the low 64 bits of \(y\) and \(z\) are multiplied, element by element, producing a 128-bit vector of \(2^{\text{esize}}\)-bit elements. The corresponding elements in the low 64 bits of \(x\) are added to each product, producing a 128-bit result vector of \(2^{\text{esize}}\)-bit elements. Overflow is ignored.

\[
r[i] = x[i] + (y[i] \times z[i]), \quad i = 0..\text{NELEM} -1
\]
_emuladdc  Multiply and Add Complex

Multiplies elements of two complex-number vectors and adds elements of a third such vector.

\[ r[i] = x[i] + (y[i] \times z[i]), \quad i = 0..\text{NELEM}/2 -1 \]
Multplies elements of two vectors and adds elements of a third vector.

\[
\begin{align*}
\text{vf16}_t & \quad _\text{emuladdf16}\{,c,f,n,z,x\}(\text{vf16}_t \ x, \ \text{vf16}_t \ y, \ \text{vf16}_t \ z) \\
\text{vf32}_t & \quad _\text{emuladdf32}\{,c,f,n,z,x\}(\text{vf32}_t \ x, \ \text{vf32}_t \ y, \ \text{vf32}_t \ z) \\
\text{vf64}_t & \quad _\text{emuladdf64}\{,c,f,n,z,x\}(\text{vf64}_t \ x, \ \text{vf64}_t \ y, \ \text{vf64}_t \ z) \\
\text{vf128}_t & \quad _\text{emuladdf128}\{,c,f,n,z,x\}(\text{vf128}_t \ x, \ \text{vf128}_t \ y, \ \text{vf128}_t \ z)
\end{align*}
\]

This function takes three 128-bit vector parameters, \(x\), \(y\), and \(z\). The vectors are interpreted as containing elements of \(\text{esize}\)-bit floating-point numbers. The vectors \(y\) and \(z\) are multiplied, element by element, and the corresponding elements of \(x\) are added to each product. The results are rounded using the specified rounding mode, or round-to-nearest if not specified, producing a 128-bit result vector of \(\text{esize}\)-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = x[i] + (y[i] \ast z[i]), \quad i = 0..\text{NELEM}-1
\]
_emuladdcf

Multiply and Add Complex Floating-Point

Multiplies elements of two complex-number vectors and adds elements of a third such vector.

vcf16_t _emuladdcf16(vcf16_t x, vcf16_t y, vcf16_t z)
vcf32_t _emuladdcf32(vcf32_t x, vcf32_t y, vcf32_t z)
vcf64_t _emuladdcf64(vcf64_t x, vcf64_t y, vcf64_t z)

This function takes three 128-bit vector parameters, x, y, and z. The vectors are interpreted as containing sets of $2^\text{esize}$-bit complex floating-point numbers, in which the real and imaginary elements are each esize bits wide. The vectors y and z are complex-multiplied, element by element, and the corresponding elements of x are added to each product. The results are rounded using round-to-nearest, producing a 128-bit result vector of $2^\text{esize}$-bit complex floating-point numbers.
\[ r[i] = x[i] + (y[i] \times z[i]), \quad i = 0..\text{NELEM} - 1 \]
_emulsub

Multiply and Subtract

Multiplies elements of two vectors and subtracts elements of a third vector.

\[ v_{16}\_t \_\text{emulsub8}(v_{16}\_t \ x, v_{8}\_t \ y, v_{8}\_t \ z) \]
\[ v_{32}\_t \_\text{emulsub16}(v_{32}\_t \ x, v_{16}\_t \ y, v_{16}\_t \ z) \]
\[ v_{64}\_t \_\text{emulsub32}(v_{64}\_t \ x, v_{32}\_t \ y, v_{32}\_t \ z) \]
\[ v_{128}\_t \_\text{emulsub64}(v_{128}\_t \ x, v_{64}\_t \ y, v_{64}\_t \ z) \]

This function takes three 128-bit vector parameters, \( x, y, \) and \( z \). Vectors \( y \) and \( z \) are interpreted as containing elements of \( esize \)-bit signed integers. Vector \( x \) is interpreted as containing elements of \( 2*esize \)-bit signed integers. The elements in the low 64 bits of \( y \) and \( z \) are multiplied, element by element, producing a 128-bit vector of \( 2*esize \)-bit elements. The corresponding elements in the low 64 bits of \( x \) are subtracted from each product, producing a 128-bit result vector of \( 2*esize \)-bit elements. Overflow is ignored.

\[
r[i] = x[i] - (y[i] \times z[i]), \quad i = 0..\text{NELEM} - 1
\]
**_emulsubu**  

**Multiply and Subtract Unsigned**

Multiplies elements of two vectors and subtracts elements of a third vector.

\[
vu16_t \ _emulsubu8(vu16_t \ x, \ vu8_t \ y, \ vu8_t \ z) \\
vu32_t \ _emulsubu16(vu32_t \ x, \ vu16_t \ y, \ vu16_t \ z) \\
vu64_t \ _emulsubu32(vu64_t \ x, \ vu32_t \ y, \ vu32_t \ z) \\
vul28_t \ _emulsubu64(vul28_t \ x, \ vul64_t \ y, \ vul64_t \ z)
\]

This function takes three 128-bit vector parameters, \(x\), \(y\), and \(z\). Vectors \(y\) and \(z\) are interpreted as containing elements of \(esize\)-bit unsigned integers. Vector \(x\) is interpreted as containing elements of \(2 \times esize\)-bit unsigned integers. The elements in the low 64 bits of \(y\) and \(z\) are multiplied, element by element, producing a 128-bit vector of \(2 \times esize\)-bit elements. The corresponding elements in the low 64 bits of \(x\) are subtracted from each product, producing a 128-bit result vector of \(2 \times esize\)-bit elements. Overflow is ignored.

\[
r[i] = x[i] - (y[i] \times z[i]), \quad i = 0..NELEM -1
\]
_emulsubm  Multiply and Subtract Mixed Sign

Multiplies elements of two vectors and subtracts elements of a third vector.

\[
\begin{align*}
\text{v16}_t\ _\text{emulsubm8}(\text{v16}_t\ x, \text{vu8}_t\ y, \text{v8}_t\ z) \\
\text{v32}_t\ _\text{emulsubm16}(\text{v32}_t\ x, \text{vu16}_t\ y, \text{v16}_t\ z) \\
\text{v64}_t\ _\text{emulsubm32}(\text{v64}_t\ x, \text{vu32}_t\ y, \text{v32}_t\ z) \\
\text{v128}_t\ _\text{emulsubm64}(\text{v128}_t\ x, \text{vu64}_t\ y, \text{v64}_t\ z)
\end{align*}
\]

This function takes three 128-bit vector parameters, \(x\), \(y\) (unsigned), and \(z\) (signed). Vector \(y\) is interpreted as containing elements of \(esize\)-bit unsigned integers, and vector \(z\) is interpreted as containing elements of \(esize\)-bit signed integers. Vector \(x\) is interpreted as containing elements of \(2 \times esize\)-bit integers. The elements in the low 64 bits of \(y\) and \(z\) are multiplied, element by element, producing a 128-bit vector of \(2 \times esize\)-bit elements. The corresponding elements in the low 64 bits of \(x\) are subtracted from each product, producing a 128-bit result vector of \(2 \times esize\)-bit elements. Overflow is ignored.

\[
r[i] = x[i] - (y[i] \times z[i]), \quad i = 0..\text{NELEM} -1
\]
**_emulsubc**  

**Multiply and Subtract Complex**

Multiplies elements of two complex-number vectors and subtracts elements of a third such vector.

\[
\begin{align*}
\text{vc16}_t \quad \text{emulsubc8}(\text{vc16}_t \ x, \ \text{vc8}_t \ y, \ \text{vc8}_t \ z) \\
\text{vc32}_t \quad \text{emulsubc16}(\text{vc32}_t \ x, \ \text{vc16}_t \ y, \ \text{vc16}_t \ z) \\
\text{vc64}_t \quad \text{emulsubc32}(\text{vc64}_t \ x, \ \text{vc32}_t \ y, \ \text{vc32}_t \ z)
\end{align*}
\]

This function takes three 128-bit vector parameters, \( x, y, \) and \( z \). Vectors \( y \) and \( z \) are interpreted as containing sets of \( 2^\text{esize} \)-bit complex integers, in which the real and imaginary elements are each \( \text{esize} \) bits wide. Vector \( x \) is interpreted as containing sets of \( 4^\text{esize} \)-bit complex integers, in which the real and imaginary elements are each \( 2^\text{esize} \) bits wide. The elements in the low 64 bits of \( y \) and \( z \) are complex-multiplied, element by element, producing a 128-bit vector of \( 2^\text{esize} \)-bit elements. The corresponding elements of \( x \) are subtracted from each product, producing a 128-bit result vector of \( 2^\text{esize} \)-bit elements. Overflow is ignored.

\[
\begin{align*}
\mathbf{r}[i] &= \mathbf{x}[i] - (\mathbf{y}[i] \times \mathbf{z}[i]), \quad i = 0..\text{NELEM} - 1
\end{align*}
\]
_emulsubf Multiply and Subtract Floating-Point

Multiplies elements of two vectors and subtracts elements of a third vector.

\[
\begin{align*}
\text{vf16}_t & \quad \_\text{emulsubf16}(\text{vf16}_t \ x, \ \text{vf16}_t \ y, \ \text{vf16}_t \ z) \\
\text{vf32}_t & \quad \_\text{emulsubf32}(\text{vf32}_t \ x, \ \text{vf32}_t \ y, \ \text{vf32}_t \ z) \\
\text{vf64}_t & \quad \_\text{emulsubf64}(\text{vf64}_t \ x, \ \text{vf64}_t \ y, \ \text{vf64}_t \ z) \\
\text{vf128}_t & \quad \_\text{emulsubf128}(\text{vf128}_t \ x, \ \text{vf128}_t \ y, \ \text{vf128}_t \ z)
\end{align*}
\]

This function takes three 128-bit vector parameters, \(x\), \(y\), and \(z\). The vectors are interpreted as containing elements of \(esize\)-bit floating-point numbers. The vectors \(y\) and \(z\) are multiplied, element by element, and the corresponding elements of \(x\) are subtracted from each product. The results are rounded using round-to-nearest, producing a 128-bit result vector of \(esize\)-bit elements.

\[
r[i] = x[i] - (y[i] \times z[i]), \quad i = 0..\text{NELEM} -1
\]
_emulsubcf

Multiply and Subtract Complex Floating-Point

Multiplies elements of two complex-number vectors and subtracts elements of a third such vector.

\[
\begin{align*}
\text{vcf16_t} & \quad \_\text{emulsubcf16}(\text{vcf16_t} \ x, \ \text{vcf16_t} \ y, \ \text{vcf16_t} \ z) \\
\text{vcf32_t} & \quad \_\text{emulsubcf32}(\text{vcf32_t} \ x, \ \text{vcf32_t} \ y, \ \text{vcf32_t} \ z) \\
\text{vcf64_t} & \quad \_\text{emulsubcf64}(\text{vcf64_t} \ x, \ \text{vcf64_t} \ y, \ \text{vcf64_t} \ z)
\end{align*}
\]

This function takes three 128-bit vector parameters, \( x, y, \) and \( z \). The vectors are interpreted as containing sets of \( 2^{*\text{esize}} \)-bit complex floating-point numbers, in which the real and imaginary elements are each \( \text{esize} \) bits wide. The vectors \( y \) and \( z \) are complex-multiplied, element by element, and the corresponding elements of \( x \) are subtracted from each product. The results are rounded using round-to-nearest, producing a 128-bit result vector of \( 2^{*\text{esize}} \)-bit complex floating-point numbers.
\[ r[i] = x[i] - (y[i] \times z[i]), \quad i = 0..\text{NELEM} - 1 \]
3.5.3 **Ensemble Multiply (Extract Immediate)**

The Ensemble Multiply (Extract Immediate) functions include:

- `_emulxi` Multiply and Extract Immediate
- `_emulxic` Multiply and Extract Immediate Complex
- `_emuladdxi` Multiply, Add, and Extract Immediate
- `_emuladdxic` Multiply, Add, and Extract Immediate Complex
_emulxi

Multiply and Extract Immediate

Multiplying elements of two vectors and extracting fields.

v8_t _emulxi8n(v8_t x, v8_t y, int sh)
v16_t _emulxi16n(v16_t x, v16_t y, int sh)
v32_t _emulxi32n(v32_t x, v32_t y, int sh)
v64_t _emulxi64n(v64_t x, v64_t y, int sh)

This function takes two 128-bit vector parameters, x and y, and an immediate scalar shift amount, sh. The vectors are interpreted as containing elements of esize-bit signed integers.

The elements of x and y are multiplied, element by element, producing a vector of 2*esize-bit elements. Each element is right-shifted by sh, with rounding to nearest. The low esize bits are taken as the result, producing a 128-bit result vector of esize-bit elements. If a result exceeds the range representable in an esize-bit signed integer, it is limited.

_emuladdxi is equivalent to _emuladdx with spos = sh, fsize = esize, and dpos = 0.
\[ r[i] = (x[i] \times y[i]) \gg \text{sh}, \text{ round to nearest, } i = 0..\text{NELEM}-1 \]
_emulxic | Multiply and Extract Immediate Complex

Multiplies elements of two complex-number vectors and extracts fields.

```
vc8_t _emulxic8(vc8_t x, vc8_t y, int sh)
vc16_t _emulxic16(vc16_t x, vc16_t y, int sh)
vc32_t _emulxic32(vc32_t x, vc32_t y, int sh)
```

This function takes two 128-bit vector parameters, \( x \) and \( y \), and an immediate scalar shift amount, \( sh \). The vectors are interpreted as containing sets of \( 2^{esize} \)-bit complex integers, in which the real and imaginary elements are each \( esize \) bits wide.

The elements of \( x \) and \( y \) are complex-multiplied, element by element, producing a vector of \( 2^{esize} \)-bit elements. Each element is right-shifted by \( sh \), with rounding to nearest. The low \( esize \) bits are taken as the result, producing a 128-bit result vector of \( 2^{esize} \)-bit complex integers, in which the real and imaginary elements are each \( esize \) bits wide. If a result exceeds the range representable in an \( esize \)-bit signed integer, it is limited.

_\emulxic is equivalent to complex-number version of _\emulx with \( spos = sh \), \( fsize = esize \), and \( dpos = 0 \).
$r[i] = (x[i] * y[i]) >> sh$, round to nearest, $i = 0..\text{NELEM}-1$
**_emuladdxi**  
**Multiply, Add, and Extract Immediate**

Multiplies elements of two vectors, adds elements of a third vector, and extracts fields.

\[
\begin{align*}
\text{v8}_t & \quad _\text{emuladdxi8}(\text{v8}_t \ x, \text{v8}_t \ y, \text{v8}_t \ z, \text{int} \ sh) \\
\text{v16}_t & \quad _\text{emuladdxi16}(\text{v16}_t \ x, \text{v16}_t \ y, \text{v16}_t \ z, \text{int} \ sh) \\
\text{v32}_t & \quad _\text{emuladdxi32}(\text{v32}_t \ x, \text{v32}_t \ y, \text{v32}_t \ z, \text{int} \ sh) \\
\text{v64}_t & \quad _\text{emuladdxi64}(\text{v64}_t \ x, \text{v64}_t \ y, \text{v64}_t \ z, \text{int} \ sh) \\
\text{v128}_t & \quad _\text{emuladdxi128}(\text{v128}_t \ x, \text{v128}_t \ y, \text{v128}_t \ z, \text{int} \ sh)
\end{align*}
\]

This function takes three 128-bit vector parameters, \( x, y, \) and \( z, \) and an immediate scalar shift amount, \( sh. \) The vectors are interpreted as containing elements of \( esize\)-bit signed integers.

The elements of \( y \) and \( z \) are multiplied, element by element, producing a vector of \( 2^{esize}\)-bit elements. The corresponding elements of \( x \) are left-shifted by \( sh \) and added to these products. Each element is right-shifted by \( sh, \) with rounding to nearest. The low \( esize \) bits are taken as the result, producing a 128-bit result vector of \( esize\)-bit elements. If a result exceeds the range representable in an \( esize\)-bit signed integer, it is limited.

\( _\text{emuladdxi} \) is equivalent to \( _\text{emuladdx} \) with \( spos = sh, fsize = esize, \) and \( dpos = 0. \)
\[ r[i] = ((x[i] \ll sh) + y[i] \times z[i]) \gg sh, \text{ round to nearest, } i = 0..\text{NELEM}-1 \]
_emuladdxic  Multiply, Add, and Extract Immediate Complex

Multiplies elements of two complex-number vectors, adds elements of a third vector, and extracts fields.

```c
vc8_t _emuladdxic8(vc8_t x, vc8_t y, vc8_t z, int sh)
vc16_t _emuladdxic16(vc16_t x, vc16_t y, vc16_t z, int sh)
vc32_t _emuladdxic32(vc32_t x, vc32_t y, vc32_t z, int sh)
vc64_t _emuladdxic64(vc64_t x, vc64_t y, vc64_t z, int sh)
```

This function takes three 128-bit vector parameters, \(x\), \(y\), and \(z\), and an immediate scalar shift amount, \(sh\). The vectors are interpreted as containing sets of \(2^{esize}\)-bit complex integers, in which the real and imaginary elements are each \(esize\) bits wide.

The elements of \(y\) and \(z\) are complex-multiplied, element by element, producing a vector of \(2^{esize}\)-bit elements. The corresponding elements of \(x\) are left-shifted by \(sh\) and added to these products. Each element is right-shifted by \(sh\), with rounding to nearest. The low \(esize\) bits are taken as the result, producing a 128-bit result vector of \(2^{esize}\)-bit complex integers, in which the real and imaginary elements are each \(esize\) bits wide. If a result exceeds the range representable in an \(esize\)-bit signed integer, it is limited.

_emuladdxic is equivalent to complex-number version of _emuladdx with spos = sh, fsize = esize, and dpos = 0.
\[ r[i] = ((x[i] \ll \text{sh}) + y[i] \times z[i]) \gg \text{sh}, \text{ round to nearest, } i = 0..\text{NELEM}-1 \]
3.5.4 **Ensemble Multiply (Generalized Extract)**

The Ensemble Multiply (Generalized Extract) functions include:

- `_emulx` Multiply and Extract
- `_emulxx` Multiply and Extract/Expand
- `_emulxc` Multiply and Extract Complex
- `_emulxxc` Multiply and Extract/Expand Complex
- `_emulx` Multiply and Extract (Generic)
- `_emuladdx` Multiply, Add, and Extract
- `_emuladdxx` Multiply, Add, and Extract/Expand
- `_emuladdxc` Multiply, Add, and Extract Complex
- `_emuladdxxc` Multiply, Add, and Extract/Expand Complex
- `_emuladdx` Multiply, Add, and Extract (Generic)
**_emulx**

Multiply and Extract

Multiplies elements of two vectors and extracts fields.

```c
v8_t _emulx8(v8_t x, v8_t y, int ctrl)
v16_t _emulx16(v16_t x, v16_t y, int ctrl)
v32_t _emulx32(v32_t x, v32_t y, int ctrl)
v64_t _emulx64(v64_t x, v64_t y, int ctrl)
v128_t _emulx64(v128_t x, v128_t y, int ctrl)
```

This function takes two 128-bit vector parameters, \( x \) and \( y \), and a scalar control parameter, \( \text{ctrl} \). The vectors are interpreted as containing elements of \( \text{esize} \)-bit integers. The low 32 bits of \( \text{ctrl} \) specify control options, including \( \text{esize} \), \( \text{fsize} \), starting position \( \text{spos} \), and destination position \( \text{dpos} \) for the extraction.

The elements of \( x \) and \( y \) are multiplied, element by element, producing an intermediate vector of \( 2^{\text{esize}} \)-bit elements. Fields of \( \text{fsize} \) bits are then extracted from this intermediate result, right-shifted by \( \text{spos} \), rounded as specified, left-shifted by \( \text{dpos} \), and placed in \( \text{esize} \)-bit elements in the 128-bit result vector.

See Section 3.5.11 on page 302 for the control parameter, \( \text{ctrl} \), and extraction options. For this function, the \( x \) bit and the \( n \) bit in \( \text{ctrl} \) are both set to 0.
_emulxx Multiply and Extract/Expand

Multiplies elements of two vectors and extracts fields.

\[
\begin{align*}
\text{v16}_t \; & \text{emulxx8}\left(\text{v8}_t \; x, \; \text{v8}_t \; y, \; \text{int} \; \text{ctrl}\right) \\
\text{v32}_t \; & \text{emulxx16}\left(\text{v16}_t \; x, \; \text{v16}_t \; y, \; \text{int} \; \text{ctrl}\right) \\
\text{v64}_t \; & \text{emulxx32}\left(\text{v32}_t \; x, \; \text{v32}_t \; y, \; \text{int} \; \text{ctrl}\right) \\
\text{v128}_t \; & \text{emulxx64}\left(\text{v64}_t \; x, \; \text{v64}_t \; y, \; \text{int} \; \text{ctrl}\right)
\end{align*}
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\), and a scalar control parameter, \(ctrl\). The vectors are interpreted as containing elements of \(esize\)-bit integers. The low 32 bits of \(ctrl\) specify control options, including \(esize\), \(fsize\), starting position \((spos)\), and destination position \((dpos)\) for the extraction.

The elements of \(x\) and \(y\) are multiplied, element by element, producing an intermediate vector of \(2\times esize\)-bit elements. Fields of \(fsize\) bits are then extracted from this intermediate result, right-shifted by \(spos\), rounded as specified, left-shifted by \(dpos\), and placed in \(2\times esize\)-bit elements in the 128-bit result vector. The high 64 bits of \(x\) and \(y\) are ignored.

See Section 3.5.11 on page 302 for the control parameter, \(ctrl\), and extraction options. For this function, the \(x\) bit in \(ctrl\) is set to 1 and the \(n\) bit in \(ctrl\) is set to 0.
Chapter 3: Vector and Matrix Functions

**Ensemble Functions**

![Diagram of ensemble functions]

**Control Field with x bit = 1**

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsize</td>
<td>dpos</td>
<td>x</td>
<td>s</td>
<td>n</td>
<td>m</td>
<td>rnd</td>
<td>gssp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Right-shift by spos
Round as specified
Left-shift by dpos

---

240 BroadMX C/C++ Functions
Multiply and Extract Complex

Multiplies elements of two complex-number vectors and extracts fields.

\[
\begin{align*}
vc8_t & \_emulxc8(vc8_t \ x, \ vc8_t \ y, \ int \ ctrl) \\
vc16_t & \_emulxc16(vc16_t \ x, \ vc16_t \ y, \ int \ ctrl) \\
vc32_t & \_emulxc32(vc32_t \ x, \ vc32_t \ y, \ int \ ctrl) \\
vc64_t & \_emulxc64(vc64_t \ x, \ vc64_t \ y, \ int \ ctrl)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \), and a scalar control parameter, \( ctrl \). The vectors are interpreted as containing elements of \( esize \)-bit complex integers. The low 32 bits of \( ctrl \) specify control options, including \( esize \), \( fsize \), starting position \( (spos) \), and destination position \( (dpos) \) for the extraction.

The elements of \( x \) and \( y \) are multiplied, element by element, producing an intermediate vector of \( 2 \times esize \)-bit elements. Fields of \( fsize \) bits are then extracted from this intermediate result, right-shifted by \( spos \), rounded as specified, left-shifted by \( dpos \), and placed in \( esize \)-bit elements in the 128-bit complex result vector.

See Section 3.5.11 on page 302 for the control parameter, \( ctrl \), and extraction options. For this function, the \( x \) bit in \( ctrl \) is set to 0 and the \( n \) bit in \( ctrl \) is set to 1.
_emulxxc Multiply and Extract/Expand Complex

Multiplies elements of two complex-number vectors and extracts fields.

\[ \text{vc16_t } \_\text{emulxxc8} (\text{vc8_t } x, \text{vc8_t } y, \text{int } ctrl) \]
\[ \text{vc32_t } \_\text{emulxxc16} (\text{vc16_t } x, \text{vc16_t } y, \text{int } ctrl) \]
\[ \text{vc64_t } \_\text{emulxxc32} (\text{vc32_t } x, \text{vc32_t } y, \text{int } ctrl) \]

This function takes two 128-bit vector parameters, \( x \) and \( y \), and a scalar control parameter, \( ctrl \). The vectors are interpreted as containing elements of \( esize \)-bit complex integers. The low 32 bits of \( ctrl \) specify control options, including \( esize \), \( fsize \), starting position (\( spos \)), and destination position (\( dpos \)) for the extraction.

The elements of \( x \) and \( y \) are multiplied, element by element, producing an intermediate vector of \( 2 \cdot esize \)-bit elements. Fields of \( fsize \) bits are then extracted from this intermediate result, right-shifted by \( spos \), rounded as specified, left-shifted by \( dpos \), and placed in \( 2 \cdot esize \)-bit elements in the 128-bit complex result vector. The high 64 bits of \( x \) and \( y \) are ignored.

See Section 3.5.11 on page 302 for the control parameter, \( ctrl \), and extraction options. For this function, the \( x \) bit and the \( n \) bit in \( ctrl \) are both set to 1.
_emulx Multiply and Extract (Generic)

Multiplies elements of two vectors and extracts fields.

    hexlet_t _emulx(hexlet_t x, hexlet_t y, int ctrl)

This function takes two 128-bit vector parameters, x and y, and a scalar control parameter, ctrl. The vectors are interpreted as containing elements of esize-bit integers. The low 32 bits of ctrl specify control options, including esize, fsize, starting position (spos), and destination position (dpos) for the extraction.

The elements of x and y are multiplied, element by element, producing a vector of 2*esize-bit elements. Fields of fsize bits are extracted from this intermediate result in one of two modes:

- **Extract to Double-Size Result:** Extracts fields from 2*esize-bit elements in the intermediate products, and places them in 2*esize-bit elements in the result. This mode is selected by setting the x bit to 1 in ctrl. The high 64 bits of x and y are ignored.

- **Extract to Single-Size Result:** Extracts fields from 2*esize-bit elements in the intermediate products, and places them in esize-bit elements in the result. This mode is selected by clearing the x bit to 0 in ctrl.

In both modes, a field of fsize bits is extracted from each intermediate result. The fields are right-shifted by spos, rounded as specified, and left-shifted by dpos, producing a 128-bit result vector of elements of the specified size (esize or 2*esize). This operation can be either real or complex. For real mode the n bit in ctrl is set to 0. For complex mode the n bit in ctrl is set to 1.

See Section 3.5.11 on page 302 for the control parameter, ctrl, and extraction options.

The generic _emulx function encompasses all the capabilities of the _emulx, _emulxx, _emulxc, and _emulxxc functions which were described in the preceding pages. These four functions are typically incorporated for ease of use with specific parameters.
_emuladdx

Multiply, Add, and Extract

Multiplies elements of two vectors, adds elements of a third vector, and extracts fields.

\[
\begin{align*}
\text{v8}_t \quad &\text{emuladdx8(v8}_t \, x, \, v8}_t \, y, \, v8}_t \, z, \, \text{int} \, \text{ctrl}) \\
\text{v16}_t \quad &\text{emuladdx16(v16}_t \, x, \, v16}_t \, y, \, v16}_t \, z, \, \text{int} \, \text{ctrl}) \\
\text{v32}_t \quad &\text{emuladdx32(v32}_t \, x, \, v32}_t \, y, \, v32}_t \, z, \, \text{int} \, \text{ctrl}) \\
\text{v64}_t \quad &\text{emuladdx64(v64}_t \, x, \, v64}_t \, y, \, v64}_t \, z, \, \text{int} \, \text{ctrl}) \\
\text{v128}_t \quad &\text{emuladdx128(v128}_t \, x, \, v128}_t \, y, \, v128}_t \, z, \, \text{int} \, \text{ctrl}) \\
\end{align*}
\]

This function takes three 128-bit vector parameters, \(x\), \(y\), and \(z\), and a scalar control parameter, \(\text{ctrl}\). The vectors are interpreted as containing elements of \(\text{esize}\)-bit integers. The low 32 bits of \(\text{ctrl}\) specify control options, including \(\text{esize}\), \(\text{fsize}\), starting position \((spos)\), and destination position \((dpos)\) for the extraction.

The elements of \(y\) and \(z\) are multiplied, element by element, producing an intermediate vector of \(2*\text{esize}\)-bit elements. The corresponding elements of \(x\) are left-shifted by \(spos\) and added to this product. Fields of \(fsize\) bits are then extracted from this intermediate result, right-shifted by \(spos\), rounded as specified, left-shifted by \(dpos\), and placed in \(\text{esize}\)-bit elements in the 128-bit result vector.

See Section 3.5.11 on page 302 for the control parameter, \(\text{ctrl}\), and extraction options. For this function, the \(x\) bit and the \(n\) bit in \(\text{ctrl}\) are both set to 0.
_emuladdxx   Multiply, Add, and Extract/Expand

Multiplies elements of two vectors, adds elements of a third vector, and extracts fields.

\[
v16_t \_\text{emuladdxx}8(v16_t \: x, \: v8_t \: y, \: v8_t \: z, \: \text{int ctrl}) \\
v32_t \_\text{emuladdxx}16(v32_t \: x, \: v16_t \: y, \: v16_t \: z, \: \text{int ctrl}) \\
v64_t \_\text{emuladdxx}32(v64_t \: x, \: v32_t \: y, \: v32_t \: z, \: \text{int ctrl}) \\
v128_t \_\text{emuladdxx}64(v128_t \: x, \: v64_t \: y, \: v64_t \: z, \: \text{int ctrl})
\]

This function takes three 128-bit vector parameters, \(x\), \(y\), and \(z\), and a scalar control parameter, \(\text{ctrl}\). The vectors are interpreted as containing elements of \(\text{esize}\)-bit integers. The low 32 bits of \(\text{ctrl}\) specify control options, including \(\text{esize}\), \(\text{fsize}\), starting position \(\text{(spos)}\), and destination position \(\text{(dpos)}\) for the extraction.

The elements of \(y\) and \(z\) are multiplied, element by element, producing a vector of \(2\times\text{esize}\)-bit elements. The corresponding elements of \(x\) are left-shifted by \(\text{spos}\) and added to this product. Fields of \(\text{fsize}\) bits are then extracted from this intermediate result, right-shifted by \(\text{spos}\), rounded as specified, left-shifted by \(\text{dpos}\), and placed in \(2\times\text{esize}\)-bit elements in the 128-bit result vector. The high 64 bits of \(y\) and \(z\) are ignored.

See Section 3.5.11 on page 302 for the control parameter, \(\text{ctrl}\), and extraction options. For this function, the \(x\) bit in \(\text{ctrl}\) is set to 1 and the \(n\) bit in \(\text{ctrl}\) is set to 0.
Chapter 3: Vector and Matrix Functions

Ensemble Functions

Control Field with x bit = 1

\[ \begin{align*}
127 & \quad x & 0 \\
& \quad \text{esize} & \quad \text{element} \\
& \quad \text{esize} & \quad \text{element} \\
127 & \quad y & 0 \\
& \quad \text{esize} & \quad \text{element} \\
& \quad \text{esize} & \quad \text{element} \\
127 & \quad z & 0 \\
& \quad \text{esize} & \quad \text{element} \\
& \quad \text{esize} & \quad \text{element} \\
\end{align*} \]

\[ \begin{align*}
\text{ctrl} & \quad 127 & \quad 31 & \quad 0 \\
& \quad \text{element} & \quad \text{element} \\
\end{align*} \]

left-shift by spos

\[ << \quad << \]

left-shift by dpos

\[ \begin{align*}
\text{fsize} & \quad \text{fsize} \\
2^{\text{esize}} & \quad 2^{\text{esize}} \\
\end{align*} \]

right-shift by spos

\[ >> \quad >> \]

round as specified

\[ \begin{align*}
\text{r} & \quad 127 \\
& \quad 2^{\text{esize}} & \quad 2^{\text{esize}} \\
\end{align*} \]

Controller Field

\[ \begin{align*}
31 & \quad 24 & \quad 23 & \quad 16 & \quad 15 & \quad 14 & \quad 13 & \quad 12 & \quad 11 & \quad 10 & \quad 9 & \quad 8 & \quad 0 \\
\text{fsize} & \quad \text{dpos} & \quad x & \quad s & \quad n & \quad m & \quad \text{rnd} & \quad \text{gssp} \\
\end{align*} \]
_emuladdxc Multiply, Add, and Extract Complex

Multiplies elements of two complex-number vectors, adds elements of a third complex-number vector, and extracts fields.

vc8_t _emuladdxc8(vc8_t x, vc8_t y, vc8_t z, int ctrl)
vc16_t _emuladdxc16(vc16_t x, vc16_t y, vc16_t z, int ctrl)
vc32_t _emuladdxc32(vc32_t x, vc32_t y, vc32_t z, int ctrl)
vc64_t _emuladdxc64(vc64_t x, vc64_t y, vc64_t z, int ctrl)

This function takes three 128-bit vector parameters, \( x, y, \) and \( z, \) and a scalar control parameter, \( ctrl. \) The vectors are interpreted as containing elements of \( esize\)-bit complex integers. The low 32 bits of \( ctrl\) specify control options, including \( esize, fsize, \) starting position \( (spos), \) and destination position \( (dpos) \) for the extraction.

The elements of \( y \) and \( z \) are multiplied, element by element, producing an intermediate vector of \( 2\times esize\)-bit elements. The corresponding elements of \( x \) are left-shifted by \( spos \) and added to this product. Fields of \( fsize \) bits are then extracted from this intermediate result, right-shifted by \( spos \), rounded as specified, left-shifted by \( dpos \), and placed in \( esize\)-bit elements in the 128-bit complex result vector.

See Section 3.5.11 on page 302 for the control parameter, \( ctrl, \) and extraction options. For this function, the \( x \) bit in \( ctrl \) is set to 0 and the \( n \) bit in \( ctrl \) is set to 1.
_emuladdxxc  Multiply, Add, and Extract/Expand Complex

Multiplies elements of two complex-number vectors, adds elements of a third complex-number vector, and extracts fields.

\[
\begin{align*}
\text{vc16_t} & \quad \_\text{emuladdxxc8}(\text{vc16_t} \ x, \ \text{vc8_t} \ y, \ \text{vc8_t} \ z, \ \text{int} \ \text{ctrl}) \\
\text{vc32_t} & \quad \_\text{emuladdxxc16}(\text{vc32_t} \ x, \ \text{vc16_t} \ y, \ \text{vc16_t} \ z, \ \text{int} \ \text{ctrl}) \\
\text{vc64_t} & \quad \_\text{emuladdxxc32}(\text{vc64_t} \ x, \ \text{vc32_t} \ y, \ \text{vc32_t} \ z, \ \text{int} \ \text{ctrl}) \\
\text{vc128_t} & \quad \_\text{emuladdxxc64}(\text{vc128_t} \ x, \ \text{vc64_t} \ y, \ \text{vc64_t} \ z, \ \text{int} \ \text{ctrl})
\end{align*}
\]

This function takes three 128-bit vector parameters, \( x \), \( y \), and \( z \), and a scalar control parameter, \( \text{ctrl} \). The vectors are interpreted as containing elements of \( \text{esize} \)-bit complex integers. The low 32 bits of \( \text{ctrl} \) specify control options, including \( \text{esize} \), \( \text{FSIZE} \), starting position \( (\text{spos}) \), and destination position \( (\text{dpos}) \) for the extraction.

The elements of \( y \) and \( z \) are multiplied, element by element, producing a vector of \( 2^*\text{esize} \)-bit elements. The corresponding elements of \( x \) are left-shifted by \( \text{spos} \) and added to this product. Fields of \( \text{FSIZE} \) bits are then extracted from this intermediate result, right-shifted by \( \text{spos} \), rounded as specified, left-shifted by \( \text{dpos} \), and placed in \( 2^*\text{esize} \)-bit elements in the 128-bit complex result vector. The high 64 bits of \( y \) and \( z \) are ignored.

See Section 3.5.11 on page 302 for the control parameter, \( \text{ctrl} \), and extraction options. For this function, the \( x \) bit and the \( n \) bit in \( \text{ctrl} \) are both set to 1.
_emuladdx  Multiply, Add, and Extract (Generic)

Multiplies elements of two vectors, adds elements of a third vector, and extracts fields.

\[
\text{hexlet_t } \_\text{emuladdx}(\text{hexlet_t } x, \text{hexlet_t } y, \text{hexlet_t } z, \text{int } ctrl)
\]

This function takes three 128-bit vector parameters, \(x\), \(y\), and \(z\), and a scalar control parameter, \(ctrl\). The vectors are interpreted as containing elements of \(esize\)-bit integers. The low 32 bits of \(ctrl\) specify control options, including \(esize\), \(fsize\), starting position (\(spos\)), and destination position (\(dpos\)) for the extraction.

The elements of \(y\) and \(z\) are multiplied, element by element, producing a vector of \(2*esize\)-bit elements. The corresponding elements of \(x\) are left-shifted by \(spos\) and added to this product. Fields of \(fsize\) bits are extracted from this intermediate result in one of two modes:

- **Extract to Double-Size Result**: Extracts fields from \(2*esize\)-bit elements in the intermediate products, and places them in \(2*esize\)-bit elements in the result. This mode is selected by setting the \(x\) bit to 1 in \(ctrl\). The high 64 bits of \(x\), \(y\), and \(z\) are ignored.

- **Extract to Single-Size Result**: Extracts fields from \(2*esize\)-bit elements in the intermediate products, and places them in \(esize\)-bit elements in the result. This mode is selected by clearing the \(x\) bit to 0 in \(ctrl\).

In both modes, a field of \(fsize\) bits is extracted from each intermediate result. The fields are right-shifted by \(spos\), rounded as specified, and left-shifted by \(dpos\), producing a 128-bit result vector of elements of the specified size (\(esize\) or \(2*esize\)). This operation can be either real or complex. For real mode the \(n\) bit in \(ctrl\) is set to 0. For complex mode the \(n\) bit in \(ctrl\) is set to 1.

See Section 3.5.11 on page 302 for the control parameter, \(ctrl\), and extraction options.

The generic _emuladdx function encompasses all the capabilities of the _emuladdx, _emuladdxx, _emuladdxc, and _emuladdxxc functions which were described in the preceding pages. These four functions are typically incorporated for ease of use with specific parameters.
3.5.5 **Ensemble Multiply Sum (Dot Product)**

The Ensemble Multiply Sum (Dot Product) functions include:

- `_emulsum` Multiply and Sum
- `_emulsumc` Multiply and Sum Complex
- `_emulsumm` Multiply and Sum Mixed Sign
- `_emulsumu` Multiply and Sum Unsigned
- `_emulsumg` Multiply and Sum Galois Field
_emulsum Multiply and Sum

Multiplies two vectors and sums them in a dot product result.

\[
v128_t\ _{emulsum8}(v8_t\ x,\ v8_t\ y) \\
v128_t\ _{emulsum16}(v16_t\ x,\ v16_t\ y) \\
v128_t\ _{emulsum32}(v32_t\ x,\ v32_t\ y) \\
v128_t\ _{emulsum64}(v64_t\ x,\ v64_t\ y) \\
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\). The vectors are interpreted as containing elements of \(esize\)-bit signed integers. The elements of \(x\) and \(y\) are multiplied, element by element, producing a 256-bit vector of \(2^*\esize\)-bit elements. The products are summed, producing a 128-bit integer result.

\[
r = \sum_{i=0}^{\text{NELEM}-1} x[i] \times y[i]
\]
_emulsumc Multiply and Sum Complex

Multiplies elements of two complex-number vectors and sums them in a dot product result.

\[
\text{vc64}_t \ _\text{emulsumc8}(\text{vc8}_t \ x, \ \text{vc8}_t \ y) \\
\text{vc64}_t \ _\text{emulsumc16}(\text{vc16}_t \ x, \ \text{vc16}_t \ y) \\
\text{vc64}_t \ _\text{emulsumc32}(\text{vc32}_t \ x, \ \text{vc32}_t \ y)
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\). The vectors are interpreted as containing elements of \(2 \times \text{esize}\)-bit complex integers, in which the real and imaginary elements are each \(\text{esize}\) bits wide. The elements of \(x\) and \(y\) are complex-multiplied, element by element, producing a 256-bit vector of \(2 \times \text{esize}\)-bit elements. The products are summed into a 128-bit complex integer, in which the real and imaginary elements are each 64 bits wide.

\[
r = y[i] \ast z[i], \quad i = 0..\text{NELEM} - 1
\]
_emulsumm 

Multiply and Sum Mixed Sign

Multiplies two vectors and sums them in a dot product result.

\[
v128_t \_emulsumm8(vu8_t \ x, v8_t \ y) \\
v128_t \_emulsumm16(vu16_t \ x, v16_t \ y) \\
v128_t \_emulsumm32(vu32_t \ x, v32_t \ y) \\
v128_t \_emulsumm64(vu64_t \ x, v64_t \ y)
\]

This function takes two 128-bit vector parameters, \(x\) (unsigned) and \(y\) (signed). Vector \(x\) is interpreted as containing elements of \(esize\)-bit unsigned integers, and vector \(y\) is interpreted as containing elements of \(esize\)-bit signed integers. The elements of \(x\) and \(y\) are multiplied, element by element, producing a 256-bit vector of \(2\times esize\)-bit elements. The products are summed, producing a 128-bit integer result.

\[
r = \sum_{i=0}^{\text{NELEM}-1} x[i] \times y[i]
\]
_emulsumu  Multiply and Sum Unsigned

Multiplies two vectors and sums them in a dot product result.

\[
\begin{align*}
&v128_t \_emulsumu8(vu8_t \ x, vu8_t \ y) \\
v128_t \_emulsumu16(vu16_t \ x, vu16_t \ y) \\
v128_t \_emulsumu32(vu32_t \ x, vu32_t \ y) \\
v128_t \_emulsumu64(vu64_t \ x, vu64_t \ y)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing elements of \( \text{esize} \)-bit unsigned integers. The elements of \( x \) and \( y \) are multiplied, element by element, producing a 256-bit vector of \( 2^\text{esize} \)-bit elements. The products are summed, producing a 128-bit integer result.

\[
r = \sum_{i=0}^{\text{NELEM}-1} x[i] \times y[i]
\]
_emulsumg Multiply and Sum Galois Field

Multiplies two vectors and sums them in a dot product result.

\[
\text{vu8_t } \text{emulsumg8}(\text{vu8_t } x, \text{vu8_t } y, \text{int } z)
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\), and a basis polynomial \(z\). The \(x\) and \(y\) vectors are interpreted as containing elements of 8 bits, representing 7th degree polynomials with binary coefficients. The integer \(z\) is interpreted as the basis polynomial, with an implied 1 bit in bit-position 8. Corresponding elements of \(x\) and \(y\) are polynomial-multiplied (see page 211 for a description of polynomial multiplication), producing an intermediate product of double-size elements. The products are x-or’ed and the result is then taken modulo the basis polynomial (with its implied 1 bit), producing a 8-bit scalar in the 128-bit result vector representing a 7th degree polynomial. Higher order bits (8 and above) are zeroed. The operation never overflows.

The basis polynomial is like a prime number, in that it cannot be factored into other polynomials. It is typically chosen so that the set of binary polynomials, together with this modulo-product operation, form a group-theoretic field known as a Galois field. The operation is used in error-correcting codes.

\[
r = \sum_{i=0}^{\text{NELEM } - 1} x[i] \cdot y[i]
\]
3.5.6 **Ensemble Divide**

The Ensemble Divide functions include:

- `_ediv` Divide
- `_edivf` Divide Floating-Point
- `_edivu` Divide Unsigned
Divides elements of two vectors.

\[ \text{v64_t \_ediv64 (v64_t \ x, \ v64_t \ y)} \]

This function takes two 128-bit parameters, \(x\) and \(y\). It interprets the low 64 bits of each parameter as a 64-bit signed integer, and divides \(x\) by \(y\), producing a 128-bit result with the quotient in the low 64 bits and the remainder in the high 64 bits.

\[ r[0] = x / y \]

\[ r[1] = x \% y \]
Divides elements of two vectors.

```c
vf16_t _edivf16(c, f, n, z, x)(vf16_t x, vf16_t y)
vf32_t _edivf32(c, f, n, z, x)(vf32_t x, vf32_t y)
vf64_t _edivf64(c, f, n, z, x)(vf64_t x, vf64_t y)
vf128_t _edivf128(c, f, n, z, x)(vf128_t x, vf128_t y)
```

This function takes two 128-bit vector parameters, \( \mathbf{x} \) and \( \mathbf{y} \). The vectors are interpreted as containing elements of \( \text{esize} \)-bit floating-point numbers. Each element of \( \mathbf{x} \) is divided by the corresponding element of \( \mathbf{y} \). The results are rounded, using the specified rounding mode, or round-to-nearest if not specified, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = x[i] / y[i], \quad i = 0..\text{NELEM} -1
\]
Divides elements of two vectors.

\[
vu64_t \text{ _edivu64}(vu64_t \ x, vu64_t \ y)
\]

This function takes two 128-bit parameters, \(x\) and \(y\). It interprets the low 64 bits of each parameter as a 64-bit unsigned integer, and divides \(x\) by \(y\), producing a 128-bit result with the quotient in the low 64 bits and the remainder in the high 64 bits.

\[
r[0] = x /^u y
\]

\[
r[1] = x \% y
\]
3.5.7 **Ensemble Scale Add**

The Ensemble Scale Add functions include:

- `_escaladdf`  Scale and Add Floating-Point
- `_escaladdx`  Scale, Add, and Extract
_escaladdf  
Scale and Add Floating-Point

Multiplies elements of two vectors, each vector using a different multiplier, and adds the products.

\[
\text{vf16_t}_\text{escaladdf16(vf16_t x, vf16_t y, vf16_t z)} \\
\text{vf32_t}_\text{escaladdf32(vf32_t x, vf32_t y, vf32_t z)} \\
\text{vf64_t}_\text{escaladdf64(vf64_t x, vf64_t y, vf64_t z)}
\]

This function takes three 128-bit vector parameters, \(x\), \(y\), and \(z\). The vectors are interpreted as containing elements of \(esize\)-bit floating-point numbers. Elements of \(x\) are multiplied by the least-significant element of \(z\), forming a first vector product. Elements of \(y\) are multiplied by the next-least-significant element of \(z\), forming a second vector product. The two vector products are summed, element by element, and rounded to the nearest, producing a 128-bit result vector of \(esize\)-bit elements. Any exceptions are handled by the default IEEE-754 rules as described in Section 2.4.2 on page 44.

\[
r[i] = (x[i] * z[0]) + (y[i] * z[1]), \quad i = 0..\text{NELEM -1}
\]
Multplies elements of two vectors, each vector using a different multiplier, adds the products, and extracts fields.

```c
hexlet_t _escaladdx8(hexlet_t x, hexlet_t y, hexlet_t ctrl)
```

This function takes two 128-bit vector parameters, x and y, and a 128-bit control parameter, ctrl. The vectors are interpreted as containing elements of `esize`-bit integers. The low 32 bits of ctrl specify control options, including `esize`, `fsize`, starting position (spos), and destination position (dpos) for the extraction. The high 64 bits of ctrl specify two scalar multipliers, s0 and s1.

Elements of x are multiplied by s0, which consists of bits 64+`esize`-1..64 of ctrl, producing a first vector of 2*`esize`-bit elements. Elements of y are multiplied by s1, which consists of bits 64+2*`esize`-1..64+`esize` of ctrl, producing a second vector of 2*`esize`-bit elements. The vectors are summed, element by element. Fields of `fsize` bits are extracted from this intermediate result in one of two modes:

- **Extract to Double-Size Result**: Extracts fields from 2*`esize`-bit elements in the intermediate products, and places them in 2*`esize`-bit elements in the result. This mode is selected by setting the x bit to 1 in ctrl. The high 64 bits of x and y are ignored.
- **Extract to Single-Size Result**: Extracts fields from 2*`esize`-bit elements in the intermediate products, and places them in `esize`-bit elements in the result. This mode is selected by clearing the x bit to 0 in ctrl.

In both modes, a field of `fsize` bits is extracted from each intermediate result. The fields are right-shifted by spos, rounded as specified, and left-shifted by dpos, producing a 128-bit result vector of `esize`-bit elements.

See Section 3.5.11 on page 302 for the control parameter, ctrl, and extraction options.
Chapter 3: Vector and Matrix Functions

Ensemble Functions

Control Field with x bit = 1
\[ r[i] = x[i] \times \text{scalx} + y[i] \times \text{scaly}, \quad i = 0..\text{NELEM}-1 \]
3.5.8 **Ensemble Convolve**

The Ensemble Convolve functions include:

- `_econ` Convolve
- `_econu` Convolve Unsigned
- `_econm` Convolve Mixed Sign
- `_econc` Convolve Complex
- `_econf` Convolve Floating-Point
- `_econcf` Convolve Complex Floating-Point
- `_econx` Convolve and Extract
- `_econxi` Convolve and Extract Immediate
- `_econxic` Convolve and Extract Immediate Complex
**_econ Convolve**

Convolves elements of two vectors.

\[
v_{16\_t} \_econ8(v_{8\_t} x, v_{8\_t} y) \\
v_{32\_t} \_econ16(v_{16\_t} x, v_{16\_t} y) \\
v_{64\_t} \_econ32(v_{32\_t} x, v_{32\_t} y) \\
v_{128\_t} \_econ64(v_{64\_t} x, v_{64\_t} y)
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing elements of `esize`-bit signed integers. Typically, the first vector represents time-domain sample data and the second vector represents filter coefficients. The vectors are convolved as shown below, resulting in a 128-bit vector of \(2^{\text{esize}}\)-bit elements.

\[
\sum_{j = 0}^{\text{NELEM}/2 - 1} x[i - j + \text{offset}] \cdot y[j], \quad i = 0..\text{NELEM}/2 - 1
\]

where: \( \text{offset} = \text{NELEM}/2 \)
**_econu**  

Convolves elements of two vectors.

\[
vu16_t \ _econu8(vu8_t \ x, \ vu8_t \ y)\n\]
\[
vu32_t \ _econu16(vu16_t \ x, \ vu16_t \ y)\n\]
\[
vu64_t \ _econu32(vu32_t \ x, \ vu32_t \ y)\n\]
\[
vu128_t \ _econu64(vu64_t \ x, \ vu64_t \ y)\n\]

This function takes two 128-bit vector parameters, \(x\) and \(y\). The vectors are interpreted as containing elements of \(esize\)-bit unsigned integers. Typically, the first vector represents time-domain sample data and the second vector represents filter coefficients. The vectors are convolved as shown below, resulting in a 128-bit vector of \(2^{esize}\)-bit elements.

\[
n_{i} = \sum_{j = 0}^{\text{NELEM/2} - 1} x_{i - j + \text{offset}} \cdot y_{j}, \quad i = 0..\text{NELEM/2} - 1
\]

where: \(\text{offset} = \text{NELEM/2}\)
_econm

Convolve Mixed Sign

Convolves elements of two vectors.

\[
v16_t \_econm8(vu8_t \ x, v8_t \ y) \\
v32_t \_econm16(vu16_t \ x, v16_t \ y) \\
v64_t \_econm32(vu32_t \ x, v32_t \ y) \\
v128_t \_econm64(vu64_t \ x, v64_t \ y)
\]

This function takes two 128-bit vector parameters, \(\mathbf{x}\) (unsigned) and \(\mathbf{y}\) (signed). Vector \(\mathbf{x}\) is interpreted as containing elements of \(esize\)-bit unsigned integers, and vector \(\mathbf{y}\) is interpreted as containing elements of \(esize\)-bit signed integers. Typically, the first vector represents time-domain sample data and the second vector represents filter coefficients. The vectors are convolved as shown below, resulting in a 128-bit vector of \(2^*esize\)-bit elements.

\[
r[i] = \sum_{j = 0}^{NELEM/2 - 1} x[i - j + offset] * m y[j], \quad i = 0..64/esize - 1
\]

where: \(\text{offset} = 64/esize\)
_econc Convolve Complex

Convolves elements of two complex-number vectors.

vc16_t _econc8 (vc8_t x, vc8_t y)
vc32_t _econc16 (vc16_t x, vc16_t y)
vc64_t _econc32 (vc32_t x, vc32_t y)

This function takes two 128-bit vector parameters, \( x \) and \( y \). The vectors are interpreted as containing sets of \( 2 \times \text{esize} \)-bit complex integers, in which the real and imaginary elements are each \( \text{esize} \) bits wide. Typically, the first vector represents time-domain sample data and the second vector represents filter coefficients. The vectors are convolved as shown below, producing a 128-bit result vector of \( 4 \times \text{esize} \)-bit complex integers, in which the real and imaginary elements are each \( 2 \times \text{esize} \) bits wide.

The next figure shows a complex convolve example that uses a smaller \( \text{esize} \).
\[ r[i] = \sum_{j=0}^{\text{NELEM}/2 - 1} x[i - j + \text{offset}] * y[j], \quad i = 0..\text{NELEM} - 1 \]

where: \( \text{offset} = 64/\text{esize} \)
_econf

Convolve Floating-Point

Convolves elements of two vectors.

\[
\begin{align*}
\text{vf16}_t & \quad \text{_econf16}(\text{vf16}_t \ xlo, \text{vf16}_t \ xhi, \text{vf16}_t \ y) \\
\text{vf32}_t & \quad \text{_econf32}(\text{vf32}_t \ xlo, \text{vf32}_t \ xhi, \text{vf32}_t \ y) \\
\text{vf64}_t & \quad \text{_econf64}(\text{vf64}_t \ xlo, \text{vf64}_t \ xhi, \text{vf64}_t \ y)
\end{align*}
\]

This function takes three 128-bit vector parameters, \(xlo\), \(xhi\), and \(y\). The vectors are interpreted as containing elements of \(esize\)-bit floating-point numbers. Vectors \(xlo\) and \(xhi\) are concatenated, \(xlo\) low and \(xhi\) high, and used as the first operand, representing time-domain sample data. Vector \(y\) is used as the second operand, representing filter coefficients. The vectors are convolved as shown below, producing a 128-bit result vector of \(esize\)-bit elements.
\[ r[i] = \sum_{j=0}^{\text{NELEM} - 1} x[i - j + \text{offset}] \times y[j], \quad i = 0..\text{NELEM} - 1 \]

where: \( x = xhi || xlo \), and offset = \( \text{NELEM}/2 \)
**_econcf**  Convolve Complex Floating-Point

Convolves vectors of complex floating-point numbers.

```c
vcf16_t _econcf16(vcf16_t xlo, vcf16_t xhi, vcf16_t y)
vcf32_t _econcf32(vcf32_t xlo, vcf32_t xhi, vcf32_t y)
```

This function takes three 128-bit vector parameters, `xlo`, `xhi`, and `y`. The vectors are interpreted as containing sets of $2 \times \text{esize}$-bit complex floating-point numbers, in which the real and imaginary elements are each `esize` bits wide. The vectors are concatenated, `xlo` low and `xhi` high and convolved with `y` as shown below. Each element is rounded to nearest, producing a 128-bit result vector of $2 \times \text{esize}$-bit elements.

The next figure shows a complex floating-point convolve example that uses a smaller `esize`. 

![](436-022.png)
\[
\sum_{j=0}^{\text{NELEM}-1} x[i - j + \text{offset}] \cdot y[j], \quad i = 0..\text{NELEM}-1
\]

where: \( x = \text{xhi} || \text{xlo} \), and \( \text{offset} = \text{NELEM}/2 \)
_econx

Convolve and Extract

Convolves elements of vectors and extracts fields.

\[
\begin{align*}
\text{v8_t } & \_\text{econx8}(\text{v8_t } xlo, \text{v8_t } xhi, \text{v8_t_t } y, \text{int } ctrl) \\
\text{v16_t } & \_\text{econx16}(\text{v16_t } xlo, \text{v16_t } xhi, \text{v16_t_t } y, \text{int } ctrl) \\
\text{v32_t } & \_\text{econx32}(\text{v32_t } xlo, \text{v32_t } xhi, \text{v32_t_t } y, \text{int } ctrl) \\
\text{v64_t } & \_\text{econx64}(\text{v64_t } xlo, \text{v64_t } xhi, \text{v64_t_t } y, \text{int } ctrl)
\end{align*}
\]

This function takes three 128-bit vector parameters, xlo, xhi, and y, and a scalar control parameter, ctrl. Vectors xlo, xhi, and y are interpreted as containing elements of esize-bit integers, which themselves contain fields of fsize bits. The low 32 bits of ctrl specify control options, including esize, fsize, starting position (spos), and destination position (dpos) for the extraction.

Vectors xlo and xhi are concatenated, xlo low and xhi high, and used as the first operand, representing time-domain sample data. Vector y is used as the second operand, representing coefficients. The vectors are convolved as shown below, producing a vector of 2*esize-bit elements. Fields of fsize bits are extracted from this intermediate result in one of two modes:

- **Extract to Double-Size Result**: Extracts fields from 2*esize-bit elements in the intermediate products, and places them in 2*esize-bit elements in the result. This mode is selected by setting the x bit to 1 in ctrl. The high 64 bits of xhi are ignored.

- **Extract to Single-Size Result**: Extracts fields from 2*esize-bit elements in the intermediate products, and places them in esize-bit elements in the result. This mode is selected by clearing the x bit to 0 in ctrl.

In both modes, a field of fsize bits is extracted from each intermediate result. The fields are right-shifted by spos, rounded as specified, and left-shifted by dpos, producing a 128-bit result vector of esize-bit elements.

See Section 3.5.11 on page 302 for the control parameter, ctrl, and extraction options.
Chapter 3: Vector and Matrix Functions

Ensemble Functions

$\mathbf{r[i]} = \sum_{j=0}^{\text{NELEM}-1} x[i - j + \text{NELEM}] \times y[j], \quad i = 0..\text{NELEM}-1$

$r[i] = \times[i - j + \text{NELEM}] \times y[j] \sum_{j=0}^{\text{NELEM}-1}$
_econxi  Convolve and Extract Immediate

Convolves elements of vectors and extracts fields.

\[
\begin{align*}
&v8_t \ _\text{econxi8n}(v8_t \ xlo, \ v8_t \ xhi, \ v8_t \ y, \ int \ sh) \\
v16_t \ _\text{econxi16n}(v16_t \ xlo, \ v16_t \ xhi, \ v16_t \ y, \ int \ sh) \\
v32_t \ _\text{econxi32n}(v32_t \ xlo, \ v32_t \ xhi, \ v32_t \ y, \ int \ sh) \\
v64_t \ _\text{econxi64n}(v64_t \ xlo, \ v64_t \ xhi, \ v64_t \ y, \ int \ sh)
\end{align*}
\]

This function takes three 128-bit vector parameters, \texttt{xlo}, \texttt{xhi}, and \texttt{y}, and an immediate scalar shift amount, \texttt{sh}. The vectors are interpreted as containing elements of \texttt{esize}-bit signed integers.

Vectors \texttt{xlo} and \texttt{xhi} are concatenated, \texttt{xlo} low and \texttt{xhi} high, and used as the first operand, representing time-domain sample data. Vector \texttt{y} is used as the second operand, representing coefficients. The vectors are convolved as shown below, producing a 128-bit result vector of \(2^{\text{esize}}\)-bit elements. Each element is right-shifted by \texttt{sh}, with rounding to nearest. The low \texttt{esize} bits are taken as the result, producing a 128-bit result vector of \texttt{esize}-bit elements. If a result exceeds the range representable in an \texttt{esize}-bit signed integer, it is limited.

\texttt{econxi} is equivalent to \texttt{econx} with \texttt{spos} = \texttt{sh}, \texttt{fsize} = \texttt{esize}, and \texttt{dpos} = 0.
$r[i] = \sum_{j=0}^{\text{NELEM} - 1} x[i - j + \text{NELEM}] \times y[j] \gg \text{sh}$, round to nearest, $i = 0..\text{NELEM} - 1$
_econxic  Convolve and Extract Immediate Complex

Convolves elements of complex-number vectors and extracts fields.

\[
\begin{align*}
\text{vc8\_t} & \ _\text{econxic8n}(\text{vc8\_t} \ xlo, \ \text{vc8\_t} \ xhi, \ \text{vc8\_t} \ y, \ \text{int} \ sh) \\
\text{vc16\_t} & \ _\text{econxic16n}(\text{vc16\_t} \ xlo, \ \text{vc16\_t} \ xhi, \ \text{vc16\_t} \ y, \ \text{int} \ sh) \\
\text{vc32\_t} & \ _\text{econxic32n}(\text{vc32\_t} \ xlo, \ \text{vc32\_t} \ xhi, \ \text{vc32\_t} \ y, \ \text{int} \ sh)
\end{align*}
\]

This function takes three 128-bit vector parameters, \text{xlo}, \text{xhi}, and \text{y}, and an immediate scalar shift amount, \text{sh}. The vectors are interpreted as containing elements of \text{esize}-bit complex integers.

Vectors \text{xlo} and \text{xhi} are concatenated, \text{xlo} low and \text{xhi} high, and used as the first operand, representing time-domain sample data. Vector \text{y} is used as the second operand, representing coefficients. The vectors are convolved as shown below, producing a 128-bit result vector of 2*\text{esize}-bit elements. Each element is right-shifted by \text{sh}, with rounding to nearest. The low \text{esize} bits are taken as the result, producing a 128-bit result vector of \text{esize}-bit elements. If a result exceeds the range representable in an \text{esize}-bit complex integer, it is limited.

\text{econxic} is equivalent to complex-number version of \text{econx} with \text{ spos = sh}, \text{ fsize = esize}, and \text{dpos = 0}. 
Chapter 3: Vector and Matrix Functions

Ensemble Functions

r[i] = Σ x[i - j + NELEM] * y[j] >> sh, round to nearest, i = 0..NELEM - 1
3.5.9 **Ensemble Conversion and Scaling**

The Ensemble Conversion and Scaling functions include:

- `_edeflatef_` Deflate Floating-Point
- `_einflatef_` Inflate Floating-Point
- `_efloatf_` Convert Integer to Floating-Point
- `_esinkf_` Convert Floating-Point to Integer
- `_eextracti_` Extract Immediate
- `_eextractiu_` Extract Immediate Unsigned
- `_eextractx_` Extract from Single-Size Source
- `_eextract_` Extract from Double-Size Source
- `_eextractm_` Extract and Merge
- `_eextract_` Extract (Generic)
_edeflatef

Deflate Floating-Point

Converts vector elements to the next-lower-precision format.

\[
\begin{align*}
\text{vf16}_t \quad & \_\text{edeflatef32}\{, c, f, n, z, x\}(\text{vf32}_t \ x) \\
\text{vf32}_t \quad & \_\text{edeflatef64}\{, c, f, n, z, x\}(\text{vf64}_t \ x) \\
\text{vf64}_t \quad & \_\text{edeflatef128}\{, c, f, n, z, x\}(\text{vf128}_t \ x)
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), containing elements of \( \text{esize} \)-bit floating-point numbers. Each element is converted to the next-lower precision floating-point format, from \( \text{esize} \) to \( \text{esize}/2 \). The results are rounded, using the specified rounding mode, or round-to-nearest if not specified. The half-size elements are compressed into the low 64 bits of the result. The high 64 bits of the result are cleared to zero.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
\begin{align*}
\text{r}[i] &= \text{x}[i], \quad i = 0..\text{NELEM} - 1 \\
\text{r}[i] &= 0, \quad i = \text{NELEM}..2*\text{NELEM} - 1
\end{align*}
\]
Inflate Floating-Point

Converts vector elements to the next-higher-precision format.

\[
\begin{align*}
\text{vf32}_t & \quad \_\text{einflatef16}(c, f, n, z, x)(\text{vf16}_t \ x) \\
\text{vf64}_t & \quad \_\text{einflatef32}(c, f, n, z, x)(\text{vf32}_t \ x) \\
\text{vf128}_t & \quad \_\text{einflatef64}(c, f, n, z, x)(\text{vf64}_t \ x)
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), containing elements of esize-bit floating-point numbers. Each element in the low 64 bits of \( x \) is converted to the next-higher precision floating-point format, producing a 128-bit result vector of 2*esize-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = x[i], \quad i = 0..\text{NELEM}/2 -1
\]
Converts vector elements from integer format to floating-point format.

\[
\begin{align*}
\text{vf16_t} & \quad \_efloatf16\{,c,f,n,z,x\}(v16\_t \ x) \\
\text{vf32_t} & \quad \_efloatf32\{,c,f,n,z,x\}(v32\_t \ x) \\
\text{vf64_t} & \quad \_efloatf64\{,c,f,n,z,x\}(v64\_t \ x) \\
\text{vf128_t} & \quad \_efloatf128\{,c,f,n,z,x\}(v128\_t \ x)
\end{align*}
\]

This function takes a 128-bit vector parameter, \( \mathbf{x} \), containing elements of \( \text{esize}\)-bit signed integers. Each element is converted from its integer format to an \( \text{esize}\)-bit floating-point format. The results are rounded using the specified rounding mode, or round-to-nearest if not specified, producing a 128-bit result vector of \( \text{esize}\)-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = x[i]
\]
**_esinkf**  

Converts vector elements from floating-point format to integer format.

\[
\begin{align*}
\text{v16}_t & \text{ _esinkf16}((c,cd,f,fd,n,z,zd,x) (vf16_t x)) \\
\text{v32}_t & \text{ _esinkf32}((c,cd,f,fd,n,z,zd,x) (vf32_t x)) \\
\text{v64}_t & \text{ _esinkf64}((c,cd,f,fd,n,z,zd,x) (vf64_t x)) \\
\text{v128}_t & \text{ _esinkf128}((c,cd,f,fd,n,z,zd,x) (vf128_t x))
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), containing elements of \( \text{esize} \)-bit floating-point numbers. Each element is converted from its floating-point format to an \( \text{esize} \)-bit integer format. The results are rounded using the specified rounding mode, or round-to-nearest if not specified, producing a 128-bit result vector of \( \text{esize} \)-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = x[i], \text{ rounding as specified}
\]
_eextracti  

Extracts fields in elements of a pair of vectors.

\[
v_8\_t\ _{\text{eextracti8n}}(v_{16}\_t\ xlo, v_{16}\_t\ xhi, \text{int} \ sh) \\
v_{16}\_t\ _{\text{eextracti16n}}(v_{32}\_t\ xlo, v_{32}\_t\ xhi, \text{int} \ sh) \\
v_{32}\_t\ _{\text{eextracti32n}}(v_{64}\_t\ xlo, v_{64}\_t\ xhi, \text{int} \ sh) \\
v_{64}\_t\ _{\text{eextracti64n}}(v_{128}\_t\ xlo, v_{128}\_t\ xhi, \text{int} \ sh)
\]

This function takes two 128-bit vector parameters, \(xlo\) and \(xhi\), and a scalar shift-amount parameter, \(sh\). The vectors are concatenated, \(xlo\) low and \(xhi\) high, and interpreted as containing elements of \(2^{esize}\)-bit signed integers. Each element is shifted right by the amount \(sh\), with rounding to nearest. The low \(esize\) bits are taken as the result, producing a 128-bit result vector of \(esize\)-bit elements. If a result exceeds the range representable in an \(esize\)-bit signed integer, it is limited.

\_eextracti is equivalent to _eextract with \(spos = sh\), \(fsize = esize\), and \(dpos = 0\).

\[
r[i] = x[i] >> sh, \text{ round to nearest}
\]

where: \(x = xhi \| xlo\)
Extract Immediate Unsigned

Extracts fields in elements of two vectors, and deposits them in elements of another vector.

\[
\begin{align*}
\text{vu8_t} & \quad \text{eextractiu8(vu16_t xlo, vu16_t xhi, int sh)} \\
\text{vu16_t} & \quad \text{eextractiu16(vu32_t xlo, vu32_t xhi, int sh)} \\
\text{vu32_t} & \quad \text{eextractiu32(vu64_t xlo, vu64_t xhi, int sh)} \\
\text{vu64_t} & \quad \text{eextractiu64(vu128_t xlo, vu128_t xhi, int sh)}
\end{align*}
\]

This function takes two 128-bit vector parameters, \(xlo\) and \(xhi\), and a scalar shift-amount parameter, \(sh\). The vectors are concatenated, \(xlo\) low and \(xhi\) high, and interpreted as containing elements of \(2^{*}\text{esize}\)-bit unsigned integers. Each element is shifted right by the amount \(sh\), with rounding to nearest. The low \(\text{esize}\) bits are taken as the result, producing a 128-bit result vector of \(\text{esize}\)-bit elements. If a result exceeds the range representable in an \(\text{esize}\)-bit signed integer, it is limited.

\text{eextracti} is equivalent to \text{eextract} with \(spos = sh\), \(fsize = esize\), and \(dpos = 0\).

\[r[i] = x[i] \gg u sh\], round to nearest

where: \(x = xhi || xlo\)
_eextractx Extract from Single-Size Source

Extracts fields in elements of two vectors, and deposits them in elements of another vector.

\[ \begin{align*}
&v_{8\_t} \_eextractx8(v_{8\_t} x, \text{int ctrl}) \\
v_{16\_t} \_eextractx16(v_{16\_t} x, \text{int ctrl}) \\
v_{32\_t} \_eextractx32(v_{32\_t} x, \text{int ctrl}) \\
v_{64\_t} \_eextractx64(v_{64\_t} x, \text{int ctrl}) \\
v_{128\_t} \_eextractx128(v_{128\_t} x, \text{int ctrl})
\end{align*} \]

This function takes a 128-bit vector parameter, \( x \), and a scalar control parameter, \( ctrl \). The vector is interpreted as containing elements of \( esize \)-bit integers. The low 32 bits of \( ctrl \) specify control options, including \( esize, fsize, \) starting position (\( spos \)), and destination position (\( dpos \)) for the extraction. The \( x \) bit of \( ctrl \) must be set to 1 and the \( m \) bit of \( ctrl \) must be set to 0. See Section 3.5.11 on page 302 for the control parameter, \( ctrl \), and extraction options.

A field of \( fsize \) bits is extracted from each source element. The fields are right-shifted by \( spos \), rounded as specified, and left-shifted by \( dpos \), producing a 128-bit result vector of \( esize \)-bit elements.

The _eextractx function is very similar to the _xextractx function on page 189, except that _eextractx results can be rounded or limited.
Extracts fields in elements of two vectors, and deposits them in elements of another vector.

\[
\begin{align*}
v8_t \_eextract8(v16_t x, v16_t y, int ctrl) \\
v16_t \_eextract16(v32_t x, v32_t y, int ctrl) \\
v32_t \_eextract32(v64_t x, v64_t y, int ctrl) \\
v64_t \_eextract64(v128_t x, v128_t y, int ctrl) \\
v128_t \_eextract128(v128_t x, v128_t y, int ctrl)
\end{align*}
\]

This function takes two 128-bit vector parameters, \( x \) and \( y \), and a scalar control parameter, \( ctrl \). The vectors are interpreted as containing elements of \( 2 \times esize \)-bit integers. The low 32 bits of \( ctrl \) specify control options, including \( esize \), \( fsize \), starting position (\( spos \)), and destination position (\( dpos \)) for the extraction. The \( x \) bit of \( ctrl \) must be set to 0 and the \( m \) bit of \( ctrl \) must be set to 0. See Section 3.5.11 on page 302 for the control parameter, \( ctrl \), and extraction options.

A field of \( fsize \) bits is extracted from each source element. The fields are right-shifted by \( spos \), rounded as specified, and left-shifted by \( dpos \), producing a 128-bit result vector of \( esize \)-bit elements.

The \( \_eextract \) function is very similar to the \( \_xextract \) function on page 189, except that \( \_eextract \) results can be rounded or limited.
_eextractm  Extract and Merge

Extracts fields in elements of two vectors, and deposits and merges them in elements of another vector.

\[
\begin{align*}
_v8_t \ _eextractm8 & (v8_t \ x, v8_t \ y, \ int \ ctrl) \\
v16_t \ _eextractm16 & (v16_t \ x, v16_t \ y, \ int \ ctrl) \\
v32_t \ _eextractm32 & (v32_t \ x, v32_t \ y, \ int \ ctrl) \\
v64_t \ _eextractm64 & (v64_t \ x, v64_t \ y, \ int \ ctrl) \\
v128_t \ _eextractm128 & (v128_t \ x, v128_t \ y, \ int \ ctrl)
\end{align*}
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\), and a scalar control parameter, \(ctrl\). The vectors are interpreted as containing elements of \(esize\)-bit integers. The low 32 bits of \(ctrl\) specify control options, including \(esize\), \(fsize\), starting position \((spos)\), and destination position \((dpos)\) for the extraction. The \(m\) bit of \(ctrl\) must be set to 1. See Section 3.5.11 on page 302 for the control parameter, \(ctrl\), and extraction options.

A field of \(fsize\) bits is extracted from each source element. The fields are right-shifted by \(spos\), rounded as specified, left-shifted by \(dpos\), and merged with the bit positions from vector \(y\) that are outside the extracted fields from vector \(x\), producing a 128-bit result vector of \(esize\)-bit elements.

The _eextractm function is very similar to the _xextractm function on page 191, except that _eextractm results can be rounded or limited.
Chapter 3: Vector and Matrix Functions

Ensemble Functions

**Control Field**

with m bit = 1
and x bit ignored

```
<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsize</td>
<td>dpos</td>
<td>x</td>
<td>s</td>
<td>n</td>
<td>m</td>
<td>l</td>
<td>md</td>
<td>gsp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
_eextract

Extracts fields in elements of two vectors, and deposits (and optionally merges) them in elements of another vector.

\[
\text{hexlet_t } _eextract(\text{hexlet_t } x, \text{hexlet_t } y, \text{int } ctrl)
\]

This function takes two 128-bit vector parameters, \(x\) and \(y\), and a scalar control parameter, \(ctrl\). The vectors are interpreted as containing elements of \(esize\)-bit or \(2^{esize}\)-bit integers. The low 32 bits of \(ctrl\) specify control options, including \(esize\), \(fsize\), starting position \((spos)\), and destination position \((dpos)\) for the extraction. All three parameters are required, even though fewer may actually be used depending on the control mode selected. For modes requiring a single vector input, that input is provided via argument \(x\), while argument \(y\) is ignored.

The following extraction modes are available:

- **Extract from Single-Size Source**: Extracts fields from \(esize\)-bit elements in vector \(x\), and places them in \(esize\)-bit elements in the result. Vector \(y\) is not used, although its reference must be present in the function syntax. This mode is selected by setting the \(x\) bit to 1 in \(ctrl\).

- **Extract from Double-Size Source**: Extracts fields from \(2^{esize}\)-bit elements in the concatenated vectors \(x\) and \(y\), and places them in \(esize\)-bit elements in the result. This mode is selected by clearing the \(x\) bit to 0 in \(ctrl\).

- **Extract-Merge**—Extracts fields from \(esize\)-bit elements in vector \(x\), merges them with the bit positions from vector \(y\) that are outside the extracted fields from vector \(x\), and places them into \(esize\)-bit elements in the result. This mode is selected by setting the \(m\) bit to 1 in \(ctrl\), in which case the \(x\) bit is ignored.

In all modes, a field of \(fsize\) bits is extracted from each source element. The fields are right-shifted by \(spos\), rounded as specified, and left-shifted by \(dpos\), producing a 128-bit result vector of \(esize\)-bit elements.

See Section 3.5.11 on page 302 for the control parameter, \(ctrl\), and extraction options.

The generic _eextract function encompasses all the capabilities of the _eextractx, _eextract, and _eextractm functions which were described in the preceding pages. These three functions are typically incorporated for ease of use with specific parameters.

The _eextract function is also very similar to the _xextract function on page 189, except that _eextract results can be rounded or limited.
3.5.10 **Ensemble Special Operations**

The Ensemble Special Operations functions include:

- `_elogmost`  Log of Most-Significant Bit
- `_elogmostu` Log of Most-Significant Bit Unsigned
- `_erecestf`  Reciprocal Estimate Floating-Point
- `_ersqestf`  Reciprocal Square-Root Estimate Floating-Point
- `_esqrf`     Square-Root Floating-Point
_elogmost Log of Most-Significant Bit

Computes position of each vector element's most-significant bit.

```
v8_t _elogmost8(v8_t x)
v16_t _elogmost16(v16_t x)
v32_t _elogmost32(v32_t x)
v64_t _elogmost64(v64_t x)
v128_t _elogmost128(v128_t x)
```

This function takes a 128-bit vector parameter, \( x \), containing elements of \( \text{esize} \)-bit signed integers. The position of each element's most-significant bit is computed, producing a 128-bit result vector of \( \text{esize} \)-bit elements. If an element's value is 0, the value -1 is returned.

\[
r[i] = \text{Floor} \left( \log_2(\text{Abs}(x[i])) \right), \quad i = 0..\text{NELEM} -1
\]
_elogmostu  

Log of Most-Significant Bit Unsigned

Computes position of each vector element's most-significant bit.

\[ r[i] = \text{Floor} \left( \log_2(x[i]) \right), \quad i = 0..\text{NELEM} -1 \]
Reciprocal Estimate Floating-Point

Computes reciprocals of vector elements.

\[
\begin{align*}
vf16_t & \ _\text{erecestf16}\{c,f,n,z,x}\} (vf16_t \ x) \\
vf32_t & \ _\text{erecestf32}\{c,f,n,z,x\} (vf32_t \ x) \\
vf64_t & \ _\text{erecestf64}\{c,f,n,z,x\} (vf64_t \ x) \\
vf128_t & \ _\text{erecestf128}\{c,f,n,z,x\} (vf128_t \ x)
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), containing elements of \( \text{esize} \)-bit floating-point numbers. An approximation to the reciprocal, \( 1/x \), is computed for each element, producing a 128-bit result vector of \( \text{esize} \)-bit elements. The result is correct for 16-bit floating-point and good to 11 bits of significance for higher-precision formats, where it can serve as a good starting-point for a Newton-Raphson iteration that achieves full accuracy.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
\begin{align*}
\text{optional trap} & \leftarrow (1/x, 1/x, 1/x, 1/x) \\
\end{align*}
\]

\[
r[i] = 1/(x[i]), \quad i = 0..\text{NELEM} -1
\]
_ersqestf Reciprocal Square-Root Estimate Floating-Point

Computes reciprocal square-roots of vector elements.

\[
\begin{align*}
\text{vf16}_t & \_\text{ersqestf16}\{,c,f,n,z,x\}(\text{vf16}_t \ x) \\
\text{vf32}_t & \_\text{ersqestf32}\{,c,f,n,z,x\}(\text{vf32}_t \ x) \\
\text{vf64}_t & \_\text{ersqestf64}\{,c,f,n,z,x\}(\text{vf64}_t \ x) \\
\text{vf128}_t & \_\text{ersqestf128}\{,c,f,n,z,x\}(\text{vf128}_t \ x)
\end{align*}
\]

This function takes a 128-bit vector parameter, \(x\), containing elements of \(\text{esize}\)-bit floating-point numbers. An approximation to the reciprocal square-root is computed for each element, producing a 128-bit result vector of \(\text{esize}\)-bit elements. The result is correct for 16-bit floating-point and good to 11 bits of significance for the higher-precision formats, where it can serve as a good starting-point for a Newton-Raphson iteration that achieves full accuracy.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = \frac{1}{\sqrt{x[i]}}, \quad i = 0..\text{NELEM} - 1
\]
Computes square-roots of vector elements.

\[
\begin{align*}
\text{vf16}_t & \quad \text{esqrf16} (, c, f, n, z, x) (\text{vf16}_t \ x) \\
\text{vf32}_t & \quad \text{esqrf32} (, c, f, n, z, x) (\text{vf32}_t \ x) \\
\text{vf64}_t & \quad \text{esqrf64} (, c, f, n, z, x) (\text{vf64}_t \ x) \\
\text{vf128}_t & \quad \text{esqrf128} (, c, f, n, z, x) (\text{vf128}_t \ x)
\end{align*}
\]

This function takes a 128-bit vector parameter, \( x \), containing elements of \( esize \)-bit floating-point numbers. The square-root of each element is computed. The result is rounded using the specified rounding mode, or round-to-nearest if not specified, producing a 128-bit result vector of \( esize \)-bit elements.

See Section 2.3 on page 39 for details on optional rounding and exception modes.

\[
r[i] = \sqrt{x[i]}, \quad i = 0..\text{NELEM} -1
\]
3.5.11 **Extraction Control**

Some of the BroadMX functions support a powerful field extraction capability as the final step of their operation. Six of the Broadband functions—one Crossbar function and five Ensemble functions—have a special capability for extracting fields of bits from source-vector elements and repositioning them in result-vector elements. Figure 30 on page 306 and Table 30 on page 307 describe the low 32 bits of the 128-bit control parameter, ctrl, used to define extraction variables in these function. The functions are described in this chapter.

- **Extract-Merge Functions:**
  - _xextract function: page 189
  - _eextractx function: page 291
  - _eextract function: page 292
  - _eextractm function: page 293

- **Extract Functions Combined with Other Functions:**
  - _econx function: page 277
  - _emuladdx function: page 244
  - _emulx function: page 237
  - _escaladdx function: page 263

3.5.11.1 **Extract-Merge Functions**

The _xextract and _eextract functions provide the most flexible extract capabilities, offering the choice between simple extract or extract-merge. The control parameter, ctrl, defines the following modes for these function:

- **Extract**—Extracts fields from one or two source vectors, as follows:
  - *Extract from Single-Size Source*: Extracts fields from esize-bit elements in vector x, and places them in esize-bit elements in the result. Vector y is not used, although its reference must be present in the function syntax. This mode is selected by setting the x bit to 1 in ctrl. See Figure 25 on page 303.
  - *Extract from Double-Size Source*: Extracts fields from 2*esize-bit elements in the concatenated vectors x and y, and places them in esize-bit elements in the result. This mode is selected by clearing the x bit to 0 in ctrl. See Figure 26 on page 303.

- **Extract-Merge**—Extracts fields from esize-bit elements in vector x, merges them with the bit positions from vector y that are outside the extracted fields from vector x, and places them into esize-bit elements in the result. This mode is selected by setting the m bit to 1 in ctrl, in which case the x bit is ignored. See Figure 27 on page 303.
Figure 25. Extract from Single-Size Source (xextract and eextract)

Figure 26. Extract from Double-Size Source (xextract and eextract)

Figure 27. Extract-Merge (xextract and eextract)
3.5.11.2 Extract Functions Combined with Other Functions

The `_econx`, `_emuladdx`, `_emulx`, and `_escaladdx` functions combine multiply or convolution operations with back-end extract functions. The control parameter, `ctrl`, defines the following modes for these function:

- **Extract to Double-Size Result**: Extracts fields from $2^{*esize}$-bit elements in the intermediate products, and places them in $2^{*esize}$-bit elements in the result. This mode is selected by setting the x bit to 1 in `ctrl`. The high 64 bits source vectors are ignored. See Figure 28.

- **Extract to Single-Size Result**: Extracts fields from $2^{*esize}$-bit elements in the intermediate products, and places them in `esize`-bit elements in the result. This mode is selected by clearing the x bit to 0 in `ctrl`. See Figure 29.

There is no merge mode for these functions.

**Figure 28. Extract to Double-Size Result (**econx, emuladdx, emulx, and escaladdx**)**

**Figure 29. Extract to Single-Size Result (**econx, emuladdx, emulx, and escaladdx**)**
3.5.11.3 **The Control Parameter (ctrl)**

Figure 30 on page 306 and Table 30 on page 307 describe the fields of the extract control parameter, \texttt{ctrl}. The fields are arranged so that the most common control parameters can be computed dynamically. For applications with multiple, independent vector elements, such as arithmetic and shift operations, \texttt{esize} is typically equal to the size of a single element in a vector. For applications which treat an entire 128- or 256-bit vector as a single unit, such as delay lines, there is no particular correspondence between \texttt{esize} and vector-element size.
### Figure 30. xextract Control Parameter (low 32 bits)

<table>
<thead>
<tr>
<th>Field Size (fsize)</th>
<th>31</th>
<th>24</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination Position (dpos)</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>x</td>
<td>s</td>
<td>n</td>
<td>m</td>
</tr>
</tbody>
</table>

- **Field Size (fsize)**

- **Extended Size**
  
eextract, xextract:
  
  - 1 = Extract from Single-Size Source (esize source to esize result)
  - 0 = Extract from Double-Size Source (2*esize source to esize result)

  econx, econx, emuladdr, emulx, escaladdr:
  
  - 1 = Extract to Double-Size Result (2*esize source to 2*esize result)
  - 0 = Extract to Single-Size Result (2*esize source to esize result)

- **Signed or Unsigned**
  
  - 1 = Signed
  - 0 = Unsigned

- **Real or Complex (ignored in eextract and xextract)**
  
  - 1 = Complex
  - 0 = Real

- **Merge or Mixed-Sign**
  
eextract, xextract:
  
  - 1 = Extract-Merge
  - 0 = Extract

  econx, econx, emuladdr, emulx, escaladdr:
  
  - 1 = Mixed Sign
  - 0 = Same Sign

- **Limit or Truncate (ignored in xextract)**
  
  - 1 = Limit (saturate)
  - 0 = Truncate

- **Rounding (ignored in xextract)**
  
  - 11 = Ceiling
  - 10 = Nearest
  - 01 = Zero
  - 00 = Floor

- **Element Size (esize) and Source Position (spos)**
  
gssp = 512 - 4*esize + spos

---

496-005.apx
Table 30. Fields in xextract Control Parameter (low 32 bits)

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| fsize | 31:24 | **Field Size**—The size of fields to be extracted. All 8-bit values are legal, with the following constraints:  
  - $fsize = 0$ specifies the maximum field size that fits the destination.  
  - If the specified field extends beyond the destination, or if it extends beyond the source, $fsize$ is clamped to:  
    $$fsize = \text{Min}(fsize, (esize - dpos), (2*esize - spos))$$  
  To obtain maximum precision, set $fsize$ equal $esize$, and $dpos$ equal to zero. |
| dpos  | 23:16 | **Destination Position**—The starting bit number in the destination where the extracted field is to be placed. Destination bits to the right of bit-number $dpos$ are:  
  - in *Extract-Merge* mode ($m = 1$): left unchanged (*eextract* and *xextract* only).  
  - in *Extract* mode ($m = 0$): zero-filled.  
  $dpos$ must be less than $esize$; if it is not, the *ReservedInstruction* exception is taken. |
| $x$   | 15    | **This bit has two different meanings, depending on the function:**  
  - **Extended Source Size**—For *eextract* and *xextract* functions, the $x$ bit specifies the relative size of the source with respect to the result:  
    - 1 = *Extract from Single-Size Source*. Extracts from $esize$ source elements to $esize$ result elements. For example, 16-bit source elements are extracted to 16-bit result elements.  
    - 0 = *Extract from Double-Size Source*. Extracts from $2*esize$ source elements to $esize$ result elements. For example, 32-bit source elements are extracted to 16-bit result elements.  
    The $x$ bit is ignored when the $m$ bit is set to 1.  
  - **Extended Result Size**—For *econx*, *emuladdx*, *emulx*, and *escaladdx* functions, the $x$ bit specifies the relative size of the result with respect to the intermediate-product source:  
    - 1 = *Extract to Double-Size Result*. Extracts from $2*esize$ elements of intermediate products to $2*esize$ result elements. For example, a 16-bit by 16-bit vector multiply, which yields $2*esize$ (32-bit) intermediate products, is extracted to 32-bit result elements.  
    - 0 = *Extract to Single-Size Result*. Extracts from $2*esize$ elements of intermediate products to $esize$ result elements. For example, a 16-bit by 16-bit vector multiply, which yields $2*esize$ (32-bit) intermediate products, is extracted to 16-bit result elements. |
Table 30. Fields in xextract Control Parameter (low 32 bits) (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| **s** | 14 | **Sign**—Specifies, along with the **m** bit, whether arithmetic is performed on signed or unsigned integers. This affects (a) sign-extension on fields that extend beyond the source’s most-significant bit, and (b) how saturation, if specified, is performed.  
1 = Signed  
0 = Unsigned |
| **n** | 13 | **Real or Complex**—Specifies whether the vectors are treated as vectors of real numbers or vectors of complex numbers. This bit is ignored for the **eextract** and **xextract** functions.  
1 = Complex.  
0 = Real. |
| **m** | 12 | This bit has two different meanings, depending on the function:  
**Merge**—For **eextract** and **xextract** functions, the **m** bit specifies how vacant bits are filled:  
1 = Extract-Merge. Takes fields from esize-bit elements in vector **x**, merges them with the bit positions from vector **y** that are not filled by the fields in vector **x**, and places them into concatenated esize-bit elements of the result vector. The **x** bit is ignored when the **m** bit is set to 1.  
0 = Extract. Shifts fields from 2*esize-bit elements in the concatenation of vector **x** (high order) and vector **y** (low order), and places them into concatenated esize-bit elements of the result vector.  
**Mixed-Sign**—For **econx**, **emuladdx**, **emulx**, and **escaladdx** functions, the **m** bit specifies, along with the **s** bit, the sign of operands:  
1 = Mixed-Sign Multiplication. The first operand is treated as signed (if the **s** bit is 1) or unsigned (if the **s** bit is 0). The second operand is treated the opposite way.  
0 = Same-Sign Multiplication. The sign of operands is specified by the **s** bit. |
| **l** | 11 | **Limiting**—Specifies the method of handling values that overflow. This bit is ignored for the **eextract** and **xextract** functions.  
1 = Limit (saturate). Examines any significant bits that are dropped as a result of the extract. If overflow occurs, the result is clamped to the minimum or maximum integer representable in esize bits.  
0 = Truncate. Ignores arithmetic overflow and simply extracts the esize number of bits. This is equivalent to performing arithmetic modulo 2^esize. |
**Table 30. Fields in xextract Control Parameter (low 32 bits) (continued)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| `rnd` | 10:9 | **Rounding**—Specifies the rounding mode. This bit is ignored for the `eextract` and `xextract` functions.  
11 = Ceiling  
10 = Nearest  
01 = Zero  
00 = Floor |
| `gssp`| 8:0  | **Element Size (esize) and Source Positions (spos)**—Specifies the element size and the source position of vector elements. Element size (esize) is the bit-length of the vector's elements. For complex operands, maximum esize is 64. Source position (spos) is the bit-position (counting from bit 0) at which extraction is to start.  
Bits 8:0 are: |

<table>
<thead>
<tr>
<th>esize</th>
<th>gssp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1 0</td>
</tr>
<tr>
<td>32</td>
<td>1 1 0</td>
</tr>
<tr>
<td>16</td>
<td>1 1 1 0</td>
</tr>
<tr>
<td>8</td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>4</td>
<td>1 1 1 1 1 0</td>
</tr>
<tr>
<td>2</td>
<td>1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1 1 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

The encoding is equivalent to:

\[
gssp = 512 - 4 * \text{esize} + \text{spos}
\]

where `spos` is in the range 0 to \((2 * \text{esize}) - 1\).  
Values of `gssp` not in the table are interpreted as `esize = 8` and `spos = bits 3..0`. 
3.6 **Wide Functions**

Wide functions perform lookups on large tables (matrices) of data in memory, or they perform memory-table-based Crossbar or Ensemble operations. Table lookups can be done on up to 128 bits by up to 256 table entries (rows). They load a table of source data from memory, perform a set of operations on partitions of bits in the source data, and return the result as a 128-bit vector of elements.

The Wide functions include the following types:

- Wide Table Lookup
- Wide Crossbar
- Wide Ensemble
3.6.1 **Wide Table Lookup**

The Wide Table Lookup functions include:

- `_wtranslate`  Wide Table Lookup
Chapter 3: Vector and Matrix Functions

Wide Functions

_wtranslate

Wide Table Lookup

Looks up indexed entries in tables of data in memory.

v8_t _wtranslate8{l,b}(int8 const *addr, v8_t x, int w, int d)
v16_t _wtranslate16{l,b}(int16 const *addr, v16_t x, int w, int d)
v32_t _wtranslate32{l,b}(int32 const *addr, v32_t x, int w, int d)
v64_t _wtranslate64{l,b}(int64 const *addr, v64_t x, int w, int d)
vc8_t _wtranslatec8{l,b}(cplxi8 const *addr, vc8_t x, int w, int d)
vc16_t _wtranslatec16{l,b}(cplxi16 const *addr, vc16_t x, int w, int d)
vc32_t _wtranslatec32{l,b}(cplxi32 const *addr, vc32_t x, int w, int d)

This function takes an address, \texttt{addr}, a 128-bit vector parameter, \texttt{x}, a scalar table width, \texttt{w}, and a scalar table depth, \texttt{d}. The address, \texttt{addr}, is interpreted as the base address of the tables in memory. The vector, \texttt{x}, is interpreted as containing elements of \texttt{esize}-bit indices into the tables. The \texttt{w} is interpreted as the number of tables; the range of \texttt{w} is 1 (in which case the same table is used for all indices) up to \texttt{NELEM} (in which case each element gets its own table). If \texttt{w} is less than the number of indices in the vector, the indices wrap around to the next-most-significant address in the tables. The \texttt{d} is interpreted as the number of \texttt{esize}-bit memory locations that constitute each table; the depth can be between 4 and 256.

The indexed entries in each table are loaded from memory, modulo the depth, and concatenated, producing a 128-bit result vector of \texttt{esize}-bit elements.

The address in memory accessed by a given index is:

\[
addr + (index \times (esize/8) \times w)
\]

The function can access entries in tables of up to 32,768 bits (128 x 256 bits).
\[ r[i] = \text{addr}[(i \mod w) + (x[i] \mod d) \times w], \ i = 0 \ldots \text{NELEM}-1 \]
3.6.2 **Wide Crossbar**

The Wide Crossbar functions include:

- `_wswitch` Wide Switch
_wswitch

Wide Switch

Selects one of 256 source bits for each bit of a 128-bit result.

v8_t _wswitch8{,l,b}(uint8 const *addr, v8_t xlo, v8_t xhi)
v16_t _wswitch16{,l,b}(uint8 const *addr, v16_t xlo, v16_t xhi)
v32_t _wswitch32{,l,b}(uint8 const *addr, v32_t xlo, v32_t xhi)
v64_t _wswitch64{,l,b}(uint8 const *addr, v64_t xlo, v64_t xhi)
v128_t _wswitch128{,l,b}(uint8 const *addr, v128_t xlo, v128_t xhi)

This function takes an address, addr, and two 128-bit vector parameters, xlo and xhi. The address, addr, is interpreted as the base address of a matrix of 128-by-8-bit switch controls in memory. The vectors are concatenated, xlo low and xhi high, and interpreted as a field of 256 bits. The 8-bit switch control for each bit position of the result selects one of the 256 bits for that result bit-position, producing a 128-bit result vector of 1-bit elements.

The byte selectors for each of the 128 output bits are stored in memory in an unusual way. The byte that selects the source of output bit-position i is assembled as the concatenation of 8 individual bits—bit 0 of the byte is obtained from bit i of addr[0], bit 1 of the byte is obtained from bit i of addr[1], and so on up to bit 7 of the byte, which is obtained from bit i of addr[7]. The matrix of 8-bit switch controls in memory is organized in this manner because it reduces the wiring needed in the architecture’s hardware implementation.
\[ r_i = x_j \]

where: \( i = 0..127 \)

\[ j = addr[7][i] \ll 7 + addr[6][i] \ll 6 + \ldots + addr[1][i] \ll 1 + addr[0][i] \ll 0 \]
3.6.3 **Wide Ensemble**

The Wide Multiply Matrix functions include:

- `_wmulmat` Wide Multiply Matrix
- `_wmulmatu` Wide Multiply Matrix Unsigned
- `_wmulmatm` Wide Multiply Matrix Mixed-Sign
- `_wmulmatc` Wide Multiply Matrix Complex
- `_wmulmatf` Wide Multiply Matrix Floating-Point
- `_wmulmatcf` Wide Multiply Matrix Complex Floating-Point
- `_wmulmatp` Wide Multiply Matrix Polynomial
- `_wmulmatg` Wide Multiply Matrix Galois
- `_wmulmatxi` Wide Multiply Matrix Extract Immediate
- `_wmulmatxic` Wide Multiply Matrix Extract Immediate Complex
- `_wmulmatx` Wide Multiply Matrix Extract
Multiplies elements of a vector with a matrix in memory.

v16_t _wmulmat8(,l,b)(int8 const *addr, v8_t x, int w, int d)
v32_t _wmulmat16(,l,b)(int16 const *addr, v16_t x, int w, int d)
v64_t _wmulmat32(,l,b)(int32 const *addr, v32_t x, int w, int d)

This function takes an address parameter, addr, a 128-bit vector parameter, x, and two scalar values width, w, and depth, d. The address, addr, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing w times d elements of esize-bit signed integers. Vector x is interpreted as containing d elements of esize-bit signed integers.

The elements of vector x are multiplied by the elements of the matrix. The results for each matrix column are summed, producing a 128-bit result vector of 2*esize-bit elements.

\[
r[i] = \left( \sum_{j=0}^{\text{depth}-1} x[j] \cdot \text{addr}[i + \text{width} \times j] \right), \quad i = 0..\text{width}-1
\]

\[
r[i] = 0, \quad i = \text{width}..\text{NELEM}/2-1
\]

where: \( 2 \leq \text{depth} \leq \text{NELEM}, \) and \( 2 \leq \text{width} \leq \text{NELEM}/2 \)
### _wmulmatu

**Wide Multiply Matrix Unsigned**

Multiplies elements of a vector with a matrix in memory.

- `vu16_t _wmulmatu8({},1,b)(int8 const *addr, vu8_t x, int w, int d)`
- `vu32_t _wmulmatu16({},1,b)(int16 const *addr, vu16_t x, int w, int d)`
- `vu64_t _wmulmatu32({},1,b)(int32 const *addr, vu32_t x, int w, int d)`

This function takes an address parameter, `addr`, a 128-bit vector parameter, `x`, and two scalar values width, `w`, and depth, `d`. The address, `addr`, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing `w` times `d` elements of `esize`-bit unsigned integers. Vector `x` is interpreted as containing `d` elements of `esize`-bit unsigned integers.

The elements of vector `x` are multiplied by the elements of the matrix. The results for each matrix column are summed, producing a 128-bit result vector of `2*esize`-bit elements.

The diagram illustrates the process:

- **Matrix in Memory**
  - `w` columns
  - `d` rows
  - Elements are multiplied and summed.

- **Vector `x`**
  - `d` elements
  - Elements are multiplied.

- **Result Vector `r`**
  - `2*esize` elements
  - Elements are summed.

Mathematically:

\[
 r[i] = \left( \sum_{j=0}^{\text{depth}-1} x[j] \cdot \text{addr}[i + \text{width} \times j] \right), \quad i = 0..\text{width}-1 \]

where:

\[
r[i] = 0, \quad i = \text{width}..\text{NELEM}/2-1
\]

and:

\[
2 \leq \text{depth} \leq \text{NELEM}, \quad \text{and} \quad 2 \leq \text{width} \leq \text{NELEM}/2
\]
wide multiply matrix mixed-sign

This function takes an address parameter, `addr`, a 128-bit vector parameter, `x`, and two scalar values width, `w`, and depth, `d`. The address, `addr`, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing `w` times `d` elements of `esize`-bit signed integers. Vector `x` is interpreted as containing `d` elements of `esize`-bit unsigned integers.

The elements of vector `x` are multiplied by the elements of the matrix. The results for each matrix column are summed, producing a 128-bit result vector of 2*`esize`-bit elements.

\[
    r[i] = \sum_{j=0}^{\text{depth} - 1} x[j] \cdot \text{addr}[i + \text{width} \times j], \quad i = 0..\text{width}-1
\]

\[
    r[i] = 0, \quad i = \text{width}..\text{NELEM}/2-1
\]

where: \(2 \leq \text{depth} \leq \text{NELEM}\), and \(2 \leq \text{width} \leq \text{NELEM}/2\)
_wmulmatc

Wide Multiply Matrix Complex

Multiplies elements of a vector of complex floating-point numbers with a matrix in memory.

vc16_t _wmulmatc8({l,b})(cplxi8 const *addr, vc8_t x, int w, int d)
vc32_t _wmulmatc16({l,b})(cplxi16 const *addr, vc16_t x, int w, int d)

This function takes an address parameter, addr, a 128-bit vector parameter, x, and two scalar values width, w, and depth, d. The address, addr, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing w times d elements of 2*esize-bit complex numbers, in which the real and imaginary elements are each esize bits wide. Vector x is interpreted as containing d elements of 2*esize-bit complex numbers.

The elements of vector x are complex-multiplied by the elements of the matrix. The results for each matrix column are summed, producing a 128-bit result vector of 2*esize-bit elements.
r[i] = \left( \sum_{j=0}^{\text{depth}-1} x_j \cdot \text{addr}[i + \text{width} \times j] \right), \quad i = 0..\text{width}-1

r[i] = 0, \quad i = \text{width}.\text{NELEM}/2-1

where: \quad 2 \leq \text{depth} \leq \text{NELEM}, \text{ and } 2 \leq \text{width} \leq \text{NELEM}/2
_wmulmatf  Wide Multiply Matrix Floating-Point

Multiplies elements of a vector of floating-point numbers with a matrix in memory.

\[
\begin{align*}
\text{vf16_t } & \text{ _wmulmatf16}(\langle 1, 1 \rangle) \text{ (float16 const *addr, vf16_t x, int w, int d)} \\
\text{vf32_t } & \text{ _wmulmatf32}(\langle 1, 1 \rangle) \text{ (float32 const *addr, vf32_t x, int w, int d)} \\
\text{vf64_t } & \text{ _wmulmatf64}(\langle 1, 1 \rangle) \text{ (float64 const *addr, vf64_t x, int w, int d)}
\end{align*}
\]

This function takes an address parameter, \texttt{addr}, a 128-bit vector parameter, \texttt{x}, and two scalar values \texttt{width}, \texttt{w}, and \texttt{depth}, \texttt{d}. The address, \texttt{addr}, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing \texttt{w} times \texttt{d} elements of \texttt{esize}-bit floating-point numbers. Vector \texttt{x} is interpreted as containing \texttt{d} elements of \texttt{esize}-bit floating-point numbers.

The elements of vector \texttt{x} are multiplied by the elements of the matrix. The results for each matrix column are summed, producing a 128-bit result vector of \texttt{esize}-bit elements.

\[
\begin{align*}
[r[i]] &= \left\{ \begin{array}{ll}
\sum_{j=0}^{\text{depth}-1} x_j \cdot \text{addr}[\text{width} \times j + i], & i = 0..\text{width}-1 \\
0, & i = \text{width}..\text{NELEM}-1
\end{array} \right.
\end{align*}
\]

where: \( 2 \leq \text{depth} \leq \text{NELEM}, \text{ and } 2 \leq \text{width} \leq \text{NELEM} \)

\[\begin{array}{c}
\text{higher}
\\\text{addresses}
\\\text{in memory}
\\\text{Matrix in Memory}
\\
\begin{array}{cccc}
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast
\end{array}
\\
\text{element}
\\\text{element}
\\\text{element}
\\\text{element}
\\
\text{esize}
\\\text{esize}
\\\text{esize}
\\\text{esize}
\\
\text{127}
\\\text{0}
\\
\ast\text{addr}
\\
\text{base address}
\end{array}\]

\[\sum_{j=0}^{\text{depth}-1} x_j \cdot \text{addr}[\text{width} \times j + i], \quad i = 0..\text{width}-1\]

where: \( 2 \leq \text{depth} \leq \text{NELEM}, \text{ and } 2 \leq \text{width} \leq \text{NELEM} \)
**_wmulmatcf**  
Wide Multiply Matrix Complex Floating-Point

Multiplies elements of a vector of complex floating-point numbers with a matrix in memory.

```c
vcf16_t _wmulmatcf16{,l,b}{cplxf16 const *addr, vcf16_t x, int w, int d}
vcf32_t _wmulmatcf32{,l,b}{cplxf32 const *addr, vcf32_t x, int w, int d}
```

This function takes an address parameter, `addr`, a 128-bit vector parameter, `x`, and two scalar values width, `w`, and depth, `d`. The address, `addr`, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing `w` times `d` elements of `2*esize`-bit complex floating-point numbers, in which the real and imaginary elements are each `esize` bits wide. Vector `x` is interpreted as containing `d` elements of `2*esize`-bit complex floating-point numbers.

The elements of vector `x` are complex-multiplied by the elements of the matrix. The results for each matrix column are summed, producing a 128-bit result vector of `2*esize`-bit complex floating-point numbers.

\[
r[i] = \left( \sum_{j=0}^{\text{depth}-1} x_j \cdot \text{addr}[\text{width} \times j + i] \right), \quad i = 0..\text{width}-1
\]
\[ r[i] = 0, \quad i = \text{width}..\text{NELEM}-1 \]

where: \( 2 \leq \text{depth} \leq \text{NELEM}, \) and \( 2 \leq \text{width} \leq \text{NELEM} \)
Wide Multiply Matrix Polynomial

_MULTIPLY§

Multiplies a vector of polynomials with a matrix of polynomials in memory.

\[
vu16_t \_wmulmatp8(l,b)(\text{uint8 const } \ast \text{addr}, \text{vu8_t x}, \text{int } w, \text{int } d)\]
\[
vu32_t \_wmulmatp16(l,b)(\text{uint16 const } \ast \text{addr}, \text{vu16_t x}, \text{int } w, \text{int } d)\]
\[
vu64_t \_wmulmatp32(l,b)(\text{uint32 const } \ast \text{addr}, \text{vu32_t x}, \text{int } w, \text{int } d)\]

This function takes an address parameter, addr, a 128-bit vector parameter, x, and two scalar values width, w, and depth, d. The address, addr, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing \(w\) times \(d\) elements, each of which represents a \((N-1)\)°-degree polynomial with binary coefficients (0 or 1). Vector x is interpreted as containing \(d\) elements, also representing \((N-1)\)° degree polynomials with binary coefficients.

The elements of vector x are polynomial-multiplied (see page 211 for a description of polynomial multiplication) by the elements of the matrix and summed, producing a 128-bit result vector of \(2^N\)-bit elements, each representing a \((2n-2)\)° degree polynomial. The operation never overflows.
The text on the page is as follows:

Chapter 3: Vector and Matrix Functions

Wide Functions

\[ r[i] = \sum_{j=0}^{\text{depth} - 1} x_j \cdot P \text{addr}[\text{width} \times j + i], \quad i = 0..\text{width}-1 \]

\[ r[i] = 0, \quad i = \text{width}..\text{NELEM}/2-1 \]

where: \( 2 \leq \text{depth} \leq \text{NELEM}, \) and \( 2 \leq \text{width} \leq \text{NELEM}/2 \)
_wmulmatg

Wide Multiply Matrix Galois Field

Multiplies a vector of 7th degree polynomials with a matrix of polynomials in memory.

v8_t _wmulmatg8(ulong const *addr, vu8_t x, int p)

This function takes an address parameter, addr, a 128-bit vector parameter, x, and a basis polynomial p. The address, addr, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing 256 elements, each of which is a byte representing a 7th-degree polynomial with binary coefficients (0 or 1). Vector x is interpreted as containing 16 elements, 8 bits each, also representing 7th degree polynomials with binary coefficients. The integer p is interpreted as the basis polynomial, with an implied 1 bit in bit-position 8.

The elements of vector x are polynomial-multiplied (see page 211 for a description of polynomial multiplication) by the elements of the matrix and summed, producing an intermediate sum of products of 16-bit elements, each representing a 14th degree polynomial. The intermediate sum of products is then taken modulo p, producing a 128-bit result vector with 8-bit elements, each representing a 7th degree polynomial. The operation never overflows.
Chapter 3: Vector and Matrix Functions

Wide Functions

Matrix in Memory

\[ \begin{array}{cccccccc}
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\end{array} \]

\[ x \]

\[ w \]

\[ d \]

\[ * \text{addr} \]

\[ \text{base address} \]

\[ r \]

\[ p \]

\[ \text{127} \]

\[ \text{0} \]
\[ r[i] = \left( \sum_{j=0}^{\text{depth}-1} x_j \cdot P \cdot \text{addr(width} \times j + i) \right) \mod p[i], \quad i = 0..\text{width}-1 \]

\[ r[i] = 0, \quad i = \text{width}..\text{NELEM}-1 \]

where: \( 2 \leq \text{depth} \leq \text{NELEM}, \) and \( 2 \leq \text{width} \leq \text{NELEM} \)
_wmulmatxi  Wide Multiply Matrix Extract Immediate

Multiplies elements of a vector with a matrix in memory and extracts fields.

```
v8_t _wmulmatx8(1, b)(int8 const *addr, v8_t x, int sh, int w, int d)
v16_t _wmulmatx16(1, b)(int16 const *addr, v16_t x, int sh, int w, int d)
v32_t _wmulmatx32(1, b)(int32 const *addr, v32_t x, int sh, int w, int d)
v64_t _wmulmatx64(1, b)(int64 const *addr, v64_t x, int sh, int w, int d)
```

This function takes an address parameter, `addr`, a 128-bit vector parameter, `x`, and three scalar values shift amount, `sh`, width, `w`, and depth, `d`. The address, `addr`, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing `w` times `d` elements of `esize`-bit signed integers. Vector `x` is interpreted as containing `d` elements of `esize`-bit signed integers. Scalar `sh` is interpreted as a shift amount, in bits.

The elements of vector `x` are multiplied by the elements of the matrix. The results for each column are summed, and each sum is right-shifted by `sh`, with rounding to nearest, producing a 128-bit result vector of `esize`-bit elements.
\[ r[i] = \left( \sum_{j=0}^{\text{depth} - 1} x_j \cdot \text{addr}[\text{width} \times j + i] \right) \gg \text{sh} \quad \text{round to nearest, } i = 0..\text{width}-1 \]

\[ r[i] = 0, \quad i = \text{width}..\text{NELEM}-1 \]

where: \( 2 \leq \text{depth} \leq \text{NELEM}, \) and \( 2 \leq \text{width} \leq \text{NELEM} \)
This function takes an address parameter, `addr`, a 128-bit vector parameter, `x`, and three scalar values shift amount, `sh`, width, `w`, and depth, `d`. The address, `addr`, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing `w` times `d` elements of `2^esize`-bit complex numbers, in which the real and imaginary elements are each `esize` bits wide. Vector `x` is interpreted as containing `d` elements of `2^esize`-bit complex numbers. Scalar `sh` is interpreted as a shift amount, in bits.

The elements of vector `x` are complex-multiplied by the elements of the matrix. The results for each matrix column are summed, and each sum is right-shifted by `sh`, with rounding to nearest, producing a 128-bit result vector of `2^esize`-bit complex numbers.
r[i] = \left( \sum_{j = 0}^{\text{depth} - 1} x_j \cdot \text{addr}[\text{width} \times j + i] \right) \gg \text{sh} \ , \ \text{round to nearest, } i = 0..\text{width-1} \\
\text{r[i] = 0, } i = \text{width..NELEM-1} \\
\text{where: } 2 \leq \text{depth} \leq \text{NELEM, and } 2 \leq \text{width} \leq \text{NELEM}
Chapter 3: Vector and Matrix Functions

Wide Functions

_wmulmatx

Wide Multiply Matrix Extract

Multiplies elements of a vector with a matrix in memory and extracts fields.

\[
\begin{align*}
\text{v8_t } & \_wmulmatx8\{,l,b\}\text{(void const }*\text{addr, v8_t }x, \text{ int } w, \text{ int } d, \text{ int } ctrl) \\
\text{v16_t } & \_wmulmatx16\{,l,b\}\text{(void const }*\text{addr, v16_t }x, \text{ int } w, \text{ int } d, \text{ int } ctrl) \\
\text{v32_t } & \_wmulmatx32\{,l,b\}\text{(void const }*\text{addr, v32_t }x, \text{ int } w, \text{ int } d, \text{ int } ctrl) \\
\text{v64_t } & \_wmulmatx64\{,l,b\}\text{(void const }*\text{addr, v64_t }x, \text{ int } w, \text{ int } d, \text{ int } ctrl)
\end{align*}
\]

This function takes an address parameter, \texttt{addr}, a 128-bit vector parameter, \texttt{x}, two scalar values \texttt{width}, \texttt{w}, and depth, \texttt{d}, and a scalar control parameter, \texttt{ctrl}. The address, \texttt{addr}, is interpreted as the base address of a matrix in memory. The matrix is interpreted as containing \texttt{w} times \texttt{d} elements of \texttt{esize}-bit integers. Vector \texttt{x} is interpreted as containing elements of \texttt{esize}-bit integers, which themselves contain fields of \texttt{fsize} bits. The low 32 bits of \texttt{ctrl} specify control options, including \texttt{esize}, \texttt{fsize}, starting position (\texttt{spos}), and destination position (\texttt{dpos}) for the extraction.

The elements of vector \texttt{x} are multiplied with the matrix, producing a vector of \texttt{2*esize}-bit elements. Fields of \texttt{fsize} bits are extracted from this intermediate result in one of two modes:

- **Extract to Double-Size Result**: Extracts fields from \texttt{2*esize}-bit elements in the intermediate products, and places them in \texttt{2*esize}-bit elements in the result. This mode is selected by setting the \texttt{x} bit to 1 in \texttt{ctrl}. The high 64 bits of the \texttt{x} input vector are ignored.

- **Extract to Single-Size Result**: Extracts fields from \texttt{2*esize}-bit elements in the intermediate products, and places them in \texttt{esize}-bit elements in the result. This mode is selected by clearing the \texttt{x} bit to 0 in \texttt{ctrl}.

In both modes, a field of \texttt{fsize} bits is extracted from each intermediate result. The fields are right-shifted by \texttt{spos}, rounded as specified, and left-shifted by \texttt{dpos}, producing a 128-bit result vector of \texttt{esize}-bit elements.

See Section 3.5.11 on page 302 for the control parameter, \texttt{ctrl}, and extraction options.
Chapter 3: Vector and Matrix Functions

Wide Functions

Matrix in Memory

higher addresses in memory

\[ \sum \] \[ \sum \]

Control Field with x bit = 1

<table>
<thead>
<tr>
<th>x</th>
<th>s</th>
<th>n</th>
<th>m</th>
<th>l</th>
<th>rnd</th>
<th>gsp</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>24</td>
<td>23</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>fs</td>
<td>pos</td>
<td>dpos</td>
<td>fs</td>
<td>size</td>
<td>0</td>
<td>fs</td>
</tr>
</tbody>
</table>
Chapter 3: Vector and Matrix Functions

Wide Functions

Matrix in Memory

$d$

\[ \sum \sum \sum \sum \]

$w$

$higher$ $addresses$ $in$ $memory$

$\times$

$x$

$0$

$esize$ $esize$ $esize$ $esize$

$element$ $element$ $element$ $element$

$*addr$

$base$ $address$

$ctrl$

$control$

$right-shift$ $by$ $spos$

$\ll$

$round$ $as$ $specified$

$\gg$

$left-shift$ $by$ $dpos$

$s$

$s$

$s$

$s$

$esize$

$esize$

$esize$

$esize$

$r$

$Control$ $Field$

$with$ $x$ $bit$ $= 0$

$31$ $24$ $23$ $16$ $15$ $14$ $13$ $12$ $11$ $10$ $9$ $8$ $0$

$fsize$ $dpos$ $x$ $s$ $n$ $m$ $l$ $md$ $gssp$

496-235 ap s
Appendix A  Glossary

**Aligned**
A memory location is said to be *aligned* when its address is an even multiple of a 128-bit memory location. Aligned addresses have their low four bits cleared to zero.

**ANSI/IEEE Standard**
The *ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic*, available from The Institute of Electrical and Electronics Engineers, Inc.

**Big-Endian**
An ordering of bits or bytes in which the most-significant bit or byte of a data structure is always labeled 0 (zero), and the least-significant bit or byte is labeled as the data structure size (in bits or bytes) minus one. In the BroadMX architecture, the ordering of bits in any data structure is always little-endian, but the ordering of bytes can be either big-endian or little-endian.

**BroadMX**
The name of MicroUnity's instruction set architecture.

**Byte**
The concatenation of eight bits, and a single addressable element of the memory array.

**Ceiling**
A rounding mode in which numbers are rounded up, towards positive infinity (+∞). The mode is specified in the *ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic*.

**Clear**
To enter a value of zero (0). Compare set.
**Convolve**

To compute a finite slice of the mathematical convolution function, \( \sum x[i] \times y[j-i] \). Convolutions are used as building blocks for direct implementation of finite impulse response filters (FIRs), where the \( x \) parameter is the signal (sample data) and the \( y \) parameter is the impulse response (coefficients).

**Crossbar Functions**

Crossbar functions take source operands from 128-bit registers (between one and three), and/or from one or two immediate values, perform shift operations on partitions of bits in the source operands, and place the concatenated result in a 128-bit register. Compare *group functions*, *ensemble functions* and *crossbar functions*.

**Dot Product**

The element-by-element multiplication and summing (accumulation) of the corresponding elements of two input vectors, resulting in a scalar output. The vectors must be equal length. Thus, the dot product of two vectors, \( x \) and \( y \), each with \( i \) elements, is equal to \( \sum (x_i \times y_i) \) over all values of \( i \). The elements of the input vectors may be real or complex. Also called scalar product or inner product. Compare *vector*.

**Doublet**


**Element**

An indexed data item in a vector. See *vector*.

**Element Size**

The size, in bits, of one element in a vector (one-dimensional array) of elements. For a given function, the element size of source vectors may be different than the element size of result vectors. In a few cases, two source vectors for a function may have different element sizes. See *esize*.

**Endian Order**

The byte ordering of multi-byte data in memory. Little-endian byte order assigns the lowest memory address to the least-significant byte (little end). Big-endian byte order assigns the lowest memory address to the most-significant byte (big end).
Ensemble Functions

Ensemble functions take source operands from 128-bit registers (between one and four), and/or from an immediate value, perform fixed-point or floating-point operations on partitions of bits in the source operands, and place the concatenated result in a 128-bit register. Compare group functions, ensemble functions and crossbar functions.

esize

The size, in bits, of one element in a vector of elements. The value of esize is defined for a given function. If an integer appears in a function’s mnemonic (e.g., econf32), it is that function’s esize. A few functions, however, define esize in a control value. In most functions, esize is the size of source-vector elements. However, in the case of xexpand, esize is the size of result-vector elements, and in the cases of some emuladd.. and emulsub.. functions, different source vectors have different element sizes (esize or 2*esize).

Exception

A program condition, which arises when a source or result operand is not as expected, and which may or may not cause a trap (jump to service routine). The exception is said to be signaled or occur when the unexpected condition occurs; the exception is said to be taken, raised or trapped if it causes a jump to the associated service routine. Certain floating-point functions have optional mnemonic suffixes (c, f, n, z, x, cd, cf, cz) that specify both rounding mode and exception handling. Including such a suffix causes related exceptions to be taken when their condition is signaled. Certain other floating-point functions have default exception handling, as defined by the ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic, in which a different-than-exact result (such as NaN or denormalized number) is returned without invoking a service routine. Certain fixed-point functions have an optional mnemonic suffix (o) that specifies overflow-exception handling. Compare trap.

Extract

A shift operation in which a field within an element of a source operand, specified by a field size (fsize) and starting position (spos), is shifted to a position in a destination element that is specified a destination position (dpos).

Floating-Point Exception

One of the exceptions defined by the ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic and supported by the BroadMX architecture: invalid operation, division by zero, overflow, underflow, or inexact.
**Floor**

A rounding mode in which numbers are rounded down, towards negative infinity ($-\infty$). The mode is specified in the *ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic*.

**Galois Field**

A field in which arithmetic is performed using polynomial multiplication, modulo a primitive polynomial, and exclusive-OR addition.

**Gateway**

A 64-bit data structure, aligned on 8-byte boundaries with little-endian byte ordering, that is used to securely invoke a system call or procedure at a higher-than-current privilege level. It contains the virtual address of the privileged procedure’s entry point (program counter) and its target privilege level (pl).

**Gateway Access Type**

A set of values assigned by an operating system that specify the minimum required privilege level for reading, writing, executing, or using a gateway.

**Group Functions**

Group functions take source operands from 128-bit registers (between one and three), and/or from an immediate value, perform fixed-point or floating-point operations on partitions of bits in the source operands, and place the concatenated result in a 128-bit register and/or generate an exception. Compare group functions, ensemble functions and crossbar functions.

**Hexlet**


**i128_t**

A fixed-point data type containing 128 bits.

**High**

Most-significant.
**Immediate**

A scalar value included as a parameter in a function’s syntax.

**Inexact**

A floating-point result that requires rounding if it is to avoid an inexact exception. Compare *rounding* and *exact*.

**Inner Product**

Same as *dot product*.

**Inplace**

A function is said to be *inplace* if a single register is used as both the source and destination for the function.

**Limit**

A mechanism that limits overflow of a function result by clamping the result value to the maximum or minimum integer value that can be represented by the result data type. The architecture supports limiting overflow by saturation or truncation. Also called *saturate*.

**Little-Endian**

An ordering of bits or bytes in which the least-significant bit or byte of a data structure is always labeled 0 (zero), and the most-significant bit or byte is labeled as the data structure size (in bits or bytes) minus one. In the BroadMX architecture, the ordering of bits in any data structure is always little-endian, but the ordering of bytes can be either big-endian or little-endian.

**Low**

Least-significant.

**NaN**

Not a number. A floating-point data type specified by the *ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic*. 

Nearest

A rounding mode in which numbers are rounded to the nearest representable number, rounding down when the fraction is less than 1/2 and rounding up when the fraction is greater than 1/2. See Table 4 on page 26 for representable values. The ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic specifies this as the default rounding mode.

Negate

(1) Arithmetic negation: to change the polarity of the sign bit on a signed value, which is the equivalent of complementing a two's-compliment number. (2) Bit-wise negation: to complement each bit of a binary value. Arithmetic negation and bit-wise negation become equivalent when a 1 bit is added to the least-significant bit position of a bit-wise-negated value.

Nibble

The concatenation of four bits.

Octlet

The concatenation of 64 bits, and the concatenation of eight bytes. Defined in IEEE standard 1596.5-1993, Standard for Shared-Data Formats.

Overflow

An exception condition in which a destination-operand format’s largest representable number (either positive or negative) is exceeded in magnitude. The term negative overflow is, in this sense, a type of overflow and is different from the term underflow. See underflow.

Peck

The concatenation of two bits.

Privilege Level

A 2-bit value in bit-positions 0 and 1 of a gateway data structure. It specifies the required level—between 0 (lowest) and 3 (highest)—for accessing code. The operating system assigns a privilege level and a gateway access type to each gateway. The gateway access type specifies the minimum required privilege level for reading, writing, executing, or using the gateway.
Quadlet


Rounding

An operation optionally performed on inexact results of certain BroadMX functions. There are four types of rounding: round to nearest, round toward zero, round toward $+\infty$ (ceiling), and round toward $-\infty$ (floor). The default rounding mode for inexact results is round to nearest for the precision of the destination register or memory location. While rounding avoids an inexact exception, it does not necessarily prevent the other types of floating-point exceptions: invalid operation, division by zero, overflow, or underflow. Compare exact and inexact.

Saturate

See limit.

Scalar

A quantity representable by its position on a scale. Compare vector.

Scalar Product

Same as dot product.

Set

To enter a value of one (1). Compare clear.

Size

Unless otherwise specified, size is measured in bits.

SIGD

Single instruction, group data. The number of bits is fixed (in the BroadMX architecture, at 128) but the number of items in the group is variable.
Symbol

A synonym for a vector element.

Table

A matrix of data located in memory. Tables are accessed by Wide functions.

Trap

A jump to a service routine that is caused by the occurrence of an exception. Compare exception.

Triclet

The concatenation of 256 bits, and the concatenation of thirty-two bytes.

Underflow

The ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic specifies this as a condition dependent on two correlated events—tininess and loss of accuracy. The term is not used to represent negative overflow. See overflow.

\texttt{v128_t, v64_t, v32_t, v16_t, v8_t, v4_t, v2_t}

A 128-bit fixed-point vector data type containing elements of the specified bit size.

\texttt{vf128_t, vf64_t, vf32_t, vf16_t}

A 128-bit floating-point vector data type containing elements of the specified bit size.

Vector

A one-dimensional array of data elements, each of which can be located with an array index. Vectors can represent a complex number. When a vector represents a complex number, the real and imaginary parts of the number are considered two parts of a single vector element. Compare scalar.

Vector Product

The element-by-element multiplication of two vectors, giving a vector result. Compare dot product and vector.
**Wide Functions**

Wide functions perform ensemble and/or crossbar operations on a large matrix of data in memory. They do this by using an address in a general register to fetch a matrix (table) of source data from memory, and a source operand from a 128-bit general register. They perform a group of operations on partitions of bits in the source operands and concatenate the results together, placing the result in a 128-bit general register. Compare *group functions*, *ensemble functions* and *crossbar functions*.

**Zero Rounding**

A rounding mode in which numbers are rounded toward zero. Positive numbers round down, and negative numbers round up. The mode is specified in the *ANSI/IEEE standard 754-1985 Binary Floating-Point Arithmetic*. 
Appendix B  Index

A
addresses........................................................................22
Aligned...........................................................................339
B
Big-Endian.......................................................................339
Boolean Operations........................................................126
BroadMX.........................................................................339
Byte................................................................................24
C
Calling Conventions..........................................................49
Carry Bit..........................................................................28
Ceiling..............................................................................339
clear................................................................................52
Compare AND Equal Zero.................................................148
Compare AND Not Equal Zero............................................148
Compare Equal....................................................................148
Compare Equal Floating-Point............................................148
Compare Greater or Equal...............................................148
Compare Greater or Equal Floating-Point.........................148
Compare Greater or Equal Unsigned..............................148
Compare Greater or Equal Zero........................................148
Compare Greater than Zero.............................................148
Compare Less.....................................................................148
Compare Less Floating-Point.............................................148
Compare Less or Equal Zero.............................................148
Compare Less or Greater Floating-Point........................148
Compare Less than Zero...................................................148
Compare Less Unsigned..................................................148
Compare Not Equal..........................................................148
compare-branched relations.............................................42
compare-set relations.......................................................42
comparison relations........................................................42
Compiler..........................................................................18
complex numbers............................................................35, 36, 38, 80
control parameter............................................................305
Convolve.........................................................................340
cplx16.............................................................................31
cplx32.............................................................................31
cplx64.............................................................................31
cplxi16............................................................................24
cplxi32............................................................................24
cplxi64............................................................................24
cplxi8.............................................................................24
Create Vector.................................................................60, 63, 64, 65

D
Data Types.........................................................................22
Debugger.............................................................................18
Denormalized Values.......................................................35
Division By Zero Exception.............................................45
Dot Product.......................................................................340
Double Precision............................................................30
Doublet............................................................................24
E
eabsf..............................................................................196
eaddf..............................................................................197
econ.................................................................................267
econc...............................................................................270
econf.................................................................................274
econ.................................................................................272
ecomm..............................................................................269
econu.................................................................................268
econx.................................................................................277
econx..........................................................280
edeflatef..........................................................................285
ediv................................................................................258
edivf................................................................................259
ediu..................................................................................260
eextract..........................................................................292
eextracti..........................................................................289
eextractiu..........................................................................290
eextractm..........................................................................293
extractx..........................................................................291
efloatf..............................................................................287
einfltf..............................................................................286
Element............................................................................340
Element Size.....................................................................51, 340
elogmost..........................................................................297
elogmostu.........................................................................298
emul.................................................................................204
emuladdc.........................................................................216
Appendix B

H
Half Precision .................................................. 30
Hexlet ......................................................... 24
hexlet_t ......................................................... 26

I
Imaginary Part of Complex Vector ...................... 84
imagpart ......................................................... 84
Immediate ..................................................... 343
Inexact .......................................................... 343
Inexact Exception .............................................. 46
Infinite Values ............................................... 35
Inner Product ................................................ 343
Inplace .......................................................... 343
instruction ....................................................... 51
interleave ...................................................... 76
Invalid Operation Exception ......................... 45

Appendix B
sv
Swizzle Copy.................................180
Swizzle Reverse............................181, 182
Symbol......................................346
Syntax Format................................58
T
Trap.............................................346
Triclet........................................24
U
Underflow....................................346
Underflow Exception.......................46
unordered.................................42
V
v1_t........................................26, 27
v128_t........................................26, 27
v16_t..........................................26, 27
v2_t...........................................26, 27
v32_t..........................................26, 27
v4_t...........................................26, 27
v64_t..........................................26, 27
v8_t...........................................26, 27
vall.........................................64
variables...................................53
vc16_t.......................................26
vc32_t.......................................26
vc64_t.......................................26
vc8_t.........................................26
vcf16_t.....................................34
vcf32_t.....................................34
vcf64_t.....................................34
VCONST.....................................62
Vector.......................................51, 346
vector......................................63
Vector functions.........................3, 51
vector-element size......................22, 29
evlem..........................................65
vf128_t.....................................32
vf16_t.......................................33
vf32_t.......................................33
vf64_t.......................................32
vget..........................................72
vgetu.........................................72
void*.........................................22
vput..........................................73
W
Wide Crossbar.............................314
Wide Multiply Matrix.....................317
Wide Table Lookup.......................311
wmulmat.....................................318
wmulmatc....................................321
wmulmatcf..................................324
wmulmatf....................................323
wmulmatg....................................328
wmulmatp....................................326
wmulmatu....................................319
wmulmatx....................................335
wmulmatxi...................................331
wmulmatxic..................................333
wsswitch....................................315
wtranslate.................................312
X
xcompress....................................165
xcompressi..................................165
xcompressiu................................166
xcompressu..................................166
xcopy........................................180
xdeposit....................................170
xdepositm....................................172
xdepositu....................................171
xexpand......................................167
xexpandi.....................................167
xexpandiu...................................168
xexpandu....................................168
xextract.....................................190, 193
xextractm....................................191
xextractx.....................................189
xfieldreverse..............................182
xreverse.....................................181
xrotl.........................................155
xroli.........................................155
xrotr.........................................156
xrotri.........................................156
xselect......................................187
xshl...........................................157
xshli..........................................157
xshlio........................................158
xshliuo......................................159
xshlm.........................................160
xshlim........................................160
xshlo..........................................158
xshluo........................................159
xshr...........................................161
xshri..........................................161
xshriu.........................................162
xshrm.........................................163
xshrmi........................................163
xshru..........................................162

Appendix B

BroadMX C/C++ Functions
xshuffle... ................................................... 176
xshuffl Sor ................................................... 177
xswizzle... ................................................... 183
xwithdraw... ............................................... 173
xwithdrawu..................................................... 174
Z
Zero Rounding... ........................................ 347
Zero Values... ............................................ 35