

ARM64 Hardware Intrinsic APIs in .NET

A handbook with C# code samples and ARM64 assembly code

MIT License

Copyright (c) 2020 Kunal Pathak

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Table of Contents

| | |
|--|----|
| Introduction | 14 |
| 1. Abs..... | 15 |
| 2. AbsoluteCompareGreaterThan | 17 |
| 3. AbsoluteCompareGreaterThanOrEqual..... | 19 |
| 4. AbsoluteCompareGreaterThanOrEqualScalar | 21 |
| 5. AbsoluteCompareGreaterThanScalar | 23 |
| 6. AbsoluteCompareLessThan | 25 |
| 7. AbsoluteCompareLessThanOrEqual..... | 27 |
| 8. AbsoluteCompareLessThanOrEqualScalar..... | 29 |
| 9. AbsoluteCompareLessThanScalar | 31 |
| 10. AbsoluteDifference | 33 |
| 11. AbsoluteDifferenceAdd | 35 |
| 12. AbsoluteDifferenceScalar | 37 |
| 13. AbsoluteDifferenceWideningLower..... | 38 |
| 14. AbsoluteDifferenceWideningLowerAndAdd | 40 |
| 15. AbsoluteDifferenceWideningUpper | 42 |
| 16. AbsoluteDifferenceWideningUpperAndAdd..... | 44 |
| 17. AbsSaturate..... | 46 |
| 18. AbsSaturateScalar | 47 |
| 19. AbsScalar | 48 |
| 20. Add | 49 |
| 21. AddAcross..... | 51 |
| 22. AddAcrossWidening | 52 |
| 23. AddHighNarrowingLower | 54 |
| 24. AddHighNarrowingUpper..... | 56 |
| 25. AddPairwise..... | 58 |
| 26. AddPairwiseScalar | 60 |
| 27. AddPairwiseWidening..... | 61 |
| 28. AddPairwiseWideningAndAdd..... | 62 |
| 29. AddPairwiseWideningAndAddScalar..... | 64 |
| 30. AddPairwiseWideningScalar | 65 |
| 31. AddRoundedHighNarrowingLower | 66 |

| | |
|---|-----|
| 32. AddRoundedHighNarrowingUpper | 68 |
| 33. AddSaturate | 70 |
| 34. AddSaturateScalar | 72 |
| 35. AddScalar | 74 |
| 36. AddWideningLower | 75 |
| 37. AddWideningUpper | 77 |
| 38. And | 79 |
| 39. BitwiseClear | 81 |
| 40. BitwiseSelect | 83 |
| 41. Ceiling | 85 |
| 42. CeilingScalar | 86 |
| 43. CompareEqual | 87 |
| 44. CompareEqualScalar | 89 |
| 45. CompareGreaterThan | 91 |
| 46. CompareGreaterThanOrEqual | 93 |
| 47. CompareGreaterThanOrEqualScalar | 95 |
| 48. CompareGreaterThanScalar | 97 |
| 49. CompareLessThan | 99 |
| 50. CompareLessThanOrEqual | 101 |
| 51. CompareLessThanOrEqualScalar | 103 |
| 52. CompareLessThanScalar | 105 |
| 53. CompareTest | 107 |
| 54. CompareTestScalar | 109 |
| 55. ConvertToDouble | 111 |
| 56. ConvertToDoubleScalar | 112 |
| 57. ConvertToDoubleUpper | 113 |
| 58. ConvertToInt32RoundAwayFromZero | 114 |
| 59. ConvertToInt32RoundAwayFromZeroScalar | 115 |
| 60. ConvertToInt32RoundToEven | 116 |
| 61. ConvertToInt32RoundToEvenScalar | 117 |
| 62. ConvertToInt32RoundToNegativeInfinity | 118 |
| 63. ConvertToInt32RoundToNegativeInfinityScalar | 119 |
| 64. ConvertToInt32RoundToPositiveInfinity | 120 |
| 65. ConvertToInt32RoundToPositiveInfinityScalar | 121 |

| | |
|--|-----|
| 66. ConvertToInt32RoundToZero | 122 |
| 67. ConvertToInt32RoundToZeroScalar | 123 |
| 68. ConvertToInt64RoundAwayFromZero | 124 |
| 69. ConvertToInt64RoundAwayFromZeroScalar | 125 |
| 70. ConvertToInt64RoundToEven | 126 |
| 71. ConvertToInt64RoundToEvenScalar | 127 |
| 72. ConvertToInt64RoundToNegativeInfinity | 128 |
| 73. ConvertToInt64RoundToNegativeInfinityScalar | 129 |
| 74. ConvertToInt64RoundToPositiveInfinity | 130 |
| 75. ConvertToInt64RoundToPositiveInfinityScalar | 131 |
| 76. ConvertToInt64RoundToZero | 132 |
| 77. ConvertToInt64RoundToZeroScalar | 133 |
| 78. ConvertToSingle | 134 |
| 79. ConvertToSingleLower..... | 135 |
| 80. ConvertToSingleRoundToOddLower | 136 |
| 81. ConvertToSingleRoundToOddUpper..... | 137 |
| 82. ConvertToSingleScalar | 138 |
| 83. ConvertToSingleUpper | 139 |
| 84. ConvertToUInt32RoundAwayFromZero | 140 |
| 85. ConvertToUInt32RoundAwayFromZeroScalar | 141 |
| 86. ConvertToUInt32RoundToEven | 142 |
| 87. ConvertToUInt32RoundToEvenScalar | 143 |
| 88. ConvertToUInt32RoundToNegativeInfinity | 144 |
| 89. ConvertToUInt32RoundToNegativeInfinityScalar | 145 |
| 90. ConvertToUInt32RoundToPositiveInfinity | 146 |
| 91. ConvertToUInt32RoundToPositiveInfinityScalar | 147 |
| 92. ConvertToUInt32RoundToZero | 148 |
| 93. ConvertToUInt32RoundToZeroScalar | 149 |
| 94. ConvertToUInt64RoundAwayFromZero | 150 |
| 95. ConvertToUInt64RoundAwayFromZeroScalar | 151 |
| 96. ConvertToUInt64RoundToEven | 152 |
| 97. ConvertToUInt64RoundToEvenScalar | 153 |
| 98. ConvertToUInt64RoundToNegativeInfinity | 154 |
| 99. ConvertToUInt64RoundToNegativeInfinityScalar | 155 |

| | |
|---|-----|
| 100. ConvertToUInt64RoundToPositiveInfinity | 156 |
| 101. ConvertToUInt64RoundToPositiveInfinityScalar | 157 |
| 102. ConvertToUInt64RoundToZero | 158 |
| 103. ConvertToUInt64RoundToZeroScalar | 159 |
| 104. Divide | 160 |
| 105. DivideScalar | 161 |
| 106. DuplicateSelectedScalarToVector128..... | 162 |
| 107. DuplicateSelectedScalarToVector64..... | 164 |
| 108. DuplicateToVector128..... | 166 |
| 109. DuplicateToVector64..... | 168 |
| 110. Extract..... | 169 |
| 111. ExtractNarrowingLower | 171 |
| 112. ExtractNarrowingSaturateLower | 172 |
| 113. ExtractNarrowingSaturateScalar..... | 173 |
| 114. ExtractNarrowingSaturateUnsignedLower | 174 |
| 115. ExtractNarrowingSaturateUnsignedScalar | 175 |
| 116. ExtractNarrowingSaturateUnsignedUpper | 176 |
| 117. ExtractNarrowingSaturateUpper | 178 |
| 118. ExtractNarrowingUpper | 180 |
| 119. ExtractVector128 | 182 |
| 120. ExtractVector64 | 184 |
| 121. Floor..... | 186 |
| 122. FloorScalar..... | 187 |
| 123. FusedAddHalving..... | 188 |
| 124. FusedAddRoundedHalving..... | 190 |
| 125. FusedMultiplyAdd..... | 192 |
| 126. FusedMultiplyAddByScalar | 194 |
| 127. FusedMultiplyAddBySelectedScalar | 196 |
| 128. FusedMultiplyAddNegatedScalar | 198 |
| 129. FusedMultiplyAddScalar..... | 200 |
| 130. FusedMultiplyAddScalarBySelectedScalar | 202 |
| 131. FusedMultiplySubtract | 204 |
| 132. FusedMultiplySubtractByScalar | 206 |
| 133. FusedMultiplySubtractBySelectedScalar | 208 |

| | |
|--|-----|
| 134. FusedMultiplySubtractNegatedScalar | 210 |
| 135. FusedMultiplySubtractScalar | 212 |
| 136. FusedMultiplySubtractScalarBySelectedScalar | 214 |
| 137. FusedSubtractHalving..... | 216 |
| 138. Insert | 218 |
| 139. InsertScalar | 220 |
| 140. InsertSelectedScalar..... | 222 |
| 141. LeadingSignCount | 225 |
| 142. LeadingZeroCount..... | 226 |
| 143. LoadAndInsertScalar | 228 |
| 144. LoadAndReplicateToVector128 | 230 |
| 145. LoadAndReplicateToVector64..... | 231 |
| 146. LoadVector128 | 232 |
| 147. LoadVector64..... | 233 |
| 148. Max | 234 |
| 149. MaxAcross..... | 236 |
| 150. MaxNumber..... | 237 |
| 151. MaxNumberAcross | 239 |
| 152. MaxNumberPairwise | 240 |
| 153. MaxNumberPairwiseScalar | 242 |
| 154. MaxNumberScalar..... | 243 |
| 155. MaxPairwise..... | 244 |
| 156. MaxPairwiseScalar..... | 246 |
| 157. MaxScalar | 247 |
| 158. Min..... | 248 |
| 159. MinAcross..... | 250 |
| 160. MinNumber | 251 |
| 161. MinNumberAcross..... | 253 |
| 162. MinNumberPairwise..... | 254 |
| 163. MinNumberPairwiseScalar | 256 |
| 164. MinNumberScalar | 257 |
| 165. MinPairwise | 258 |
| 166. MinPairwiseScalar | 260 |
| 167. MinScalar..... | 261 |

| | |
|---|-----|
| 168. Multiply | 262 |
| 169. MultiplyAdd..... | 264 |
| 170. MultiplyAddByScalar | 266 |
| 171. MultiplyAddBySelectedScalar | 268 |
| 172. MultiplyByScalar..... | 270 |
| 173. MultiplyBySelectedScalar | 272 |
| 174. MultiplyBySelectedScalarWideningLower | 274 |
| 175. MultiplyBySelectedScalarWideningLowerAndAdd | 276 |
| 176. MultiplyBySelectedScalarWideningLowerAndSubtract..... | 278 |
| 177. MultiplyBySelectedScalarWideningUpper | 280 |
| 178. MultiplyBySelectedScalarWideningUpperAndAdd..... | 282 |
| 179. MultiplyBySelectedScalarWideningUpperAndSubtract | 284 |
| 180. MultiplyDoublingByScalarSaturateHigh | 286 |
| 181. MultiplyDoublingBySelectedScalarSaturateHigh | 288 |
| 182. MultiplyDoublingSaturateHigh..... | 290 |
| 183. MultiplyDoublingSaturateHighScalar..... | 292 |
| 184. MultiplyDoublingScalarBySelectedScalarSaturateHigh | 293 |
| 185. MultiplyDoublingWideningAndAddSaturateScalar | 295 |
| 186. MultiplyDoublingWideningAndSubtractSaturateScalar | 297 |
| 187. MultiplyDoublingWideningLowerAndAddSaturate | 299 |
| 188. MultiplyDoublingWideningLowerAndSubtractSaturate..... | 301 |
| 189. MultiplyDoublingWideningLowerByScalarAndAddSaturate | 303 |
| 190. MultiplyDoublingWideningLowerByScalarAndSubtractSaturate | 305 |
| 191. MultiplyDoublingWideningLowerBySelectedScalarAndAddSaturate..... | 307 |
| 192. MultiplyDoublingWideningLowerBySelectedScalarAndSubtractSaturate | 309 |
| 193. MultiplyDoublingWideningSaturateLower | 311 |
| 194. MultiplyDoublingWideningSaturateLowerByScalar | 313 |
| 195. MultiplyDoublingWideningSaturateLowerBySelectedScalar | 315 |
| 196. MultiplyDoublingWideningSaturateScalar | 317 |
| 197. MultiplyDoublingWideningSaturateScalarBySelectedScalar | 318 |
| 198. MultiplyDoublingWideningSaturateUpper | 320 |
| 199. MultiplyDoublingWideningSaturateUpperByScalar | 321 |
| 200. MultiplyDoublingWideningSaturateUpperBySelectedScalar | 323 |
| 201. MultiplyDoublingWideningScalarBySelectedScalarAndAddSaturate | 325 |

| | |
|--|-----|
| 202. MultiplyDoublingWideningScalarBySelectedScalarAndSubtractSaturate | 327 |
| 203. MultiplyDoublingWideningUpperAndAddSaturate | 329 |
| 204. MultiplyDoublingWideningUpperAndSubtractSaturate | 331 |
| 205. MultiplyDoublingWideningUpperByScalarAndAddSaturate | 333 |
| 206. MultiplyDoublingWideningUpperByScalarAndSubtractSaturate | 335 |
| 207. MultiplyDoublingWideningUpperBySelectedScalarAndAddSaturate | 337 |
| 208. MultiplyDoublingWideningUpperBySelectedScalarAndSubtractSaturate..... | 339 |
| 209. MultiplyExtended | 341 |
| 210. MultiplyExtendedByScalar | 343 |
| 211. MultiplyExtendedBySelectedScalar | 344 |
| 212. MultiplyExtendedScalar | 346 |
| 213. MultiplyExtendedScalarBySelectedScalar | 347 |
| 214. MultiplyRoundedDoublingByScalarSaturateHigh | 349 |
| 215. MultiplyRoundedDoublingBySelectedScalarSaturateHigh | 351 |
| 216. MultiplyRoundedDoublingSaturateHigh | 353 |
| 217. MultiplyRoundedDoublingSaturateHighScalar | 355 |
| 218. MultiplyRoundedDoublingScalarBySelectedScalarSaturateHigh | 357 |
| 219. MultiplyScalar | 359 |
| 220. MultiplyScalarBySelectedScalar | 360 |
| 221. MultiplySubtract | 362 |
| 222. MultiplySubtractByScalar | 364 |
| 223. MultiplySubtractBySelectedScalar | 366 |
| 224. MultiplyWideningLower | 368 |
| 225. MultiplyWideningLowerAndAdd | 370 |
| 226. MultiplyWideningLowerAndSubtract | 372 |
| 227. MultiplyWideningUpper | 374 |
| 228. MultiplyWideningUpperAndAdd | 376 |
| 229. MultiplyWideningUpperAndSubtract | 378 |
| 230. Negate | 380 |
| 231. NegateSaturate | 381 |
| 232. NegateSaturateScalar | 382 |
| 233. NegateScalar | 383 |
| 234. Not | 384 |
| 235. Or | 386 |

| | |
|---|-----|
| 236. OrNot..... | 388 |
| 237. PolynomialMultiply | 390 |
| 238. PolynomialMultiplyWideningLower | 392 |
| 239. PolynomialMultiplyWideningUpper | 393 |
| 240. PopCount | 394 |
| 241. ReciprocalEstimate..... | 395 |
| 242. ReciprocalEstimateScalar | 396 |
| 243. ReciprocalExponentScalar | 397 |
| 244. ReciprocalSquareRootEstimate..... | 398 |
| 245. ReciprocalSquareRootEstimateScalar | 399 |
| 246. ReciprocalSquareRootStep | 400 |
| 247. ReciprocalSquareRootStepScalar | 402 |
| 248. ReciprocalStep | 403 |
| 249. ReciprocalStepScalar | 405 |
| 250. ReverseElement16..... | 406 |
| 251. ReverseElement32..... | 407 |
| 252. ReverseElement8 | 408 |
| 253. ReverseElementBits..... | 409 |
| 254. RoundAwayFromZero..... | 410 |
| 255. RoundAwayFromZeroScalar | 411 |
| 256. RoundToNearest..... | 412 |
| 257. RoundToNearestScalar | 413 |
| 258. RoundToNegativeInfinity | 414 |
| 259. RoundToNegativeInfinityScalar | 415 |
| 260. RoundToPositiveInfinity..... | 416 |
| 261. RoundToPositiveInfinityScalar | 417 |
| 262. RoundToZero..... | 418 |
| 263. RoundToZeroScalar..... | 419 |
| 264. ShiftArithmetic | 420 |
| 265. ShiftArithmeticRounded | 422 |
| 266. ShiftArithmeticRoundedSaturate | 424 |
| 267. ShiftArithmeticRoundedSaturateScalar | 426 |
| 268. ShiftArithmeticRoundedScalar | 428 |
| 269. ShiftArithmeticSaturate | 429 |

| | |
|--|-----|
| 270. ShiftArithmeticSaturateScalar | 431 |
| 271. ShiftArithmeticScalar | 433 |
| 272. ShiftLeftAndInsert..... | 434 |
| 273. ShiftLeftAndInsertScalar..... | 436 |
| 274. ShiftLeftLogical..... | 438 |
| 275. ShiftLeftLogicalSaturate..... | 440 |
| 276. ShiftLeftLogicalSaturateScalar..... | 442 |
| 277. ShiftLeftLogicalSaturateUnsigned..... | 444 |
| 278. ShiftLeftLogicalSaturateUnsignedScalar..... | 446 |
| 279. ShiftLeftLogicalScalar..... | 448 |
| 280. ShiftLeftLogicalWideningLower | 449 |
| 281. ShiftLeftLogicalWideningUpper..... | 451 |
| 282. ShiftLogical..... | 453 |
| 283. ShiftLogicalRounded..... | 455 |
| 284. ShiftLogicalRoundedSaturate..... | 457 |
| 285. ShiftLogicalRoundedSaturateScalar | 459 |
| 286. ShiftLogicalRoundedScalar | 461 |
| 287. ShiftLogicalSaturate | 462 |
| 288. ShiftLogicalSaturateScalar | 464 |
| 289. ShiftLogicalScalar | 466 |
| 290. ShiftRightAndInsert | 467 |
| 291. ShiftRightAndInsertScalar | 469 |
| 292. ShiftRightArithmetic..... | 471 |
| 293. ShiftRightArithmeticAdd | 472 |
| 294. ShiftRightArithmeticAddScalar | 474 |
| 295. ShiftRightArithmeticNarrowingSaturateLower | 475 |
| 296. ShiftRightArithmeticNarrowingSaturateScalar | 477 |
| 297. ShiftRightArithmeticNarrowingSaturateUnsignedLower..... | 479 |
| 298. ShiftRightArithmeticNarrowingSaturateUnsignedScalar | 481 |
| 299. ShiftRightArithmeticNarrowingSaturateUnsignedUpper | 483 |
| 300. ShiftRightArithmeticNarrowingSaturateUpper | 485 |
| 301. ShiftRightArithmeticRounded..... | 487 |
| 302. ShiftRightArithmeticRoundedAdd | 489 |
| 303. ShiftRightArithmeticRoundedAddScalar | 491 |

| | |
|---|-----|
| 304. ShiftRightArithmeticRoundedNarrowingSaturateLower | 492 |
| 305. ShiftRightArithmeticRoundedNarrowingSaturateScalar | 494 |
| 306. ShiftRightArithmeticRoundedNarrowingSaturateUnsignedLower | 496 |
| 307. ShiftRightArithmeticRoundedNarrowingSaturateUnsignedScalar | 498 |
| 308. ShiftRightArithmeticRoundedNarrowingSaturateUnsignedUpper | 500 |
| 309. ShiftRightArithmeticRoundedNarrowingSaturateUpper | 502 |
| 310. ShiftRightArithmeticRoundedScalar | 504 |
| 311. ShiftRightArithmeticScalar | 505 |
| 312. ShiftRightLogical | 506 |
| 313. ShiftRightLogicalAdd | 508 |
| 314. ShiftRightLogicalAddScalar | 510 |
| 315. ShiftRightLogicalNarrowingLower | 511 |
| 316. ShiftRightLogicalNarrowingSaturateLower | 513 |
| 317. ShiftRightLogicalNarrowingSaturateScalar | 515 |
| 318. ShiftRightLogicalNarrowingSaturateUpper | 517 |
| 319. ShiftRightLogicalNarrowingUpper | 519 |
| 320. ShiftRightLogicalRounded | 521 |
| 321. ShiftRightLogicalRoundedAdd | 523 |
| 322. ShiftRightLogicalRoundedAddScalar | 525 |
| 323. ShiftRightLogicalRoundedNarrowingLower | 526 |
| 324. ShiftRightLogicalRoundedNarrowingSaturateLower | 528 |
| 325. ShiftRightLogicalRoundedNarrowingSaturateScalar | 530 |
| 326. ShiftRightLogicalRoundedNarrowingSaturateUpper | 532 |
| 327. ShiftRightLogicalRoundedNarrowingUpper | 534 |
| 328. ShiftRightLogicalRoundedScalar | 536 |
| 329. ShiftRightLogicalScalar | 537 |
| 330. SignExtendWideningLower | 538 |
| 331. SignExtendWideningUpper | 539 |
| 332. Sqrt | 540 |
| 333. SqrtScalar | 541 |
| 334. Store | 542 |
| 335. StorePair | 544 |
| 336. StorePairNonTemporal | 546 |
| 337. StorePairScalar | 548 |

| | |
|--|-----|
| 338. StorePairScalarNonTemporal | 549 |
| 339. StoreSelectedScalar | 550 |
| 340. Subtract..... | 552 |
| 341. SubtractHighNarrowingLower | 554 |
| 342. SubtractHighNarrowingUpper..... | 556 |
| 343. SubtractRoundedHighNarrowingLower | 558 |
| 344. SubtractRoundedHighNarrowingUpper | 560 |
| 345. SubtractSaturate | 562 |
| 346. SubtractSaturateScalar | 564 |
| 347. SubtractScalar | 566 |
| 348. SubtractWideningLower..... | 567 |
| 349. SubtractWideningUpper | 569 |
| 350. TransposeEven | 571 |
| 351. TransposeOdd..... | 573 |
| 352. UnzipEven | 575 |
| 353. UnzipOdd..... | 577 |
| 354. VectorTableLookup | 579 |
| 355. VectorTableLookupExtension..... | 581 |
| 356. Xor..... | 583 |
| 357. ZeroExtendWideningLower | 585 |
| 358. ZeroExtendWideningUpper | 586 |
| 359. ZipHigh | 587 |
| 360. ZipLow..... | 589 |

Introduction

In my [vectorization using .NET APIs](#) blog, I have described the SIMD datatypes `Vector64<T>` and `Vector128<T>` that operates on 'ARM64 hardware intrinsic' APIs present under [System.Runtime.Intrinsics.Arm.Arm64](#) and [System.Runtime.Intrinsics.Arm.Arm64](#) class. In this handbook, I will describe those hardware intrinsic APIs by showing sample code usage along with examples and generated ARM64 code. This will help people in understanding these APIs so they can use them to optimize their .NET code written to target ARM64.

Most of the description of these APIs is adapted and referenced from [Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile document](#). You can also refer to the description of SIMD and Floating-point instructions description at [Arm developer docs page](#).

1. Abs

Vector64<ushort> Abs(Vector64<short> value)

This method calculates the absolute value of each vector element value, stores in a result vector and returns the result vector.

```
private Vector64<ushort> AbsTest(Vector64<short> value)
{
    return AdvSimd.Abs(value);
}
// value = <-11, -12, -13, 14>
// Result = <11, 12, 13, 14>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<uint> Abs(Vector64<int> value)
Vector64<byte> Abs(Vector64<sbyte> value)
Vector64<float> Abs(Vector64<float> value)
Vector128<ushort> Abs(Vector128<short> value)
Vector128<uint> Abs(Vector128<int> value)
Vector128<byte> Abs(Vector128<sbyte> value)
Vector128<float> Abs(Vector128<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Abs(Vector128<double> value)
Vector128<ulong> Abs(Vector128<long> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsTest(System.Runtime.Intrinsics.Vector64`1[Int16]):System.Ru
ntime.Intrinsics.Vector64`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    abs    v16.4h, v0.4h
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


2. AbsoluteCompareGreaterThan

Vector64<float> AbsoluteCompareGreaterThan(Vector64<float> left, Vector64<float> right)

This method performs comparison of absolute value of corresponding vector elements in left with those of right vector and if the left's value is greater than the right's value, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<float> AbsoluteCompareGreaterThanTest(Vector64<float> left,
Vector64<float> right)
{
    return AdvSimd.AbsoluteCompareGreaterThan(left, right);
}
// left = <-11.5f, -12.5f>
// right = <10.5f, -22.5f>
// Result = <NaN, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> AbsoluteCompareGreaterThan(Vector128<float> left,
Vector128<float> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> AbsoluteCompareGreaterThan(Vector128<double> left,
Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteCompareGreaterThanTest(System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    facgt  v16.2s, v0.2s, v1.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

3. AbsoluteCompareGreaterThanOrEqual

Vector64<float> AbsoluteCompareGreaterThanOrEqual(Vector64<float> left, Vector64<float> right)

This method performs comparison of absolute value of corresponding vector elements in left with those of right vector and if the left's value is greater than or equal to the right's value, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<float> AbsoluteCompareGreaterThanOrEqualTest(Vector64<float>
left, Vector64<float> right)
{
    return AdvSimd.AbsoluteCompareGreaterThanOrEqual(left, right);
}
// left = <-11.5f, -12.5f>
// right = <11.5f, -22.5f>
// Result = <NaN, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> AbsoluteCompareGreaterThanOrEqual(Vector128<float> left,
Vector128<float> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> AbsoluteCompareGreaterThanOrEqual(Vector128<double> left,
Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteCompareGreaterThanOrEqualTest(System.Runtime.Intrinsic
s.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Run
time.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    facge  v16.2s, v0.2s, v1.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

4. AbsoluteCompareGreaterThanOrEqualScalar

Vector64<double> AbsoluteCompareGreaterThanOrEqualScalar(Vector64<double> left, Vector64<double> right)

This method compares the absolute value of corresponding vector elements of left and right vector and if the left's element value is greater than or equal to the right's element value, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double>
AbsoluteCompareGreaterThanOrEqualScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.AbsoluteCompareGreaterThanOrEqualScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <NaN>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<float> AbsoluteCompareGreaterThanOrEqualScalar(Vector64<float> left,
Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteCompareGreaterThanOrEqualScalarTest(System.Runtime.Intrinsics.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    facge  d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

5. AbsoluteCompareGreaterThanScalar

Vector64<double> AbsoluteCompareGreaterThanScalar(Vector64<double> left, Vector64<double> right)

This method compares the absolute value of corresponding vector elements of left and right vector and if the left's element value is greater than the right's element value, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double>
AbsoluteCompareGreaterThanScalarTest(Vector64<double> left, Vector64<double>
right)
{
    return AdvSimd.Arm64.AbsoluteCompareGreaterThanScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> AbsoluteCompareGreaterThanScalar(Vector64<float> left,
Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteCompareGreaterThanScalarTest(System.Runtime.Intrinsics
.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runt
ime.Intrinsics.Vector64`1[Double]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    facgt  d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

6. AbsoluteCompareLessThan

Vector64<float> AbsoluteCompareLessThan(Vector64<float> left, Vector64<float> right)

This method performs comparison of absolute value of corresponding vector elements in left and right vector and if the left's value is less than to the right's value, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<float> AbsoluteCompareLessThanTest(Vector64<float> left,
Vector64<float> right)
{
    return AdvSimd.AbsoluteCompareLessThan(left, right);
}
// left = <-11.5f, -12.5f>
// right = <10.5f, -22.5f>
// Result = <0, NaN>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> AbsoluteCompareLessThan(Vector128<float> left,
Vector128<float> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> AbsoluteCompareLessThan(Vector128<double> left,
Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteCompareLessThanTest(System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    facgt  v16.2s, v1.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

7. AbsoluteCompareLessThanOrEqual

Vector64<float> AbsoluteCompareLessThanOrEqual(Vector64<float> left, Vector64<float> right)

This method performs comparison of absolute value of each vector element in left with the absolute value of the corresponding vector element in right and if the left's value is less than or equal to the right's value, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<float> AbsoluteCompareLessThanOrEqualTest(Vector64<float>
left, Vector64<float> right)
{
    return AdvSimd.AbsoluteCompareLessThanOrEqual(left, right);
}
// left = <-11.5f, -12.5f>
// right = <11.5f, -22.5f>
// Result = <0, NaN>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> AbsoluteCompareLessThanOrEqual(Vector128<float> left,
Vector128<float> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> AbsoluteCompareLessThanOrEqual(Vector128<double> left,
Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteCompareLessThanOrEqualTest(System.Runtime.Intrinsics.V
ector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtim
e.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    facge  v16.2s, v1.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

8. AbsoluteCompareLessThanOrEqualScalar

Vector64<double> AbsoluteCompareLessThanOrEqualScalar(Vector64<double> left, Vector64<double> right)

This method compares the absolute value of corresponding vector elements of left and right vector and if the left's element value is less than or equal to the right's element value, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double>
AbsoluteCompareLessThanOrEqualScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.AbsoluteCompareLessThanOrEqualScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <NaN>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Avsimd.Arm64
Vector64<float> AbsoluteCompareLessThanOrEqualScalar(Vector64<float> left,
Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteCompareLessThanOrEqualScalarTest(System.Runtime.Intrin
sics.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.
Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    facge  d16, d1, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

9. AbsoluteCompareLessThanScalar

Vector64<double> AbsoluteCompareLessThanScalar(Vector64<double> left, Vector64<double> right)

This method compares the absolute value of corresponding vector elements of left and right vector and if the left's element value is less than the right's element value, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double> AbsoluteCompareLessThanScalarTest(Vector64<double>
left, Vector64<double> right)
{
    return AdvSimd.Arm64.AbsoluteCompareLessThanScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> AbsoluteCompareLessThanScalar(Vector64<float> left,
Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteCompareLessThanScalarTest(System.Runtime.Intrinsics.Ve
ctor64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime
.Intrinsics.Vector64`1[Double]
;
; V00 arg0      [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
; V01 arg1      [V01,T01] (  3,  3  )  simd8  ->  d1
HFA(simd8)
;# V02 OutArgs  [V02   ] (  1,  1  )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    facgt  d16, d1, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

10. AbsoluteDifference

Vector64<byte> AbsoluteDifference(Vector64<byte> left, Vector64<byte> right)

This method subtracts the corresponding vector elements of right vector from those of left vector, places the absolute values of the results in a result vector, and writes the vector to the result vector.

```
private Vector64<byte> AbsoluteDifferenceTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.AbsoluteDifference(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 37, 17>
// Result = <10, 10, 10, 10, 10, 10, 20, 1>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ushort> AbsoluteDifference(Vector64<short> left, Vector64<short>
right)
Vector64<uint> AbsoluteDifference(Vector64<int> left, Vector64<int> right)
Vector64<byte> AbsoluteDifference(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector64<float> AbsoluteDifference(Vector64<float> left, Vector64<float>
right)
Vector64<ushort> AbsoluteDifference(Vector64<ushort> left, Vector64<ushort>
right)
Vector64<uint> AbsoluteDifference(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> AbsoluteDifference(Vector128<byte> left, Vector128<byte>
right)
Vector128<ushort> AbsoluteDifference(Vector128<short> left, Vector128<short>
right)
Vector128<uint> AbsoluteDifference(Vector128<int> left, Vector128<int> right)
Vector128<byte> AbsoluteDifference(Vector128<sbyte> left, Vector128<sbyte>
right)
Vector128<float> AbsoluteDifference(Vector128<float> left, Vector128<float>
right)
Vector128<ushort> AbsoluteDifference(Vector128<ushort> left,
Vector128<ushort> right)
Vector128<uint> AbsoluteDifference(Vector128<uint> left, Vector128<uint>
right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> AbsoluteDifference(Vector128<double> left,
Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:AbsoluteDifferenceTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
;# V02 OutArgs      [V02   ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uabd    v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8

```

11. AbsoluteDifferenceAdd

Vector64<byte> AbsoluteDifferenceAdd(Vector64<byte> addend, Vector64<byte> left, Vector64<byte> right)

This method subtracts the corresponding vector elements of the right vector from those of left vector, and accumulates the absolute values of the results along with the values of addend and returns the accumulated result.

```
private Vector64<byte> AbsoluteDifferenceAddTest(Vector64<byte> addend,
Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.AbsoluteDifferenceAdd(addend, left, right);
}
// addend = <11, 12, 13, 14, 15, 16, 17, 18>
// left = <21, 52, 23, 24, 25, 26, 27, 28>
// right = <41, 32, 33, 34, 35, 36, 37, 38>
// Result = <31, 32, 23, 24, 25, 26, 27, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> AbsoluteDifferenceAdd(Vector64<short> addend, Vector64<short>
left, Vector64<short> right)
Vector64<int> AbsoluteDifferenceAdd(Vector64<int> addend, Vector64<int> left,
Vector64<int> right)
Vector64<sbyte> AbsoluteDifferenceAdd(Vector64<sbyte> addend, Vector64<sbyte>
left, Vector64<sbyte> right)
Vector64<ushort> AbsoluteDifferenceAdd(Vector64<ushort> addend,
Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> AbsoluteDifferenceAdd(Vector64<uint> addend, Vector64<uint>
left, Vector64<uint> right)
Vector128<byte> AbsoluteDifferenceAdd(Vector128<byte> addend, Vector128<byte>
left, Vector128<byte> right)
Vector128<short> AbsoluteDifferenceAdd(Vector128<short> addend,
Vector128<short> left, Vector128<short> right)
Vector128<int> AbsoluteDifferenceAdd(Vector128<int> addend, Vector128<int>
left, Vector128<int> right)
Vector128<sbyte> AbsoluteDifferenceAdd(Vector128<sbyte> addend,
Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<ushort> AbsoluteDifferenceAdd(Vector128<ushort> addend,
Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> AbsoluteDifferenceAdd(Vector128<uint> addend, Vector128<uint>
left, Vector128<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteDifferenceAddTest(System.Runtime.Intrinsics.Vector64`1
```

```

[Byte],System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.V
ector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 )   simd8  ->  d2
HFA(simd8)
;# V03 OutArgs      [V03      ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uaba   v0.8b, v1.8b, v2.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

12. AbsoluteDifferenceScalar

Vector64<double> AbsoluteDifferenceScalar(Vector64<double> left, Vector64<double> right)

This method subtracts the floating-point values in the elements of the right vector from that of left vector, stores the absolute value of into a result vector, and returns the result vector.

```
private Vector64<double> AbsoluteDifferenceScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.AbsoluteDifferenceScalar(left, right);
}
// left = <11.5>
// right = <16.5>
// Result = <5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> AbsoluteDifferenceScalar(Vector64<float> left,
Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteDifferenceScalarTest(System.Runtime.Intrinsics.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fabd   d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

13. AbsoluteDifferenceWideningLower

Vector128<ushort> AbsoluteDifferenceWideningLower(Vector64<byte> left, Vector64<byte> right)

This method subtracts the corresponding vector elements in the right from those of left and places the absolute value in a result vector and returns the result vector. The result vector Vector128<ushort> as seen in below example is twice the size of input parameter Vector4<byte>.

```
private Vector128<ushort> AbsoluteDifferenceWideningLowerTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.AbsoluteDifferenceWideningLower(left, right);
}
// left = <11, 2, 113, 104, 180, 11, 120, 121>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <10, 20, 90, 80, 155, 15, 93, 93>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<uint> AbsoluteDifferenceWideningLower(Vector64<short> left, Vector64<short> right)
Vector128<ulong> AbsoluteDifferenceWideningLower(Vector64<int> left, Vector64<int> right)
Vector128<ushort> AbsoluteDifferenceWideningLower(Vector64<sbyte> left, Vector64<sbyte> right)
Vector128<uint> AbsoluteDifferenceWideningLower(Vector64<ushort> left, Vector64<ushort> right)
Vector128<ulong> AbsoluteDifferenceWideningLower(Vector64<uint> left, Vector64<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteDifferenceWideningLowerTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1 [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
```

```
uabdl    v16.8h, v0.8b, v1.8b
mov      v0.16b, v16.16b
ldp      fp, lr, [sp],#16
ret      lr
```

; Total bytes of code 24, prolog size 8

14. AbsoluteDifferenceWideningLowerAndAdd

Vector128<ushort> AbsoluteDifferenceWideningLowerAndAdd(Vector128<ushort> addend, Vector64<byte> left, Vector64<byte> right)

This method subtracts the corresponding vector elements of *right* from that of *left*, and accumulates the absolute values of the result along with the elements of *addend* and return the accumulated vector. The result vector *Vector128<ushort>* as seen in below example is twice the size of input parameter *Vector4<byte>*.

```
private Vector128<ushort>
AbsoluteDifferenceWideningLowerAndAddTest(Vector128<ushort> addend,
Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.AbsoluteDifferenceWideningLowerAndAdd(addend, left, right);
}
// addend = <100, 200, 300, 100, 100, 100, 100, 100>
// left = <11, 2, 113, 104, 180, 11, 120, 121>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <110, 220, 390, 180, 1155, 115, 193, 193>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> AbsoluteDifferenceWideningLowerAndAdd(Vector128<int> addend,
Vector64<short> left, Vector64<short> right)
Vector128<long> AbsoluteDifferenceWideningLowerAndAdd(Vector128<long> addend,
Vector64<int> left, Vector64<int> right)
Vector128<short> AbsoluteDifferenceWideningLowerAndAdd(Vector128<short>
addend, Vector64<sbyte> left, Vector64<sbyte> right)
Vector128<uint> AbsoluteDifferenceWideningLowerAndAdd(Vector128<uint> addend,
Vector64<ushort> left, Vector64<ushort> right)
Vector128<ulong> AbsoluteDifferenceWideningLowerAndAdd(Vector128<ulong>
addend, Vector64<uint> left, Vector64<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteDifferenceWideningLowerAndAddTest(System.Runtime.Intri
nsics.Vector128`1[UInt16],System.Runtime.Intrinsics.Vector64`1[Byte],System.R
untime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector128`1[UI
nt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
```



```

;# V03 OutArgs      [V03      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp     fp, lr, [sp,#-16]!
        mov     fp, sp
        uabal   v0.8h, v1.8b, v2.8b
        ldp     fp, lr, [sp],#16
        ret     lr

; Total bytes of code 20, prolog size 8

```

15. AbsoluteDifferenceWideningUpper

Vector128<ushort> AbsoluteDifferenceWideningUpper(Vector128<byte> left, Vector128<byte> right)

This method subtracts the corresponding vector elements in upper half of right vector from those of left, places the absolute value of the result in a result vector and returns the result vector. The size of individual element of result Vector128<ushort> as seen in below example is twice the size of input parameter Vector128<byte>.

```
private Vector128<ushort> AbsoluteDifferenceWideningUpperTest(Vector128<byte>
left, Vector128<byte> right)
{
    return AdvSimd.AbsoluteDifferenceWideningUpper(left, right);
}
// left = <11, 208, 103, 184, 180, 21, 130, 151, 31, 2, 113, 104, 180, 11,
120, 121>
// right = <21, 22, 23, 24, 25, 26, 27, 28, 20, 122, 231, 24, 25, 26, 27, 28>
// Result = <11, 120, 118, 80, 155, 15, 93, 93>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<uint> AbsoluteDifferenceWideningUpper(Vector128<short> left,
Vector128<short> right)
Vector128<ulong> AbsoluteDifferenceWideningUpper(Vector128<int> left,
Vector128<int> right)
Vector128<ushort> AbsoluteDifferenceWideningUpper(Vector128<sbyte> left,
Vector128<sbyte> right)
Vector128<uint> AbsoluteDifferenceWideningUpper(Vector128<ushort> left,
Vector128<ushort> right)
Vector128<ulong> AbsoluteDifferenceWideningUpper(Vector128<uint> left,
Vector128<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteDifferenceWideningUpperTest(System.Runtime.Intrinsics.
Vector128`1[Byte],System.Runtime.Intrinsics.Vector128`1[Byte]):System.Runtime
.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
stp fp, lr, [sp,#-16]!
```

```
mov    fp, sp
uabdl2 v16.8h, v0.16b, v1.16b
mov    v0.16b, v16.16b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

16. AbsoluteDifferenceWideningUpperAndAdd

Vector128<ushort> AbsoluteDifferenceWideningUpperAndAdd(Vector128<ushort> addend, Vector128<byte> left, Vector128<byte> right)

This method subtracts the corresponding vector elements in upper half of right from those of left, accumulates the absolute value of the result along with addended and return the accumulated vector. The size of individual element of result Vector128<ushort> as seen in below example is twice the size of input parameter Vector128<byte>.

```
private Vector128<ushort>
AbsoluteDifferenceWideningUpperAndAddTest(Vector128<ushort> addend,
Vector128<byte> left, Vector128<byte> right)
{
    return AdvSimd.AbsoluteDifferenceWideningUpperAndAdd(addend, left, right);
}
// addend = <100, 200, 300, 100, 100, 100, 100, 100>
// left = <11, 208, 103, 184, 180, 21, 130, 151, 31, 2, 113, 104, 180, 11, 120, 121>
// right = <21, 22, 23, 24, 25, 26, 27, 28, 20, 122, 231, 24, 25, 26, 27, 28>
// Result = <111, 320, 418, 180, 255, 115, 193, 193>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> AbsoluteDifferenceWideningUpperAndAdd(Vector128<int> addend,
Vector128<short> left, Vector128<short> right)
Vector128<long> AbsoluteDifferenceWideningUpperAndAdd(Vector128<long> addend,
Vector128<int> left, Vector128<int> right)
Vector128<short> AbsoluteDifferenceWideningUpperAndAdd(Vector128<short>
addend, Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<uint> AbsoluteDifferenceWideningUpperAndAdd(Vector128<uint> addend,
Vector128<ushort> left, Vector128<ushort> right)
Vector128<ulong> AbsoluteDifferenceWideningUpperAndAdd(Vector128<ulong>
addend, Vector128<uint> left, Vector128<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsoluteDifferenceWideningUpperAndAddTest(System.Runtime.Intrinsics.Vector128`1[UInt16],System.Runtime.Intrinsics.Vector128`1[Byte],System.Runtime.Intrinsics.Vector128`1[Byte]):System.Runtime.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2          [V02,T02] ( 3, 3 ) simd16 -> d2
```

```

HFA(simd16)
;# V03 OutArgs      [V03      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp      fp, lr, [sp,#-16]!
        mov      fp, sp
        uabal2   v0.8h, v1.16b, v2.16b
        ldp      fp, lr, [sp],#16
        ret      lr

; Total bytes of code 20, prolog size 8

```

17. AbsSaturate

Vector64<short> AbsSaturate(Vector64<short> value)

This method calculates saturated absolute value of each vector element of value. If any element's absolute value is outside the range, the result is saturated. In below example, 1st lane value is -32768 which is ushort.MinValue. It's absolute value would be 32768, but since it is out of range, it is saturated to 32767 which is ushort.MaxValue.

```
private Vector64<short> AbsSaturateTest(Vector64<short> value)
{
    return AdvSimd.AbsSaturate(value);
}
// value = <-32768, -12, -13, 32767>
// Result = <32767, 12, 13, 32767>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> AbsSaturate(Vector64<int> value)
Vector64<sbyte> AbsSaturate(Vector64<sbyte> value)
Vector128<short> AbsSaturate(Vector128<short> value)
Vector128<int> AbsSaturate(Vector128<int> value)
Vector128<sbyte> AbsSaturate(Vector128<sbyte> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<long> AbsSaturate(Vector128<long> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsSaturateTest(System.Runtime.Intrinsics.Vector64`1[Int16]):S
ystem.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
;# V01 OutArgs      [V01   ] (  1,  1  )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp     fp, lr, [sp,#-16]!
        mov     fp, sp
        sqabs   v16.4h, v0.4h
        mov     v0.8b, v16.8b
        ldp     fp, lr, [sp],#16
        ret     lr

; Total bytes of code 24, prolog size 8
```

18. AbsSaturateScalar

Vector64<short> AbsSaturateScalar(Vector64<short> value)

This method reads 0th vector element from the value vector, stores the absolute value of the result into a result vector and returns the result vector. This method operates on signed integer values.

```
private Vector64<short> AbsSaturateScalarTest(Vector64<short> value)
{
    return AdvSimd.Arm64.AbsSaturateScalar(value);
}
// value = <11, 12, 13, 14>
// Result = <11, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<int> AbsSaturateScalar(Vector64<int> value)
Vector64<long> AbsSaturateScalar(Vector64<long> value)
Vector64<sbyte> AbsSaturateScalar(Vector64<sbyte> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:AbsSaturateScalarTest(System.Runtime.Intrinsics.Vector64`1[Int
16]):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqabs  h16, h0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

19. AbsScalar

Vector64<double> AbsScalar(Vector64<double> value)

This method calculates floating-point absolute value, similar to `Abs()` and stores them in result vector and return the result vector.

```
private Vector64<double> AbsScalarTest(Vector64<double> value)
{
    return AdvSimd.AbsScalar(value);
}
// value = <-11.5>
// Result = <11.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> AbsScalar(Vector64<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<ulong> AbsScalar(Vector64<long> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AbsScalarTest(System.Runtime.Intrinsics.Vector64`1[Double]):Sy
stem.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fabs    d16, d0
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

20. Add

Vector64<byte> Add(Vector64<byte> left, Vector64<byte> right)

This method adds the corresponding vector elements in the left and right vector, and returns the result vector.

```
private Vector64<byte> AddTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.Add(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <32, 34, 36, 38, 40, 42, 44, 46>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> Add(Vector64<short> left, Vector64<short> right)
Vector64<int> Add(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> Add(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> Add(Vector64<float> left, Vector64<float> right)
Vector64<ushort> Add(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> Add(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> Add(Vector128<byte> left, Vector128<byte> right)
Vector128<short> Add(Vector128<short> left, Vector128<short> right)
Vector128<int> Add(Vector128<int> left, Vector128<int> right)
Vector128<long> Add(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> Add(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> Add(Vector128<float> left, Vector128<float> right)
Vector128<ushort> Add(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> Add(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> Add(Vector128<ulong> left, Vector128<ulong> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Add(Vector128<double> left, Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    add    v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

21. AddAcross

Vector64<byte> AddAcross(Vector64<byte> value)

This method adds every vector element in the value vector together, and writes the result to the 0th element of result vector, while other elements of result vector set to 0.

```
private Vector64<byte> AddAcrossTest(Vector64<byte> value)
{
    return AdvSimd.Arm64.AddAcross(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <116, 0, 0, 0, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<short> AddAcross(Vector64<short> value)
Vector64<sbyte> AddAcross(Vector64<sbyte> value)
Vector64<ushort> AddAcross(Vector64<ushort> value)
Vector64<byte> AddAcross(Vector128<byte> value)
Vector64<short> AddAcross(Vector128<short> value)
Vector64<int> AddAcross(Vector128<int> value)
Vector64<sbyte> AddAcross(Vector128<sbyte> value)
Vector64<ushort> AddAcross(Vector128<ushort> value)
Vector64<uint> AddAcross(Vector128<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddAcrossTest(System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    addv   b16, v0.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

22. AddAcrossWidening

Vector64<ushort> AddAcrossWidening(Vector64<byte> value)

This method adds every vector element in the value vector together, and writes the result to the 0th element of result vector, while other elements of result vector set to 0. As seen in below example, the result vector's element size ushort is twice as long as the input parameter's element size byte.

```
private Vector64<ushort> AddAcrossWideningTest(Vector64<byte> value)
{
    return AdvSimd.Arm64.AddAcrossWidening(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <116, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<int> AddAcrossWidening(Vector64<short> value)
Vector64<short> AddAcrossWidening(Vector64<sbyte> value)
Vector64<uint> AddAcrossWidening(Vector64<ushort> value)
Vector64<ushort> AddAcrossWidening(Vector128<byte> value)
Vector64<int> AddAcrossWidening(Vector128<short> value)
Vector64<long> AddAcrossWidening(Vector128<int> value)
Vector64<short> AddAcrossWidening(Vector128<sbyte> value)
Vector64<uint> AddAcrossWidening(Vector128<ushort> value)
Vector64<ulong> AddAcrossWidening(Vector128<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:AddAcrossWideningTest(System.Runtime.Intrinsics.Vector64`1[Byte
e]):System.Runtime.Intrinsics.Vector64`1[UInt16]
;
; V00 arg0          [V00,T00]( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uaddlv h16, v0.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


23. AddHighNarrowingLower

Vector64<byte> AddHighNarrowingLower(Vector128<ushort> left, Vector128<ushort> right)

This method adds corresponding vector elements in the left and right vector, places the most significant half of the result into the result vector and return the result vector. As seen in below example, elements in result vector Vector64<byte> is half the size of that of input Vector128<ushort> although number of total elements are same.

```
private Vector64<byte> AddHighNarrowingLowerTest(Vector128<ushort> left,
Vector128<ushort> right)
{
    return AdvSimd.AddHighNarrowingLower(left, right);
}
// left = <100, 200, 300, 400, 500, 600, 700, 800>
// right = <900, 1000, 1100, 1200, 1300, 1400, 1500, 1600>
// Result = <3, 4, 5, 6, 7, 7, 8, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> AddHighNarrowingLower(Vector128<int> left, Vector128<int>
right)
Vector64<int> AddHighNarrowingLower(Vector128<long> left, Vector128<long>
right)
Vector64<sbyte> AddHighNarrowingLower(Vector128<short> left, Vector128<short>
right)
Vector64<ushort> AddHighNarrowingLower(Vector128<uint> left, Vector128<uint>
right)
Vector64<uint> AddHighNarrowingLower(Vector128<ulong> left, Vector128<ulong>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddHighNarrowingLowerTest(System.Runtime.Intrinsics.Vector128`
1[UInt16],System.Runtime.Intrinsics.Vector128`1[UInt16]):System.Runtime.Intri
nsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
```

```
    addhn    v16.8b, v0.8h, v1.8h
    mov      v0.8b, v16.8b
    ldp      fp, lr, [sp],#16
    ret      lr
```

; Total bytes of code 24, prolog size 8

24. AddHighNarrowingUpper

Vector128<byte> AddHighNarrowingUpper(Vector64<byte> lower, Vector128<ushort> left, Vector128<ushort> right)

This method adds corresponding vector elements in the left and right vector, places the most significant half of the result into upper half of the result vector while the lower half of vector is set to the elements in lower.

```
private Vector128<byte> AddHighNarrowingUpperTest(Vector64<byte> lower,
Vector128<ushort> left, Vector128<ushort> right)
{
    return AdvSimd.AddHighNarrowingUpper(lower, left, right);
}
// lower = <1, 255, 13, 41, 54, 61, 71, 18>
// left = <100, 200, 300, 400, 500, 600, 700, 800>
// right = <900, 1000, 1100, 1200, 1300, 1400, 1500, 1600>
// Result = <1, 255, 13, 41, 54, 61, 71, 18, 3, 4, 5, 6, 7, 7, 8, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> AddHighNarrowingUpper(Vector64<short> lower, Vector128<int>
left, Vector128<int> right)
Vector128<int> AddHighNarrowingUpper(Vector64<int> lower, Vector128<long>
left, Vector128<long> right)
Vector128<sbyte> AddHighNarrowingUpper(Vector64<sbyte> lower,
Vector128<short> left, Vector128<short> right)
Vector128<ushort> AddHighNarrowingUpper(Vector64<ushort> lower,
Vector128<uint> left, Vector128<uint> right)
Vector128<uint> AddHighNarrowingUpper(Vector64<uint> lower, Vector128<ulong>
left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddHighNarrowingUpperTest(System.Runtime.Intrinsics.Vector64`1
[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16],System.Runtime.Intrinsic
s.Vector128`1[UInt16]):System.Runtime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2          [V02,T02] ( 3, 3 ) simd16 -> d2
HFA(simd16)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
```



```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    addhn2 v0.16b, v1.8h, v2.8h  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 20, prolog size 8

25. AddPairwise

Vector64<byte> AddPairwise(Vector64<byte> left, Vector64<byte> right)

This method creates a vector by concatenating the vector elements of left vector followed by those of the right vector, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values and places them in result vector and returns the result vector.

```
private Vector64<byte> AddPairwiseTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.AddPairwise(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <23, 27, 31, 35, 43, 47, 51, 55>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> AddPairwise(Vector64<short> left, Vector64<short> right)
Vector64<int> AddPairwise(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> AddPairwise(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> AddPairwise(Vector64<float> left, Vector64<float> right)
Vector64<ushort> AddPairwise(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> AddPairwise(Vector64<uint> left, Vector64<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<byte> AddPairwise(Vector128<byte> left, Vector128<byte> right)
Vector128<double> AddPairwise(Vector128<double> left, Vector128<double>
right)
Vector128<short> AddPairwise(Vector128<short> left, Vector128<short> right)
Vector128<int> AddPairwise(Vector128<int> left, Vector128<int> right)
Vector128<long> AddPairwise(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> AddPairwise(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> AddPairwise(Vector128<float> left, Vector128<float> right)
Vector128<ushort> AddPairwise(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> AddPairwise(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> AddPairwise(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddPairwiseTest(System.Runtime.Intrinsics.Vector64`1[Byte],Sys
tem.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1
[Byte]
;
```

```

; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    addp   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

26. AddPairwiseScalar

Vector64<float> AddPairwiseScalar(Vector64<float> value)

This method adds vector elements in the value vector and writes the result to the 0th element of result vector, while other elements of result vector set to 0.

```
private Vector64<float> AddPairwiseScalarTest(Vector64<float> value)
{
    return AdvSimd.Arm64.AddPairwiseScalar(value);
}
// value = <11.5, 12.5>
// Result = <24, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<double> AddPairwiseScalar(Vector128<double> value)
Vector64<long> AddPairwiseScalar(Vector128<long> value)
Vector64<ulong> AddPairwiseScalar(Vector128<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddPairwiseScalarTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    faddp   s16, v0.2s
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

27. AddPairwiseWidening

Vector64<ushort> AddPairwiseWidening(Vector64<byte> value)

This method adds pairs of adjacent integer values from the value vector, stores them in a result vector and returns the vector. As seen in example below, the result vector elements ushort is twice as long as the input's vector elements size byte.

```
private Vector64<ushort> AddPairwiseWideningTest(Vector64<byte> value)
{
    return AdvSimd.AddPairwiseWidening(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <23, 27, 31, 35>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> AddPairwiseWidening(Vector64<short> value)
Vector64<short> AddPairwiseWidening(Vector64<sbyte> value)
Vector64<uint> AddPairwiseWidening(Vector64<ushort> value)
Vector128<ushort> AddPairwiseWidening(Vector128<byte> value)
Vector128<int> AddPairwiseWidening(Vector128<short> value)
Vector128<long> AddPairwiseWidening(Vector128<int> value)
Vector128<short> AddPairwiseWidening(Vector128<sbyte> value)
Vector128<uint> AddPairwiseWidening(Vector128<ushort> value)
Vector128<ulong> AddPairwiseWidening(Vector128<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddPairwiseWideningTest(System.Runtime.Intrinsics.Vector64`1[B
yte]):System.Runtime.Intrinsics.Vector64`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uaddlp v16.4h, v0.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

28. AddPairwiseWideningAndAdd

Vector64<ushort> AddPairwiseWideningAndAdd(Vector64<ushort> addend, Vector64<byte> value)

This method adds pairs of adjacent integer values from the value vector and accumulates the results with those of addend vector and return the result vector. As seen in below example, the result vector element size ushort is twice as long as that of input parameter's size byte.

```
private Vector64<ushort> AddPairwiseWideningAndAddTest(Vector64<ushort>
addend, Vector64<byte> value)
{
    return AdvSimd.AddPairwiseWideningAndAdd(addend, value);
}
// addend = <11, 12, 13, 14>
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <34, 39, 44, 49>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> AddPairwiseWideningAndAdd(Vector64<int> addend, Vector64<short>
value)
Vector64<short> AddPairwiseWideningAndAdd(Vector64<short> addend,
Vector64<sbyte> value)
Vector64<uint> AddPairwiseWideningAndAdd(Vector64<uint> addend,
Vector64<ushort> value)
Vector128<ushort> AddPairwiseWideningAndAdd(Vector128<ushort> addend,
Vector128<byte> value)
Vector128<int> AddPairwiseWideningAndAdd(Vector128<int> addend,
Vector128<short> value)
Vector128<long> AddPairwiseWideningAndAdd(Vector128<long> addend,
Vector128<int> value)
Vector128<short> AddPairwiseWideningAndAdd(Vector128<short> addend,
Vector128<sbyte> value)
Vector128<uint> AddPairwiseWideningAndAdd(Vector128<uint> addend,
Vector128<ushort> value)
Vector128<ulong> AddPairwiseWideningAndAdd(Vector128<ulong> addend,
Vector128<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddPairwiseWideningAndAddTest(System.Runtime.Intrinsics.Vector
64`1[UInt16],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intri
nsics.Vector64`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
```

```

HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;# V02 OutArgs  [V02    ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uadalp v0.4h, v1.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

29. AddPairwiseWideningAndAddScalar

Vector64<long> AddPairwiseWideningAndAddScalar(Vector64<long> addend, Vector64<int> value)

This method adds pairs of adjacent integer values from value vector and accumulates the results with the vector elements of addend and returns the result vector. As seen in below example, the result vector element's size long is twice as long as that of value vector element's size int.

```
private Vector64<long> AddPairwiseWideningAndAddScalarTest(Vector64<long>
addend, Vector64<int> value)
{
    return AdvSimd.AddPairwiseWideningAndAddScalar(addend, value);
}
// addend = <11>
// value = <11, 12>
// Result = <34>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> AddPairwiseWideningAndAddScalar(Vector64<ulong> addend,
Vector64<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddPairwiseWideningAndAddScalarTest(System.Runtime.Intrinsics.
Vector64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int32]):System.Runtime
.Intrinsics.Vector64`1[Int64]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sadalp v0.1d, v1.2s
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

30. AddPairwiseWideningScalar

Vector64<long> AddPairwiseWideningScalar(Vector64<int> value)

This method adds pairs of adjacent integer values from the value vector, stores them in a result vector and returns the result vector. As seen in below example, the result vector element's size long is twice as long as the input value's element size int.

```
private Vector64<long> AddPairwiseWideningScalarTest(Vector64<int> value)
{
    return AdvSimd.AddPairwiseWideningScalar(value);
}
// value = <11, 12>
// Result = <23>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> AddPairwiseWideningScalar(Vector64<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddPairwiseWideningScalarTest(System.Runtime.Intrinsics.Vector
64`1[Int32]):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
;# V01 OutArgs      [V01      ] (  1,  1  )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp     fp, lr, [sp,#-16]!
    mov     fp, sp
    saddlp  v16.1d, v0.2s
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

31. AddRoundedHighNarrowingLower

Vector64<byte> AddRoundedHighNarrowingLower(Vector128<ushort> left, Vector128<ushort> right)

This method adds corresponding vector elements in left vector to those of right vector, stores the most significant half of the result into the result vector such that the result is rounded and return the result.

```
private Vector64<byte> AddRoundedHighNarrowingLowerTest(Vector128<ushort>
left, Vector128<ushort> right)
{
    return AdvSimd.AddRoundedHighNarrowingLower(left, right);
}
// left = <100, 200, 300, 400, 500, 600, 700, 800>
// right = <900, 1000, 1100, 1200, 1300, 1400, 1500, 1600>
// Result = <4, 5, 5, 6, 7, 8, 9, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> AddRoundedHighNarrowingLower(Vector128<int> left,
Vector128<int> right)
Vector64<int> AddRoundedHighNarrowingLower(Vector128<long> left,
Vector128<long> right)
Vector64<sbyte> AddRoundedHighNarrowingLower(Vector128<short> left,
Vector128<short> right)
Vector64<ushort> AddRoundedHighNarrowingLower(Vector128<uint> left,
Vector128<uint> right)
Vector64<uint> AddRoundedHighNarrowingLower(Vector128<ulong> left,
Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddRoundedHighNarrowingLowerTest(System.Runtime.Intrinsics.Vec
tor128`1[UInt16],System.Runtime.Intrinsics.Vector128`1[UInt16]):System.Runtim
e.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    raddhn v16.8b, v0.8h, v1.8h
```

```
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret     lr
```

; Total bytes of code 24, prolog size 8

32. AddRoundedHighNarrowingUpper

Vector128<byte> AddRoundedHighNarrowingUpper(Vector64<byte> lower, Vector128<ushort> left, Vector128<ushort> right)

This method adds corresponding vector elements in left vector to those of right vector, places the most significant half of the result (after rounding) into the upper half of the result vector while the lower half is set to the elements in lower.

```
private Vector128<byte> AddRoundedHighNarrowingUpperTest(Vector64<byte>
lower, Vector128<ushort> left, Vector128<ushort> right)
{
    return AdvSimd.AddRoundedHighNarrowingUpper(lower, left, right);
}
// lower = <1, 255, 13, 41, 54, 61, 71, 18>
// left = <100, 200, 300, 400, 500, 600, 700, 800>
// right = <900, 1000, 1100, 1200, 1300, 1400, 1500, 1600>
// Result = <1, 255, 13, 41, 54, 61, 71, 18, 4, 5, 5, 6, 7, 8, 9, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> AddRoundedHighNarrowingUpper(Vector64<short> lower,
Vector128<int> left, Vector128<int> right)
Vector128<int> AddRoundedHighNarrowingUpper(Vector64<int> lower,
Vector128<long> left, Vector128<long> right)
Vector128<sbyte> AddRoundedHighNarrowingUpper(Vector64<sbyte> lower,
Vector128<short> left, Vector128<short> right)
Vector128<ushort> AddRoundedHighNarrowingUpper(Vector64<ushort> lower,
Vector128<uint> left, Vector128<uint> right)
Vector128<uint> AddRoundedHighNarrowingUpper(Vector64<uint> lower,
Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddRoundedHighNarrowingUpperTest(System.Runtime.Intrinsics.Vec
tor64`1[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16],System.Runtime.In
trinsics.Vector128`1[UInt16]):System.Runtime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2          [V02,T02] ( 3, 3 ) simd16 -> d2
HFA(simd16)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    raddhn2 v0.16b, v1.8h, v2.8h  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 20, prolog size 8

33. AddSaturate

Vector64<byte> AddSaturate(Vector64<byte> left, Vector64<byte> right)

This method adds the values of corresponding elements of the left and right vectors, stores the results in a vector and returns the result vector. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<byte> AddSaturateTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.AddSaturate(left, right);
}
// left = <155, 200, 200, 1, 5, 16, 17, 18>
// right = <155, 100, 100, 2, 25, 26, 27, 28>
// Result = <255, 255, 255, 3, 30, 42, 44, 46>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> AddSaturate(Vector64<short> left, Vector64<short> right)
Vector64<int> AddSaturate(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> AddSaturate(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<ushort> AddSaturate(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> AddSaturate(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> AddSaturate(Vector128<byte> left, Vector128<byte> right)
Vector128<short> AddSaturate(Vector128<short> left, Vector128<short> right)
Vector128<int> AddSaturate(Vector128<int> left, Vector128<int> right)
Vector128<long> AddSaturate(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> AddSaturate(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<ushort> AddSaturate(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> AddSaturate(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> AddSaturate(Vector128<ulong> left, Vector128<ulong> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<byte> AddSaturate(Vector64<byte> left, Vector64<sbyte> right)
Vector64<short> AddSaturate(Vector64<short> left, Vector64<ushort> right)
Vector64<int> AddSaturate(Vector64<int> left, Vector64<uint> right)
Vector64<sbyte> AddSaturate(Vector64<sbyte> left, Vector64<byte> right)
Vector64<ushort> AddSaturate(Vector64<ushort> left, Vector64<short> right)
Vector64<uint> AddSaturate(Vector64<uint> left, Vector64<int> right)
Vector128<byte> AddSaturate(Vector128<byte> left, Vector128<sbyte> right)
Vector128<short> AddSaturate(Vector128<short> left, Vector128<ushort> right)
Vector128<int> AddSaturate(Vector128<int> left, Vector128<uint> right)
Vector128<long> AddSaturate(Vector128<long> left, Vector128<ulong> right)
Vector128<sbyte> AddSaturate(Vector128<sbyte> left, Vector128<byte> right)
Vector128<ushort> AddSaturate(Vector128<ushort> left, Vector128<short> right)
Vector128<uint> AddSaturate(Vector128<uint> left, Vector128<int> right)
Vector128<ulong> AddSaturate(Vector128<ulong> left, Vector128<long> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:AddSaturateTest(System.Runtime.Intrinsics.Vector64`1[Byte],Sys
tem.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1
[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqadd  v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

34. AddSaturateScalar

Vector64<long> AddSaturateScalar(Vector64<long> left, Vector64<long> right)

This method scalar variant, adds the values of corresponding elements of the left and right vectors, stores the results in a vector and returns the result vector. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<long> AddSaturateScalarTest(Vector64<long> left,
Vector64<long> right)
{
    return AdvSimd.AddSaturateScalar(left, right);
}
// left = <11>
// right = <11>
// Result = <22>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> AddSaturateScalar(Vector64<ulong> left, Vector64<ulong>
right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<byte> AddSaturateScalar(Vector64<byte> left, Vector64<byte> right)
Vector64<byte> AddSaturateScalar(Vector64<byte> left, Vector64<sbyte> right)
Vector64<short> AddSaturateScalar(Vector64<short> left, Vector64<short>
right)
Vector64<short> AddSaturateScalar(Vector64<short> left, Vector64<ushort>
right)
Vector64<int> AddSaturateScalar(Vector64<int> left, Vector64<int> right)
Vector64<int> AddSaturateScalar(Vector64<int> left, Vector64<uint> right)
Vector64<long> AddSaturateScalar(Vector64<long> left, Vector64<ulong> right)
Vector64<sbyte> AddSaturateScalar(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector64<sbyte> AddSaturateScalar(Vector64<sbyte> left, Vector64<byte> right)
Vector64<ushort> AddSaturateScalar(Vector64<ushort> left, Vector64<ushort>
right)
Vector64<ushort> AddSaturateScalar(Vector64<ushort> left, Vector64<short>
right)
Vector64<uint> AddSaturateScalar(Vector64<uint> left, Vector64<uint> right)
Vector64<uint> AddSaturateScalar(Vector64<uint> left, Vector64<int> right)
Vector64<ulong> AddSaturateScalar(Vector64<ulong> left, Vector64<long> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddSaturateScalarTest(System.Runtime.Intrinsics.Vector64`1[Int
64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Runtime.Intrinsics.Ve
```



```

ctor64`1[Int64]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqadd  d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

```

```

; Total bytes of code 24, prolog size 8

```

35. AddScalar

Vector64<double> AddScalar(Vector64<double> left, Vector64<double> right)

This method adds the floating-point values of the two source vectors, and writes the result to the result. This performs scalar operation.

```
private Vector64<double> AddScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.AddScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <23>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<long> AddScalar(Vector64<long> left, Vector64<long> right)
Vector64<float> AddScalar(Vector64<float> left, Vector64<float> right)
Vector64<ulong> AddScalar(Vector64<ulong> left, Vector64<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddScalarTest(System.Runtime.Intrinsics.Vector64`1[Double],Sys
tem.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64
`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fadd    d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

36. AddWideningLower

Vector128<ushort> AddWideningLower(Vector64<byte> left, Vector64<byte> right)

This method adds corresponding vector elements in the left to those of right vector, stores the result in a vector, and returns the result vector. As seen in below example, the result vector element's size ushort is twice that of input parameter element size byte.

```
private Vector128<ushort> AddWideningLowerTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.AddWideningLower(left, right);
}
// left = <155, 200, 200, 1, 5, 16, 17, 18>
// right = <155, 100, 100, 2, 25, 26, 27, 28>
// Result = <310, 300, 300, 3, 30, 42, 44, 46>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> AddWideningLower(Vector64<short> left, Vector64<short> right)
Vector128<long> AddWideningLower(Vector64<int> left, Vector64<int> right)
Vector128<short> AddWideningLower(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector128<uint> AddWideningLower(Vector64<ushort> left, Vector64<ushort>
right)
Vector128<ulong> AddWideningLower(Vector64<uint> left, Vector64<uint> right)
Vector128<short> AddWideningLower(Vector128<short> left, Vector64<sbyte>
right)
Vector128<int> AddWideningLower(Vector128<int> left, Vector64<short> right)
Vector128<long> AddWideningLower(Vector128<long> left, Vector64<int> right)
Vector128<ushort> AddWideningLower(Vector128<ushort> left, Vector64<byte>
right)
Vector128<uint> AddWideningLower(Vector128<uint> left, Vector64<ushort>
right)
Vector128<ulong> AddWideningLower(Vector128<ulong> left, Vector64<uint>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddWideningLowerTest(System.Runtime.Intrinsics.Vector64`1[Byte
],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vecto
r128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
```

```

;# V02 OutArgs      [V02      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uaddl   v16.8h, v0.8b, v1.8b
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

37. AddWideningUpper

Vector128<ushort> AddWideningUpper(Vector128<byte> left, Vector128<byte> right)

This method adds corresponding vector elements in the upper half of left to those of right vector, stores the result into a result vector, and returns the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input parameter's element size byte.

```
private Vector128<ushort> AddWideningUpperTest(Vector128<byte> left,
Vector128<byte> right)
{
    return AdvSimd.AddWideningUpper(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// right = <21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36>
// Result = <48, 50, 52, 54, 56, 58, 60, 62>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> AddWideningUpper(Vector128<short> left, Vector128<short>
right)
Vector128<short> AddWideningUpper(Vector128<short> left, Vector128<sbyte>
right)
Vector128<int> AddWideningUpper(Vector128<int> left, Vector128<short> right)
Vector128<long> AddWideningUpper(Vector128<int> left, Vector128<int> right)
Vector128<long> AddWideningUpper(Vector128<long> left, Vector128<int> right)
Vector128<short> AddWideningUpper(Vector128<sbyte> left, Vector128<sbyte>
right)
Vector128<ushort> AddWideningUpper(Vector128<ushort> left, Vector128<byte>
right)
Vector128<uint> AddWideningUpper(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> AddWideningUpper(Vector128<uint> left, Vector128<ushort>
right)
Vector128<ulong> AddWideningUpper(Vector128<uint> left, Vector128<uint>
right)
Vector128<ulong> AddWideningUpper(Vector128<ulong> left, Vector128<uint>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.AddWideningUpperTest(System.Runtime.Intrinsics.Vector128`1[Byte
e],System.Runtime.Intrinsics.Vector128`1[Byte]):System.Runtime.Intrinsics.Vec
tor128`1[UInt16]
;
```

```

; V00 arg0      [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1      [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uaddl2 v16.8h, v0.16b, v1.16b
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

```

```

; Total bytes of code 24, prolog size 8

```

38. And

Vector64<byte> And(Vector64<byte> left, Vector64<byte> right)

This method ands the vector elements in the left and right vector, and returns the result vector.

```
private Vector64<byte> AndTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.And(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <1, 4, 5, 8, 9, 16, 17, 16>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<double> And(Vector64<double> left, Vector64<double> right)
Vector64<short> And(Vector64<short> left, Vector64<short> right)
Vector64<int> And(Vector64<int> left, Vector64<int> right)
Vector64<long> And(Vector64<long> left, Vector64<long> right)
Vector64<sbyte> And(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> And(Vector64<float> left, Vector64<float> right)
Vector64<ushort> And(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> And(Vector64<uint> left, Vector64<uint> right)
Vector64<ulong> And(Vector64<ulong> left, Vector64<ulong> right)
Vector128<byte> And(Vector128<byte> left, Vector128<byte> right)
Vector128<double> And(Vector128<double> left, Vector128<double> right)
Vector128<short> And(Vector128<short> left, Vector128<short> right)
Vector128<int> And(Vector128<int> left, Vector128<int> right)
Vector128<long> And(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> And(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> And(Vector128<float> left, Vector128<float> right)
Vector128<ushort> And(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> And(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> And(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:AndTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    and    v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

39. BitwiseClear

Vector64<byte> BitwiseClear(Vector64<byte> value, Vector64<byte> mask)

This method performs AND of corresponding vector elements in value and complement of mask vector and returns the result vector containing the result of this operation.

```
private Vector64<byte> BitwiseClearTest(Vector64<byte> value, Vector64<byte>
mask)
{
    return AdvSimd.BitwiseClear(value, mask);
}
// value = <255, 255, 255, 255, 255, 255, 255, 255>
// mask = <1, 2, 4, 8, 16, 32, 64, 128>
// Result = <254, 253, 251, 247, 239, 223, 191, 127>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<double> BitwiseClear(Vector64<double> value, Vector64<double> mask)
Vector64<short> BitwiseClear(Vector64<short> value, Vector64<short> mask)
Vector64<int> BitwiseClear(Vector64<int> value, Vector64<int> mask)
Vector64<long> BitwiseClear(Vector64<long> value, Vector64<long> mask)
Vector64<sbyte> BitwiseClear(Vector64<sbyte> value, Vector64<sbyte> mask)
Vector64<float> BitwiseClear(Vector64<float> value, Vector64<float> mask)
Vector64<ushort> BitwiseClear(Vector64<ushort> value, Vector64<ushort> mask)
Vector64<uint> BitwiseClear(Vector64<uint> value, Vector64<uint> mask)
Vector64<ulong> BitwiseClear(Vector64<ulong> value, Vector64<ulong> mask)
Vector128<byte> BitwiseClear(Vector128<byte> value, Vector128<byte> mask)
Vector128<double> BitwiseClear(Vector128<double> value, Vector128<double>
mask)
Vector128<short> BitwiseClear(Vector128<short> value, Vector128<short> mask)
Vector128<int> BitwiseClear(Vector128<int> value, Vector128<int> mask)
Vector128<long> BitwiseClear(Vector128<long> value, Vector128<long> mask)
Vector128<sbyte> BitwiseClear(Vector128<sbyte> value, Vector128<sbyte> mask)
Vector128<float> BitwiseClear(Vector128<float> value, Vector128<float> mask)
Vector128<ushort> BitwiseClear(Vector128<ushort> value, Vector128<ushort>
mask)
Vector128<uint> BitwiseClear(Vector128<uint> value, Vector128<uint> mask)
Vector128<ulong> BitwiseClear(Vector128<ulong> value, Vector128<ulong> mask)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.BitwiseClearTest(System.Runtime.Intrinsics.Vector64`1[Byte],Sy
stem.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`
1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
```

```

HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;# V02 OutArgs  [V02    ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    bic    v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

40. BitwiseSelect

Vector64<byte> BitwiseSelect(Vector64<byte> select, Vector64<byte> left, Vector64<byte> right)

This method sets each bit in the result to the corresponding bit from the left vector when the select vector's bit was 1, otherwise from the right vector.

```
private Vector64<byte> BitwiseSelectTest(Vector64<byte> select,
Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.BitwiseSelect(select, left, right);
}
// select = <11, 12, 13, 14, 15, 16, 17, 18>
// left = <21, 22, 23, 24, 25, 26, 27, 28>
// right = <31, 32, 33, 34, 35, 36, 37, 38>
// Result = <21, 36, 37, 40, 41, 52, 53, 52>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<double> BitwiseSelect(Vector64<double> select, Vector64<double>
left, Vector64<double> right)
Vector64<short> BitwiseSelect(Vector64<short> select, Vector64<short> left,
Vector64<short> right)
Vector64<int> BitwiseSelect(Vector64<int> select, Vector64<int> left,
Vector64<int> right)
Vector64<long> BitwiseSelect(Vector64<long> select, Vector64<long> left,
Vector64<long> right)
Vector64<sbyte> BitwiseSelect(Vector64<sbyte> select, Vector64<sbyte> left,
Vector64<sbyte> right)
Vector64<float> BitwiseSelect(Vector64<float> select, Vector64<float> left,
Vector64<float> right)
Vector64<ushort> BitwiseSelect(Vector64<ushort> select, Vector64<ushort>
left, Vector64<ushort> right)
Vector64<uint> BitwiseSelect(Vector64<uint> select, Vector64<uint> left,
Vector64<uint> right)
Vector64<ulong> BitwiseSelect(Vector64<ulong> select, Vector64<ulong> left,
Vector64<ulong> right)
Vector128<byte> BitwiseSelect(Vector128<byte> select, Vector128<byte> left,
Vector128<byte> right)
Vector128<double> BitwiseSelect(Vector128<double> select, Vector128<double>
left, Vector128<double> right)
Vector128<short> BitwiseSelect(Vector128<short> select, Vector128<short>
left, Vector128<short> right)
Vector128<int> BitwiseSelect(Vector128<int> select, Vector128<int> left,
Vector128<int> right)
Vector128<long> BitwiseSelect(Vector128<long> select, Vector128<long> left,
Vector128<long> right)
Vector128<sbyte> BitwiseSelect(Vector128<sbyte> select, Vector128<sbyte>
left, Vector128<sbyte> right)
```

```

Vector128<float> BitwiseSelect(Vector128<float> select, Vector128<float>
left, Vector128<float> right)
Vector128<ushort> BitwiseSelect(Vector128<ushort> select, Vector128<ushort>
left, Vector128<ushort> right)
Vector128<uint> BitwiseSelect(Vector128<uint> select, Vector128<uint> left,
Vector128<uint> right)
Vector128<ulong> BitwiseSelect(Vector128<ulong> select, Vector128<ulong>
left, Vector128<ulong> right)

```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods.BitwiseSelectTest(System.Runtime.Intrinsics.Vector64`1[Byte],S
ystem.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`
1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] (  3,  3  )  simd8  ->  d1
HFA(simd8)
; V02 arg2          [V02,T02] (  3,  3  )  simd8  ->  d2
HFA(simd8)
;# V03 OutArgs      [V03      ] (  1,  1  )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    mov    v16.8b, v0.8b
    bsl    v16.8b, v1.8b, v2.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 28, prolog size 8

```

41. Ceiling

Vector64<float> Ceiling(Vector64<float> value)

This method rounds each vector element of value having floating-point values to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<float> CeilingTest(Vector64<float> value)
{
    return AdvSimd.Ceiling(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> Ceiling(Vector128<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Ceiling(Vector128<double> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CeilingTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintp v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

42. CeilingScalar

Vector64<double> CeilingScalar(Vector64<double> value)

Same as Ceiling above but operates at scalar level.

```
private Vector64<double> CeilingScalarTest(Vector64<double> value)
{
    return AdvSimd.CeilingScalar(value);
}
// value = <11.5>
// Result = <12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> CeilingScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CeilingScalarTest(System.Runtime.Intrinsics.Vector64`1[Double]
):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintp d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

43. CompareEqual

Vector64<byte> CompareEqual(Vector64<byte> left, Vector64<byte> right)

This method compares corresponding vector elements from *left* with those in *right*, and if the comparison is equal sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and returns the result vector.

```
private Vector64<byte> CompareEqualTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.CompareEqual(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <11, 22,13, 14, 25, 26, 27, 28>
// Result = <255, 0, 255, 255, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> CompareEqual(Vector64<short> left, Vector64<short> right)
Vector64<int> CompareEqual(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> CompareEqual(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> CompareEqual(Vector64<float> left, Vector64<float> right)
Vector64<ushort> CompareEqual(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> CompareEqual(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> CompareEqual(Vector128<byte> left, Vector128<byte> right)
Vector128<short> CompareEqual(Vector128<short> left, Vector128<short> right)
Vector128<int> CompareEqual(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> CompareEqual(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> CompareEqual(Vector128<float> left, Vector128<float> right)
Vector128<ushort> CompareEqual(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> CompareEqual(Vector128<uint> left, Vector128<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> CompareEqual(Vector128<double> left, Vector128<double>
right)
Vector128<long> CompareEqual(Vector128<long> left, Vector128<long> right)
Vector128<ulong> CompareEqual(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareEqualTest(System.Runtime.Intrinsics.Vector64`1[Byte],Sy
stem.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`
1[Byte]
;
```

```

; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    cmeq   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

```

```

; Total bytes of code 24, prolog size 8

```

44. CompareEqualScalar

Vector64<double> CompareEqualScalar(Vector64<double> left, Vector64<double> right)

This method compares corresponding floating-point values from the left and right vector, and if the comparison is equal sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double> CompareEqualScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.CompareEqualScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <NaN>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<long> CompareEqualScalar(Vector64<long> left, Vector64<long> right)
Vector64<float> CompareEqualScalar(Vector64<float> left, Vector64<float>
right)
Vector64<ulong> CompareEqualScalar(Vector64<ulong> left, Vector64<ulong>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareEqualScalarTest(System.Runtime.Intrinsics.Vector64`1[Do
uble],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics
.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcmeq   d16, d0, d1
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr
```

; Total bytes of code 24, prolog size 8

45. CompareGreaterThan

Vector64<byte> CompareGreaterThan(Vector64<byte> left, Vector64<byte> right)

This method compares corresponding vector elements in the left and right vector, and if the left's value is greater than the right's value sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<byte> CompareGreaterThanTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.CompareGreaterThan(left, right);
}
// left = <31, 12, 33, 34, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <255, 0, 255, 255, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> CompareGreaterThan(Vector64<short> left, Vector64<short>
right)
Vector64<int> CompareGreaterThan(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> CompareGreaterThan(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector64<float> CompareGreaterThan(Vector64<float> left, Vector64<float>
right)
Vector64<ushort> CompareGreaterThan(Vector64<ushort> left, Vector64<ushort>
right)
Vector64<uint> CompareGreaterThan(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> CompareGreaterThan(Vector128<byte> left, Vector128<byte>
right)
Vector128<short> CompareGreaterThan(Vector128<short> left, Vector128<short>
right)
Vector128<int> CompareGreaterThan(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> CompareGreaterThan(Vector128<sbyte> left, Vector128<sbyte>
right)
Vector128<float> CompareGreaterThan(Vector128<float> left, Vector128<float>
right)
Vector128<ushort> CompareGreaterThan(Vector128<ushort> left,
Vector128<ushort> right)
Vector128<uint> CompareGreaterThan(Vector128<uint> left, Vector128<uint>
right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> CompareGreaterThan(Vector128<double> left,
Vector128<double> right)
Vector128<long> CompareGreaterThan(Vector128<long> left, Vector128<long>
right)
```

Vector128<ulong> CompareGreaterThan(Vector128<ulong> left, Vector128<ulong> right)

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareGreaterThanTest(System.Runtime.Intrinsics.Vector64`1[Byte]
, System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vec
tor64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    cmhi   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

46. CompareGreaterThanOrEqual

Vector64<byte> CompareGreaterThanOrEqual(Vector64<byte> left, Vector64<byte> right)

This method compares corresponding vector elements in the left and right vector, and if the left's value is greater than or equal to the right's value sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<byte> CompareGreaterThanOrEqualTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.CompareGreaterThanOrEqual(left, right);
}
// left = <31, 22, 33, 34, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <255, 255, 255, 255, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> CompareGreaterThanOrEqual(Vector64<short> left,
Vector64<short> right)
Vector64<int> CompareGreaterThanOrEqual(Vector64<int> left, Vector64<int>
right)
Vector64<sbyte> CompareGreaterThanOrEqual(Vector64<sbyte> left,
Vector64<sbyte> right)
Vector64<float> CompareGreaterThanOrEqual(Vector64<float> left,
Vector64<float> right)
Vector64<ushort> CompareGreaterThanOrEqual(Vector64<ushort> left,
Vector64<ushort> right)
Vector64<uint> CompareGreaterThanOrEqual(Vector64<uint> left, Vector64<uint>
right)
Vector128<byte> CompareGreaterThanOrEqual(Vector128<byte> left,
Vector128<byte> right)
Vector128<short> CompareGreaterThanOrEqual(Vector128<short> left,
Vector128<short> right)
Vector128<int> CompareGreaterThanOrEqual(Vector128<int> left, Vector128<int>
right)
Vector128<sbyte> CompareGreaterThanOrEqual(Vector128<sbyte> left,
Vector128<sbyte> right)
Vector128<float> CompareGreaterThanOrEqual(Vector128<float> left,
Vector128<float> right)
Vector128<ushort> CompareGreaterThanOrEqual(Vector128<ushort> left,
Vector128<ushort> right)
Vector128<uint> CompareGreaterThanOrEqual(Vector128<uint> left,
Vector128<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> CompareGreaterThanOrEqual(Vector128<double> left,
```

```

Vector128<double> right)
Vector128<long> CompareGreaterThanOrEqual(Vector128<long> left,
Vector128<long> right)
Vector128<ulong> CompareGreaterThanOrEqual(Vector128<ulong> left,
Vector128<ulong> right)

```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:CompareGreaterThanOrEqualTest(System.Runtime.Intrinsics.Vector
64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrins
ics.Vector64`1[Byte]
;
; V00 arg0      [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    cmhs   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

47. CompareGreaterThanOrEqualScalar

Vector64<double> CompareGreaterThanOrEqualScalar(Vector64<double> left, Vector64<double> right)

This method compares corresponding vector elements in the left and right vector, and if the left's value is greater than or equal to the right's value sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double> CompareGreaterThanOrEqualScalarTest(Vector64<double>
left, Vector64<double> right)
{
    return AdvSimd.Arm64.CompareGreaterThanOrEqualScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <NaN>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<long> CompareGreaterThanOrEqualScalar(Vector64<long> left,
Vector64<long> right)
Vector64<float> CompareGreaterThanOrEqualScalar(Vector64<float> left,
Vector64<float> right)
Vector64<ulong> CompareGreaterThanOrEqualScalar(Vector64<ulong> left,
Vector64<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareGreaterThanOrEqualScalarTest(System.Runtime.Intrinsics.
Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runti
me.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcmge  d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

48. CompareGreaterThanScalar

Vector64<double> CompareGreaterThanScalar(Vector64<double> left, Vector64<double> right)

This method compares corresponding vector elements in the left and right vector, and if the left's value is greater than the right's value sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double> CompareGreaterThanScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.CompareGreaterThanScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<long> CompareGreaterThanScalar(Vector64<long> left, Vector64<long>
right)
Vector64<float> CompareGreaterThanScalar(Vector64<float> left,
Vector64<float> right)
Vector64<ulong> CompareGreaterThanScalar(Vector64<ulong> left,
Vector64<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareGreaterThanScalarTest(System.Runtime.Intrinsics.Vector6
4`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intr
insics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcmgt   d16, d0, d1
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr
```

; Total bytes of code 24, prolog size 8

49. CompareLessThan

Vector64<byte> CompareLessThan(Vector64<byte> left, Vector64<byte> right)

This method compares corresponding vector elements in the left and right vector, and if the left's value is less than the right's value sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<byte> CompareLessThanTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.CompareLessThan(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <1, 22, 3, 4, 25, 26, 27, 28>
// Result = <0, 255, 0, 0, 255, 255, 255, 255>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> CompareLessThan(Vector64<short> left, Vector64<short> right)
Vector64<int> CompareLessThan(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> CompareLessThan(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> CompareLessThan(Vector64<float> left, Vector64<float> right)
Vector64<ushort> CompareLessThan(Vector64<ushort> left, Vector64<ushort>
right)
Vector64<uint> CompareLessThan(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> CompareLessThan(Vector128<byte> left, Vector128<byte> right)
Vector128<short> CompareLessThan(Vector128<short> left, Vector128<short>
right)
Vector128<int> CompareLessThan(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> CompareLessThan(Vector128<sbyte> left, Vector128<sbyte>
right)
Vector128<float> CompareLessThan(Vector128<float> left, Vector128<float>
right)
Vector128<ushort> CompareLessThan(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> CompareLessThan(Vector128<uint> left, Vector128<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> CompareLessThan(Vector128<double> left, Vector128<double>
right)
Vector128<long> CompareLessThan(Vector128<long> left, Vector128<long> right)
Vector128<ulong> CompareLessThan(Vector128<ulong> left, Vector128<ulong>
right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:CompareLessThanTest(System.Runtime.Intrinsics.Vector64`1[Byte]
, System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector
64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
;# V02 OutArgs      [V02   ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    cmhi   v16.8b, v1.8b, v0.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

50. CompareLessThanOrEqual

Vector64<byte> CompareLessThanOrEqual(Vector64<byte> left, Vector64<byte> right)

This method compares corresponding vector elements in the left and right vector, and if the left's value is less than or equal to the right's value sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<byte> CompareLessThanOrEqualTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.CompareLessThanOrEqual(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <1, 12, 3, 4, 25, 26, 27, 28>
// Result = <0, 0, 0, 0, 255, 255, 255, 255>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> CompareLessThanOrEqual(Vector64<short> left, Vector64<short>
right)
Vector64<int> CompareLessThanOrEqual(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> CompareLessThanOrEqual(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector64<float> CompareLessThanOrEqual(Vector64<float> left, Vector64<float>
right)
Vector64<ushort> CompareLessThanOrEqual(Vector64<ushort> left,
Vector64<ushort> right)
Vector64<uint> CompareLessThanOrEqual(Vector64<uint> left, Vector64<uint>
right)
Vector128<byte> CompareLessThanOrEqual(Vector128<byte> left, Vector128<byte>
right)
Vector128<short> CompareLessThanOrEqual(Vector128<short> left,
Vector128<short> right)
Vector128<int> CompareLessThanOrEqual(Vector128<int> left, Vector128<int>
right)
Vector128<sbyte> CompareLessThanOrEqual(Vector128<sbyte> left,
Vector128<sbyte> right)
Vector128<float> CompareLessThanOrEqual(Vector128<float> left,
Vector128<float> right)
Vector128<ushort> CompareLessThanOrEqual(Vector128<ushort> left,
Vector128<ushort> right)
Vector128<uint> CompareLessThanOrEqual(Vector128<uint> left, Vector128<uint>
right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> CompareLessThanOrEqual(Vector128<double> left,
Vector128<double> right)
```

```
Vector128<long> CompareLessThanOrEqual(Vector128<long> left, Vector128<long>
right)
Vector128<ulong> CompareLessThanOrEqual(Vector128<ulong> left,
Vector128<ulong> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareLessThanOrEqualTest(System.Runtime.Intrinsics.Vector64`
1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics
.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    cmhs    v16.8b, v1.8b, v0.8b
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

51. CompareLessThanOrEqualScalar

Vector64<double> CompareLessThanOrEqualScalar(Vector64<double> left, Vector64<double> right)

This method compares corresponding vector elements in the left and right vector, and if the left's value is less than or equal to the right's value sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double> CompareLessThanOrEqualScalarTest(Vector64<double>
left, Vector64<double> right)
{
    return AdvSimd.Arm64.CompareLessThanOrEqualScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <NaN>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<long> CompareLessThanOrEqualScalar(Vector64<long> left,
Vector64<long> right)
Vector64<float> CompareLessThanOrEqualScalar(Vector64<float> left,
Vector64<float> right)
Vector64<ulong> CompareLessThanOrEqualScalar(Vector64<ulong> left,
Vector64<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareLessThanOrEqualScalarTest(System.Runtime.Intrinsics.Vec
tor64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.
Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcmge  d16, d1, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

52. CompareLessThanScalar

Vector64<double> CompareLessThanScalar(Vector64<double> left, Vector64<double> right)

This method compares corresponding vector elements in the left and right vector, and if the left's value is less than the right's value sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double> CompareLessThanScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.CompareLessThanScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<long> CompareLessThanScalar(Vector64<long> left, Vector64<long>
right)
Vector64<float> CompareLessThanScalar(Vector64<float> left, Vector64<float>
right)
Vector64<ulong> CompareLessThanScalar(Vector64<ulong> left, Vector64<ulong>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareLessThanScalarTest(System.Runtime.Intrinsics.Vector64`1
[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrins
ics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcmgt   d16, d1, d0
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr
```

; Total bytes of code 24, prolog size 8

53. CompareTest

Vector64<byte> CompareTest(Vector64<byte> left, Vector64<byte> right)

This method performs AND of corresponding vector elements in the left and right vector, and if the result is not zero, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<byte> CompareTestTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.CompareTest(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <4, 22, 23, 24, 25, 26, 27, 28>
// Result = <0, 255, 255, 255, 255, 255, 255, 255>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> CompareTest(Vector64<short> left, Vector64<short> right)
Vector64<int> CompareTest(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> CompareTest(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> CompareTest(Vector64<float> left, Vector64<float> right)
Vector64<ushort> CompareTest(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> CompareTest(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> CompareTest(Vector128<byte> left, Vector128<byte> right)
Vector128<short> CompareTest(Vector128<short> left, Vector128<short> right)
Vector128<int> CompareTest(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> CompareTest(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> CompareTest(Vector128<float> left, Vector128<float> right)
Vector128<ushort> CompareTest(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> CompareTest(Vector128<uint> left, Vector128<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> CompareTest(Vector128<double> left, Vector128<double>
right)
Vector128<long> CompareTest(Vector128<long> left, Vector128<long> right)
Vector128<ulong> CompareTest(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareTestTest(System.Runtime.Intrinsics.Vector64`1[Byte],Sys
tem.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1
[Byte]
;
```

```

; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    cmtst  v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

54. CompareTestScalar

Vector64<double> CompareTestScalar(Vector64<double> left, Vector64<double> right)

This method performs AND of corresponding vector elements in the left and right vector, and if the result is not zero, sets every bit of the corresponding vector element in the result vector to one, otherwise sets every bit of the corresponding vector element in the result vector to zero and return the result vector.

```
private Vector64<double> CompareTestScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.CompareTestScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <NaN>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Avsimd.Arm64
Vector64<long> CompareTestScalar(Vector64<long> left, Vector64<long> right)
Vector64<ulong> CompareTestScalar(Vector64<ulong> left, Vector64<ulong>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:CompareTestScalarTest(System.Runtime.Intrinsics.Vector64`1[Dou
ble],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.
Vector64`1[Double]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    cmtst  d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


55. ConvertToDouble

Vector128<double> ConvertToDouble(Vector64<float> value)

This method converts each element in a value vector to double the precision of the input element using the rounding mode that as per ARM docs, is determined by the [FPCR](#), and returns the result vector.

```
private Vector128<double> ConvertToDoubleTest(Vector64<float> value)
{
    return AdvSimd.Arm64.ConvertToDouble(value);
}
// value = <11.5, 12.5>
// Result = <11.5, 12.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector128<double> ConvertToDouble(Vector128<long> value)
Vector128<double> ConvertToDouble(Vector128<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToDoubleTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector128`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtl  v16.2d, v0.2s
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

56. ConvertToDoubleScalar

Vector64<double> ConvertToDoubleScalar(Vector64<long> value)

This method converts each element in a value vector to double the precision of the input element using the rounding mode that as per ARM docs, is determined by the [FPCR](#), and returns the result vector.

```
private Vector64<double> ConvertToDoubleScalarTest(Vector64<long> value)
{
    return AdvSimd.Arm64.ConvertToDoubleScalar(value);
}
// value = <11>
// Result = <11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<double> ConvertToDoubleScalar(Vector64<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToDoubleScalarTest(System.Runtime.Intrinsics.Vector64`1
[Int64]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    scvtf  d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

57. ConvertToDoubleUpper

Vector128<double> ConvertToDoubleUpper(Vector128<float> value)

This method converts each element in the upper half of value vector to double the precision of the input element using the rounding mode that as per ARM docs, is determined by the [FPCR](#), and returns the result vector. As seen in below example, the result vector element's size is double that is twice as long as that of input parameter's element size float.

```
private Vector128<double> ConvertToDoubleUpperTest(Vector128<float> value)
{
    return AdvSimd.Arm64.ConvertToDoubleUpper(value);
}
// value = <11.5, 12.5, 13.5, 14.5>
// Result = <13.5, 14.5>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToDoubleUpperTest(System.Runtime.Intrinsics.Vector128`1
[Single]):System.Runtime.Intrinsics.Vector128`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtl2 v16.2d, v0.4s
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

58. ConvertToInt32RoundAwayFromZero

Vector64<int> ConvertToInt32RoundAwayFromZero(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round to Nearest with [Ties to Away rounding mode](#), stores in the result vector and returns the result vector.

```
private Vector64<int> ConvertToInt32RoundAwayFromZeroTest(Vector64<float>
value)
{
    return AdvSimd.ConvertToInt32RoundAwayFromZero(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ConvertToInt32RoundAwayFromZero(Vector128<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt32RoundAwayFromZeroTest(System.Runtime.Intrinsics.
Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtas v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

59. ConvertToInt32RoundAwayFromZeroScalar

Vector64<int> ConvertToInt32RoundAwayFromZeroScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round to Nearest with [Ties to Away rounding mode](#), stores in the result vector and returns the result vector.

```
private Vector64<int>
ConvertToInt32RoundAwayFromZeroScalarTest(Vector64<float> value)
{
    return AdvSimd.ConvertToInt32RoundAwayFromZeroScalar(value);
}
// value = <11.5, 12.5>
// Result = <12, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ConvertToInt32RoundAwayFromZeroScalarTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtas s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

60. ConvertToInt32RoundToEven

Vector64<int> ConvertToInt32RoundToEven(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round to Nearest rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<int> ConvertToInt32RoundToEvenTest(Vector64<float> value)
{
    return AdvSimd.ConvertToInt32RoundToEven(value);
}
// value = <11.5, 12.5>
// Result = <12, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ConvertToInt32RoundToEven(Vector128<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt32RoundToEvenTest(System.Runtime.Intrinsics.Vector
64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtns v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

61. ConvertToInt32RoundToEvenScalar

Vector64<int> ConvertToInt32RoundToEvenScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round to Nearest rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<int> ConvertToInt32RoundToEvenScalarTest(Vector64<float>
value)
{
    return AdvSimd.ConvertToInt32RoundToEvenScalar(value);
}
// value = <11.5, 12.5>
// Result = <12, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt32RoundToEvenScalarTest(System.Runtime.Intrinsics.
Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtns s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

62. ConvertToInt32RoundToNegativeInfinity

Vector64<int> ConvertToInt32RoundToNegativeInfinity(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round towards Minus Infinity rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<int>
ConvertToInt32RoundToNegativeInfinityTest(Vector64<float> value)
{
    return AdvSimd.ConvertToInt32RoundToNegativeInfinity(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ConvertToInt32RoundToNegativeInfinity(Vector128<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt32RoundToNegativeInfinityTest(System.Runtime.Intri
nsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtms v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

63. ConvertToInt32RoundToNegativeInfinityScalar

Vector64<int> ConvertToInt32RoundToNegativeInfinityScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round towards Minus Infinity rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<int>
ConvertToInt32RoundToNegativeInfinityScalarTest(Vector64<float> value)
{
    return AdvSimd.ConvertToInt32RoundToNegativeInfinityScalar(value);
}
// value = <11.5, 12.5>
// Result = <11, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt32RoundToNegativeInfinityScalarTest(System.Runtime
.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtms s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

64. ConvertToInt32RoundToPositiveInfinity

Vector64<int> ConvertToInt32RoundToPositiveInfinity(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round towards Plus Infinity rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<int>
ConvertToInt32RoundToPositiveInfinityTest(Vector64<float> value)
{
    return AdvSimd.ConvertToInt32RoundToPositiveInfinity(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ConvertToInt32RoundToPositiveInfinity(Vector128<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt32RoundToPositiveInfinityTest(System.Runtime.Intri
nsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtps v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

65. ConvertToInt32RoundToPositiveInfinityScalar

Vector64<int> ConvertToInt32RoundToPositiveInfinityScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round towards Plus Infinity rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<int>
ConvertToInt32RoundToPositiveInfinityScalarTest(Vector64<float> value)
{
    return AdvSimd.ConvertToInt32RoundToPositiveInfinityScalar(value);
}
// value = <11.5, 12.5>
// Result = <12, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt32RoundToPositiveInfinityScalarTest(System.Runtime
.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtps s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

66. ConvertToInt32RoundToZero

Vector64<int> ConvertToInt32RoundToZero(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round to Nearest with [toward zero rounding mode](#), stores in the result vector and returns the result vector.

```
private Vector64<int> ConvertToInt32RoundToZeroTest(Vector64<float> value)
{
    return AdvSimd.ConvertToInt32RoundToZero(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ConvertToInt32RoundToZero(Vector128<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ConvertToInt32RoundToZeroTest(System.Runtime.Intrinsics.Vector
64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtzs v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

67. ConvertToInt32RoundToZeroScalar

Vector64<int> ConvertToInt32RoundToZeroScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to a signed integer value using the Round to Nearest with [toward zero rounding mode](#), stores in the result vector and returns the result vector.

```
private Vector64<int> ConvertToInt32RoundToZeroScalarTest(Vector64<float>
value)
{
    return AdvSimd.ConvertToInt32RoundToZeroScalar(value);
}
// value = <11.5, 12.5>
// Result = <11, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt32RoundToZeroScalarTest(System.Runtime.Intrinsics.
Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtzs s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

68. ConvertToInt64RoundAwayFromZero

Vector128<long> ConvertToInt64RoundAwayFromZero(Vector128<double> value)

This method converts each element in the value vector from a floating-point to a 64-bits signed integer value using the Round to Nearest with Ties to Away rounding mode, stores in the result vector and returns the result vector.

```
private Vector128<long> ConvertToInt64RoundAwayFromZeroTest(Vector128<double>
value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundAwayFromZero(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundAwayFromZeroTest(System.Runtime.Intrinsics.
Vector128`1[Double]):System.Runtime.Intrinsics.Vector128`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtas v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

69. ConvertToInt64RoundAwayFromZeroScalar

Vector64<long> ConvertToInt64RoundAwayFromZeroScalar(Vector64<double> value)

This method converts each element in the value vector from a floating-point to a 64-bits signed integer value using the Round to Nearest with Ties to Away rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<long>
ConvertToInt64RoundAwayFromZeroScalarTest(Vector64<double> value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundAwayFromZeroScalar(value);
}
// value = <11.5>
// Result = <12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundAwayFromZeroScalarTest(System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtas d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

70. ConvertToInt64RoundToEven

Vector128<long> ConvertToInt64RoundToEven(Vector128<double> value)

This method converts each element in the value vector from a floating-point to a 64-bits signed integer value using the Round to Nearest rounding mode, stores in the result vector and returns the result vector.

```
private Vector128<long> ConvertToInt64RoundToEvenTest(Vector128<double>
value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundToEven(value);
}
// value = <11.5, 12.5>
// Result = <12, 12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundToEvenTest(System.Runtime.Intrinsics.Vector
128`1[Double]):System.Runtime.Intrinsics.Vector128`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtns v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

71. ConvertToInt64RoundToEvenScalar

Vector64<long> ConvertToInt64RoundToEvenScalar(Vector64<double> value)

This method converts each element in the value vector from a floating-point to a 64-bits signed integer value using the Round to Nearest rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<long> ConvertToInt64RoundToEvenScalarTest(Vector64<double>
value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundToEvenScalar(value);
}
// value = <11.5>
// Result = <12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundToEvenScalarTest(System.Runtime.Intrinsics.
Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtts d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

72. ConvertToInt64RoundToNegativeInfinity

Vector128<long> ConvertToInt64RoundToNegativeInfinity(Vector128<double> value)

This method converts each element in a vector from a floating-point value to a 64-bits signed integer value using the Round towards Minus Infinity rounding mode, and returns the result.

```
private Vector128<long>
ConvertToInt64RoundToNegativeInfinityTest(Vector128<double> value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundToNegativeInfinity(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundToNegativeInfinityTest(System.Runtime.Intrinsics.Vector128`1[Double]):System.Runtime.Intrinsics.Vector128`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtms v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

73. ConvertToInt64RoundToNegativeInfinityScalar

Vector64<long> ConvertToInt64RoundToNegativeInfinityScalar(Vector64<double> value)

This method converts each element in a vector from a floating-point value to a 64-bits signed integer value using the Round towards Minus Infinity rounding mode, and returns the result.

```
private Vector64<long>
ConvertToInt64RoundToNegativeInfinityScalarTest(Vector64<double> value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundToNegativeInfinityScalar(value);
}
// value = <11.5>
// Result = <11>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundToNegativeInfinityScalarTest(System.Runtime
.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtms d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

74. ConvertToInt64RoundToPositiveInfinity

Vector128<long> ConvertToInt64RoundToPositiveInfinity(Vector128<double> value)

This method converts each element in a vector from a floating-point value to a 64-bits signed integer value using the Round towards Plus Infinity rounding mode, and returns the result.

```
private Vector128<long>
ConvertToInt64RoundToPositiveInfinityTest(Vector128<double> value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundToPositiveInfinity(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundToPositiveInfinityTest(System.Runtime.Intri
nsics.Vector128`1[Double]):System.Runtime.Intrinsics.Vector128`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtps v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

75. ConvertToInt64RoundToPositiveInfinityScalar

Vector64<long> ConvertToInt64RoundToPositiveInfinityScalar(Vector64<double> value)

This method converts each element in a vector from a floating-point value to a 64-bits signed integer value using the Round towards Plus Infinity rounding mode, and returns the result.

```
private Vector64<long>
ConvertToInt64RoundToPositiveInfinityScalarTest(Vector64<double> value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundToPositiveInfinityScalar(value);
}
// value = <11.5>
// Result = <12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundToPositiveInfinityScalarTest(System.Runtime
.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtps d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

76. ConvertToInt64RoundToZero

Vector128<long> ConvertToInt64RoundToZero(Vector128<double> value)

This method converts each element in a vector from a floating-point value to a 64-bits signed integer value using the Round towards Zero rounding mode, and returns the result.

```
private Vector128<long> ConvertToInt64RoundToZeroTest(Vector128<double>
value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundToZero(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundToZeroTest(System.Runtime.Intrinsics.Vector
128`1[Double]):System.Runtime.Intrinsics.Vector128`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtzs v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

77. ConvertToInt64RoundToZeroScalar

Vector64<long> ConvertToInt64RoundToZeroScalar(Vector64<double> value)

This method converts each element in a vector from a floating-point value to a 64-bits signed integer value using the Round towards Zero rounding mode, and returns the result.

```
private Vector64<long> ConvertToInt64RoundToZeroScalarTest(Vector64<double>
value)
{
    return AdvSimd.Arm64.ConvertToInt64RoundToZeroScalar(value);
}
// value = <11.5>
// Result = <11>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToInt64RoundToZeroScalarTest(System.Runtime.Intrinsics.
Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtzs d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

78. ConvertToSingle

Vector64<float> ConvertToSingle(Vector64<int> value)

This method converts each element in a vector from fixed-point to floating-point using the rounding mode that, as per ARM docs, is specified by the [FPCR](#), and returns the result.

```
private Vector64<float> ConvertToSingleTest(Vector64<int> value)
{
    return AdvSimd.ConvertToSingle(value);
}
// value = <11, 12>
// Result = <11, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> ConvertToSingle(Vector64<uint> value)
Vector128<float> ConvertToSingle(Vector128<int> value)
Vector128<float> ConvertToSingle(Vector128<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToSingleTest(System.Runtime.Intrinsics.Vector64`1[Int32
]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    scvtf  v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

79. ConvertToSingleLower

Vector64<float> ConvertToSingleLower(Vector128<double> value)

This method converts each vector element in the value vector to half the precision of the source element, stores it in a result vector. As seen below, the result vector element's size float is half as long as the input vector element's size double. The rounding mode is determined by the [FPCR](#).

```
private Vector64<float> ConvertToSingleLowerTest(Vector128<double> value)
{
    return AdvSimd.Arm64.ConvertToSingleLower(value);
}
// value = <11.5, 12.5>
// Result = <11.5, 12.5>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToSingleLowerTest(System.Runtime.Intrinsics.Vector128`1
[Double]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtb  v16.2s, v0.2d
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

80. ConvertToSingleRoundToOddLower

Vector64<float> ConvertToSingleRoundToOddLower(Vector128<double> value)

This method narrows each vector element in the value vector to half the precision using the Round to Odd rounding mode, and stores the result in result vector. For details see the ARM docs.

```
private Vector64<float> ConvertToSingleRoundToOddLowerTest(Vector128<double>
value)
{
    return AdvSimd.Arm64.ConvertToSingleRoundToOddLower(value);
}
// value = <11.5, 12.5>
// Result = <11.5, 12.5>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToSingleRoundToOddLowerTest(System.Runtime.Intrinsics.V
ector128`1[Double]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] (  3,  3  ) simd16  ->  d0
HFA(simd16)
;# V01 OutArgs      [V01      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp     fp, lr, [sp,#-16]!
    mov     fp, sp
    fcvtzn  v16.2s, v0.2d
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

81. ConvertToSingleRoundToOddUpper

Vector128<float> ConvertToSingleRoundToOddUpper(Vector64<float> lower, Vector128<double> value)

This method narrows each vector element in the upper-half of value vector to half the precision using the Round to Odd rounding mode, and stores the result in the upper half of result vector, lower half being the values from lower vector. For details see the ARM docs.

```
private Vector128<float> ConvertToSingleRoundToOddUpperTest(Vector64<float>
lower, Vector128<double> value)
{
    return AdvSimd.Arm64.ConvertToSingleRoundToOddUpper(lower, value);
}
// lower = <11.5, 12.5>
// value = <11.5, 12.5>
// Result = <11.5, 12.5, 11.5, 12.5>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToSingleRoundToOddUpperTest(System.Runtime.Intrinsics.V
ector64`1[Single],System.Runtime.Intrinsics.Vector128`1[Double]):System.Runti
me.Intrinsics.Vector128`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtxn2 v0.4s, v1.2d
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

82. ConvertToSingleScalar

Vector64<float> ConvertToSingleScalar(Vector64<int> value)

This method converts the value vector from fixed-point to floating-point using the rounding mode that is specified by the [FPCR](#), and returns the result.

```
private Vector64<float> ConvertToSingleScalarTest(Vector64<int> value)
{
    return AdvSimd.ConvertToSingleScalar(value);
}
// value = <11, 12>
// Result = <11, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> ConvertToSingleScalar(Vector64<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToSingleScalarTest(System.Runtime.Intrinsics.Vector64`1
[Int32]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    scvtf   s16, s0
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

83. ConvertToSingleUpper

**Vector128<float> ConvertToSingleUpper(Vector64<float> lower,
Vector128<double> value)**

This method converts each vector element in the upper-half of value vector to half the precision and stores the result in upper-half of result vector, lower half being the values from lower vector. As seen in example below, the result vector element's size float is half as long as the input vector element's size double. The rounding mode, as per ARM docs, is determined by the [FPCR](#).

```
private Vector128<float> ConvertToSingleUpperTest(Vector64<float> lower,  
Vector128<double> value)  
{  
    return AdvSimd.Arm64.ConvertToSingleUpper(lower, value);  
}  
// lower = <5.1, 5.1>  
// value = <11.5, 12.5>  
// Result = <5.1, 5.1, 11.5, 12.5>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods:ConvertToSingleUpperTest(System.Runtime.Intrinsics.Vector64`1[  
Single],System.Runtime.Intrinsics.Vector128`1[Double]):System.Runtime.Intrins  
ics.Vector128`1[Single]  
;  
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0  
HFA(simd8)  
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1  
HFA(simd16)  
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]  
"OutgoingArgSpace"  
; Lcl frame size = 0  
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    fcvtn2 v0.4s, v1.2d  
    ldp    fp, lr, [sp],#16  
    ret    lr  
  
; Total bytes of code 20, prolog size 8
```

84. ConvertToUInt32RoundAwayFromZero

Vector64<uint> ConvertToUInt32RoundAwayFromZero(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round to Nearest with [Ties to Away rounding mode](#), stores in the result vector and returns the result vector.

```
private Vector64<uint> ConvertToUInt32RoundAwayFromZeroTest(Vector64<float>
value)
{
    return AdvSimd.ConvertToUInt32RoundAwayFromZero(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<uint> ConvertToUInt32RoundAwayFromZero(Vector128<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt32RoundAwayFromZeroTest(System.Runtime.Intrinsics
.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvttau v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

85. ConvertToUInt32RoundAwayFromZeroScalar

Vector64<uint> ConvertToUInt32RoundAwayFromZeroScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round to Nearest with [Ties to Away rounding mode](#), stores in the result vector and returns the result vector.

```
private Vector64<uint>
ConvertToUInt32RoundAwayFromZeroScalarTest(Vector64<float> value)
{
    return AdvSimd.ConvertToUInt32RoundAwayFromZeroScalar(value);
}
// value = <11.5, 12.5>
// Result = <12, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ConvertToUInt32RoundAwayFromZeroScalarTest(System.Runtime.Intr
insics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvttau s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

86. ConvertToUInt32RoundToEven

Vector64<uint> ConvertToUInt32RoundToEven(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round to Nearest rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<uint> ConvertToUInt32RoundToEvenTest(Vector64<float> value)
{
    return AdvSimd.ConvertToUInt32RoundToEven(value);
}
// value = <11.5, 12.5>
// Result = <12, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<uint> ConvertToUInt32RoundToEven(Vector128<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt32RoundToEvenTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtnu v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

87. ConvertToUInt32RoundToEvenScalar

Vector64<uint> ConvertToUInt32RoundToEvenScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round to Nearest rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<uint> ConvertToUInt32RoundToEvenScalarTest(Vector64<float>
value)
{
    return AdvSimd.ConvertToUInt32RoundToEvenScalar(value);
}
// value = <11.5, 12.5>
// Result = <12, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt32RoundToEvenScalarTest(System.Runtime.Intrinsics
.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtnu s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

88. ConvertToUInt32RoundToNegativeInfinity

Vector64<uint> ConvertToUInt32RoundToNegativeInfinity(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round towards Minus Infinity rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<uint>
ConvertToUInt32RoundToNegativeInfinityTest(Vector64<float> value)
{
    return AdvSimd.ConvertToUInt32RoundToNegativeInfinity(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<uint> ConvertToUInt32RoundToNegativeInfinity(Vector128<float>
value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt32RoundToNegativeInfinityTest(System.Runtime.Intr
insics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtmu v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

89. ConvertToUInt32RoundToNegativeInfinityScalar

Vector64<uint> ConvertToUInt32RoundToNegativeInfinityScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round towards Minus Infinity rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<uint>
ConvertToUInt32RoundToNegativeInfinityScalarTest(Vector64<float> value)
{
    return AdvSimd.ConvertToUInt32RoundToNegativeInfinityScalar(value);
}
// value = <11.5, 12.5>
// Result = <11, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt32RoundToNegativeInfinityScalarTest(System.Runtime
e.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtmu s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

90. ConvertToUInt32RoundToPositiveInfinity

Vector64<uint> ConvertToUInt32RoundToPositiveInfinity(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round towards Plus Infinity rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<uint>
ConvertToUInt32RoundToPositiveInfinityTest(Vector64<float> value)
{
    return AdvSimd.ConvertToUInt32RoundToPositiveInfinity(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<uint> ConvertToUInt32RoundToPositiveInfinity(Vector128<float>
value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt32RoundToPositiveInfinityTest(System.Runtime.Intr
insics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtpu v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

91. ConvertToUInt32RoundToPositiveInfinityScalar

Vector64<uint> ConvertToUInt32RoundToPositiveInfinityScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round towards Plus Infinity rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<uint>
ConvertToUInt32RoundToPositiveInfinityScalarTest(Vector64<float> value)
{
    return AdvSimd.ConvertToUInt32RoundToPositiveInfinityScalar(value);
}
// value = <11.5, 12.5>
// Result = <12, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt32RoundToPositiveInfinityScalarTest(System.Runtime
e.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtpu s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

92. ConvertToUInt32RoundToZero

Vector64<uint> ConvertToUInt32RoundToZero(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round to Nearest with [toward zero rounding mode](#), stores in the result vector and returns the result vector.

```
private Vector64<uint> ConvertToUInt32RoundToZeroTest(Vector64<float> value)
{
    return AdvSimd.ConvertToUInt32RoundToZero(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<uint> ConvertToUInt32RoundToZero(Vector128<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt32RoundToZeroTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtzu v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

93. ConvertToUInt32RoundToZeroScalar

Vector64<uint> ConvertToUInt32RoundToZeroScalar(Vector64<float> value)

This method converts each element in the value vector from a floating-point to an unsigned integer value using the Round to Nearest with [toward zero rounding mode](#), stores in the result vector and returns the result vector.

```
private Vector64<uint> ConvertToUInt32RoundToZeroScalarTest(Vector64<float>
value)
{
    return AdvSimd.ConvertToUInt32RoundToZeroScalar(value);
}
// value = <11.5, 12.5>
// Result = <11, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt32RoundToZeroScalarTest(System.Runtime.Intrinsics
.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[UInt32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtzu s16, s0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

94. ConvertToUInt64RoundAwayFromZero

Vector128<ulong> ConvertToUInt64RoundAwayFromZero(Vector128<double> value)

This method converts each element in the value vector from a floating-point to a 64-bits unsigned integer value using the Round to Nearest with Ties to Away rounding mode, stores in the result vector and returns the result vector.

```
private Vector128<ulong>
ConvertToUInt64RoundAwayFromZeroTest(Vector128<double> value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundAwayFromZero(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundAwayFromZeroTest(System.Runtime.Intrinsics
.Vector128`1[Double]):System.Runtime.Intrinsics.Vector128`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvttau v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

95. ConvertToUInt64RoundAwayFromZeroScalar

Vector64<ulong> ConvertToUInt64RoundAwayFromZeroScalar(Vector64<double> value)

This method converts each element in the value vector from a floating-point to a 64-bits unsigned integer value using the Round to Nearest with Ties to Away rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<ulong>
ConvertToUInt64RoundAwayFromZeroScalarTest(Vector64<double> value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundAwayFromZeroScalar(value);
}
// value = <11.5>
// Result = <12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundAwayFromZeroScalarTest(System.Runtime.Intr
insics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvttau d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

96. ConvertToUInt64RoundToEven

Vector128<ulong> ConvertToUInt64RoundToEven(Vector128<double> value)

This method converts each element in the value vector from a floating-point to a 64-bits unsigned integer value using the Round to Nearest rounding mode, stores in the result vector and returns the result vector.

```
private Vector128<ulong> ConvertToUInt64RoundToEvenTest(Vector128<double>
value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundToEven(value);
}
// value = <11.5, 12.5>
// Result = <12, 12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundToEvenTest(System.Runtime.Intrinsics.Vecto
r128`1[Double]):System.Runtime.Intrinsics.Vector128`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtnu v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

97. ConvertToUInt64RoundToEvenScalar

Vector64<ulong> ConvertToUInt64RoundToEvenScalar(Vector64<double> value)

This method converts each element in the value vector from a floating-point to a 64-bits unsigned integer value using the Round to Nearest rounding mode, stores in the result vector and returns the result vector.

```
private Vector64<ulong> ConvertToUInt64RoundToEvenScalarTest(Vector64<double>
value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundToEvenScalar(value);
}
// value = <11.5>
// Result = <12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundToEvenScalarTest(System.Runtime.Intrinsics
.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtnu d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

98. ConvertToUInt64RoundToNegativeInfinity

Vector128<ulong> ConvertToUInt64RoundToNegativeInfinity(Vector128<double> value)

This method converts each element in the value vector from a floating-point value to a 64-bits unsigned integer value using the Round towards Minus Infinity rounding mode, and returns the result.

```
private Vector128<ulong>
ConvertToUInt64RoundToNegativeInfinityTest(Vector128<double> value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundToNegativeInfinity(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundToNegativeInfinityTest(System.Runtime.Intr
insics.Vector128`1[Double]):System.Runtime.Intrinsics.Vector128`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtmu v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

99. ConvertToUInt64RoundToNegativeInfinityScalar

Vector64<ulong> ConvertToUInt64RoundToNegativeInfinityScalar(Vector64<double> value)

This method converts each element in the value vector from a floating-point value to a 64-bits unsigned integer value using the Round towards Minus Infinity rounding mode, and returns the result.

```
private Vector64<ulong>
ConvertToUInt64RoundToNegativeInfinityScalarTest(Vector64<double> value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundToNegativeInfinityScalar(value);
}
// value = <11.5>
// Result = <11>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundToNegativeInfinityScalarTest(System.Runtime
e.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtmu d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

100. ConvertToUInt64RoundToPositiveInfinity

Vector128<ulong> ConvertToUInt64RoundToPositiveInfinity(Vector128<double> value)

This method converts each element in the value vector from a floating-point value to a 64-bits unsigned integer value using the Round towards Plus Infinity rounding mode, and returns the result.

```
private Vector128<ulong>
ConvertToUInt64RoundToPositiveInfinityTest(Vector128<double> value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundToPositiveInfinity(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundToPositiveInfinityTest(System.Runtime.Intr
insics.Vector128`1[Double]):System.Runtime.Intrinsics.Vector128`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtpu v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

101. ConvertToUInt64RoundToPositiveInfinityScalar

Vector64<ulong> ConvertToUInt64RoundToPositiveInfinityScalar(Vector64<double> value)

This method converts each element in the value vector from a floating-point value to a 64-bits unsigned integer value using the Round towards Plus Infinity rounding mode, and returns the result.

```
private Vector64<ulong>
ConvertToUInt64RoundToPositiveInfinityScalarTest(Vector64<double> value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundToPositiveInfinityScalar(value);
}
// value = <11.5>
// Result = <12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundToPositiveInfinityScalarTest(System.Runtime
e.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtpu d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

102. ConvertToUInt64RoundToZero

Vector128<ulong> ConvertToUInt64RoundToZero(Vector128<double> value)

This method converts each element in the value vector from a floating-point value to a 64-bits unsigned integer value using the Round towards Zero rounding mode, and returns the result.

```
private Vector128<ulong> ConvertToUInt64RoundToZeroTest(Vector128<double>
value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundToZero(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundToZeroTest(System.Runtime.Intrinsics.Vecto
r128`1[Double]):System.Runtime.Intrinsics.Vector128`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtzu v16.2d, v0.2d
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

103. ConvertToUInt64RoundToZeroScalar

Vector64<ulong> ConvertToUInt64RoundToZeroScalar(Vector64<double> value)

This method converts each element in the value vector from a floating-point value to a 64-bits unsigned integer value using the Round towards Zero rounding mode, and returns the result.

```
private Vector64<ulong> ConvertToUInt64RoundToZeroScalarTest(Vector64<double>
value)
{
    return AdvSimd.Arm64.ConvertToUInt64RoundToZeroScalar(value);
}
// value = <11.5>
// Result = <11>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ConvertToUInt64RoundToZeroScalarTest(System.Runtime.Intrinsics
.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fcvtzu d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

104. Divide

Vector64<float> Divide(Vector64<float> left, Vector64<float> right)

This method divides the corresponding floating-point values in the left vector, by those in the right vector, stores the result in a result vector, and returns the result vector.

```
private Vector64<float> DivideTest(Vector64<float> left, Vector64<float>
right)
{
    return AdvSimd.Arm64.Divide(left, right);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// Result = <0.53488374, 0.5555556>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector128<double> Divide(Vector128<double> left, Vector128<double> right)
Vector128<float> Divide(Vector128<float> left, Vector128<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:DivideTest(System.Runtime.Intrinsics.Vector64`1[Single],System
.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[
Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fdiv   v16.2s, v0.2s, v1.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

105. DivideScalar

Vector64<double> DivideScalar(Vector64<double> left, Vector64<double> right)

This method divides the corresponding floating-point values in the left vector, by those in the right vector, stores the result in a result vector, and returns the result vector.

```
private Vector64<double> DivideScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.DivideScalar(left, right);
}
// left = <11>
// right = <3.1>
// Result = <3.5483873>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> DivideScalar(Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:DivideScalarTest(System.Runtime.Intrinsics.Vector64`1[Double],
System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vecto
r64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fdiv   d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

106. DuplicateSelectedScalarToVector128

Vector128<byte> DuplicateSelectedScalarToVector128(Vector64<byte> value, byte index)

This method creates a vector by duplicating the vector element at index `index` into each element of the result vector. As seen in below example, the result vector elements count is double that of input parameter `value`.

```
private Vector128<byte> DuplicateSelectedScalarToVector128Test(Vector64<byte>
value, byte index)
{
    return AdvSimd.DuplicateSelectedScalarToVector128(value, 3);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// index = 3
// Result = <14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> DuplicateSelectedScalarToVector128(Vector64<short> value,
byte index)
Vector128<int> DuplicateSelectedScalarToVector128(Vector64<int> value, byte
index)
Vector128<float> DuplicateSelectedScalarToVector128(Vector64<float> value,
byte index)
Vector128<sbyte> DuplicateSelectedScalarToVector128(Vector64<sbyte> value,
byte index)
Vector128<ushort> DuplicateSelectedScalarToVector128(Vector64<ushort> value,
byte index)
Vector128<uint> DuplicateSelectedScalarToVector128(Vector64<uint> value, byte
index)
Vector128<byte> DuplicateSelectedScalarToVector128(Vector128<byte> value,
byte index)
Vector128<short> DuplicateSelectedScalarToVector128(Vector128<short> value,
byte index)
Vector128<int> DuplicateSelectedScalarToVector128(Vector128<int> value, byte
index)
Vector128<float> DuplicateSelectedScalarToVector128(Vector128<float> value,
byte index)
Vector128<sbyte> DuplicateSelectedScalarToVector128(Vector128<sbyte> value,
byte index)
Vector128<ushort> DuplicateSelectedScalarToVector128(Vector128<ushort> value,
byte index)
Vector128<uint> DuplicateSelectedScalarToVector128(Vector128<uint> value,
byte index)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> DuplicateSelectedScalarToVector128(Vector128<double> value,
```

```
byte index)
Vector128<long> DuplicateSelectedScalarToVector128(Vector128<long> value,
byte index)
Vector128<ulong> DuplicateSelectedScalarToVector128(Vector128<ulong> value,
byte index)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:DuplicateSelectedScalarToVector128Test(System.Runtime.Intrinsi
cs.Vector64`1[Byte],ubyte):System.Runtime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01  ] ( 0, 0 )  ubyte  ->  zero-ref
;# V02 OutArgs       [V02  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    dup    v16.16b, v0.b[3]
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

107. DuplicateSelectedScalarToVector64

Vector64<byte> DuplicateSelectedScalarToVector64(Vector64<byte> value, byte index)

This method creates a vector by duplicating the vector element at index in value vector into each element of the result vector.

```
private Vector64<byte> DuplicateSelectedScalarToVector64Test(Vector64<byte>
value, byte index)
{
    return AdvSimd.DuplicateSelectedScalarToVector64(value, 3);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// index = 3
// Result = <14, 14, 14, 14, 14, 14, 14, 14>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> DuplicateSelectedScalarToVector64(Vector64<short> value, byte
index)
Vector64<int> DuplicateSelectedScalarToVector64(Vector64<int> value, byte
index)
Vector64<float> DuplicateSelectedScalarToVector64(Vector64<float> value, byte
index)
Vector64<sbyte> DuplicateSelectedScalarToVector64(Vector64<sbyte> value, byte
index)
Vector64<ushort> DuplicateSelectedScalarToVector64(Vector64<ushort> value,
byte index)
Vector64<uint> DuplicateSelectedScalarToVector64(Vector64<uint> value, byte
index)
Vector64<byte> DuplicateSelectedScalarToVector64(Vector128<byte> value, byte
index)
Vector64<short> DuplicateSelectedScalarToVector64(Vector128<short> value,
byte index)
Vector64<int> DuplicateSelectedScalarToVector64(Vector128<int> value, byte
index)
Vector64<float> DuplicateSelectedScalarToVector64(Vector128<float> value,
byte index)
Vector64<sbyte> DuplicateSelectedScalarToVector64(Vector128<sbyte> value,
byte index)
Vector64<ushort> DuplicateSelectedScalarToVector64(Vector128<ushort> value,
byte index)
Vector64<uint> DuplicateSelectedScalarToVector64(Vector128<uint> value, byte
index)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:DuplicateSelectedScalarToVector64Test(System.Runtime.Intrinsic
s.Vector64`1[Byte],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01  ] ( 0, 0 )   ubyte  ->  zero-ref
;# V02 OutArgs       [V02  ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    dup    v16.8b, v0.b[3]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

108. DuplicateToVector128

Vector128<byte> DuplicateToVector128(byte value)

This method creates a vector by duplicating the value into each element in the result vector.

```
private Vector128<byte> DuplicateToVector128Test(byte value)
{
    return AdvSimd.DuplicateToVector128(value);
}
// value = 7
// Result = <7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> DuplicateToVector128(short value)
Vector128<int> DuplicateToVector128(int value)
Vector128<sbyte> DuplicateToVector128(sbyte value)
Vector128<float> DuplicateToVector128(float value)
Vector128<ushort> DuplicateToVector128(ushort value)
Vector128<uint> DuplicateToVector128(uint value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> DuplicateToVector128(double value)
Vector128<long> DuplicateToVector128(long value)
Vector128<ulong> DuplicateToVector128(ulong value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.DuplicateToVector128Test(ubyte):System.Runtime.Intrinsics.Vect
or128`1[Byte]
;
; V00 arg0 [V00,T00] ( 3, 3 ) ubyte -> x0
;# V01 OutArgs [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uxtb   w0, w0
    dup    v16.16b, w0
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 28, prolog size 8
```


109. DuplicateToVector64

Vector64<byte> DuplicateToVector64(byte value)

This method creates a vector by duplicating the value into each element in the result vector.

```
private Vector64<byte> DuplicateToVector64Test(byte value)
{
    return AdvSimd.DuplicateToVector64(value);
}
// value = 5
// Result = <5, 5, 5, 5, 5, 5, 5, 5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> DuplicateToVector64(short value)
Vector64<int> DuplicateToVector64(int value)
Vector64<sbyte> DuplicateToVector64(sbyte value)
Vector64<float> DuplicateToVector64(float value)
Vector64<ushort> DuplicateToVector64(ushort value)
Vector64<uint> DuplicateToVector64(uint value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:DuplicateToVector64Test(ubyte):System.Runtime.Intrinsics.Vecto
r64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) ubyte -> x0
;# V01 OutArgs      [V01  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uxtb    w0, w0
    dup    v16.8b, w0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 28, prolog size 8
```

110. Extract

byte Extract(Vector64<byte> vector, byte index)

This method extracts an element from vector at index and returns it.

```
private byte ExtractTest(Vector64<byte> vector, byte index)
{
    return AdvSimd.Extract(vector, 3);
}
// vector = <11, 12, 13, 14, 15, 16, 17, 18>
// index = 3
// Result = 14
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
short Extract(Vector64<short> vector, byte index)
int Extract(Vector64<int> vector, byte index)
sbyte Extract(Vector64<sbyte> vector, byte index)
float Extract(Vector64<float> vector, byte index)
ushort Extract(Vector64<ushort> vector, byte index)
uint Extract(Vector64<uint> vector, byte index)
byte Extract(Vector128<byte> vector, byte index)
double Extract(Vector128<double> vector, byte index)
short Extract(Vector128<short> vector, byte index)
int Extract(Vector128<int> vector, byte index)
long Extract(Vector128<long> vector, byte index)
sbyte Extract(Vector128<sbyte> vector, byte index)
float Extract(Vector128<float> vector, byte index)
ushort Extract(Vector128<ushort> vector, byte index)
uint Extract(Vector128<uint> vector, byte index)
ulong Extract(Vector128<ulong> vector, byte index)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ExtractTest(System.Runtime.Intrinsics.Vector64`1[Byte],ubyte):
ubyte
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    umov   w0, v0.b[3]
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

111. ExtractNarrowingLower

Vector64<byte> ExtractNarrowingLower(Vector128<ushort> value)

This method narrows each element in the value vector to half the original width, stores the result into a result vector and returns the vector. As seen in below example, the result vector element's size byte is half as long as that of input parameter element's size ushort.

```
private Vector64<byte> ExtractNarrowingLowerTest(Vector128<ushort> value)
{
    return AdvSimd.ExtractNarrowingLower(value);
}
// value = <300, 12, 413, 514, 15, 216, 117, 618>
// Result = <44, 12, 157, 2, 15, 216, 117, 106>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ExtractNarrowingLower(Vector128<int> value)
Vector64<int> ExtractNarrowingLower(Vector128<long> value)
Vector64<sbyte> ExtractNarrowingLower(Vector128<short> value)
Vector64<ushort> ExtractNarrowingLower(Vector128<uint> value)
Vector64<uint> ExtractNarrowingLower(Vector128<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ExtractNarrowingLowerTest(System.Runtime.Intrinsics.Vector128`
1[UInt16]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp     fp, lr, [sp,#-16]!
        mov     fp, sp
        xtn     v16.8b, v0.8h
        mov     v0.8b, v16.8b
        ldp     fp, lr, [sp],#16
        ret     lr

; Total bytes of code 24, prolog size 8
```

112. ExtractNarrowingSaturateLower

Vector64<byte> ExtractNarrowingSaturateLower(Vector128<ushort> value)

This method saturates each element in the value vector to half the original width, stores the result into a result vector, and returns the result vector.

```
private Vector64<byte> ExtractNarrowingSaturateLowerTest(Vector128<ushort>
value)
{
    return AdvSimd.ExtractNarrowingSaturateLower(value);
}
// value = <300, 12, 413, 514, 15, 216, 117, 618>
// Result = <255, 12, 255, 255, 15, 216, 117, 255>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ExtractNarrowingSaturateLower(Vector128<int> value)
Vector64<int> ExtractNarrowingSaturateLower(Vector128<long> value)
Vector64<sbyte> ExtractNarrowingSaturateLower(Vector128<short> value)
Vector64<ushort> ExtractNarrowingSaturateLower(Vector128<uint> value)
Vector64<uint> ExtractNarrowingSaturateLower(Vector128<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ExtractNarrowingSaturateLowerTest(System.Runtime.Intrinsics.Ve
ctor128`1[UInt16]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqxtn  v16.8b, v0.8h
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

113. ExtractNarrowingSaturateScalar

Vector64<byte> ExtractNarrowingSaturateScalar(Vector64<ushort> value)

This method saturates 0th element in the value vector to half the original width, stores the result into a result vector, and returns the result vector. Other elements except 0th element are initialized to 0.

```
private Vector64<byte> ExtractNarrowingSaturateScalarTest(Vector64<ushort>
value)
{
    return AdvSimd.Arm64.ExtractNarrowingSaturateScalar(value);
}
// value = <500, 500, 500, 500>
// Result = <255, 0, 0, 0, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<short> ExtractNarrowingSaturateScalar(Vector64<int> value)
Vector64<int> ExtractNarrowingSaturateScalar(Vector64<long> value)
Vector64<sbyte> ExtractNarrowingSaturateScalar(Vector64<short> value)
Vector64<ushort> ExtractNarrowingSaturateScalar(Vector64<uint> value)
Vector64<uint> ExtractNarrowingSaturateScalar(Vector64<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ExtractNarrowingSaturateScalarTest(System.Runtime.Intrinsics.V
ector64`1[UInt16]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqxtn  b16, h0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

114. ExtractNarrowingSaturateUnsignedLower

Vector64<byte> ExtractNarrowingSaturateUnsignedLower(Vector128<short> value)

This method saturates each element (which is always signed integer value) in the value vector to an unsigned integer value that is half the original width, stores the result in a result vector, and returns the result vector. As seen in below example, the result vector element's size byte is half as long as the input parameter value's element's size short.

```
private Vector64<byte>
ExtractNarrowingSaturateUnsignedLowerTest(Vector128<short> value)
{
    return AdvSimd.ExtractNarrowingSaturateUnsignedLower(value);
}
// value = <-300, -12, 413, 514, 15, 216, 117, 618>
// Result = <0, 0, 255, 255, 15, 216, 117, 255>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ushort> ExtractNarrowingSaturateUnsignedLower(Vector128<int> value)
Vector64<uint> ExtractNarrowingSaturateUnsignedLower(Vector128<long> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ExtractNarrowingSaturateUnsignedLowerTest(System.Runtime.Intri
nsics.Vector128`1[Int16]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqxtun v16.8b, v0.8h
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

115. ExtractNarrowingSaturateUnsignedScalar

Vector64<byte> ExtractNarrowingSaturateUnsignedScalar(Vector64<short> value)

This method saturates 0th element (which is always signed integer value) in the value vector to an unsigned integer value that is half the original width, stores the result in a result vector, and returns the result vector. As seen in below example, the result vector element's size byte is half as long as the input parameter value's element's size short. All the other elements of result vector except 0th element is initialized to 0.

```
private Vector64<byte>
ExtractNarrowingSaturateUnsignedScalarTest(Vector64<short> value)
{
    return AdvSimd.Arm64.ExtractNarrowingSaturateUnsignedScalar(value);
}
// value = <11, 12, 13, 14>
// Result = <11, 0, 0, 0, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<ushort> ExtractNarrowingSaturateUnsignedScalar(Vector64<int> value)
Vector64<uint> ExtractNarrowingSaturateUnsignedScalar(Vector64<long> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ExtractNarrowingSaturateUnsignedScalarTest(System.Runtime.Intr
insics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqxtun b16, h0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

116. ExtractNarrowingSaturateUnsignedUpper

Vector128<byte> ExtractNarrowingSaturateUnsignedUpper(Vector64<byte> lower, Vector128<short> value)

This method saturates each element (which is always signed integer value) in the upper half of value vector to an unsigned integer value that is half the original width, stores the result in the upper-half of result vector, and returns the result vector, the lower-half of the result vector contains values from lower vector. As seen in below example, the result vector element's size byte is half as long as the input parameter value's element's size short.

```
private Vector128<byte>
ExtractNarrowingSaturateUnsignedUpperTest(Vector64<byte> lower,
Vector128<short> value)
{
    return AdvSimd.ExtractNarrowingSaturateUnsignedUpper(lower, value);
}
// lower = <125, 12, 13, 14, 15, 216, 117, 18>
// value = <-500, 500, 12, 14, 257, 16, 17, 18>
// Result = <125, 12, 13, 14, 15, 216, 117, 18, 0, 255, 12, 14, 255, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<ushort> ExtractNarrowingSaturateUnsignedUpper(Vector64<ushort>
lower, Vector128<int> value)
Vector128<uint> ExtractNarrowingSaturateUnsignedUpper(Vector64<uint> lower,
Vector128<long> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ExtractNarrowingSaturateUnsignedUpperTest(System.Runtime.Intri
nsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector128`1[Int16]):System.R
untime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqxtun2 v0.16b, v1.8h
    ldp    fp, lr, [sp],#16
```



```
ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

117. ExtractNarrowingSaturateUpper

Vector128<byte> ExtractNarrowingSaturateUpper(Vector64<byte> lower, Vector128<ushort> value)

This method saturates each element in the upper-half of value vector to half the original width, stores the result into the upper-half of result vector, and returns the result vector, the lower half of result vector containing the values from lower vector.

```
private Vector128<byte> ExtractNarrowingSaturateUpperTest(Vector64<byte>
lower, Vector128<ushort> value)
{
    return AdvSimd.ExtractNarrowingSaturateUpper(lower, value);
}
// lower = <125, 12, 13, 14, 15, 216, 117, 18>
// value = <500, 500, 12, 14, 257, 16, 17, 18>
// Result = <125, 12, 13, 14, 15, 216, 117, 18, 255, 255, 12, 14, 255, 16,
17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> ExtractNarrowingSaturateUpper(Vector64<short> lower,
Vector128<int> value)
Vector128<int> ExtractNarrowingSaturateUpper(Vector64<int> lower,
Vector128<long> value)
Vector128<sbyte> ExtractNarrowingSaturateUpper(Vector64<sbyte> lower,
Vector128<short> value)
Vector128<ushort> ExtractNarrowingSaturateUpper(Vector64<ushort> lower,
Vector128<uint> value)
Vector128<uint> ExtractNarrowingSaturateUpper(Vector64<uint> lower,
Vector128<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ExtractNarrowingSaturateUpperTest(System.Runtime.Intrinsics.Ve
ctor64`1[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16]):System.Runtime.
Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
```

```
uqxtn2  v0.16b, v1.8h
ldp      fp, lr, [sp],#16
ret      lr
```

; Total bytes of code 20, prolog size 8

118. ExtractNarrowingUpper

Vector128<byte> ExtractNarrowingUpper(Vector64<byte> lower, Vector128<ushort> value)

This method narrows each element in the upper half of value vector to half the original width, stores the result in the upper half of result vector and returns the vector. The lower half of result vector contains values from lower vector. As seen in below example, the result vector element's size byte is half as long as that of input parameter element's size ushort.

```
private Vector128<byte> ExtractNarrowingUpperTest(Vector64<byte> lower,
Vector128<ushort> value)
{
    return AdvSimd.ExtractNarrowingUpper(lower, value);
}
// lower = <125, 12, 13, 14, 15, 216, 117, 18>
// value = <500, 500, 12, 14, 257, 16, 17, 18>
// Result = <125, 12, 13, 14, 15, 216, 117, 18, 244, 244, 12, 14, 1, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> ExtractNarrowingUpper(Vector64<short> lower, Vector128<int>
value)
Vector128<int> ExtractNarrowingUpper(Vector64<int> lower, Vector128<long>
value)
Vector128<sbyte> ExtractNarrowingUpper(Vector64<sbyte> lower,
Vector128<short> value)
Vector128<ushort> ExtractNarrowingUpper(Vector64<ushort> lower,
Vector128<uint> value)
Vector128<uint> ExtractNarrowingUpper(Vector64<uint> lower, Vector128<ulong>
value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ExtractNarrowingUpperTest(System.Runtime.Intrinsics.Vector64`1
[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16]):System.Runtime.Intrinsi
cs.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
stp fp, lr, [sp,#-16]!
```

```
mov    fp, sp
xtn2   v0.16b, v1.8h
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 20, prolog size 8

119. ExtractVector128

Vector128<byte> ExtractVector128(Vector128<byte> upper, Vector128<byte> lower, byte index)

This method extracts the vector elements from upper starting at index (and hence should be less than the size of vector) and fills the result vector. Once the upper vector runs out and there is room to fill in, elements from lower elements are picked.

```
private Vector128<byte> ExtractVector128Test(Vector128<byte> upper,
Vector128<byte> lower, byte index)
{
    return AdvSimd.ExtractVector128(upper, lower, 5);
}
// upper = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// lower = <31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 42, 44, 45, 46>
// index = 5
// Result = <16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 31, 32, 33, 34, 35>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<double> ExtractVector128(Vector128<double> upper, Vector128<double>
lower, byte index)
Vector128<short> ExtractVector128(Vector128<short> upper, Vector128<short>
lower, byte index)
Vector128<int> ExtractVector128(Vector128<int> upper, Vector128<int> lower,
byte index)
Vector128<long> ExtractVector128(Vector128<long> upper, Vector128<long>
lower, byte index)
Vector128<sbyte> ExtractVector128(Vector128<sbyte> upper, Vector128<sbyte>
lower, byte index)
Vector128<float> ExtractVector128(Vector128<float> upper, Vector128<float>
lower, byte index)
Vector128<ushort> ExtractVector128(Vector128<ushort> upper, Vector128<ushort>
lower, byte index)
Vector128<uint> ExtractVector128(Vector128<uint> upper, Vector128<uint>
lower, byte index)
Vector128<ulong> ExtractVector128(Vector128<ulong> upper, Vector128<ulong>
lower, byte index)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ExtractVector128Test(System.Runtime.Intrinsics.Vector128`1[Byte
e],System.Runtime.Intrinsics.Vector128`1[Byte],ubyte):System.Runtime.Intrinsi
cs.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
```

```

HFA(simd16)
; V01 arg1      [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;* V02 arg2     [V02  ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs  [V03  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ext    v16.16b, v0.16b, v1.16b, #5
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

120. ExtractVector64

Vector64<byte> ExtractVector64(Vector64<byte> upper, Vector64<byte> lower, byte index)

This method extracts the vector elements from upper starting at index (and hence should be less than the size of vector) and fills the result vector. Once the upper vector runs out and there is room to fill in, elements from lower elements are picked. This method is same as ExtractVector128() except it operates on Vector64<T>.

```
private Vector64<byte> ExtractVector64Test(Vector64<byte> upper,
Vector64<byte> lower, byte index)
{
    return AdvSimd.ExtractVector64(upper, lower, 5);
}
// upper = <11, 12, 13, 14, 15, 16, 17, 18>
// lower = <21, 22, 23, 24, 25, 26, 27, 28>
// index = 5
// Result = <16, 17, 18, 21, 22, 23, 24, 25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ExtractVector64(Vector64<short> upper, Vector64<short> lower,
byte index)
Vector64<int> ExtractVector64(Vector64<int> upper, Vector64<int> lower, byte
index)
Vector64<sbyte> ExtractVector64(Vector64<sbyte> upper, Vector64<sbyte> lower,
byte index)
Vector64<float> ExtractVector64(Vector64<float> upper, Vector64<float> lower,
byte index)
Vector64<ushort> ExtractVector64(Vector64<ushort> upper, Vector64<ushort>
lower, byte index)
Vector64<uint> ExtractVector64(Vector64<uint> upper, Vector64<uint> lower,
byte index)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ExtractVector64Test(System.Runtime.Intrinsics.Vector64`1[Byte]
,System.Runtime.Intrinsics.Vector64`1[Byte],ubyte):System.Runtime.Intrinsics.
Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
```



```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ext    v16.8b, v0.8b, v1.8b, #5
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

121. Floor

Vector64<float> Floor(Vector64<float> value)

This method rounds each element in the value vector containing floating-point values to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, places the result in a vector and return the result vector. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<float> FloorTest(Vector64<float> value)
{
    return AdvSimd.Floor(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> Floor(Vector128<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Floor(Vector128<double> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FloorTest(System.Runtime.Intrinsics.Vector64`1[Single]):System
.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintm v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

122. FloorScalar

Vector64<double> FloorScalar(Vector64<double> value)

This method rounds each element in the value vector containing floating-point values to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, places the result in a vector and return the result vector. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<double> FloorScalarTest(Vector64<double> value)
{
    return AdvSimd.FloorScalar(value);
}
// value = <11.5>
// Result = <11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> FloorScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.FloorScalarTest(System.Runtime.Intrinsics.Vector64`1[Double]):
System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintm d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

123. FusedAddHalving

Vector64<byte> FusedAddHalving(Vector64<byte> left, Vector64<byte> right)

This method adds corresponding element values from the left and right vectors, shifts each result right one bit, places the truncated results in a vector, and returns the result vector.

```
private Vector64<byte> FusedAddHalvingTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.FusedAddHalving(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <16, 17, 18, 19, 20, 21, 22, 23>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> FusedAddHalving(Vector64<short> left, Vector64<short> right)
Vector64<int> FusedAddHalving(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> FusedAddHalving(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<ushort> FusedAddHalving(Vector64<ushort> left, Vector64<ushort>
right)
Vector64<uint> FusedAddHalving(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> FusedAddHalving(Vector128<byte> left, Vector128<byte> right)
Vector128<short> FusedAddHalving(Vector128<short> left, Vector128<short>
right)
Vector128<int> FusedAddHalving(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> FusedAddHalving(Vector128<sbyte> left, Vector128<sbyte>
right)
Vector128<ushort> FusedAddHalving(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> FusedAddHalving(Vector128<uint> left, Vector128<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedAddHalvingTest(System.Runtime.Intrinsics.Vector64`1[Byte]
,System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector
64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uhadd  v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

124. FusedAddRoundedHalving

Vector64<byte> FusedAddRoundedHalving(Vector64<byte> left, Vector64<byte> right)

This method adds corresponding element values from the left and right vectors, shifts each result right one bit, places the rounded results in a vector, and returns the result vector.

```
private Vector64<byte> FusedAddRoundedHalvingTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.FusedAddRoundedHalving(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <16, 17, 18, 19, 20, 21, 22, 23>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> FusedAddRoundedHalving(Vector64<short> left, Vector64<short>
right)
Vector64<int> FusedAddRoundedHalving(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> FusedAddRoundedHalving(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector64<ushort> FusedAddRoundedHalving(Vector64<ushort> left,
Vector64<ushort> right)
Vector64<uint> FusedAddRoundedHalving(Vector64<uint> left, Vector64<uint>
right)
Vector128<byte> FusedAddRoundedHalving(Vector128<byte> left, Vector128<byte>
right)
Vector128<short> FusedAddRoundedHalving(Vector128<short> left,
Vector128<short> right)
Vector128<int> FusedAddRoundedHalving(Vector128<int> left, Vector128<int>
right)
Vector128<sbyte> FusedAddRoundedHalving(Vector128<sbyte> left,
Vector128<sbyte> right)
Vector128<ushort> FusedAddRoundedHalving(Vector128<ushort> left,
Vector128<ushort> right)
Vector128<uint> FusedAddRoundedHalving(Vector128<uint> left, Vector128<uint>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedAddRoundedHalvingTest(System.Runtime.Intrinsics.Vector64`
1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics
.Vector64`1[Byte]
```

```

;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    urhadd v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

125. FusedMultiplyAdd

Vector64<float> FusedMultiplyAdd(Vector64<float> addend, Vector64<float> left, Vector64<float> right)

This method multiplies corresponding floating-point values in the vectors in the left and right vectors, adds the product to the vector elements of the addended vector, and returns the accumulated result vector.

```
private Vector64<float> FusedMultiplyAddTest(Vector64<float> addend,
Vector64<float> left, Vector64<float> right)
{
    return AdvSimd.FusedMultiplyAdd(addend, left, right);
}
// addend = <11.5, 12.5>
// left = <21.5, 22.5>
// right = <11.5, 12.5>
// Result = <258.75, 293.75>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> FusedMultiplyAdd(Vector128<float> addend, Vector128<float>
left, Vector128<float> right)
```

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> FusedMultiplyAdd(Vector128<double> addend,
Vector128<double> left, Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplyAddTest(System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmla   v0.2s, v1.2s, v2.2s
    ldp    fp, lr, [sp],#16
```



```
ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

126. FusedMultiplyAddByScalar

Vector64<float> FusedMultiplyAddByScalar(Vector64<float> addend, Vector64<float> left, Vector64<float> right)

This method multiplies floating-point value element at 0th index of right vector with elements in the left vector, adds the product to the vector elements of the addended vector, and returns the accumulated result vector.

```
private Vector64<float> FusedMultiplyAddByScalarTest(Vector64<float> addend,
Vector64<float> left, Vector64<float> right)
{
    return AdvSimd.Arm64.FusedMultiplyAddByScalar(addend, left, right);
}
// addend = <11.5, 12.5>
// left = <21.5, 22.5>
// right = <11.5, 12.5>
// Result = <258.75, 271.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector128<double> FusedMultiplyAddByScalar(Vector128<double> addend,
Vector128<double> left, Vector64<double> right)
Vector128<float> FusedMultiplyAddByScalar(Vector128<float> addend,
Vector128<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplyAddByScalarTest(System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmla   v0.2s, v1.2s, v2.s[0]
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 20, prolog size 8

127. FusedMultiplyAddBySelectedScalar

Vector64<float> FusedMultiplyAddBySelectedScalar(Vector64<float> addend, Vector64<float> left, Vector64<float> right, byte rightIndex)

This method multiplies floating-point value element at `rightIndex` index of `right` vector with elements in the `left` vector, adds the product to the vector elements of the `addend` vector, and returns the accumulated result vector.

```
private Vector64<float> FusedMultiplyAddBySelectedScalarTest(Vector64<float>
addend, Vector64<float> left, Vector64<float> right, byte rightIndex)
{
    return AdvSimd.Arm64.FusedMultiplyAddBySelectedScalar(addend, left, right,
0);
}
// addend = <11.5, 12.5>
// left = <21.5, 22.5>
// right = <11.5, 12.5>
// rightIndex = 0
// Result = <258.75, 271.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> FusedMultiplyAddBySelectedScalar(Vector64<float> addend,
Vector64<float> left, Vector128<float> right, byte rightIndex)
Vector128<double> FusedMultiplyAddBySelectedScalar(Vector128<double> addend,
Vector128<double> left, Vector128<double> right, byte rightIndex)
Vector128<float> FusedMultiplyAddBySelectedScalar(Vector128<float> addend,
Vector128<float> left, Vector64<float> right, byte rightIndex)
Vector128<float> FusedMultiplyAddBySelectedScalar(Vector128<float> addend,
Vector128<float> left, Vector128<float> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplyAddBySelectedScalarTest(System.Runtime.Intrinsics
.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single],System.Runti
me.Intrinsics.Vector64`1[Single],ubyte):System.Runtime.Intrinsics.Vector64`1[
Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;* V03 arg3          [V03 ] ( 0, 0 ) ubyte -> zero-ref
;# V04 OutArgs       [V04 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmla   v0.2s, v1.2s, v2.s[0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

128. FusedMultiplyAddNegatedScalar

Vector64<double> FusedMultiplyAddNegatedScalar(Vector64<double> addend, Vector64<double> left, Vector64<double> right)

This method multiplies the values of the left and right vector, negates the product, subtracts the value of the addend vector from the product, and returns the result.

```
private Vector64<double> FusedMultiplyAddNegatedScalarTest(Vector64<double>
addend, Vector64<double> left, Vector64<double> right)
{
    return AdvSimd.FusedMultiplyAddNegatedScalar(addend, left, right);
}
// addend = <100.5>
// left = <5.5>
// right = <15.5>
// Result = <-185.75>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> FusedMultiplyAddNegatedScalar(Vector64<float> addend,
Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplyAddNegatedScalarTest(System.Runtime.Intrinsics.Ve
ctor64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double],System.Runtime.
Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;# V03 OutArgs  [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fnmadd d16, d1, d2, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


129. FusedMultiplyAddScalar

**Vector64<double> FusedMultiplyAddScalar(Vector64<double> addend,
Vector64<double> left, Vector64<double> right)**

This method multiplies corresponding floating-point values in the vectors in the left and right vectors, adds the product to the vector elements of the addended vector, and returns the accumulated result vector.

```
private Vector64<double> FusedMultiplyAddScalarTest(Vector64<double> addend,  
Vector64<double> left, Vector64<double> right)  
{  
    return AdvSimd.FusedMultiplyAddScalar(addend, left, right);  
}  
// addend = <100.5>  
// left = <5.5>  
// right = <15.5>  
// Result = <185.75>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector64<float> FusedMultiplyAddScalar(Vector64<float> addend,  
Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods:FusedMultiplyAddScalarTest(System.Runtime.Intrinsics.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]  
;  
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0  
HFA(simd8)  
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1  
HFA(simd8)  
; V02 arg2      [V02,T02] ( 3, 3 ) simd8 -> d2  
HFA(simd8)  
;# V03 OutArgs  [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]  
"OutgoingArgSpace"  
; Lcl frame size = 0  
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    fmadd  d16, d1, d2, d0  
    mov    v0.8b, v16.8b  
    ldp    fp, lr, [sp],#16  
    ret    lr  
  
; Total bytes of code 24, prolog size 8
```


130. FusedMultiplyAddScalarBySelectedScalar

Vector64<double> FusedMultiplyAddScalarBySelectedScalar(Vector64<double> addend, Vector64<double> left, Vector128<double> right, byte rightIndex)

This method multiplies the vector elements in the left vector by an element at rightIndex of the right vector, and accumulates the product to the corresponding vector elements of the addend vector and returns the result vector.

```
private Vector64<double>
FusedMultiplyAddScalarBySelectedScalarTest(Vector64<double> addend,
Vector64<double> left, Vector128<double> right, byte rightIndex)
{
    return AdvSimd.Arm64.FusedMultiplyAddScalarBySelectedScalar(addend, left,
right, 0);
}
// addend = <11.5>
// left = <11.5>
// right = <11.5, 12.5>
// rightIndex = 0
// Result = <143.75>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> FusedMultiplyAddScalarBySelectedScalar(Vector64<float>
addend, Vector64<float> left, Vector64<float> right, byte rightIndex)
Vector64<float> FusedMultiplyAddScalarBySelectedScalar(Vector64<float>
addend, Vector64<float> left, Vector128<float> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplyAddScalarBySelectedScalarTest(System.Runtime.Intr
insics.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double],System
.Runtime.Intrinsics.Vector128`1[Double],ubyte):System.Runtime.Intrinsics.Vect
or64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 )   simd16 ->  d2
HFA(simd16)
;* V03 arg3          [V03      ] ( 0, 0 )   ubyte  ->  zero-ref
;# V04 OutArgs       [V04      ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
                stp    fp, lr, [sp,#-16]!
```

```
mov    fp, sp
fmla   d0, d1, v2.d[0]
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 20, prolog size 8

131. FusedMultiplySubtract

Vector64<float> FusedMultiplySubtract(Vector64<float> minuend, Vector64<float> left, Vector64<float> right)

This method multiplies corresponding floating-point values in the vectors in the left and right vectors, negates the product, adds the product to the corresponding vector element of minuend vector, and returns the result.

```
private Vector64<float> FusedMultiplySubtractTest(Vector64<float> minuend,
Vector64<float> left, Vector64<float> right)
{
    return AdvSimd.FusedMultiplySubtract(minuend, left, right);
}
// minuend = <11.5, 12.5>
// left = <21.5, 22.5>
// right = <11.5, 12.5>
// Result = <-235.75, -268.75>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> FusedMultiplySubtract(Vector128<float> minuend,
Vector128<float> left, Vector128<float> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> FusedMultiplySubtract(Vector128<double> minuend,
Vector128<double> left, Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplySubtractTest(System.Runtime.Intrinsics.Vector64`1
[Single],System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsi
cs.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmls   v0.2s, v1.2s, v2.2s
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

132. FusedMultiplySubtractByScalar

Vector64<float> FusedMultiplySubtractByScalar(Vector64<float> minuend, Vector64<float> left, Vector64<float> right)

This method multiplies floating-point value element at 0th index of right vector with elements in the left vector, negates the product, adds the product to the vector elements of the minuend vector, and returns the accumulated result vector.

```
private Vector64<float> FusedMultiplySubtractByScalarTest(Vector64<float>
minuend, Vector64<float> left, Vector64<float> right)
{
    return AdvSimd.Arm64.FusedMultiplySubtractByScalar(minuend, left, right);
}
// minuend = <11.5, 12.5>
// left = <21.5, 22.5>
// right = <11.5, 12.5>
// Result = <-235.75, -246.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector128<double> FusedMultiplySubtractByScalar(Vector128<double> minuend,
Vector128<double> left, Vector64<double> right)
Vector128<float> FusedMultiplySubtractByScalar(Vector128<float> minuend,
Vector128<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplySubtractByScalarTest(System.Runtime.Intrinsics.Ve
ctor64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.
Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmls   v0.2s, v1.2s, v2.s[0]
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 20, prolog size 8

133. FusedMultiplySubtractBySelectedScalar

Vector64<float> FusedMultiplySubtractBySelectedScalar(Vector64<float> minuend, Vector64<float> left, Vector64<float> right, byte rightIndex)

This method multiplies floating-point value element at `rightIndex` index of `right` vector with elements in the `left` vector, negates the product, adds the product to the vector elements of the `minuend` vector, and returns the accumulated result vector.

```
private Vector64<float>
FusedMultiplySubtractBySelectedScalarTest(Vector64<float> minuend,
Vector64<float> left, Vector64<float> right, byte rightIndex)
{
    return AdvSimd.Arm64.FusedMultiplySubtractBySelectedScalar(minuend, left,
right, 0);
}
// minuend = <11.5, 12.5>
// left = <21.5, 22.5>
// right = <11.5, 12.5>
// rightIndex = 0
// Result = <-235.75, -246.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> FusedMultiplySubtractBySelectedScalar(Vector64<float>
minuend, Vector64<float> left, Vector128<float> right, byte rightIndex)
Vector128<double> FusedMultiplySubtractBySelectedScalar(Vector128<double>
minuend, Vector128<double> left, Vector128<double> right, byte rightIndex)
Vector128<float> FusedMultiplySubtractBySelectedScalar(Vector128<float>
minuend, Vector128<float> left, Vector64<float> right, byte rightIndex)
Vector128<float> FusedMultiplySubtractBySelectedScalar(Vector128<float>
minuend, Vector128<float> left, Vector128<float> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplySubtractBySelectedScalarTest(System.Runtime.Intri
nsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single],System.
Runtime.Intrinsics.Vector64`1[Single],ubyte):System.Runtime.Intrinsics.Vector
64`1[Single]
;
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] (  3,  3  )  simd8  ->  d1
HFA(simd8)
; V02 arg2          [V02,T02] (  3,  3  )  simd8  ->  d2
HFA(simd8)
;* V03 arg3          [V03      ] (  0,  0  )  ubyte  ->  zero-ref
```



```

;# V04 OutArgs      [V04      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmls   v0.2s, v1.2s, v2.s[0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

134. FusedMultiplySubtractNegatedScalar

Vector64<double> FusedMultiplySubtractNegatedScalar(Vector64<double> minuend, Vector64<double> left, Vector64<double> right)

This method multiplies the values of the left and right vectors, subtracts the value of the minuend vector, and returns the result.

```
private Vector64<double>
FusedMultiplySubtractNegatedScalarTest(Vector64<double> minuend,
Vector64<double> left, Vector64<double> right)
{
    return AdvSimd.FusedMultiplySubtractNegatedScalar(minuend, left, right);
}
// minuend = <11.5>
// left = <11.5>
// right = <11>
// Result = <115>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> FusedMultiplySubtractNegatedScalar(Vector64<float> minuend,
Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplySubtractNegatedScalarTest(System.Runtime.Intrinsi
cs.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double],System.Run
time.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Doub
le]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fnmsub d16, d1, d2, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

135. FusedMultiplySubtractScalar

Vector64<double> FusedMultiplySubtractScalar(Vector64<double> minuend, Vector64<double> left, Vector64<double> right)

This method multiplies the values of the left and right vectors, negates the product, adds that to the value of the minuend vector, and returns the result.

```
private Vector64<double> FusedMultiplySubtractScalarTest(Vector64<double>
minuend, Vector64<double> left, Vector64<double> right)
{
    return AdvSimd.FusedMultiplySubtractScalar(minuend, left, right);
}
// minuend = <11.5>
// left = <11.5>
// right = <11>
// Result = <-115>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> FusedMultiplySubtractScalar(Vector64<float> minuend,
Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplySubtractScalarTest(System.Runtime.Intrinsics.Vect
or64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double],System.Runtime.In
trinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmsub  d16, d1, d2, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


136. FusedMultiplySubtractScalarBySelectedScalar

Vector64<double> FusedMultiplySubtractScalarBySelectedScalar(Vector64<double> minuend, Vector64<double> left, Vector128<double> right, byte rightIndex)

This method multiplies the vector elements in the left vector by the rightIndex element in the right vector, and subtracts the results from the vector elements of the minuend vector and returns the result.

```
private Vector64<double>
FusedMultiplySubtractScalarBySelectedScalarTest(Vector64<double> minuend,
Vector64<double> left, Vector128<double> right, byte rightIndex)
{
    return AdvSimd.Arm64.FusedMultiplySubtractScalarBySelectedScalar(minuend,
left, right, 0);
}
// minuend = <11.5>
// left = <11.5>
// right = <11.5, 12.5>
// rightIndex = 0
// Result = <-120.75>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> FusedMultiplySubtractScalarBySelectedScalar(Vector64<float>
minuend, Vector64<float> left, Vector64<float> right, byte rightIndex)
Vector64<float> FusedMultiplySubtractScalarBySelectedScalar(Vector64<float>
minuend, Vector64<float> left, Vector128<float> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedMultiplySubtractScalarBySelectedScalarTest(System.Runtime
.Intrinsics.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double],S
ystem.Runtime.Intrinsics.Vector128`1[Double],ubyte):System.Runtime.Intrinsics
.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd16 -> d2
HFA(simd16)
;* V03 arg3         [V03 ] ( 0, 0 ) ubyte -> zero-ref
;# V04 OutArgs      [V04 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
                stp    fp, lr, [sp,#-16]!
```

```
mov    fp, sp
fmls   d0, d1, v2.d[0]
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 20, prolog size 8

137. FusedSubtractHalving

Vector64<byte> FusedSubtractHalving(Vector64<byte> left, Vector64<byte> right)

This method subtracts the corresponding vector elements in the right vector from those of left vector, shifts each result right one bit, stores the result in a vector, and returns the result vector.

```
private Vector64<byte> FusedSubtractHalvingTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.FusedSubtractHalving(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <251, 251, 251, 251, 251, 251, 251, 251>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> FusedSubtractHalving(Vector64<short> left, Vector64<short>
right)
Vector64<int> FusedSubtractHalving(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> FusedSubtractHalving(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector64<ushort> FusedSubtractHalving(Vector64<ushort> left, Vector64<ushort>
right)
Vector64<uint> FusedSubtractHalving(Vector64<uint> left, Vector64<uint>
right)
Vector128<byte> FusedSubtractHalving(Vector128<byte> left, Vector128<byte>
right)
Vector128<short> FusedSubtractHalving(Vector128<short> left, Vector128<short>
right)
Vector128<int> FusedSubtractHalving(Vector128<int> left, Vector128<int>
right)
Vector128<sbyte> FusedSubtractHalving(Vector128<sbyte> left, Vector128<sbyte>
right)
Vector128<ushort> FusedSubtractHalving(Vector128<ushort> left,
Vector128<ushort> right)
Vector128<uint> FusedSubtractHalving(Vector128<uint> left, Vector128<uint>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:FusedSubtractHalvingTest(System.Runtime.Intrinsics.Vector64`1[
Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.V
ector64`1[Byte]
```



```

;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uhsb   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

138. Insert

Vector64<byte> Insert(Vector64<byte> vector, byte index, byte data)

This method copies the vector vector in result vector with the element at index set to data value.

```
private Vector64<byte> InsertTest(Vector64<byte> vector, byte index, byte data)
{
    return AdvSimd.Insert(vector, 4, 200);
}
// vector = <11, 12, 13, 14, 15, 16, 17, 18>
// index = 4
// data = 200
// Result = <11, 12, 13, 14, 200, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> Insert(Vector64<short> vector, byte index, short data)
Vector64<int> Insert(Vector64<int> vector, byte index, int data)
Vector64<sbyte> Insert(Vector64<sbyte> vector, byte index, sbyte data)
Vector64<float> Insert(Vector64<float> vector, byte index, float data)
Vector64<ushort> Insert(Vector64<ushort> vector, byte index, ushort data)
Vector64<uint> Insert(Vector64<uint> vector, byte index, uint data)
Vector128<byte> Insert(Vector128<byte> vector, byte index, byte data)
Vector128<double> Insert(Vector128<double> vector, byte index, double data)
Vector128<short> Insert(Vector128<short> vector, byte index, short data)
Vector128<int> Insert(Vector128<int> vector, byte index, int data)
Vector128<long> Insert(Vector128<long> vector, byte index, long data)
Vector128<sbyte> Insert(Vector128<sbyte> vector, byte index, sbyte data)
Vector128<float> Insert(Vector128<float> vector, byte index, float data)
Vector128<ushort> Insert(Vector128<ushort> vector, byte index, ushort data)
Vector128<uint> Insert(Vector128<uint> vector, byte index, uint data)
Vector128<ulong> Insert(Vector128<ulong> vector, byte index, ulong data)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.InsertTest(System.Runtime.Intrinsics.Vector64`1[Byte],ubyte,ub
yte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;* V02 arg2         [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    mov    w0, #200
    ins    v0.b[4], w0
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

139. InsertScalar

Vector128<double> InsertScalar(Vector128<double> result, byte resultIndex, Vector64<double> value)

This method copies the result vector in a result vector, except the element at resultIndex of result is set to that from value vector.

```
private Vector128<double> InsertScalarTest(Vector128<double> result, byte
resultIndex, Vector64<double> value)
{
    return AdvSimd.InsertScalar(result, 1, value);
}
// result = <5.5, 5.5>
// resultIndex = 1
// value = <15.5>
// Result = <5.5, 15.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long> InsertScalar(Vector128<long> result, byte resultIndex,
Vector64<long> value)
Vector128<ulong> InsertScalar(Vector128<ulong> result, byte resultIndex,
Vector64<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:InsertScalarTest(System.Runtime.Intrinsics.Vector128`1[Double]
,ubyte,System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsic
s.Vector128`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
; V02 arg2          [V02,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ins    v0.d[1], v1.d[0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```


140. InsertSelectedScalar

Vector64<byte> InsertSelectedScalar(Vector64<byte> result, byte resultIndex, Vector64<byte> value, byte valueIndex)

This method copies the result vector in a result vector, except the element at resultIndex of result is set to that of valueIndex element of value vector.

```
private Vector64<byte> InsertSelectedScalarTest(Vector64<byte> result, byte
resultIndex, Vector64<byte> value, byte valueIndex)
{
    return AdvSimd.Arm64.InsertSelectedScalar(result, 0, value, 1);
}
// result = <11, 12, 13, 14, 15, 16, 17, 18>
// resultIndex = 0
// value = <21, 22, 23, 24, 25, 26, 27, 28>
// valueIndex = 1
// Result = <22, 12, 13, 14, 15, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<byte> InsertSelectedScalar(Vector64<byte> result, byte resultIndex,
Vector128<byte> value, byte valueIndex)
Vector64<short> InsertSelectedScalar(Vector64<short> result, byte
resultIndex, Vector64<short> value, byte valueIndex)
Vector64<short> InsertSelectedScalar(Vector64<short> result, byte
resultIndex, Vector128<short> value, byte valueIndex)
Vector64<int> InsertSelectedScalar(Vector64<int> result, byte resultIndex,
Vector64<int> value, byte valueIndex)
Vector64<int> InsertSelectedScalar(Vector64<int> result, byte resultIndex,
Vector128<int> value, byte valueIndex)
Vector64<sbyte> InsertSelectedScalar(Vector64<sbyte> result, byte
resultIndex, Vector64<sbyte> value, byte valueIndex)
Vector64<sbyte> InsertSelectedScalar(Vector64<sbyte> result, byte
resultIndex, Vector128<sbyte> value, byte valueIndex)
Vector64<float> InsertSelectedScalar(Vector64<float> result, byte
resultIndex, Vector64<float> value, byte valueIndex)
Vector64<float> InsertSelectedScalar(Vector64<float> result, byte
resultIndex, Vector128<float> value, byte valueIndex)
Vector64<ushort> InsertSelectedScalar(Vector64<ushort> result, byte
resultIndex, Vector64<ushort> value, byte valueIndex)
Vector64<ushort> InsertSelectedScalar(Vector64<ushort> result, byte
resultIndex, Vector128<ushort> value, byte valueIndex)
Vector64<uint> InsertSelectedScalar(Vector64<uint> result, byte resultIndex,
Vector64<uint> value, byte valueIndex)
Vector64<uint> InsertSelectedScalar(Vector64<uint> result, byte resultIndex,
Vector128<uint> value, byte valueIndex)
Vector128<byte> InsertSelectedScalar(Vector128<byte> result, byte
resultIndex, Vector64<byte> value, byte valueIndex)
Vector128<byte> InsertSelectedScalar(Vector128<byte> result, byte
```

```

resultIndex, Vector128<byte> value, byte valueIndex)
Vector128<double> InsertSelectedScalar(Vector128<double> result, byte
resultIndex, Vector128<double> value, byte valueIndex)
Vector128<short> InsertSelectedScalar(Vector128<short> result, byte
resultIndex, Vector64<short> value, byte valueIndex)
Vector128<short> InsertSelectedScalar(Vector128<short> result, byte
resultIndex, Vector128<short> value, byte valueIndex)
Vector128<int> InsertSelectedScalar(Vector128<int> result, byte resultIndex,
Vector64<int> value, byte valueIndex)
Vector128<int> InsertSelectedScalar(Vector128<int> result, byte resultIndex,
Vector128<int> value, byte valueIndex)
Vector128<long> InsertSelectedScalar(Vector128<long> result, byte
resultIndex, Vector128<long> value, byte valueIndex)
Vector128<sbyte> InsertSelectedScalar(Vector128<sbyte> result, byte
resultIndex, Vector64<sbyte> value, byte valueIndex)
Vector128<sbyte> InsertSelectedScalar(Vector128<sbyte> result, byte
resultIndex, Vector128<sbyte> value, byte valueIndex)
Vector128<float> InsertSelectedScalar(Vector128<float> result, byte
resultIndex, Vector64<float> value, byte valueIndex)
Vector128<float> InsertSelectedScalar(Vector128<float> result, byte
resultIndex, Vector128<float> value, byte valueIndex)
Vector128<ushort> InsertSelectedScalar(Vector128<ushort> result, byte
resultIndex, Vector64<ushort> value, byte valueIndex)
Vector128<ushort> InsertSelectedScalar(Vector128<ushort> result, byte
resultIndex, Vector128<ushort> value, byte valueIndex)
Vector128<uint> InsertSelectedScalar(Vector128<uint> result, byte
resultIndex, Vector64<uint> value, byte valueIndex)
Vector128<uint> InsertSelectedScalar(Vector128<uint> result, byte
resultIndex, Vector128<uint> value, byte valueIndex)
Vector128<ulong> InsertSelectedScalar(Vector128<ulong> result, byte
resultIndex, Vector128<ulong> value, byte valueIndex)

```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:InsertSelectedScalarTest(System.Runtime.Intrinsics.Vector64`1[
Byte],ubyte,System.Runtime.Intrinsics.Vector64`1[Byte],ubyte):System.Runtime.
Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01      ] ( 0, 0 )  ubyte  ->  zero-ref
; V02 arg2          [V02,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;* V03 arg3          [V03      ] ( 0, 0 )  ubyte  ->  zero-ref
;# V04 OutArgs       [V04      ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0

```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    ins    v0.b[0], v1.b[1]  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 20, prolog size 8

141. LeadingSignCount

Vector64<short> LeadingSignCount(Vector64<short> value)

This method counts the number of leading bits of individual elements of value vector that have the same value as the most significant bit and stores the result in result vector. This count does not include the most significant bit of the input.

```
private Vector64<short> LeadingSignCountTest(Vector64<short> value)
{
    return AdvSimd.LeadSignCount(value);
}
// value = <32757, 165, 0, 15>
// Result = <0, 7, 15, 11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> LeadingSignCount(Vector64<int> value)
Vector64<sbyte> LeadingSignCount(Vector64<sbyte> value)
Vector128<short> LeadingSignCount(Vector128<short> value)
Vector128<int> LeadingSignCount(Vector128<int> value)
Vector128<sbyte> LeadingSignCount(Vector128<sbyte> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:LeadingSignCountTest(System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    cls    v16.4h, v0.4h
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

142. LeadingZeroCount

Vector64<byte> LeadingZeroCount(Vector64<byte> value)

This method counts the number of binary zero bits before the first binary one bit in individual elements of the value vector, and writes the result to the result vector.

```
private Vector64<byte> LeadingZeroCountTest(Vector64<byte> value)
{
    return AdvSimd.LeadingZeroCount(value);
}
// value = <32757, 165, 0, 15>
// Result = <1, 8, 16, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> LeadingZeroCount(Vector64<short> value)
Vector64<int> LeadingZeroCount(Vector64<int> value)
Vector64<sbyte> LeadingZeroCount(Vector64<sbyte> value)
Vector64<ushort> LeadingZeroCount(Vector64<ushort> value)
Vector64<uint> LeadingZeroCount(Vector64<uint> value)
Vector128<byte> LeadingZeroCount(Vector128<byte> value)
Vector128<short> LeadingZeroCount(Vector128<short> value)
Vector128<int> LeadingZeroCount(Vector128<int> value)
Vector128<sbyte> LeadingZeroCount(Vector128<sbyte> value)
Vector128<ushort> LeadingZeroCount(Vector128<ushort> value)
Vector128<uint> LeadingZeroCount(Vector128<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:LeadingZeroCountTest(System.Runtime.Intrinsics.Vector64`1[Byte]
):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    clz    v16.8b, v0.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


143. LoadAndInsertScalar

Vector64<byte> LoadAndInsertScalar(Vector64<byte> value, byte index, byte* address)

This method loads a single-element structure from memory at address and writes the result to the specified index of the value vector without affecting the other elements of the result vector.

```
private Vector64<byte> LoadAndInsertScalarTest(Vector64<byte> value, byte
index, byte* address)
{
    return AdvSimd.LoadAndInsertScalar(value, 2, address);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// index = 2
// address = Address of byte[] { 21, 22, 23, 24, 25, 26, 27, 28 }
// Result = <11, 12, 21, 14, 15, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> LoadAndInsertScalar(Vector64<short> value, byte index, short*
address)
Vector64<int> LoadAndInsertScalar(Vector64<int> value, byte index, int*
address)
Vector64<sbyte> LoadAndInsertScalar(Vector64<sbyte> value, byte index, sbyte*
address)
Vector64<float> LoadAndInsertScalar(Vector64<float> value, byte index, float*
address)
Vector64<ushort> LoadAndInsertScalar(Vector64<ushort> value, byte index,
ushort* address)
Vector64<uint> LoadAndInsertScalar(Vector64<uint> value, byte index, uint*
address)
Vector128<byte> LoadAndInsertScalar(Vector128<byte> value, byte index, byte*
address)
Vector128<double> LoadAndInsertScalar(Vector128<double> value, byte index,
double* address)
Vector128<short> LoadAndInsertScalar(Vector128<short> value, byte index,
short* address)
Vector128<int> LoadAndInsertScalar(Vector128<int> value, byte index, int*
address)
Vector128<long> LoadAndInsertScalar(Vector128<long> value, byte index, long*
address)
Vector128<sbyte> LoadAndInsertScalar(Vector128<sbyte> value, byte index,
sbyte* address)
Vector128<float> LoadAndInsertScalar(Vector128<float> value, byte index,
float* address)
Vector128<ushort> LoadAndInsertScalar(Vector128<ushort> value, byte index,
ushort* address)
Vector128<uint> LoadAndInsertScalar(Vector128<uint> value, byte index, uint*
```

```
address)
Vector128<ulong> LoadAndInsertScalar(Vector128<ulong> value, byte index,
ulong* address)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:LoadAndInsertScalarTest(System.Runtime.Intrinsics.Vector64`1[By
yte],ubyte,long):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T01] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01      ] ( 0, 0 )   ubyte  ->  zero-ref
; V02 arg2          [V02,T00] ( 3, 3 )   long   ->  x1
;# V03 OutArgs      [V03      ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    mov    v16.8b, v0.8b
    ld1    {v16.b}[2], [x1]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 28, prolog size 8
```

144. LoadAndReplicateToVector128

Vector128<byte> LoadAndReplicateToVector128(byte* address)

This method loads a single-element structure from memory at address and replicates the value to all the elements of the result vector.

```
private Vector128<byte> LoadAndReplicateToVector128Test(byte* address)
{
    return AdvSimd.LoadAndReplicateToVector128(address);
}
// address = Address of byte[] { 11}
// Result = <11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> LoadAndReplicateToVector128(short* address)
Vector128<int> LoadAndReplicateToVector128(int* address)
Vector128<sbyte> LoadAndReplicateToVector128(sbyte* address)
Vector128<float> LoadAndReplicateToVector128(float* address)
Vector128<ushort> LoadAndReplicateToVector128(ushort* address)
Vector128<uint> LoadAndReplicateToVector128(uint* address)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> LoadAndReplicateToVector128(double* address)
Vector128<long> LoadAndReplicateToVector128(long* address)
Vector128<ulong> LoadAndReplicateToVector128(ulong* address)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:LoadAndReplicateToVector128Test(long):System.Runtime.Intrinsic
s.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )    long -> x0
;# V01 OutArgs      [V01  ] ( 1, 1 )    lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ld1r   {v16.16b}, [x0]
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

145. LoadAndReplicateToVector64

Vector64<byte> LoadAndReplicateToVector64(byte* address)

This method loads a single-element structure from memory at address and replicates the value to all the elements of the result vector.

```
private Vector64<byte> LoadAndReplicateToVector64Test(byte* address)
{
    return AdvSimd.LoadAndReplicateToVector64(address);
}
// address = Address of byte[] { 11 }
// Result = <11, 11, 11, 11, 11, 11, 11, 11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> LoadAndReplicateToVector64(short* address)
Vector64<int> LoadAndReplicateToVector64(int* address)
Vector64<sbyte> LoadAndReplicateToVector64(sbyte* address)
Vector64<float> LoadAndReplicateToVector64(float* address)
Vector64<ushort> LoadAndReplicateToVector64(ushort* address)
Vector64<uint> LoadAndReplicateToVector64(uint* address)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:LoadAndReplicateToVector64Test(long):System.Runtime.Intrinsics
.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) long -> x0
;# V01 OutArgs      [V01  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ld1r   {v16.8b}, [x0]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

146. LoadVector128

Vector128<byte> LoadVector128(byte* address)

This method loads a multiple-element structure like array from memory at address and writes it to the result vector. If the elements in memory don't fill up all the elements of result vector, then the remaining are set to 0.

```
private Vector128<byte> LoadVector128Test(byte* address)
{
    return AdvSimd.LoadVector128(address);
}
// address = Address of new byte[14] { 21, 22, 23, 24, 25, 26, 27, 28, 1, 2,
23, 24, 25, 26}
// Result = <21, 22, 23, 24, 25, 26, 27, 28, 1, 2, 23, 24, 25, 26, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<double> LoadVector128(double* address)
Vector128<short> LoadVector128(short* address)
Vector128<int> LoadVector128(int* address)
Vector128<long> LoadVector128(long* address)
Vector128<sbyte> LoadVector128(sbyte* address)
Vector128<float> LoadVector128(float* address)
Vector128<ushort> LoadVector128(ushort* address)
Vector128<uint> LoadVector128(uint* address)
Vector128<ulong> LoadVector128(ulong* address)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:LoadVector128Test(long):System.Runtime.Intrinsics.Vector128`1[
Byte]
;
; V00 arg0 [V00,T00] ( 3, 3 ) long -> x0
;# V01 OutArgs [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ld1    {v16.16b}, [x0]
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

147. LoadVector64

Vector64<byte> LoadVector64(byte* address)

This method loads a multiple-element structure like array from memory at address and writes it to the result vector. If the elements in memory don't fill up all the elements of result vector, then the remaining are set to 0.

```
private Vector64<byte> LoadVector64Test(byte* address)
{
    return AdvSimd.LoadVector64(address);
}
// address = Address of new byte[14] { 21, 22, 23, 24, 25, 26, 27, 28, 1, 2,
// 23, 24, 25, 26}
// Result = <21, 22, 23, 24, 25, 26, 27, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<double> LoadVector64(double* address)
Vector64<short> LoadVector64(short* address)
Vector64<int> LoadVector64(int* address)
Vector64<long> LoadVector64(long* address)
Vector64<sbyte> LoadVector64(sbyte* address)
Vector64<float> LoadVector64(float* address)
Vector64<ushort> LoadVector64(ushort* address)
Vector64<uint> LoadVector64(uint* address)
Vector64<ulong> LoadVector64(ulong* address)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:LoadVector64Test(long):System.Runtime.Intrinsics.Vector64`1[By
te]
;
; V00 arg0 [V00,T00] ( 3, 3 ) long -> x0
;# V01 OutArgs [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ld1    {v16.8b}, [x0]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

148. Max

Vector64<byte> Max(Vector64<byte> left, Vector64<byte> right)

This method compares corresponding elements in the left and right vectors, places the larger of each pair in the result vector, and returns the result vector.

```
private Vector64<byte> MaxTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.Max(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <21, 22, 23, 24, 25, 26, 27, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> Max(Vector64<short> left, Vector64<short> right)
Vector64<int> Max(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> Max(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> Max(Vector64<float> left, Vector64<float> right)
Vector64<ushort> Max(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> Max(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> Max(Vector128<byte> left, Vector128<byte> right)
Vector128<short> Max(Vector128<short> left, Vector128<short> right)
Vector128<int> Max(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> Max(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> Max(Vector128<float> left, Vector128<float> right)
Vector128<ushort> Max(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> Max(Vector128<uint> left, Vector128<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Max(Vector128<double> left, Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
                stp    fp, lr, [sp,#-16]!
```

```
mov    fp, sp
umax   v16.8b, v0.8b, v1.8b
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

149. MaxAcross

Vector64<byte> MaxAcross(Vector64<byte> value)

This method compares all the vector elements in the value vector, and writes the largest value element in result vector at 0th index while other elements are set to 0.

```
private Vector64<byte> MaxAcrossTest(Vector64<byte> value)
{
    return AdvSimd.Arm64.MaxAcross(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <18, 0, 0, 0, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<short> MaxAcross(Vector64<short> value)
Vector64<sbyte> MaxAcross(Vector64<sbyte> value)
Vector64<ushort> MaxAcross(Vector64<ushort> value)
Vector64<byte> MaxAcross(Vector128<byte> value)
Vector64<short> MaxAcross(Vector128<short> value)
Vector64<int> MaxAcross(Vector128<int> value)
Vector64<sbyte> MaxAcross(Vector128<sbyte> value)
Vector64<float> MaxAcross(Vector128<float> value)
Vector64<ushort> MaxAcross(Vector128<ushort> value)
Vector64<uint> MaxAcross(Vector128<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxAcrossTest(System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs [V01 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    umaxv  b16, v0.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

150. MaxNumber

Vector64<float> MaxNumber(Vector64<float> left, Vector64<float> right)

This method compares corresponding elements in the left and right vectors, places the larger of each pair in the result vector, and returns the result vector. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value.

```
private Vector64<float> MaxNumberTest(Vector64<float> left, Vector64<float>
right)
{
    return AdvSimd.MaxNumber(left, right);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// Result = <21.5, 22.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> MaxNumber(Vector128<float> left, Vector128<float> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> MaxNumber(Vector128<double> left, Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxNumberTest(System.Runtime.Intrinsics.Vector64`1[Single],Sys
tem.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64
`1[Single]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmaxnm v16.2s, v0.2s, v1.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

151. MaxNumberAcross

Vector64<float> MaxNumberAcross(Vector128<float> value)

This method compares all the vector elements in the value vector, and writes the largest value element in result vector at 0th index while other elements are set to 0. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to MaxScalar().

```
private Vector64<float> MaxNumberAcrossTest(Vector128<float> value)
{
    return AdvSimd.Arm64.MaxNumberAcross(value);
}
// value = <11.5, 12.5, 13.5, 14.5>
// Result = <14.5, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxNumberAcrossTest(System.Runtime.Intrinsics.Vector128`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmaxnmv s16, v0.4s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

152. MaxNumberPairwise

Vector64<float> MaxNumberPairwise(Vector64<float> left, Vector64<float> right)

This method creates a vector by concatenating the vector elements of left vector followed by those of the right vector, compares adjacent vector elements and writes the largest of each pair in a result vector. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value.

```
private Vector64<float> MaxNumberPairwiseTest(Vector64<float> left,
Vector64<float> right)
{
    return AdvSimd.Arm64.MaxNumberPairwise(left, right);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// Result = <12.5, 22.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector128<double> MaxNumberPairwise(Vector128<double> left, Vector128<double>
right)
Vector128<float> MaxNumberPairwise(Vector128<float> left, Vector128<float>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxNumberPairwiseTest(System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.
Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmaxnmp v16.2s, v0.2s, v1.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```


; Total bytes of code 24, prolog size 8

153. MaxNumberPairwiseScalar

Vector64<float> MaxNumberPairwiseScalar(Vector64<float> value)

This method creates a vector by concatenating the vector elements of left vector followed by those of the right vector, compares adjacent vector elements and writes the largest of each pair in 0th element of result vector. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value.

```
private Vector64<float> MaxNumberPairwiseScalarTest(Vector64<float> value)
{
    return AdvSimd.Arm64.MaxNumberPairwiseScalar(value);
}
// value = <11.5, 12.5>
// Result = <12.5, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<double> MaxNumberPairwiseScalar(Vector128<double> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxNumberPairwiseScalarTest(System.Runtime.Intrinsics.Vector64
`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs  [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmaxnmp s16, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

154. MaxNumberScalar

Vector64<double> MaxNumberScalar(Vector64<double> left, Vector64<double> right)

This method compares corresponding vector elements in left and right vector and stores the larger value in a result vector. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value.

```
private Vector64<double> MaxNumberScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.MaxNumberScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <11.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> MaxNumberScalar(Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxNumberScalarTest(System.Runtime.Intrinsics.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmaxnm d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

155. MaxPairwise

Vector64<byte> MaxPairwise(Vector64<byte> left, Vector64<byte> right)

This method creates a vector by concatenating the vector elements of the left after the vector elements of the right vector, reads each pair of adjacent vector elements in the vectors, writes the largest of each pair into a result vector, and writes the vector to the result vector.

```
private Vector64<byte> MaxPairwiseTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.MaxPairwise(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <12, 14, 16, 18, 22, 24, 26, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> MaxPairwise(Vector64<short> left, Vector64<short> right)
Vector64<int> MaxPairwise(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> MaxPairwise(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> MaxPairwise(Vector64<float> left, Vector64<float> right)
Vector64<ushort> MaxPairwise(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> MaxPairwise(Vector64<uint> left, Vector64<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<byte> MaxPairwise(Vector128<byte> left, Vector128<byte> right)
Vector128<double> MaxPairwise(Vector128<double> left, Vector128<double>
right)
Vector128<short> MaxPairwise(Vector128<short> left, Vector128<short> right)
Vector128<int> MaxPairwise(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> MaxPairwise(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> MaxPairwise(Vector128<float> left, Vector128<float> right)
Vector128<ushort> MaxPairwise(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> MaxPairwise(Vector128<uint> left, Vector128<uint> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxPairwiseTest(System.Runtime.Intrinsics.Vector64`1[Byte],Sys
tem.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1
[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
```

```

; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    umaxp  v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

156. MaxPairwiseScalar

Vector64<float> MaxPairwiseScalar(Vector64<float> value)

This method compares two vector elements in the value vector and writes the largest of the floating-point values as a scalar to the result vector.

```
private Vector64<float> MaxPairwiseScalarTest(Vector64<float> value)
{
    return AdvSimd.Arm64.MaxPairwiseScalar(value);
}
// value = <11.5, 12.5>
// Result = <12.5, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<double> MaxPairwiseScalar(Vector128<double> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxPairwiseScalarTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmaxp   s16, v0.2s
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

157. MaxScalar

Vector64<double> MaxScalar(Vector64<double> left, Vector64<double> right)

This method compares the left and right vector, and writes the larger of the two floating-point values to the result vector.

```
private Vector64<double> MaxScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.MaxScalar(left, right);
}
// left = <11.5>
// right = <10.5>
// Result = <11.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<float> MaxScalar(Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MaxScalarTest(System.Runtime.Intrinsics.Vector64`1[Double],Sys
tem.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64
`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmax    d16, d0, d1
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

158. Min

Vector64<byte> Min(Vector64<byte> left, Vector64<byte> right)

This method compares corresponding elements in the left and right vectors, places the smaller of each pair in the result vector, and returns the result vector.

```
private Vector64<byte> MinTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.Min(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <11, 12, 13, 14, 15, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> Min(Vector64<short> left, Vector64<short> right)
Vector64<int> Min(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> Min(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> Min(Vector64<float> left, Vector64<float> right)
Vector64<ushort> Min(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> Min(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> Min(Vector128<byte> left, Vector128<byte> right)
Vector128<short> Min(Vector128<short> left, Vector128<short> right)
Vector128<int> Min(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> Min(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> Min(Vector128<float> left, Vector128<float> right)
Vector128<ushort> Min(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> Min(Vector128<uint> left, Vector128<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Min(Vector128<double> left, Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
```



```
mov    fp, sp
umin   v16.8b, v0.8b, v1.8b
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

159. MinAcross

Vector64<byte> MinAcross(Vector64<byte> value)

This method compares all the vector elements in the value vector, and writes the smaller value element in result vector at 0th index while other elements are set to 0.

```
private Vector64<byte> MinAcrossTest(Vector64<byte> value)
{
    return AdvSimd.Arm64.MinAcross(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <11, 0, 0, 0, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<short> MinAcross(Vector64<short> value)
Vector64<sbyte> MinAcross(Vector64<sbyte> value)
Vector64<ushort> MinAcross(Vector64<ushort> value)
Vector64<byte> MinAcross(Vector128<byte> value)
Vector64<short> MinAcross(Vector128<short> value)
Vector64<int> MinAcross(Vector128<int> value)
Vector64<sbyte> MinAcross(Vector128<sbyte> value)
Vector64<float> MinAcross(Vector128<float> value)
Vector64<ushort> MinAcross(Vector128<ushort> value)
Vector64<uint> MinAcross(Vector128<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinAcrossTest(System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs [V01 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uminv  b16, v0.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

160. MinNumber

Vector64<float> MinNumber(Vector64<float> left, Vector64<float> right)

This method compares corresponding elements in the left and right vectors, places the smaller of each pair in the result vector, and returns the result vector. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value.

```
private Vector64<float> MinNumberTest(Vector64<float> left, Vector64<float>
right)
{
    return AdvSimd.MinNumber(left, right);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// Result = <11.5, 12.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> MinNumber(Vector128<float> left, Vector128<float> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> MinNumber(Vector128<double> left, Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinNumberTest(System.Runtime.Intrinsics.Vector64`1[Single],Sys
tem.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64
`1[Single]
;
; V00 arg0      [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
; V01 arg1      [V01,T01] (  3,  3  )  simd8  ->  d1
HFA(simd8)
;# V02 OutArgs  [V02   ] (  1,  1  )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp     fp, lr, [sp,#-16]!
    mov     fp, sp
    fminnm  v16.2s, v0.2s, v1.2s
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr
```

; Total bytes of code 24, prolog size 8

161. MinNumberAcross

Vector64<float> MinNumberAcross(Vector128<float> value)

This method compares all the vector elements in the value vector, and writes the smaller value element in result vector at 0th index while other elements are set to 0. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to MaxScalar().

```
private Vector64<float> MinNumberAcrossTest(Vector128<float> value)
{
    return AdvSimd.Arm64.MinNumberAcross(value);
}
// value = <11.5, 12.5, 13.5, 14.5>
// Result = <11.5, 0>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinNumberAcrossTest(System.Runtime.Intrinsics.Vector128`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0 [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fminnmv s16, v0.4s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

162. MinNumberPairwise

Vector64<float> MinNumberPairwise(Vector64<float> left, Vector64<float> right)

This method creates a vector by concatenating the vector elements of left vector followed by those of the right vector, compares adjacent vector elements and writes the smallest of each pair in a result vector. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value.

```
private Vector64<float> MinNumberPairwiseTest(Vector64<float> left,
Vector64<float> right)
{
    return AdvSimd.Arm64.MinNumberPairwise(left, right);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// Result = <11.5, 21.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Avsimd.Arm64
Vector128<double> MinNumberPairwise(Vector128<double> left, Vector128<double>
right)
Vector128<float> MinNumberPairwise(Vector128<float> left, Vector128<float>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinNumberPairwiseTest(System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.
Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fminnmp v16.2s, v0.2s, v1.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

163. MinNumberPairwiseScalar

Vector64<float> MinNumberPairwiseScalar(Vector64<float> value)

This method creates a vector by concatenating the vector elements of left vector followed by those of the right vector, compares adjacent vector elements and writes the smallest of each pair in 0th element of result vector. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value.

```
private Vector64<float> MinNumberPairwiseScalarTest(Vector64<float> value)
{
    return AdvSimd.Arm64.MinNumberPairwiseScalar(value);
}
// value = <11.5, 12.5>
// Result = <11.5, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<double> MinNumberPairwiseScalar(Vector128<double> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinNumberPairwiseScalarTest(System.Runtime.Intrinsics.Vector64
`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs  [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fminnmp s16, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

164. MinNumberScalar

Vector64<double> MinNumberScalar(Vector64<double> left, Vector64<double> right)

This method compares corresponding vector elements in `left` and `right` vector and stores the smaller value in a result vector. As per ARM docs, NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value.

```
private Vector64<double> MinNumberScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.MinNumberScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <11.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> MinNumberScalar(Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinNumberScalarTest(System.Runtime.Intrinsics.Vector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fminnm d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

165. MinPairwise

Vector64<byte> MinPairwise(Vector64<byte> left, Vector64<byte> right)

This method creates a vector by concatenating the vector elements of the left after the vector elements of the right vector, reads each pair of adjacent vector elements in the vectors, writes the smallest of each pair into a result vector, and writes the vector to the result vector.

```
private Vector64<byte> MinPairwiseTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.MinPairwise(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <11, 13, 15, 17, 21, 23, 25, 27>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> MinPairwise(Vector64<short> left, Vector64<short> right)
Vector64<int> MinPairwise(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> MinPairwise(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> MinPairwise(Vector64<float> left, Vector64<float> right)
Vector64<ushort> MinPairwise(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> MinPairwise(Vector64<uint> left, Vector64<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<byte> MinPairwise(Vector128<byte> left, Vector128<byte> right)
Vector128<double> MinPairwise(Vector128<double> left, Vector128<double>
right)
Vector128<short> MinPairwise(Vector128<short> left, Vector128<short> right)
Vector128<int> MinPairwise(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> MinPairwise(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> MinPairwise(Vector128<float> left, Vector128<float> right)
Vector128<ushort> MinPairwise(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> MinPairwise(Vector128<uint> left, Vector128<uint> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinPairwiseTest(System.Runtime.Intrinsics.Vector64`1[Byte],Sys
tem.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1
[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
```

```

; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02    ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uminp  v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

166. MinPairwiseScalar

Vector64<float> MinPairwiseScalar(Vector64<float> value)

This method compares two vector elements in the value vector and writes the smallest of the floating-point values as a scalar to the result vector.

```
private Vector64<float> MinPairwiseScalarTest(Vector64<float> value)
{
    return AdvSimd.Arm64.MinPairwiseScalar(value);
}
// value = <11.5, 12.5>
// Result = <11.5, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<double> MinPairwiseScalar(Vector128<double> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinPairwiseScalarTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fminp  s16, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

167. MinScalar

Vector64<double> MinScalar(Vector64<double> left, Vector64<double> right)

This method compares the left and right vector, and writes the smaller of the two floating-point values to the result vector.

```
private Vector64<double> MinScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.MinScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <11.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Avsimd.Arm64
Vector64<float> MinScalar(Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MinScalarTest(System.Runtime.Intrinsics.Vector64`1[Double], Sys
tem.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64
`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmin    d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

168. Multiply

Vector64<byte> Multiply(Vector64<byte> left, Vector64<byte> right)

This method performs multiplication of corresponding vector elements in left and right vectors, writes the product to the result vector.

```
private Vector64<byte> MultiplyTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.Multiply(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <231, 8, 43, 80, 119, 160, 203, 248>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> Multiply(Vector64<short> left, Vector64<short> right)
Vector64<int> Multiply(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> Multiply(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> Multiply(Vector64<float> left, Vector64<float> right)
Vector64<ushort> Multiply(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> Multiply(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> Multiply(Vector128<byte> left, Vector128<byte> right)
Vector128<short> Multiply(Vector128<short> left, Vector128<short> right)
Vector128<int> Multiply(Vector128<int> left, Vector128<int> right)
Vector128<sbyte> Multiply(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> Multiply(Vector128<float> left, Vector128<float> right)
Vector128<ushort> Multiply(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> Multiply(Vector128<uint> left, Vector128<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Multiply(Vector128<double> left, Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyTest(System.Runtime.Intrinsics.Vector64`1[Byte],System
.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[By
te]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    mul    v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

169. MultiplyAdd

Vector64<byte> MultiplyAdd(Vector64<byte> addend, Vector64<byte> left, Vector64<byte> right)

This method multiplies corresponding elements in the vectors of the left and right vectors, and accumulates the product with the vector elements of the addend and returns the accumulated result.

```
private Vector64<byte> MultiplyAddTest(Vector64<byte> addend, Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.MultiplyAdd(addend, left, right);
}
// addend = <11, 12, 13, 14, 15, 16, 17, 18>
// left = <2, 22, 23, 24, 25, 26, 27, 28>
// right = <3, 32, 33, 34, 35, 36, 37, 38>
// Result = <17, 204, 4, 62, 122, 184, 248, 58>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> MultiplyAdd(Vector64<short> addend, Vector64<short> left, Vector64<short> right)
Vector64<int> MultiplyAdd(Vector64<int> addend, Vector64<int> left, Vector64<int> right)
Vector64<sbyte> MultiplyAdd(Vector64<sbyte> addend, Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<ushort> MultiplyAdd(Vector64<ushort> addend, Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> MultiplyAdd(Vector64<uint> addend, Vector64<uint> left, Vector64<uint> right)
Vector128<byte> MultiplyAdd(Vector128<byte> addend, Vector128<byte> left, Vector128<byte> right)
Vector128<short> MultiplyAdd(Vector128<short> addend, Vector128<short> left, Vector128<short> right)
Vector128<int> MultiplyAdd(Vector128<int> addend, Vector128<int> left, Vector128<int> right)
Vector128<sbyte> MultiplyAdd(Vector128<sbyte> addend, Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<ushort> MultiplyAdd(Vector128<ushort> addend, Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> MultiplyAdd(Vector128<uint> addend, Vector128<uint> left, Vector128<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyAddTest(System.Runtime.Intrinsics.Vector64`1[Byte], Sys
```



```

tem.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[
Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0      [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
; V02 arg2      [V02,T02] ( 3, 3 )  simd8  ->  d2
HFA(simd8)
;# V03 OutArgs  [V03    ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    mla    v0.8b, v1.8b, v2.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

170. MultiplyAddByScalar

Vector64<short> MultiplyAddByScalar(Vector64<short> addend, Vector64<short> left, Vector64<short> right)

This method multiplies the vector elements in the left by the 0th element value in the right, and accumulates the product with the vector elements of the addend vector and return the result vector.

```
private Vector64<short> MultiplyAddByScalarTest(Vector64<short> addend,
Vector64<short> left, Vector64<short> right)
{
    return AdvSimd.MultiplyAddByScalar(addend, left, right);
}
// addend = <11, 12, 13, 14>
// left = <21, 22, 23, 24>
// right = <31, 32, 33, 34>
// Result = <662, 694, 726, 758>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> MultiplyAddByScalar(Vector64<int> addend, Vector64<int> left,
Vector64<int> right)
Vector64<ushort> MultiplyAddByScalar(Vector64<ushort> addend,
Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> MultiplyAddByScalar(Vector64<uint> addend, Vector64<uint>
left, Vector64<uint> right)
Vector128<short> MultiplyAddByScalar(Vector128<short> addend,
Vector128<short> left, Vector64<short> right)
Vector128<int> MultiplyAddByScalar(Vector128<int> addend, Vector128<int>
left, Vector64<int> right)
Vector128<ushort> MultiplyAddByScalar(Vector128<ushort> addend,
Vector128<ushort> left, Vector64<ushort> right)
Vector128<uint> MultiplyAddByScalar(Vector128<uint> addend, Vector128<uint>
left, Vector64<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyAddByScalarTest(System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
```

```

HFA(simd8)
;# V03 OutArgs      [V03      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp    fp, lr, [sp,#-16]!
        mov    fp, sp
        mla    v0.4h, v1.4h, v2.h[0]
        ldp    fp, lr, [sp],#16
        ret    lr

; Total bytes of code 20, prolog size 8

```

171. MultiplyAddBySelectedScalar

Vector64<short> MultiplyAddBySelectedScalar(Vector64<short> addend, Vector64<short> left, Vector64<short> right, byte rightIndex)

This method multiplies the vector elements in the left by the rightIndex element value in the right, and accumulates the product with the vector elements of the addend vector and return the result vector.

```
private Vector64<short> MultiplyAddBySelectedScalarTest(Vector64<short>
addend, Vector64<short> left, Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplyAddBySelectedScalar(addend, left, right, 3);
}
// addend = <100, 100, 100, 100>
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// rightIndex = 3
// Result = <364, 388, 412, 436>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> MultiplyAddBySelectedScalar(Vector64<short> addend,
Vector64<short> left, Vector128<short> right, byte rightIndex)
Vector64<int> MultiplyAddBySelectedScalar(Vector64<int> addend, Vector64<int>
left, Vector64<int> right, byte rightIndex)
Vector64<int> MultiplyAddBySelectedScalar(Vector64<int> addend, Vector64<int>
left, Vector128<int> right, byte rightIndex)
Vector64<ushort> MultiplyAddBySelectedScalar(Vector64<ushort> addend,
Vector64<ushort> left, Vector64<ushort> right, byte rightIndex)
Vector64<ushort> MultiplyAddBySelectedScalar(Vector64<ushort> addend,
Vector64<ushort> left, Vector128<ushort> right, byte rightIndex)
Vector64<uint> MultiplyAddBySelectedScalar(Vector64<uint> addend,
Vector64<uint> left, Vector64<uint> right, byte rightIndex)
Vector64<uint> MultiplyAddBySelectedScalar(Vector64<uint> addend,
Vector64<uint> left, Vector128<uint> right, byte rightIndex)
Vector128<short> MultiplyAddBySelectedScalar(Vector128<short> addend,
Vector128<short> left, Vector64<short> right, byte rightIndex)
Vector128<short> MultiplyAddBySelectedScalar(Vector128<short> addend,
Vector128<short> left, Vector128<short> right, byte rightIndex)
Vector128<int> MultiplyAddBySelectedScalar(Vector128<int> addend,
Vector128<int> left, Vector64<int> right, byte rightIndex)
Vector128<int> MultiplyAddBySelectedScalar(Vector128<int> addend,
Vector128<int> left, Vector128<int> right, byte rightIndex)
Vector128<ushort> MultiplyAddBySelectedScalar(Vector128<ushort> addend,
Vector128<ushort> left, Vector64<ushort> right, byte rightIndex)
Vector128<ushort> MultiplyAddBySelectedScalar(Vector128<ushort> addend,
Vector128<ushort> left, Vector128<ushort> right, byte rightIndex)
Vector128<uint> MultiplyAddBySelectedScalar(Vector128<uint> addend,
Vector128<uint> left, Vector64<uint> right, byte rightIndex)
```

Vector128<uint> MultiplyAddBySelectedScalar(Vector128<uint> addend,
Vector128<uint> left, Vector128<uint> right, byte rightIndex)

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyAddBySelectedScalarTest(System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 )   simd8  ->  d2
HFA(simd8)
;* V03 arg3          [V03      ] ( 0, 0 )   ubyte  ->  zero-ref
;# V04 OutArgs       [V04      ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    mla     v0.4h, v1.4h, v2.h[3]
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 20, prolog size 8
```

172. MultiplyByScalar

Vector64<short> MultiplyByScalar(Vector64<short> left, Vector64<short> right)

This method multiplies corresponding vector elements in the left by the 0th element of right vector and returns the result vector.

```
private Vector64<short> MultiplyByScalarTest(Vector64<short> left,
Vector64<short> right)
{
    return AdvSimd.MultiplyByScalar(left, right);
}
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// Result = <231, 252, 273, 294>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> MultiplyByScalar(Vector64<int> left, Vector64<int> right)
Vector64<float> MultiplyByScalar(Vector64<float> left, Vector64<float> right)
Vector64<ushort> MultiplyByScalar(Vector64<ushort> left, Vector64<ushort>
right)
Vector64<uint> MultiplyByScalar(Vector64<uint> left, Vector64<uint> right)
Vector128<short> MultiplyByScalar(Vector128<short> left, Vector64<short>
right)
Vector128<int> MultiplyByScalar(Vector128<int> left, Vector64<int> right)
Vector128<float> MultiplyByScalar(Vector128<float> left, Vector64<float>
right)
Vector128<ushort> MultiplyByScalar(Vector128<ushort> left, Vector64<ushort>
right)
Vector128<uint> MultiplyByScalar(Vector128<uint> left, Vector64<uint> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> MultiplyByScalar(Vector128<double> left, Vector64<double>
right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyByScalarTest(System.Runtime.Intrinsics.Vector64`1[Int1
6],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vec
tor64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    mul    v16.4h, v0.4h, v1.h[0]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

173. MultiplyBySelectedScalar

Vector64<short> MultiplyBySelectedScalar(Vector64<short> left, Vector64<short> right, byte rightIndex)

This method multiplies corresponding vector elements in the left by the rightIndex element of right vector and returns the result vector.

```
private Vector64<short> MultiplyBySelectedScalarTest(Vector64<short> left,
Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplyBySelectedScalar(left, right, 3);
}
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// rightIndex = 3
// Result = <264, 288, 312, 336>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> MultiplyBySelectedScalar(Vector64<short> left,
Vector128<short> right, byte rightIndex)
Vector64<int> MultiplyBySelectedScalar(Vector64<int> left, Vector64<int>
right, byte rightIndex)
Vector64<int> MultiplyBySelectedScalar(Vector64<int> left, Vector128<int>
right, byte rightIndex)
Vector64<float> MultiplyBySelectedScalar(Vector64<float> left,
Vector64<float> right, byte rightIndex)
Vector64<float> MultiplyBySelectedScalar(Vector64<float> left,
Vector128<float> right, byte rightIndex)
Vector64<ushort> MultiplyBySelectedScalar(Vector64<ushort> left,
Vector64<ushort> right, byte rightIndex)
Vector64<ushort> MultiplyBySelectedScalar(Vector64<ushort> left,
Vector128<ushort> right, byte rightIndex)
Vector64<uint> MultiplyBySelectedScalar(Vector64<uint> left, Vector64<uint>
right, byte rightIndex)
Vector64<uint> MultiplyBySelectedScalar(Vector64<uint> left, Vector128<uint>
right, byte rightIndex)
Vector128<short> MultiplyBySelectedScalar(Vector128<short> left,
Vector64<short> right, byte rightIndex)
Vector128<short> MultiplyBySelectedScalar(Vector128<short> left,
Vector128<short> right, byte rightIndex)
Vector128<int> MultiplyBySelectedScalar(Vector128<int> left, Vector64<int>
right, byte rightIndex)
Vector128<int> MultiplyBySelectedScalar(Vector128<int> left, Vector128<int>
right, byte rightIndex)
Vector128<float> MultiplyBySelectedScalar(Vector128<float> left,
Vector64<float> right, byte rightIndex)
Vector128<float> MultiplyBySelectedScalar(Vector128<float> left,
Vector128<float> right, byte rightIndex)
```



```

Vector128<ushort> MultiplyBySelectedScalar(Vector128<ushort> left,
Vector64<ushort> right, byte rightIndex)
Vector128<ushort> MultiplyBySelectedScalar(Vector128<ushort> left,
Vector128<ushort> right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalar(Vector128<uint> left, Vector64<uint>
right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalar(Vector128<uint> left,
Vector128<uint> right, byte rightIndex)

// class System.Runtime.Intrinsics.Arm64
Vector128<double> MultiplyBySelectedScalar(Vector128<double> left,
Vector128<double> right, byte rightIndex)

```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:MultiplyBySelectedScalarTest(System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02  ] ( 0, 0 )  ubyte  ->  zero-ref
;# V03 OutArgs       [V03  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    mul    v16.4h, v0.4h, v1.h[3]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

174. MultiplyBySelectedScalarWideningLower

Vector128<int> MultiplyBySelectedScalarWideningLower(Vector64<short> left, Vector64<short> right, byte rightIndex)

This method multiplies each vector element in the left vector by the rightIndex vector element of the right vector, places the product in a result vector, and returns the result vector. As seen in example below, the result vector element int size is twice as long as the elements that are multiplied short.

```
private Vector128<int>
MultiplyBySelectedScalarWideningLowerTest(Vector64<short> left,
Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplyBySelectedScalarWideningLower(left, right, 3);
}
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// rightIndex = 3
// Result = <264, 288, 312, 336>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyBySelectedScalarWideningLower(Vector64<short> left,
Vector128<short> right, byte rightIndex)
Vector128<long> MultiplyBySelectedScalarWideningLower(Vector64<int> left,
Vector64<int> right, byte rightIndex)
Vector128<long> MultiplyBySelectedScalarWideningLower(Vector64<int> left,
Vector128<int> right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalarWideningLower(Vector64<ushort> left,
Vector64<ushort> right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalarWideningLower(Vector64<ushort> left,
Vector128<ushort> right, byte rightIndex)
Vector128<ulong> MultiplyBySelectedScalarWideningLower(Vector64<uint> left,
Vector64<uint> right, byte rightIndex)
Vector128<ulong> MultiplyBySelectedScalarWideningLower(Vector64<uint> left,
Vector128<uint> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyBySelectedScalarWideningLowerTest(System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
```

```

HFA(simd8)
;* V02 arg2      [V02   ] (  0,  0  )  ubyte  ->  zero-ref
;# V03 OutArgs   [V03   ] (  1,  1  )  lclBlk (  0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    smull  v16.4s, v0.4h, v1.h[3]
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

175. MultiplyBySelectedScalarWideningLowerAndAdd

Vector128<int> MultiplyBySelectedScalarWideningLowerAndAdd(Vector128<int> addend, Vector64<short> left, Vector64<short> right, byte rightIndex)

This method multiplies corresponding values in the left and right vectors, and accumulates the results with the vector elements of the addend vector and returns the result vector. As seen in example below, the result vector element's size `int` is twice as long as that of input's size `short`.

```
private Vector128<int>
MultiplyBySelectedScalarWideningLowerAndAddTest(Vector128<int> addend,
Vector64<short> left, Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplyBySelectedScalarWideningLowerAndAdd(addend, left,
right, 2);
}
// addend = <1000, 1000, 1000, 1000>
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// rightIndex = 2
// Result = <1253, 1276, 1299, 1322>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyBySelectedScalarWideningLowerAndAdd(Vector128<int>
addend, Vector64<short> left, Vector128<short> right, byte rightIndex)
Vector128<long> MultiplyBySelectedScalarWideningLowerAndAdd(Vector128<long>
addend, Vector64<int> left, Vector64<int> right, byte rightIndex)
Vector128<long> MultiplyBySelectedScalarWideningLowerAndAdd(Vector128<long>
addend, Vector64<int> left, Vector128<int> right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalarWideningLowerAndAdd(Vector128<uint>
addend, Vector64<ushort> left, Vector64<ushort> right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalarWideningLowerAndAdd(Vector128<uint>
addend, Vector64<ushort> left, Vector128<ushort> right, byte rightIndex)
Vector128<ulong> MultiplyBySelectedScalarWideningLowerAndAdd(Vector128<ulong>
addend, Vector64<uint> left, Vector64<uint> right, byte rightIndex)
Vector128<ulong> MultiplyBySelectedScalarWideningLowerAndAdd(Vector128<ulong>
addend, Vector64<uint> left, Vector128<uint> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyBySelectedScalarWideningLowerAndAddTest(System.Runtime
.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector64`1[Int16],Sy
stem.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Ve
ctor128`1[Int32]
;
```

```

; V00 arg0      [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8  -> d1
HFA(simd8)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;* V03 arg3      [V03  ] ( 0, 0 ) ubyte  -> zero-ref
;# V04 OutArgs   [V04  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    smlal  v0.4s, v1.4h, v2.h[2]
    ldp    fp, lr, [sp],#16
    ret    lr

```

```

; Total bytes of code 20, prolog size 8

```

176. MultiplyBySelectedScalarWideningLowerAndSubtract

```
Vector128<int>  
MultiplyBySelectedScalarWideningLowerAndSubtract(Vector128<int> minuend,  
Vector64<short> left, Vector64<short> right, byte rightIndex)
```

This method multiplies corresponding values in the left and right vectors, and subtracts the results from the vector elements of the minuend vector and returns the result. As seen in example below, the result vector element's size `int` is twice as long as that of input's size `short`.

```
private Vector128<int>  
MultiplyBySelectedScalarWideningLowerAndSubtractTest(Vector128<int> minuend,  
Vector64<short> left, Vector64<short> right, byte rightIndex)  
{  
    return AdvSimd.MultiplyBySelectedScalarWideningLowerAndSubtract(minuend,  
left, right, 2);  
}  
// minuend = <1000, 1000, 1000, 1000>  
// left = <11, 12, 13, 14>  
// right = <21, 22, 23, 24>  
// rightIndex = 2  
// Result = <747, 724, 701, 678>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector128<int>  
MultiplyBySelectedScalarWideningLowerAndSubtract(Vector128<int> minuend,  
Vector64<short> left, Vector128<short> right, byte rightIndex)  
Vector128<long>  
MultiplyBySelectedScalarWideningLowerAndSubtract(Vector128<long> minuend,  
Vector64<int> left, Vector64<int> right, byte rightIndex)  
Vector128<long>  
MultiplyBySelectedScalarWideningLowerAndSubtract(Vector128<long> minuend,  
Vector64<int> left, Vector128<int> right, byte rightIndex)  
Vector128<uint>  
MultiplyBySelectedScalarWideningLowerAndSubtract(Vector128<uint> minuend,  
Vector64<ushort> left, Vector64<ushort> right, byte rightIndex)  
Vector128<uint>  
MultiplyBySelectedScalarWideningLowerAndSubtract(Vector128<uint> minuend,  
Vector64<ushort> left, Vector128<ushort> right, byte rightIndex)  
Vector128<ulong>  
MultiplyBySelectedScalarWideningLowerAndSubtract(Vector128<ulong> minuend,  
Vector64<uint> left, Vector64<uint> right, byte rightIndex)  
Vector128<ulong>  
MultiplyBySelectedScalarWideningLowerAndSubtract(Vector128<ulong> minuend,  
Vector64<uint> left, Vector128<uint> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyBySelectedScalarWideningLowerAndSubtractTest(System.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8  -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;* V03 arg3          [V03      ] ( 0, 0 ) ubyte  -> zero-ref
;# V04 OutArgs       [V04      ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    smlsl1 v0.4s, v1.4h, v2.h[2]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

177. MultiplyBySelectedScalarWideningUpper

Vector128<int> MultiplyBySelectedScalarWideningUpper(Vector128<short> left, Vector64<short> right, byte rightIndex)

This method multiplies each vector element in the left vector by the rightIndex vector element of the right vector, places the product in a result vector, and returns the result vector. As seen in example below, the result vector element int size is twice as long as the elements that are multiplied short.

```
private Vector128<int>
MultiplyBySelectedScalarWideningUpperTest(Vector128<short> left,
Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplyBySelectedScalarWideningUpper(left, right, 2);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <11, 12, 13, 14>
// rightIndex = 2
// Result = <345, 368, 391, 414>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyBySelectedScalarWideningUpper(Vector128<short> left,
Vector128<short> right, byte rightIndex)
Vector128<long> MultiplyBySelectedScalarWideningUpper(Vector128<int> left,
Vector64<int> right, byte rightIndex)
Vector128<long> MultiplyBySelectedScalarWideningUpper(Vector128<int> left,
Vector128<int> right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalarWideningUpper(Vector128<ushort> left,
Vector64<ushort> right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalarWideningUpper(Vector128<ushort> left,
Vector128<ushort> right, byte rightIndex)
Vector128<ulong> MultiplyBySelectedScalarWideningUpper(Vector128<uint> left,
Vector64<uint> right, byte rightIndex)
Vector128<ulong> MultiplyBySelectedScalarWideningUpper(Vector128<uint> left,
Vector128<uint> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyBySelectedScalarWideningUpperTest(System.Runtime.Intrinsics.Vector128`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
```



```

HFA(simd8)
;* V02 arg2      [V02   ] (  0,  0  )  ubyte  ->  zero-ref
;# V03 OutArgs   [V03   ] (  1,  1  )  lclBlk (  0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    smull2 v16.4s, v0.8h, v1.h[2]
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

178. MultiplyBySelectedScalarWideningUpperAndAdd

Vector128<int> MultiplyBySelectedScalarWideningUpperAndAdd(Vector128<int> addend, Vector128<short> left, Vector64<short> right, byte rightIndex)

This method multiplies corresponding values in left and right vectors, and accumulates the results with the vector elements of the addend vector and return the result vector. As seen in example below, the result vector element int size is twice as long as the elements that are multiplied short.

```
private Vector128<int>
MultiplyBySelectedScalarWideningUpperAndAddTest(Vector128<int> addend,
Vector128<short> left, Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplyBySelectedScalarWideningUpperAndAdd(addend, left,
right, 0);
}
// addend = <1000, 1000, 1000, 1000>
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <11, 12, 13, 14>
// rightIndex = 0
// Result = <1165, 1176, 1187, 1198>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyBySelectedScalarWideningUpperAndAdd(Vector128<int>
addend, Vector128<short> left, Vector128<short> right, byte rightIndex)
Vector128<long> MultiplyBySelectedScalarWideningUpperAndAdd(Vector128<long>
addend, Vector128<int> left, Vector64<int> right, byte rightIndex)
Vector128<long> MultiplyBySelectedScalarWideningUpperAndAdd(Vector128<long>
addend, Vector128<int> left, Vector128<int> right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalarWideningUpperAndAdd(Vector128<uint>
addend, Vector128<ushort> left, Vector64<ushort> right, byte rightIndex)
Vector128<uint> MultiplyBySelectedScalarWideningUpperAndAdd(Vector128<uint>
addend, Vector128<ushort> left, Vector128<ushort> right, byte rightIndex)
Vector128<ulong> MultiplyBySelectedScalarWideningUpperAndAdd(Vector128<ulong>
addend, Vector128<uint> left, Vector64<uint> right, byte rightIndex)
Vector128<ulong> MultiplyBySelectedScalarWideningUpperAndAdd(Vector128<ulong>
addend, Vector128<uint> left, Vector128<uint> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyBySelectedScalarWideningUpperAndAddTest(System.Runtime
.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector128`1[Int16],S
ystem.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.V
ector128`1[Int32]
;
```

```

; V00 arg0      [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1      [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;* V03 arg3      [V03  ] ( 0, 0 ) ubyte  -> zero-ref
;# V04 OutArgs   [V04  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    smlal2 v0.4s, v1.8h, v2.h[0]
    ldp    fp, lr, [sp],#16
    ret    lr

```

```

; Total bytes of code 20, prolog size 8

```

179. MultiplyBySelectedScalarWideningUpperAndSubtract

```
Vector128<int>  
MultiplyBySelectedScalarWideningUpperAndSubtract(Vector128<int> minuend,  
Vector128<short> left, Vector64<short> right, byte rightIndex)
```

This method multiplies corresponding values in `left` and `right` vectors, and subtracts the results with the vector elements of the `minuend` vector and return the result vector. As seen in example below, the result vector element `int` size is twice as long as the elements that are multiplied `short`.

```
private Vector128<int>  
MultiplyBySelectedScalarWideningUpperAndSubtractTest(Vector128<int> minuend,  
Vector128<short> left, Vector64<short> right, byte rightIndex)  
{  
    return AdvSimd.MultiplyBySelectedScalarWideningUpperAndSubtract(minuend,  
left, right, 0);  
}  
// minuend = <11, 12, 13, 14>  
// left = <11, 12, 13, 14, 15, 16, 17, 18>  
// right = <11, 12, 13, 14>  
// rightIndex = 0  
// Result = <-154, -164, -174, -184>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector128<int>  
MultiplyBySelectedScalarWideningUpperAndSubtract(Vector128<int> minuend,  
Vector128<short> left, Vector128<short> right, byte rightIndex)  
Vector128<long>  
MultiplyBySelectedScalarWideningUpperAndSubtract(Vector128<long> minuend,  
Vector128<int> left, Vector64<int> right, byte rightIndex)  
Vector128<long>  
MultiplyBySelectedScalarWideningUpperAndSubtract(Vector128<long> minuend,  
Vector128<int> left, Vector128<int> right, byte rightIndex)  
Vector128<uint>  
MultiplyBySelectedScalarWideningUpperAndSubtract(Vector128<uint> minuend,  
Vector128<ushort> left, Vector64<ushort> right, byte rightIndex)  
Vector128<uint>  
MultiplyBySelectedScalarWideningUpperAndSubtract(Vector128<uint> minuend,  
Vector128<ushort> left, Vector128<ushort> right, byte rightIndex)  
Vector128<ulong>  
MultiplyBySelectedScalarWideningUpperAndSubtract(Vector128<ulong> minuend,  
Vector128<uint> left, Vector64<uint> right, byte rightIndex)  
Vector128<ulong>  
MultiplyBySelectedScalarWideningUpperAndSubtract(Vector128<ulong> minuend,  
Vector128<uint> left, Vector128<uint> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyBySelectedScalarWideningUpperAndSubtractTest(System.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector128`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;* V03 arg3          [V03      ] ( 0, 0 ) ubyte  -> zero-ref
;# V04 OutArgs       [V04      ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    smisl2 v0.4s, v1.8h, v2.h[0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

180. MultiplyDoublingByScalarSaturateHigh

Vector64<short> MultiplyDoublingByScalarSaturateHigh(Vector64<short> left, Vector64<short> right)

This method multiplies each vector element in the left by the rightIndex vector element of the right vector, doubles the results, places the most significant half of the final results in a result vector.

```
private Vector64<short>
MultiplyDoublingByScalarSaturateHighTest(Vector64<short> left,
Vector64<short> right)
{
    return AdvSimd.MultiplyDoublingByScalarSaturateHigh(left, right);
}
// left = <1000, 12, 13, 14>
// right = <100, 22, 23, 24>
// Result = <3, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> MultiplyDoublingByScalarSaturateHigh(Vector64<int> left,
Vector64<int> right)
Vector128<short> MultiplyDoublingByScalarSaturateHigh(Vector128<short> left,
Vector64<short> right)
Vector128<int> MultiplyDoublingByScalarSaturateHigh(Vector128<int> left,
Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyDoublingByScalarSaturateHighTest(System.Runtime.Intrin
sics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Ru
ntime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmulh v16.4h, v0.4h, v1.h[0]
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr
```

; Total bytes of code 24, prolog size 8

181. MultiplyDoublingBySelectedScalarSaturateHigh

Vector64<short> MultiplyDoublingBySelectedScalarSaturateHigh(Vector64<short> left, Vector64<short> right, byte rightIndex)

This method multiplies each vector element in the left by the specified vector element at rightIndex of the right vector, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the result vector.

```
private Vector64<short>
MultiplyDoublingBySelectedScalarSaturateHighTest(Vector64<short> left,
Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplyDoublingBySelectedScalarSaturateHigh(left, right,
0);
}
// left = <1000, 500, 13, 14>
// right = <500, 22, 23, 24>
// rightIndex = 0
// Result = <15, 7, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> MultiplyDoublingBySelectedScalarSaturateHigh(Vector64<short>
left, Vector128<short> right, byte rightIndex)
Vector64<int> MultiplyDoublingBySelectedScalarSaturateHigh(Vector64<int>
left, Vector64<int> right, byte rightIndex)
Vector64<int> MultiplyDoublingBySelectedScalarSaturateHigh(Vector64<int>
left, Vector128<int> right, byte rightIndex)
Vector128<short>
MultiplyDoublingBySelectedScalarSaturateHigh(Vector128<short> left,
Vector64<short> right, byte rightIndex)
Vector128<short>
MultiplyDoublingBySelectedScalarSaturateHigh(Vector128<short> left,
Vector128<short> right, byte rightIndex)
Vector128<int> MultiplyDoublingBySelectedScalarSaturateHigh(Vector128<int>
left, Vector64<int> right, byte rightIndex)
Vector128<int> MultiplyDoublingBySelectedScalarSaturateHigh(Vector128<int>
left, Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyDoublingBySelectedScalarSaturateHighTest(System.Runtim
e.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ub
yte):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
```



```

HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;* V02 arg2      [V02  ] ( 0, 0 )  ubyte  ->  zero-ref
;# V03 OutArgs   [V03  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmulh v16.4h, v0.4h, v1.h[0]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

182. MultiplyDoublingSaturateHigh

Vector64<short> MultiplyDoublingSaturateHigh(Vector64<short> left, Vector64<short> right)

This method multiplies the values of corresponding elements of the left and right vectors, doubles the results, places the most significant half of the result in a result vector, and returns the result vector.

```
private Vector64<short> MultiplyDoublingSaturateHighTest(Vector64<short>
left, Vector64<short> right)
{
    return AdvSimd.MultiplyDoublingSaturateHigh(left, right);
}
// left = <1000, 500, 13, 14>
// right = <500, 22, 23, 24>
// Result = <15, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> MultiplyDoublingSaturateHigh(Vector64<int> left, Vector64<int>
right)
Vector128<short> MultiplyDoublingSaturateHigh(Vector128<short> left,
Vector128<short> right)
Vector128<int> MultiplyDoublingSaturateHigh(Vector128<int> left,
Vector128<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingSaturateHighTest(System.Runtime.Intrinsics.Vec
tor64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.In
trinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmulh v16.4h, v0.4h, v1.4h
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

183. MultiplyDoublingSaturateHighScalar

Vector64<short> MultiplyDoublingSaturateHighScalar(Vector64<short> left, Vector64<short> right)

This method multiplies the values of corresponding elements of the left and right vectors, doubles the results, places the most significant half of the result in a result vector, and returns the result vector.

```
private Vector64<short>
MultiplyDoublingSaturateHighScalarTest(Vector64<short> left, Vector64<short>
right)
{
    return AdvSimd.Arm64.MultiplyDoublingSaturateHighScalar(left, right);
}
// left = <11, 12, 13, 14>
// right = <10210, 20020, 230, 240>
// Result = <3, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<int> MultiplyDoublingSaturateHighScalar(Vector64<int> left,
Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingSaturateHighScalarTest(System.Runtime.Intrinsi
cs.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runt
ime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmulh h16, h0, h1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

184. MultiplyDoublingScalarBySelectedScalarSaturateHigh

```
Vector64<short>  
MultiplyDoublingScalarBySelectedScalarSaturateHigh(Vector64<short> left,  
Vector64<short> right, byte rightIndex)
```

This method multiplies vector elements in the left vector by the rightIndex vector element of the right vector, doubles the results, places the most significant half of the truncated result in a result vector, and returns the result vector. All the other elements of result vector other than 0th element are set to 0.

```
private Vector64<short>  
MultiplyDoublingScalarBySelectedScalarSaturateHighTest(Vector64<short> left,  
Vector64<short> right, byte rightIndex)  
{  
    return  
    AdvSimd.Arm64.MultiplyDoublingScalarBySelectedScalarSaturateHigh(left, right,  
0);  
}  
// left = <11, 12, 13, 14>  
// right = <10000, 22, 23, 24>  
// rightIndex = 0  
// Result = <3, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64  
Vector64<short>  
MultiplyDoublingScalarBySelectedScalarSaturateHigh(Vector64<short> left,  
Vector128<short> right, byte rightIndex)  
Vector64<int>  
MultiplyDoublingScalarBySelectedScalarSaturateHigh(Vector64<int> left,  
Vector64<int> right, byte rightIndex)  
Vector64<int>  
MultiplyDoublingScalarBySelectedScalarSaturateHigh(Vector64<int> left,  
Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods.MultiplyDoublingScalarBySelectedScalarSaturateHighTest(System.  
Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int  
16],ubyte):System.Runtime.Intrinsics.Vector64`1[Int16]  
;  
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0  
HFA(simd8)  
; V01 arg1          [V01,T01] (  3,  3  )  simd8  ->  d1  
HFA(simd8)  
;* V02 arg2          [V02  ] (  0,  0  )  ubyte   ->  zero-ref
```

```

;# V03 OutArgs      [V03      ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmulh h16, h0, v1.h[0]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

185. MultiplyDoublingWideningAndAddSaturateScalar

Vector64<int> MultiplyDoublingWideningAndAddSaturateScalar(Vector64<int> addend, Vector64<short> left, Vector64<short> right)

This method multiplies corresponding signed integer values in the left and right vectors, doubles the results, and accumulates the final results with the vector elements of the addend vector. The result vector elements are twice as long as the elements that are multiplied. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<int>
MultiplyDoublingWideningAndAddSaturateScalarTest(Vector64<int> addend,
Vector64<short> left, Vector64<short> right)
{
    return AdvSimd.Arm64.MultiplyDoublingWideningAndAddSaturateScalar(addend,
left, right);
}
// addend = <11, 12>
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// Result = <473, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<long> MultiplyDoublingWideningAndAddSaturateScalar(Vector64<long>
addend, Vector64<int> left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningAndAddSaturateScalarTest(System.Runtim
e.Intrinsics.Vector64`1[Int32],System.Runtime.Intrinsics.Vector64`1[Int16],Sy
stem.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vector64
`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmlal s0, h1, h2
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

186. MultiplyDoublingWideningAndSubtractSaturateScalar

Vector64<int> MultiplyDoublingWideningAndSubtractSaturateScalar(Vector64<int> minuend, Vector64<short> left, Vector64<short> right)

This method multiplies corresponding signed integer values in the left and right vectors, doubles the results, and subtracts the final results from the vector elements of the minuend. The result vector elements are twice as long as the elements that are multiplied. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<int>
MultiplyDoublingWideningAndSubtractSaturateScalarTest(Vector64<int> minuend,
Vector64<short> left, Vector64<short> right)
{
    return
AdvSimd.Arm64.MultiplyDoublingWideningAndSubtractSaturateScalar(minuend,
left, right);
}
// minuend = <11, 12>
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// Result = <-451, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<long>
MultiplyDoublingWideningAndSubtractSaturateScalar(Vector64<long> minuend,
Vector64<int> left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningAndSubtractSaturateScalarTest(System.R
untime.Intrinsics.Vector64`1[Int32],System.Runtime.Intrinsics.Vector64`1[Int1
6],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vec
tor64`1[Int32]
;
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] (  3,  3  )  simd8  ->  d1
HFA(simd8)
; V02 arg2          [V02,T02] (  3,  3  )  simd8  ->  d2
HFA(simd8)
;# V03 OutArgs      [V03      ] (  1,  1  )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp     fp, lr, [sp,#-16]!
        mov     fp, sp
```

```
sqdm1s1 s0, h1, h2
ldp      fp, lr, [sp],#16
ret      lr
```

; Total bytes of code 20, prolog size 8

187. MultiplyDoublingWideningLowerAndAddSaturate

Vector128<int> MultiplyDoublingWideningLowerAndAddSaturate(Vector128<int> addend, Vector64<short> left, Vector64<short> right)

This method multiplies corresponding signed integer values in the `left` and `right` vectors, doubles the results, and accumulates the final results with the vector elements of the `addend` vector and return the accumulated result. The destination vector elements are twice as long as the elements that are multiplied. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
MultiplyDoublingWideningLowerAndAddSaturateTest(Vector128<int> addend,
Vector64<short> left, Vector64<short> right)
{
    return AdvSimd.MultiplyDoublingWideningLowerAndAddSaturate(addend, left,
right);
}
// addend = <11, 12, 13, 14>
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// Result = <473, 540, 611, 686>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long> MultiplyDoublingWideningLowerAndAddSaturate(Vector128<long>
addend, Vector64<int> left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyDoublingWideningLowerAndAddSaturateTest(System.Runtime
.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector64`1[Int16],Sy
stem.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vector12
8`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8  -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;# V03 OutArgs      [V03      ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmlal v0.4s, v1.4h, v2.4h
```

```
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 20, prolog size 8

188. MultiplyDoublingWideningLowerAndSubtractSaturate

```
Vector128<int>  
MultiplyDoublingWideningLowerAndSubtractSaturate(Vector128<int> minuend,  
Vector64<short> left, Vector64<short> right)
```

This method multiplies corresponding signed integer values in the `left` and `right` vectors, doubles the results, and subtracts the final results from the vector elements of the `minuend` vector and return the result. The destination vector elements are twice as long as the elements that are multiplied. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>  
MultiplyDoublingWideningLowerAndSubtractSaturateTest(Vector128<int> minuend,  
Vector64<short> left, Vector64<short> right)  
{  
    return AdvSimd.MultiplyDoublingWideningLowerAndSubtractSaturate(minuend,  
left, right);  
}  
// minuend = <11, 12, 13, 14>  
// left = <11, 12, 13, 14>  
// right = <21, 22, 23, 24>  
// Result = <-451, -516, -585, -658>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector128<long>  
MultiplyDoublingWideningLowerAndSubtractSaturate(Vector128<long> minuend,  
Vector64<int> left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods.MultiplyDoublingWideningLowerAndSubtractSaturateTest(System.Ru  
ntime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector64`1[Int1  
6],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vec  
tor128`1[Int32]  
;  
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0  
HFA(simd16)  
; V01 arg1          [V01,T01] ( 3, 3 ) simd8  -> d1  
HFA(simd8)  
; V02 arg2          [V02,T02] ( 3, 3 ) simd8  -> d2  
HFA(simd8)  
;# V03 OutArgs      [V03  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]  
"OutgoingArgSpace"  
; Lcl frame size = 0  
    stp    fp, lr, [sp,#-16]!
```

```
mov    fp, sp
sqdm1sl v0.4s, v1.4h, v2.4h
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 20, prolog size 8

189. MultiplyDoublingWideningLowerByScalarAndAddSaturate

```
Vector128<int>
MultiplyDoublingWideningLowerByScalarAndAddSaturate(Vector128<int> addend,
Vector64<short> left, Vector64<short> right)
```

This method multiplies each element in the left vector by the 0th element of the right vector, doubles the results, and accumulates the product with corresponding vector elements of the addend vector and return the accumulated result. As seen in below example, the result vector element's size int is twice as long as that of input vector element's size short. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
MultiplyDoublingWideningLowerByScalarAndAddSaturateTest(Vector128<int>
addend, Vector64<short> left, Vector64<short> right)
{
    return AdvSimd.MultiplyDoublingWideningLowerByScalarAndAddSaturate(addend,
left, right);
}
// addend = <11, 12, 13, 14>
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// Result = <473, 516, 559, 602>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long>
MultiplyDoublingWideningLowerByScalarAndAddSaturate(Vector128<long> addend,
Vector64<int> left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyDoublingWideningLowerByScalarAndAddSaturateTest(System
.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector64`1[I
nt16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.
Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8  -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;# V03 OutArgs      [V03  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp      fp, lr, [sp,#-16]!
```

```
mov    fp, sp
sqdmlal v0.4s, v1.4h, v2.h[0]
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 20, prolog size 8

190. MultiplyDoublingWideningLowerByScalarAndSubtractSaturate

```
Vector128<int>  
MultiplyDoublingWideningLowerByScalarAndSubtractSaturate(Vector128<int>  
minuend, Vector64<short> left, Vector64<short> right)
```

This method multiplies each element in the left vector by the 0th element of the right vector, doubles the results, and subtracts the product with corresponding vector elements of the minuend vector and return the result. As seen in below example, the result vector element's size int is twice as long as that of input vector element's size short. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>  
MultiplyDoublingWideningLowerByScalarAndSubtractSaturateTest(Vector128<int>  
minuend, Vector64<short> left, Vector64<short> right)  
{  
    return  
AdvSimd.MultiplyDoublingWideningLowerByScalarAndSubtractSaturate(minuend,  
left, right);  
}  
// minuend = <11, 12, 13, 14>  
// left = <11, 12, 13, 14>  
// right = <21, 22, 23, 24>  
// Result = <-451, -492, -533, -574>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector128<long>  
MultiplyDoublingWideningLowerByScalarAndSubtractSaturate(Vector128<long>  
minuend, Vector64<int> left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods:MultiplyDoublingWideningLowerByScalarAndSubtractSaturateTest(S  
ystem.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector6  
4`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrin  
sics.Vector128`1[Int32]  
;  
; V00 arg0 [V00,T00] ( 3, 3 ) simd16 -> d0  
HFA(simd16)  
; V01 arg1 [V01,T01] ( 3, 3 ) simd8 -> d1  
HFA(simd8)  
; V02 arg2 [V02,T02] ( 3, 3 ) simd8 -> d2  
HFA(simd8)  
;# V03 OutArgs [V03 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]  
"OutgoingArgSpace"  
; Lcl frame size = 0
```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    sqdmmlsl v0.4s, v1.4h, v2.h[0]  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 20, prolog size 8

191. MultiplyDoublingWideningLowerBySelectedScalarAndAddSaturate

Vector128<int>

MultiplyDoublingWideningLowerBySelectedScalarAndAddSaturate(Vector128<int> addend, Vector64<short> left, Vector64<short> right, byte rightIndex)

This method multiplies each element in the left vector by the rightIndex element of the right vector, doubles the results, and accumulates the product with corresponding vector elements of the addend vector and return the accumulated result. As seen in below example, the result vector element's size int is twice as long as that of input vector element's size short. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
MultiplyDoublingWideningLowerBySelectedScalarAndAddSaturateTest(Vector128<int>
> addend, Vector64<short> left, Vector64<short> right, byte rightIndex)
{
    return
AdvSimd.MultiplyDoublingWideningLowerBySelectedScalarAndAddSaturate(addend,
left, right, 0);
}
// addend = <11, 12, 13, 14>
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// rightIndex = 0
// Result = <473, 516, 559, 602>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int>
MultiplyDoublingWideningLowerBySelectedScalarAndAddSaturate(Vector128<int>
addend, Vector64<short> left, Vector128<short> right, byte rightIndex)
Vector128<long>
MultiplyDoublingWideningLowerBySelectedScalarAndAddSaturate(Vector128<long>
addend, Vector64<int> left, Vector64<int> right, byte rightIndex)
Vector128<long>
MultiplyDoublingWideningLowerBySelectedScalarAndAddSaturate(Vector128<long>
addend, Vector64<int> left, Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningLowerBySelectedScalarAndAddSaturateTes
t(System.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vect
or64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runti
me.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
```

```

; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;* V03 arg3      [V03  ] ( 0, 0 ) ubyte -> zero-ref
;# V04 OutArgs   [V04  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmlal v0.4s, v1.4h, v2.h[0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

192. MultiplyDoublingWideningLowerBySelectedScalarAndSubtractSaturate

Vector128<int>

MultiplyDoublingWideningLowerBySelectedScalarAndSubtractSaturate(Vector128<int> minuend, Vector64<short> left, Vector64<short> right, byte rightIndex)

This method multiplies each element in the left vector by the rightIndex element of the right vector, doubles the results, and subtracts the product with corresponding vector elements of the minuend vector and return the result. As seen in below example, the result vector element's size int is twice as long as that of input vector element's size short. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
```

```
MultiplyDoublingWideningLowerBySelectedScalarAndSubtractSaturateTest(Vector128<int> minuend, Vector64<short> left, Vector64<short> right, byte rightIndex)
{
```

```
    return
```

```
AdvSimd.MultiplyDoublingWideningLowerBySelectedScalarAndSubtractSaturate(minuend, left, right, 0);
}
```

```
// minuend = <11, 12, 13, 14>
```

```
// left = <11, 12, 13, 14>
```

```
// right = <21, 22, 23, 24>
```

```
// rightIndex = 0
```

```
// Result = <-451, -492, -533, -574>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
```

```
Vector128<int>
```

```
MultiplyDoublingWideningLowerBySelectedScalarAndSubtractSaturate(Vector128<int> minuend, Vector64<short> left, Vector128<short> right, byte rightIndex)
```

```
Vector128<long>
```

```
MultiplyDoublingWideningLowerBySelectedScalarAndSubtractSaturate(Vector128<long> minuend, Vector64<int> left, Vector64<int> right, byte rightIndex)
```

```
Vector128<long>
```

```
MultiplyDoublingWideningLowerBySelectedScalarAndSubtractSaturate(Vector128<long> minuend, Vector64<int> left, Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
```

```
AdvSimdMethods.MultiplyDoublingWideningLowerBySelectedScalarAndSubtractSaturateTest(System.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Int32]
```

```
;
```

```
; V00 arg0 [V00,T00] ( 3, 3 ) simd16 -> d0
```

```
HFA(simd16)
```

```

; V01 arg1      [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
; V02 arg2      [V02,T02] ( 3, 3 )  simd8  ->  d2
HFA(simd8)
;* V03 arg3      [V03      ] ( 0, 0 )  ubyte  ->  zero-ref
;# V04 OutArgs   [V04      ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
      stp      fp, lr, [sp,#-16]!
      mov      fp, sp
      sqdmlsl  v0.4s, v1.4h, v2.h[0]
      ldp      fp, lr, [sp],#16
      ret      lr

; Total bytes of code 20, prolog size 8

```

193. MultiplyDoublingWideningSaturateLower

Vector128<int> MultiplyDoublingWideningSaturateLower(Vector64<short> left, Vector64<short> right)

This method multiplies corresponding vector elements in the left and right vectors, doubles the results, stores the result in a vector, and returns the result vector. If overflow occurs with any of the results, those results are saturated. As seen in below example, the result vector element's int size is twice as long as that of input vector element's short size.

```
private Vector128<int>
MultiplyDoublingWideningSaturateLowerTest(Vector64<short> left,
Vector64<short> right)
{
    return AdvSimd.MultiplyDoublingWideningSaturateLower(left, right);
}
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// Result = <462, 528, 598, 672>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long> MultiplyDoublingWideningSaturateLower(Vector64<int> left,
Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods: MultiplyDoublingWideningSaturateLowerTest(System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmull v16.4s, v0.4h, v1.4h
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


194. MultiplyDoublingWideningSaturateLowerByScalar

Vector128<int> MultiplyDoublingWideningSaturateLowerByScalar(Vector64<short> left, Vector64<short> right)

This method multiplies each vector element in the left vector by the 0th vector element of the right vector, doubles the results, stores the results in a vector and returns the result vector. If overflow occurs with any of the results, those results are saturated. As seen in below example, the result vector element's int size is twice as long as that of input vector element's short size.

```
private Vector128<int>
MultiplyDoublingWideningSaturateLowerByScalarTest(Vector64<short> left,
Vector64<short> right)
{
    return AdvSimd.MultiplyDoublingWideningSaturateLowerByScalar(left, right);
}
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// Result = <462, 504, 546, 588>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long> MultiplyDoublingWideningSaturateLowerByScalar(Vector64<int>
left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningSaturateLowerByScalarTest(System.Runti
me.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):
System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmull v16.4s, v0.4h, v1.h[0]
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

195. MultiplyDoublingWideningSaturateLowerBySelectedScalar

```
Vector128<int>  
MultiplyDoublingWideningSaturateLowerBySelectedScalar(Vector64<short> left,  
Vector64<short> right, byte rightIndex)
```

This method multiplies each vector element in the left vector by the rightIndex vector element of the right vector, doubles the results, stores the results in a vector and returns the result vector. If overflow occurs with any of the results, those results are saturated. As seen in below example, the result vector element's int size is twice as long as that of input vector element's short size.

```
private Vector128<int>  
MultiplyDoublingWideningSaturateLowerBySelectedScalarTest(Vector64<short>  
left, Vector64<short> right, byte rightIndex)  
{  
    return AdvSimd.MultiplyDoublingWideningSaturateLowerBySelectedScalar(left,  
right, 2);  
}  
// left= <11, 12, 13, 14>  
// right = <21, 22, 23, 24>  
// rightIndex = 2  
// Result = <506, 552, 598, 644>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector128<int>  
MultiplyDoublingWideningSaturateLowerBySelectedScalar(Vector64<short> left,  
Vector128<short> right, byte rightIndex)  
Vector128<long>  
MultiplyDoublingWideningSaturateLowerBySelectedScalar(Vector64<int> left,  
Vector64<int> right, byte rightIndex)  
Vector128<long>  
MultiplyDoublingWideningSaturateLowerBySelectedScalar(Vector64<int> left,  
Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods: MultiplyDoublingWideningSaturateLowerBySelectedScalarTest(System.Runtime.Intrinsics.Vector64`1[Int16], System.Runtime.Intrinsics.Vector64`1[Int16], ubyte): System.Runtime.Intrinsics.Vector128`1[Int32]  
;  
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0  
HFA(simd8)  
; V01 arg1 [V01,T01] ( 3, 3 ) simd8 -> d1  
HFA(simd8)  
;* V02 arg2 [V02 ] ( 0, 0 ) ubyte -> zero-ref
```

```

;# V03 OutArgs      [V03      ] ( 1, 1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp      fp, lr, [sp,#-16]!
    mov      fp, sp
    sqdmull  v16.4s, v0.4h, v1.h[2]
    mov      v0.16b, v16.16b
    ldp      fp, lr, [sp],#16
    ret      lr

; Total bytes of code 24, prolog size 8

```

196. MultiplyDoublingWideningSaturateScalar

Vector64<int> MultiplyDoublingWideningSaturateScalar(Vector64<short> left, Vector64<short> right)

This method multiplies corresponding vector elements in the left and right vector, doubles the results, stores the result in a vector, and returns the result vector. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<int>
MultiplyDoublingWideningSaturateScalarTest(Vector64<short> left,
Vector64<short> right)
{
    return AdvSimd.Arm64.MultiplyDoublingWideningSaturateScalar(left, right);
}
// left = <11, 12, 13, 14>
// right = <21, 22, 23, 24>
// Result = <462, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<long> MultiplyDoublingWideningSaturateScalar(Vector64<int> left,
Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods: MultiplyDoublingWideningSaturateScalarTest(System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1 [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs [V02 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmull s16, h0, h1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

197. MultiplyDoublingWideningSaturateScalarBySelectedScalar

```
Vector64<int>  
MultiplyDoublingWideningSaturateScalarBySelectedScalar(Vector64<short> left,  
Vector64<short> right, byte rightIndex)
```

This method multiplies each vector element in the left vector by the rightIndex vector element of the right, doubles the results, stores the result in a vector, and returns the result vector. All the values in this method are signed integer values. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<int>  
MultiplyDoublingWideningSaturateScalarBySelectedScalarTest(Vector64<short>  
left, Vector64<short> right, byte rightIndex)  
{  
    return  
AdvSimd.Arm64.MultiplyDoublingWideningSaturateScalarBySelectedScalar(left,  
right, 0);  
}  
// left = <11, 12, 13, 14>  
// right = <21, 22, 23, 24>  
// rightIndex = 0  
// Result = <462, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64  
Vector64<int>  
MultiplyDoublingWideningSaturateScalarBySelectedScalar(Vector64<short> left,  
Vector128<short> right, byte rightIndex)  
Vector64<long>  
MultiplyDoublingWideningSaturateScalarBySelectedScalar(Vector64<int> left,  
Vector64<int> right, byte rightIndex)  
Vector64<long>  
MultiplyDoublingWideningSaturateScalarBySelectedScalar(Vector64<int> left,  
Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods: MultiplyDoublingWideningSaturateScalarBySelectedScalarTest(System.Runtime.Intrinsics.Vector64`1[Int16], System.Runtime.Intrinsics.Vector64`1[Int16], ubyte): System.Runtime.Intrinsics.Vector64`1[Int32]  
;  
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0  
HFA(simd8)  
; V01 arg1 [V01,T01] ( 3, 3 ) simd8 -> d1  
HFA(simd8)  
;* V02 arg2 [V02 ] ( 0, 0 ) ubyte -> zero-ref
```

```

;# V03 OutArgs      [V03      ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmull s16, h0, v1.h[0]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

198. MultiplyDoublingWideningSaturateUpper

Vector128<int> MultiplyDoublingWideningSaturateUpper(Vector128<short> left, Vector128<short> right)

This method multiplies upper half of corresponding vector elements in the left and right vectors, doubles the results, stores the results in a vector, and returns the result vector. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
MultiplyDoublingWideningSaturateUpperTest(Vector128<short> left,
Vector128<short> right)
{
    return AdvSimd.MultiplyDoublingWideningSaturateUpper(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <750, 832, 918, 1008>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long> MultiplyDoublingWideningSaturateUpper(Vector128<int> left,
Vector128<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningSaturateUpperTest(System.Runtime.Intri
nsics.Vector128`1[Int16],System.Runtime.Intrinsics.Vector128`1[Int16]):System
.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmul12 v16.4s, v0.8h, v1.8h
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

199. MultiplyDoublingWideningSaturateUpperByScalar

Vector128<int> MultiplyDoublingWideningSaturateUpperByScalar(Vector128<short> left, Vector64<short> right)

This method multiplies upper half of each vector element in the left vector by the 0th vector element of the right vector, doubles the results, stores the results in a vector, and returns the result vector. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
MultiplyDoublingWideningSaturateUpperByScalarTest(Vector128<short> left,
Vector64<short> right)
{
    return AdvSimd.MultiplyDoublingWideningSaturateUpperByScalar(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <11, 12, 13, 14>
// Result = <330, 352, 374, 396>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long> MultiplyDoublingWideningSaturateUpperByScalar(Vector128<int>
left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningSaturateUpperByScalarTest(System.Runti
me.Intrinsics.Vector128`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16])
:System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmull2 v16.4s, v0.8h, v1.h[0]
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


200. MultiplyDoublingWideningSaturateUpperBySelectedScalar

Vector128<int>
MultiplyDoublingWideningSaturateUpperBySelectedScalar(Vector128<short> left, Vector64<short> right, byte rightIndex)

This method multiplies upper half of each vector element in the left vector by the rightIndex vector element of the right vector, doubles the results, stores the results in a vector, and returns the result vector. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
MultiplyDoublingWideningSaturateUpperBySelectedScalarTest(Vector128<short>
left, Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplyDoublingWideningSaturateUpperBySelectedScalar(left,
right, 2);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <11, 12, 13, 14>
// rightIndex = 2
// Result = <390, 416, 442, 468>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int>
MultiplyDoublingWideningSaturateUpperBySelectedScalar(Vector128<short> left,
Vector128<short> right, byte rightIndex)
Vector128<long>
MultiplyDoublingWideningSaturateUpperBySelectedScalar(Vector128<int> left,
Vector64<int> right, byte rightIndex)
Vector128<long>
MultiplyDoublingWideningSaturateUpperBySelectedScalar(Vector128<int> left,
Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningSaturateUpperBySelectedScalarTest(Syst
em.Runtime.Intrinsics.Vector128`1[Int16],System.Runtime.Intrinsics.Vector64`1
[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8  -> d1
HFA(simd8)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte  -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmul12 v16.4s, v0.8h, v1.h[2]
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

201. MultiplyDoublingWideningScalarBySelectedScalarAndAddSaturate

```
Vector64<int>  
MultiplyDoublingWideningScalarBySelectedScalarAndAddSaturate(Vector64<int>  
addend, Vector64<short> left, Vector64<short> right, byte rightIndex)
```

This method multiplies each vector element in the left vector by the rightIndex vector element of the right vector, doubles the results, and accumulates the results with the corresponding vector elements of the addend vector and return the accumulated result. As seen in example below, the result vector element's size int is twice as long as that of input vector's element's short size. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<int>  
MultiplyDoublingWideningScalarBySelectedScalarAndAddSaturateTest(Vector64<int>  
> addend, Vector64<short> left, Vector64<short> right, byte rightIndex)  
{  
    return  
AdvSimd.Arm64.MultiplyDoublingWideningScalarBySelectedScalarAndAddSaturate(ad  
dend, left, right, 0);  
}  
// addend = <11, 12>  
// left = <11, 12, 13, 14>  
// right = <21, 22, 23, 24>  
// rightIndex = 0  
// Result = <473, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64  
Vector64<int>  
MultiplyDoublingWideningScalarBySelectedScalarAndAddSaturate(Vector64<int>  
addend, Vector64<short> left, Vector128<short> right, byte rightIndex)  
Vector64<long>  
MultiplyDoublingWideningScalarBySelectedScalarAndAddSaturate(Vector64<long>  
addend, Vector64<int> left, Vector64<int> right, byte rightIndex)  
Vector64<long>  
MultiplyDoublingWideningScalarBySelectedScalarAndAddSaturate(Vector64<long>  
addend, Vector64<int> left, Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods: MultiplyDoublingWideningScalarBySelectedScalarAndAddSaturateTe  
st(System.Runtime.Intrinsics.Vector64`1[Int32], System.Runtime.Intrinsics.Vect  
or64`1[Int16], System.Runtime.Intrinsics.Vector64`1[Int16], ubyte): System.Runti  
me.Intrinsics.Vector64`1[Int32]  
;  
; V00 arg0 [V00, T00] ( 3, 3 ) simd8 -> d0
```

```

HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;* V03 arg3      [V03  ] ( 0, 0 ) ubyte -> zero-ref
;# V04 OutArgs   [V04  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmlal s0, h1, v2.h[0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

202. MultiplyDoublingWideningScalarBySelectedScalarAndSubtractSaturate

```
Vector64<int>  
MultiplyDoublingWideningScalarBySelectedScalarAndSubtractSaturate(Vector64<int> minuend, Vector64<short> left, Vector64<short> right, byte rightIndex)
```

This method multiplies each vector element in the left vector by the rightIndex vector element of the right vector, doubles the results, and subtracts the results from the corresponding vector elements of the minuend vector and return the result. As seen in example below, the result vector element's size int is twice as long as that of input vector's element's short size. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<int>  
MultiplyDoublingWideningScalarBySelectedScalarAndSubtractSaturateTest(Vector64<int> minuend, Vector64<short> left, Vector64<short> right, byte rightIndex)  
{  
    return  
    AdvSimd.Arm64.MultiplyDoublingWideningScalarBySelectedScalarAndSubtractSaturate(minuend, left, right, 0);  
}  
// minuend = <11, 12>  
// left = <11, 12, 13, 14>  
// right = <21, 22, 23, 24>  
// rightIndex = 0  
// Result = <-451, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64  
Vector64<int>  
MultiplyDoublingWideningScalarBySelectedScalarAndSubtractSaturate(Vector64<int> minuend, Vector64<short> left, Vector128<short> right, byte rightIndex)  
Vector64<long>  
MultiplyDoublingWideningScalarBySelectedScalarAndSubtractSaturate(Vector64<long> minuend, Vector64<int> left, Vector64<int> right, byte rightIndex)  
Vector64<long>  
MultiplyDoublingWideningScalarBySelectedScalarAndSubtractSaturate(Vector64<long> minuend, Vector64<int> left, Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods: MultiplyDoublingWideningScalarBySelectedScalarAndSubtractSaturateTest(System.Runtime.Intrinsics.Vector64`1[Int32], System.Runtime.Intrinsics.Vector64`1[Int16], System.Runtime.Intrinsics.Vector64`1[Int16], ubyte): System.Runtime.Intrinsics.Vector64`1[Int32]  
;  
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0  
HFA(simd8)
```

```

; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;* V03 arg3      [V03  ] ( 0, 0 ) ubyte -> zero-ref
;# V04 OutArgs   [V04  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmlsl s0, h1, v2.h[0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

203. MultiplyDoublingWideningUpperAndAddSaturate

Vector128<int> MultiplyDoublingWideningUpperAndAddSaturate(Vector128<int> addend, Vector128<short> left, Vector128<short> right)

This method multiplies corresponding elements in upper half of left and right vectors, doubles the results, and accumulates the results with the vector elements of the addend vector. The result vector element's size `int` is twice as long as the input element's size `short`. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
MultiplyDoublingWideningUpperAndAddSaturateTest(Vector128<int> addend,
Vector128<short> left, Vector128<short> right)
{
    return AdvSimd.MultiplyDoublingWideningUpperAndAddSaturate(addend, left,
right);
}
// addend = <11, 12, 13, 14>
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <761, 844, 931, 1022>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long> MultiplyDoublingWideningUpperAndAddSaturate(Vector128<long>
addend, Vector128<int> left, Vector128<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningUpperAndAddSaturateTest(System.Runtime
.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector128`1[Int16],S
ystem.Runtime.Intrinsics.Vector128`1[Int16]):System.Runtime.Intrinsics.Vector
128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2          [V02,T02] ( 3, 3 ) simd16 -> d2
HFA(simd16)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmlal2 v0.4s, v1.8h, v2.8h
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

204. MultiplyDoublingWideningUpperAndSubtractSaturate

```
Vector128<int>  
MultiplyDoublingWideningUpperAndSubtractSaturate(Vector128<int> minuend,  
Vector128<short> left, Vector128<short> right)
```

This method multiplies corresponding elements in upper half of left and right vectors, doubles the results, and subtracts the results with the vector elements of the minuend vector. As seen in below example, the result vector element's size `int` is twice as long as the input element's size `short`. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>  
MultiplyDoublingWideningUpperAndSubtractSaturateTest(Vector128<int> minuend,  
Vector128<short> left, Vector128<short> right)  
{  
    return AdvSimd.MultiplyDoublingWideningUpperAndSubtractSaturate(minuend,  
left, right);  
}  
// minuend = <11, 12, 13, 14>  
// left = <11, 12, 13, 14, 15, 16, 17, 18>  
// right = <21, 22, 23, 24, 25, 26, 27, 28>  
// Result = <-739, -820, -905, -994>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector128<long>  
MultiplyDoublingWideningUpperAndSubtractSaturate(Vector128<long> minuend,  
Vector128<int> left, Vector128<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods.MultiplyDoublingWideningUpperAndSubtractSaturateTest(System.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector128`1[Int16],System.Runtime.Intrinsics.Vector128`1[Int16]):System.Runtime.Intrinsics.Vector128`1[Int32]  
;  
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0  
HFA(simd16)  
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1  
HFA(simd16)  
; V02 arg2          [V02,T02] ( 3, 3 ) simd16 -> d2  
HFA(simd16)  
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]  
"OutgoingArgSpace"  
; Lcl frame size = 0  
    stp    fp, lr, [sp,#-16]!
```

```
mov    fp, sp
sqdmlsl2 v0.4s, v1.8h, v2.8h
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 20, prolog size 8

205. MultiplyDoublingWideningUpperByScalarAndAddSaturate

```
Vector128<int>
MultiplyDoublingWideningUpperByScalarAndAddSaturate(Vector128<int> addend,
Vector128<short> left, Vector64<short> right)
```

This method multiplies each vector element in the upper half of left vector by the 0th vector element of the right vector, doubles the results, and accumulates the final results with the vector elements of the addend. As seen in below example, the result vector element's size int is twice as long as the input element's size short. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
MultiplyDoublingWideningUpperByScalarAndAddSaturateTest(Vector128<int>
addend, Vector128<short> left, Vector64<short> right)
{
    return AdvSimd.MultiplyDoublingWideningUpperByScalarAndAddSaturate(addend,
left, right);
}
// addend = <11, 12, 13, 14>
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <11, 12, 13, 14>
// Result = <341, 364, 387, 410>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long>
MultiplyDoublingWideningUpperByScalarAndAddSaturate(Vector128<long> addend,
Vector128<int> left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningUpperByScalarAndAddSaturateTest(System
.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector128`1[
Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics
.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;# V03 OutArgs      [V03  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp     fp, lr, [sp,#-16]!
```

```
mov    fp, sp
sqdmlal2 v0.4s, v1.8h, v2.h[0]
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 20, prolog size 8

206. MultiplyDoublingWideningUpperByScalarAndSubtractSaturate

```
Vector128<int>
MultiplyDoublingWideningUpperByScalarAndSubtractSaturate(Vector128<int>
minuend, Vector128<short> left, Vector64<short> right)
```

This method multiplies each vector element in the upper half of left vector by the 0th vector element of the right vector, doubles the results, and subtracts the product from the vector elements of the minuend. As seen in below example, the result vector element's size int is twice as long as the input element's size short. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>
MultiplyDoublingWideningUpperByScalarAndSubtractSaturateTest(Vector128<int>
minuend, Vector128<short> left, Vector64<short> right)
{
    return
AdvSimd.MultiplyDoublingWideningUpperByScalarAndSubtractSaturate(minuend,
left, right);
}
// minuend = <11, 12, 13, 14>
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <11, 12, 13, 14>
// Result = <-319, -340, -361, -382>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long>
MultiplyDoublingWideningUpperByScalarAndSubtractSaturate(Vector128<long>
minuend, Vector128<int> left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyDoublingWideningUpperByScalarAndSubtractSaturateTest(S
ystem.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector1
28`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intri
nsics.Vector128`1[Int32]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1      [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;# V03 OutArgs  [V03  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    sqdmulsl2 v0.4s, v1.8h, v2.h[0]  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 20, prolog size 8

207. MultiplyDoublingWideningUpperBySelectedScalarAndAddSaturate

```
Vector128<int>  
MultiplyDoublingWideningUpperBySelectedScalarAndAddSaturate(Vector128<int>  
addend, Vector128<short> left, Vector64<short> right, byte rightIndex)
```

This method multiplies each vector element in the upper half of left vector by the rightIndex vector element of the right vector, doubles the results, and accumulates the final results with the vector elements of the addend. As seen in below example, the result vector element's size int is twice as long as the input element's size short. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>  
MultiplyDoublingWideningUpperBySelectedScalarAndAddSaturateTest(Vector128<int>  
> addend, Vector128<short> left, Vector64<short> right, byte rightIndex)  
{  
    return  
    AdvSimd.MultiplyDoublingWideningUpperBySelectedScalarAndAddSaturate(addend,  
    left, right, 2);  
}  
// addend = <11, 12, 13, 14>  
// left = <11, 12, 13, 14, 15, 16, 17, 18>  
// right = <11, 12, 13, 14>  
// rightIndex = 2  
// Result = <401, 428, 455, 482>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector128<int>  
MultiplyDoublingWideningUpperBySelectedScalarAndAddSaturate(Vector128<int>  
addend, Vector128<short> left, Vector128<short> right, byte rightIndex)  
Vector128<long>  
MultiplyDoublingWideningUpperBySelectedScalarAndAddSaturate(Vector128<long>  
addend, Vector128<int> left, Vector64<int> right, byte rightIndex)  
Vector128<long>  
MultiplyDoublingWideningUpperBySelectedScalarAndAddSaturate(Vector128<long>  
addend, Vector128<int> left, Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods.MultiplyDoublingWideningUpperBySelectedScalarAndAddSaturateTest(System.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector128`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Int32]  
;  
; V00 arg0 [V00,T00] ( 3, 3 ) simd16 -> d0  
HFA(simd16)
```

```

; V01 arg1      [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;* V03 arg3      [V03  ] ( 0, 0 ) ubyte  -> zero-ref
;# V04 OutArgs   [V04  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmlal2 v0.4s, v1.8h, v2.h[2]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

208. MultiplyDoublingWideningUpperBySelectedScalarAndSubtractSaturate

```
Vector128<int>  
MultiplyDoublingWideningUpperBySelectedScalarAndSubtractSaturate(Vector128<int> minuend, Vector128<short> left, Vector64<short> right, byte rightIndex)
```

This method multiplies each vector element in the upper half of left vector by the rightIndex vector element of the right vector, doubles the results, and subtracts the product from the vector elements of the minuend. As seen in below example, the result vector element's size int is twice as long as the input element's size short. If overflow occurs with any of the results, those results are saturated.

```
private Vector128<int>  
MultiplyDoublingWideningUpperBySelectedScalarAndSubtractSaturateTest(Vector128<int> minuend, Vector128<short> left, Vector64<short> right, byte rightIndex)  
{  
    return  
    AdvSimd.MultiplyDoublingWideningUpperBySelectedScalarAndSubtractSaturate(minuend, left, right, 2);  
}  
// minuend = <11, 12, 13, 14>  
// left = <11, 12, 13, 14, 15, 16, 17, 18>  
// right = <11, 12, 13, 14>  
// rightIndex = 2  
// Result = <-379, -404, -429, -454>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector128<int>  
MultiplyDoublingWideningUpperBySelectedScalarAndSubtractSaturate(Vector128<int> minuend, Vector128<short> left, Vector128<short> right, byte rightIndex)  
Vector128<long>  
MultiplyDoublingWideningUpperBySelectedScalarAndSubtractSaturate(Vector128<long> minuend, Vector128<int> left, Vector64<int> right, byte rightIndex)  
Vector128<long>  
MultiplyDoublingWideningUpperBySelectedScalarAndSubtractSaturate(Vector128<long> minuend, Vector128<int> left, Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods.MultiplyDoublingWideningUpperBySelectedScalarAndSubtractSaturateTest(System.Runtime.Intrinsics.Vector128`1[Int32],System.Runtime.Intrinsics.Vector128`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Int32]  
;  
; V00 arg0 [V00,T00] ( 3, 3 ) simd16 -> d0
```

```

HFA(simd16)
; V01 arg1      [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;* V03 arg3      [V03  ] ( 0, 0 ) ubyte  -> zero-ref
;# V04 OutArgs   [V04  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqdmlsl2 v0.4s, v1.8h, v2.h[2]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

209. MultiplyExtended

Vector64<float> MultiplyExtended(Vector64<float> left, Vector64<float> right)

This method multiplies corresponding floating-point values in the left and right vectors, stores the result in a vector and returns the result vector. As per ARM docs, if one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

```
private Vector64<float> MultiplyExtendedTest(Vector64<float> left,
Vector64<float> right)
{
    return AdvSimd.Arm64.MultiplyExtended(left, right);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// Result = <247.25, 281.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector128<double> MultiplyExtended(Vector128<double> left, Vector128<double>
right)
Vector128<float> MultiplyExtended(Vector128<float> left, Vector128<float>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyExtendedTest(System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmulx   v16.2s, v0.2s, v1.2s
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```


210. MultiplyExtendedByScalar

Vector128<double> MultiplyExtendedByScalar(Vector128<double> left, Vector64<double> right)

This method multiplies the floating-point values in the vector elements in the left vector by the floating-point element in the right vector, stores the result in a vector and returns the result vector. As per ARM docs, if one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

```
private Vector128<double> MultiplyExtendedByScalarTest(Vector128<double>
left, Vector64<double> right)
{
    return AdvSimd.Arm64.MultiplyExtendedByScalar(left, right);
}
// left = <11.5, 12.5>
// right = <11.5>
// Result = <132.25, 143.75>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyExtendedByScalarTest(System.Runtime.Intrinsics.Vector1
28`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Int
rinsics.Vector128`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8  -> d1
HFA(simd8)
;# V02 OutArgs      [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmulx   v16.2d, v0.2d, v1.d[0]
    mov     v0.16b, v16.16b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

211. MultiplyExtendedBySelectedScalar

Vector64<float> MultiplyExtendedBySelectedScalar(Vector64<float> left, Vector64<float> right, byte rightIndex)

This method multiplies the floating-point values in the left vector elements by the rightIndex floating-point value in the right vector, stores the result in a vector and returns the result vector. As per ARM docs, if one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

```
private Vector64<float> MultiplyExtendedBySelectedScalarTest(Vector64<float>
left, Vector64<float> right, byte rightIndex)
{
    return AdvSimd.Arm64.MultiplyExtendedBySelectedScalar(left, right, 0);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// rightIndex = 0
// Result = <247.25, 268.75>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> MultiplyExtendedBySelectedScalar(Vector64<float> left,
Vector128<float> right, byte rightIndex)
Vector128<double> MultiplyExtendedBySelectedScalar(Vector128<double> left,
Vector128<double> right, byte rightIndex)
Vector128<float> MultiplyExtendedBySelectedScalar(Vector128<float> left,
Vector64<float> right, byte rightIndex)
Vector128<float> MultiplyExtendedBySelectedScalar(Vector128<float> left,
Vector128<float> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyExtendedBySelectedScalarTest(System.Runtime.Intrinsics
.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single],ubyte):Syste
m.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02      ] ( 0, 0 )  ubyte  ->  zero-ref
;# V03 OutArgs       [V03      ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
                stp      fp, lr, [sp,#-16]!
```



```
mov    fp, sp
fmulx  v16.2s, v0.2s, v1.s[0]
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

212. MultiplyExtendedScalar

Vector64<double> MultiplyExtendedScalar(Vector64<double> left, Vector64<double> right)

This method multiplies corresponding floating-point values in the left and right vectors, stores the resulting floating-point values in a vector, and returns the result vector. As per ARM docs, if one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

```
private Vector64<double> MultiplyExtendedScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.MultiplyExtendedScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <132.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Avsimd.Arm64
Vector64<float> MultiplyExtendedScalar(Vector64<float> left, Vector64<float>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods: MultiplyExtendedScalarTest(System.Runtime.Intrinsics.Vector64`
1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrin
sics.Vector64`1[Double]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmulx   d16, d0, d1
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

213. MultiplyExtendedScalarBySelectedScalar

Vector64<double> MultiplyExtendedScalarBySelectedScalar(Vector64<double> left, Vector128<double> right, byte rightIndex)

This method multiplies corresponding floating-point values in the left vector by the rightIndex floating-point value in the right vector, stores the results in a vector, and returns the result vector. As per ARM docs if one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

```
private Vector64<double>
MultiplyExtendedScalarBySelectedScalarTest(Vector64<double> left,
Vector128<double> right, byte rightIndex)
{
    return AdvSimd.Arm64.MultiplyExtendedScalarBySelectedScalar(left, right,
0);
}
// left = <11.5>
// right = <11.5, 12.5>
// rightIndex = 0
// Result = <132.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> MultiplyExtendedScalarBySelectedScalar(Vector64<float> left,
Vector64<float> right, byte rightIndex)
Vector64<float> MultiplyExtendedScalarBySelectedScalar(Vector64<float> left,
Vector128<float> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyExtendedScalarBySelectedScalarTest(System.Runtime.Intrinsics.Vector64`1[Double],System.Runtime.Intrinsics.Vector128`1[Double],ubyte):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 ->  d1
HFA(simd16)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte  ->  zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmulx   d16, d0, v1.d[0]
```

```
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

214. MultiplyRoundedDoublingByScalarSaturateHigh

Vector64<short> MultiplyRoundedDoublingByScalarSaturateHigh(Vector64<short> left, Vector64<short> right)

This method multiplies each vector element in the left by the 0th vector element of the right, doubles the results, stores the most significant half of the final results into a vector, and returns the result vector. The results are rounded.

```
private Vector64<short>
MultiplyRoundedDoublingByScalarSaturateHighTest(Vector64<short> left,
Vector64<short> right)
{
    return AdvSimd.MultiplyRoundedDoublingByScalarSaturateHigh(left, right);
}
// left = <1000, 2000, 3000, 4000>
// right = <30, 40, 50, 60>
// Result = <1, 2, 3, 4>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> MultiplyRoundedDoublingByScalarSaturateHigh(Vector64<int> left,
Vector64<int> right)
Vector128<short> MultiplyRoundedDoublingByScalarSaturateHigh(Vector128<short>
left, Vector64<short> right)
Vector128<int> MultiplyRoundedDoublingByScalarSaturateHigh(Vector128<int>
left, Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods: MultiplyRoundedDoublingByScalarSaturateHighTest(System.Runtime
.Intrinsics.Vector64`1[Int16], System.Runtime.Intrinsics.Vector64`1[Int16]):Sy
stem.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;# V02 OutArgs      [V02  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqrdmulh v16.4h, v0.4h, v1.h[0]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

215. MultiplyRoundedDoublingBySelectedScalarSaturateHigh

Vector64<short>
MultiplyRoundedDoublingBySelectedScalarSaturateHigh(Vector64<short> left, Vector64<short> right, byte rightIndex)

This method multiplies each vector element in the left by the rightIndex vector element of the right, doubles the results, stores the most significant half of the final results into a vector, and returns the result vector. The results are rounded.

```
private Vector64<short>
MultiplyRoundedDoublingBySelectedScalarSaturateHighTest(Vector64<short> left,
Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplyRoundedDoublingBySelectedScalarSaturateHigh(left,
right, 2);
}
// left = <1000, 2000, 3000, 4000>
// right = <30, 40, 50, 60>
// rightIndex = 2
// Result = <2, 3, 5, 6>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short>
MultiplyRoundedDoublingBySelectedScalarSaturateHigh(Vector64<short> left,
Vector128<short> right, byte rightIndex)
Vector64<int>
MultiplyRoundedDoublingBySelectedScalarSaturateHigh(Vector64<int> left,
Vector64<int> right, byte rightIndex)
Vector64<int>
MultiplyRoundedDoublingBySelectedScalarSaturateHigh(Vector64<int> left,
Vector128<int> right, byte rightIndex)
Vector128<short>
MultiplyRoundedDoublingBySelectedScalarSaturateHigh(Vector128<short> left,
Vector64<short> right, byte rightIndex)
Vector128<short>
MultiplyRoundedDoublingBySelectedScalarSaturateHigh(Vector128<short> left,
Vector128<short> right, byte rightIndex)
Vector128<int>
MultiplyRoundedDoublingBySelectedScalarSaturateHigh(Vector128<int> left,
Vector64<int> right, byte rightIndex)
Vector128<int>
MultiplyRoundedDoublingBySelectedScalarSaturateHigh(Vector128<int> left,
Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:MultiplyRoundedDoublingBySelectedScalarSaturateHighTest(System
.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02      ] ( 0, 0 )   ubyte  ->  zero-ref
;# V03 OutArgs       [V03      ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp     fp, lr, [sp,#-16]!
        mov     fp, sp
        sqrdmulh v16.4h, v0.4h, v1.h[2]
        mov     v0.8b, v16.8b
        ldp     fp, lr, [sp],#16
        ret     lr

; Total bytes of code 24, prolog size 8

```

216. MultiplyRoundedDoublingSaturateHigh

Vector64<short> MultiplyRoundedDoublingSaturateHigh(Vector64<short> left, Vector64<short> right)

This method multiplies corresponding elements of the left and right vectors, doubles the results, stores the most significant half of the results in a vector, and returns the result vector.

```
private Vector64<short>
MultiplyRoundedDoublingSaturateHighTest(Vector64<short> left, Vector64<short>
right)
{
    return AdvSimd.MultiplyRoundedDoublingSaturateHigh(left, right);
}
// left = <1000, 2000, 3000, 4000>
// right = <30, 40, 50, 60>
// Result = <1, 2, 5, 7>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> MultiplyRoundedDoublingSaturateHigh(Vector64<int> left,
Vector64<int> right)
Vector128<short> MultiplyRoundedDoublingSaturateHigh(Vector128<short> left,
Vector128<short> right)
Vector128<int> MultiplyRoundedDoublingSaturateHigh(Vector128<int> left,
Vector128<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyRoundedDoublingSaturateHighTest(System.Runtime.Intrins
ics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Run
time.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqrdmulh v16.4h, v0.4h, v1.4h
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

217. MultiplyRoundedDoublingSaturateHighScalar

Vector64<short> MultiplyRoundedDoublingSaturateHighScalar(Vector64<short> left, Vector64<short> right)

This method multiplies the values of corresponding elements of the left and right vectors, doubles the results, places the most significant half of the result in a result vector at 0th index. Other vector elements are set to 0.

```
private Vector64<short>
MultiplyRoundedDoublingSaturateHighScalarTest(Vector64<short> left,
Vector64<short> right)
{
    return AdvSimd.Arm64.MultiplyRoundedDoublingSaturateHighScalar(left,
right);
}
// left = <11, 12, 13, 14>
// right = <10210, 20020, 230, 240>
// Result = <3, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<int> MultiplyRoundedDoublingSaturateHighScalar(Vector64<int> left,
Vector64<int> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyRoundedDoublingSaturateHighScalarTest(System.Runtime.I
ntrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):Syst
em.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqrdmulh h16, h0, h1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


218. MultiplyRoundedDoublingScalarBySelectedScalarSaturateHigh

```
Vector64<short>  
MultiplyRoundedDoublingScalarBySelectedScalarSaturateHigh(Vector64<short>  
left, Vector64<short> right, byte rightIndex)
```

This method multiplies vector elements in the left vector by the rightIndex vector element of the right vector, doubles the results, stores the most significant half of the result in a vector, and returns the result vector. If any of the results overflows, they are saturated. The results are rounded.

```
private Vector64<short>  
MultiplyRoundedDoublingScalarBySelectedScalarSaturateHighTest(Vector64<short>  
left, Vector64<short> right, byte rightIndex)  
{  
    return  
AdvSimd.Arm64.MultiplyRoundedDoublingScalarBySelectedScalarSaturateHigh(left,  
right, 0);  
}  
// left = <11, 12, 13, 14>  
// right = <10000, 22, 23, 24>  
// rightIndex = 0  
// Result = <3, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64  
Vector64<short>  
MultiplyRoundedDoublingScalarBySelectedScalarSaturateHigh(Vector64<short>  
left, Vector128<short> right, byte rightIndex)  
Vector64<int>  
MultiplyRoundedDoublingScalarBySelectedScalarSaturateHigh(Vector64<int> left,  
Vector64<int> right, byte rightIndex)  
Vector64<int>  
MultiplyRoundedDoublingScalarBySelectedScalarSaturateHigh(Vector64<int> left,  
Vector128<int> right, byte rightIndex)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods: MultiplyRoundedDoublingScalarBySelectedScalarSaturateHighTest(  
System.Runtime.Intrinsics.Vector64`1[Int16], System.Runtime.Intrinsics.Vector64`1[Int16], ubyte): System.Runtime.Intrinsics.Vector64`1[Int16]  
;  
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0  
HFA(simd8)  
; V01 arg1 [V01,T01] ( 3, 3 ) simd8 -> d1  
HFA(simd8)  
;* V02 arg2 [V02 ] ( 0, 0 ) ubyte -> zero-ref
```

```

;# V03 OutArgs      [V03      ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqrdmulh h16, h0, v1.h[0]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

219. MultiplyScalar

Vector64<double> MultiplyScalar(Vector64<double> left, Vector64<double> right)

This method multiplies the floating-point values of the left and right vectors, and returns the result.

```
private Vector64<double> MultiplyScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.MultiplyScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <132.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> MultiplyScalar(Vector64<float> left, Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplyScalarTest(System.Runtime.Intrinsics.Vector64`1[Double]
,System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vec
tor64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmul    d16, d0, d1
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

220. MultiplyScalarBySelectedScalar

Vector64<float> MultiplyScalarBySelectedScalar(Vector64<float> left, Vector64<float> right, byte rightIndex)

This method multiplies the vector elements in the left vector by the element at rightIndex in the right vector, stores the results in a vector, and returns the result vector. All the values in this method are floating-point values.

```
private Vector64<float> MultiplyScalarBySelectedScalarTest(Vector64<float>
left, Vector64<float> right, byte rightIndex)
{
    return AdvSimd.MultiplyScalarBySelectedScalar(left, right, 0);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// rightIndex = 0
// Result = <247.25, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> MultiplyScalarBySelectedScalar(Vector64<float> left,
Vector128<float> right, byte rightIndex)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<double> MultiplyScalarBySelectedScalar(Vector64<double> left,
Vector128<double> right, byte rightIndex)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyScalarBySelectedScalarTest(System.Runtime.Intrinsics.V
ector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single],ubyte):System.
Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fmul    s16, s0, v1.s[0]
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
```



```
ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

221. MultiplySubtract

Vector64<byte> MultiplySubtract(Vector64<byte> minuend, Vector64<byte> left, Vector64<byte> right)

This method multiplies corresponding elements in the vectors of the left and right vectors, and subtracts the results from the vector elements of the minuend vector and returns the result.

```
private Vector64<byte> MultiplySubtractTest(Vector64<byte> minuend,
Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.MultiplySubtract(minuend, left, right);
}
// minuend = <11, 12, 13, 14, 15, 16, 17, 18>
// left = <21, 22, 23, 24, 25, 26, 27, 28>
// right = <31, 32, 33, 34, 35, 36, 37, 38>
// Result = <128, 76, 22, 222, 164, 104, 42, 234>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> MultiplySubtract(Vector64<short> minuend, Vector64<short>
left, Vector64<short> right)
Vector64<int> MultiplySubtract(Vector64<int> minuend, Vector64<int> left,
Vector64<int> right)
Vector64<sbyte> MultiplySubtract(Vector64<sbyte> minuend, Vector64<sbyte>
left, Vector64<sbyte> right)
Vector64<ushort> MultiplySubtract(Vector64<ushort> minuend, Vector64<ushort>
left, Vector64<ushort> right)
Vector64<uint> MultiplySubtract(Vector64<uint> minuend, Vector64<uint> left,
Vector64<uint> right)
Vector128<byte> MultiplySubtract(Vector128<byte> minuend, Vector128<byte>
left, Vector128<byte> right)
Vector128<short> MultiplySubtract(Vector128<short> minuend, Vector128<short>
left, Vector128<short> right)
Vector128<int> MultiplySubtract(Vector128<int> minuend, Vector128<int> left,
Vector128<int> right)
Vector128<sbyte> MultiplySubtract(Vector128<sbyte> minuend, Vector128<sbyte>
left, Vector128<sbyte> right)
Vector128<ushort> MultiplySubtract(Vector128<ushort> minuend,
Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> MultiplySubtract(Vector128<uint> minuend, Vector128<uint>
left, Vector128<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplySubtractTest(System.Runtime.Intrinsics.Vector64`1[Byte
```

```

],System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector
64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 )   simd8  ->  d2
HFA(simd8)
;# V03 OutArgs      [V03      ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    mls    v0.8b, v1.8b, v2.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

222. MultiplySubtractByScalar

Vector64<short> MultiplySubtractByScalar(Vector64<short> minuend, Vector64<short> left, Vector64<short> right)

This method multiplies the vector elements in the left vector by the 0th element value in the right vector, and subtracts the results from the vector elements of the minuend and returns the result.

```
private Vector64<short> MultiplySubtractByScalarTest(Vector64<short> minuend,
Vector64<short> left, Vector64<short> right)
{
    return AdvSimd.MultiplySubtractByScalar(minuend, left, right);
}
// minuend = <11, 12, 13, 14>
// left = <21, 22, 23, 24>
// right = <31, 32, 33, 34>
// Result = <-640, -670, -700, -730>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> MultiplySubtractByScalar(Vector64<int> minuend, Vector64<int>
left, Vector64<int> right)
Vector64<ushort> MultiplySubtractByScalar(Vector64<ushort> minuend,
Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> MultiplySubtractByScalar(Vector64<uint> minuend,
Vector64<uint> left, Vector64<uint> right)
Vector128<short> MultiplySubtractByScalar(Vector128<short> minuend,
Vector128<short> left, Vector64<short> right)
Vector128<int> MultiplySubtractByScalar(Vector128<int> minuend,
Vector128<int> left, Vector64<int> right)
Vector128<ushort> MultiplySubtractByScalar(Vector128<ushort> minuend,
Vector128<ushort> left, Vector64<ushort> right)
Vector128<uint> MultiplySubtractByScalar(Vector128<uint> minuend,
Vector128<uint> left, Vector64<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.MultiplySubtractByScalarTest(System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8 -> d2
```

```

HFA(simd8)
;# V03 OutArgs      [V03      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp    fp, lr, [sp,#-16]!
        mov    fp, sp
        mls    v0.4h, v1.4h, v2.h[0]
        ldp    fp, lr, [sp],#16
        ret    lr

; Total bytes of code 20, prolog size 8

```

223. MultiplySubtractBySelectedScalar

Vector64<short> MultiplySubtractBySelectedScalar(Vector64<short> minuend, Vector64<short> left, Vector64<short> right, byte rightIndex)

This method multiplies the vector elements in the left vector by the rightIndex element value in the right vector, and subtracts the results from the vector elements of the minuend and returns the result.

```
private Vector64<short> MultiplySubtractBySelectedScalarTest(Vector64<short>
minuend, Vector64<short> left, Vector64<short> right, byte rightIndex)
{
    return AdvSimd.MultiplySubtractBySelectedScalar(minuend, left, right, 2);
}
// minuend = <11, 12, 13, 14>
// left = <21, 22, 23, 24>
// right = <31, 32, 33, 34>
// rightIndex = 2
// Result = <-682, -714, -746, -778>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> MultiplySubtractBySelectedScalar(Vector64<short> minuend,
Vector64<short> left, Vector128<short> right, byte rightIndex)
Vector64<int> MultiplySubtractBySelectedScalar(Vector64<int> minuend,
Vector64<int> left, Vector64<int> right, byte rightIndex)
Vector64<int> MultiplySubtractBySelectedScalar(Vector64<int> minuend,
Vector64<int> left, Vector128<int> right, byte rightIndex)
Vector64<ushort> MultiplySubtractBySelectedScalar(Vector64<ushort> minuend,
Vector64<ushort> left, Vector64<ushort> right, byte rightIndex)
Vector64<ushort> MultiplySubtractBySelectedScalar(Vector64<ushort> minuend,
Vector64<ushort> left, Vector128<ushort> right, byte rightIndex)
Vector64<uint> MultiplySubtractBySelectedScalar(Vector64<uint> minuend,
Vector64<uint> left, Vector64<uint> right, byte rightIndex)
Vector64<uint> MultiplySubtractBySelectedScalar(Vector64<uint> minuend,
Vector64<uint> left, Vector128<uint> right, byte rightIndex)
Vector128<short> MultiplySubtractBySelectedScalar(Vector128<short> minuend,
Vector128<short> left, Vector64<short> right, byte rightIndex)
Vector128<short> MultiplySubtractBySelectedScalar(Vector128<short> minuend,
Vector128<short> left, Vector128<short> right, byte rightIndex)
Vector128<int> MultiplySubtractBySelectedScalar(Vector128<int> minuend,
Vector128<int> left, Vector64<int> right, byte rightIndex)
Vector128<int> MultiplySubtractBySelectedScalar(Vector128<int> minuend,
Vector128<int> left, Vector128<int> right, byte rightIndex)
Vector128<ushort> MultiplySubtractBySelectedScalar(Vector128<ushort> minuend,
Vector128<ushort> left, Vector64<ushort> right, byte rightIndex)
Vector128<ushort> MultiplySubtractBySelectedScalar(Vector128<ushort> minuend,
Vector128<ushort> left, Vector128<ushort> right, byte rightIndex)
Vector128<uint> MultiplySubtractBySelectedScalar(Vector128<uint> minuend,
Vector128<uint> left, Vector64<uint> right, byte rightIndex)
```

Vector128<uint> MultiplySubtractBySelectedScalar(Vector128<uint> minuend,
Vector128<uint> left, Vector128<uint> right, byte rightIndex)

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplySubtractBySelectedScalarTest(System.Runtime.Intrinsics
.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime
.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector64`1[Int
16]
;
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] (  3,  3  )  simd8  ->  d1
HFA(simd8)
; V02 arg2          [V02,T02] (  3,  3  )  simd8  ->  d2
HFA(simd8)
;* V03 arg3          [V03      ] (  0,  0  )  ubyte  ->  zero-ref
;# V04 OutArgs       [V04      ] (  1,  1  )  lclBlk (  0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp    fp, lr, [sp,#-16]!
        mov    fp, sp
        mls    v0.4h, v1.4h, v2.h[2]
        ldp    fp, lr, [sp],#16
        ret    lr

; Total bytes of code 20, prolog size 8
```

224. MultiplyWideningLower

Vector128<ushort> MultiplyWideningLower(Vector64<byte> left, Vector64<byte> right)

This method multiplies corresponding vector elements in the left and right vector, stores the result in a vector, and returns the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input element's size byte.

```
private Vector128<ushort> MultiplyWideningLowerTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.MultiplyWideningLower(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <231, 264, 299, 336, 375, 416, 459, 504>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyWideningLower(Vector64<short> left, Vector64<short>
right)
Vector128<long> MultiplyWideningLower(Vector64<int> left, Vector64<int>
right)
Vector128<short> MultiplyWideningLower(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector128<uint> MultiplyWideningLower(Vector64<ushort> left, Vector64<ushort>
right)
Vector128<ulong> MultiplyWideningLower(Vector64<uint> left, Vector64<uint>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyWideningLowerTest(System.Runtime.Intrinsics.Vector64`1
[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.
Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    umull   v16.8h, v0.8b, v1.8b
```



```
mov    v0.16b, v16.16b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

225. MultiplyWideningLowerAndAdd

Vector128<ushort> MultiplyWideningLowerAndAdd(Vector128<ushort> addend, Vector64<byte> left, Vector64<byte> right)

This method multiplies the vector elements in the left by the corresponding vector elements of the right vector, and accumulates the results with the vector elements of the addend vector and return the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input element's size byte.

```
private Vector128<ushort> MultiplyWideningLowerAndAddTest(Vector128<ushort>
addend, Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.MultiplyWideningLowerAndAdd(addend, left, right);
}
// addend = <11, 12, 13, 14, 15, 16, 17, 18>
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <242, 276, 312, 350, 390, 432, 476, 522>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyWideningLowerAndAdd(Vector128<int> addend,
Vector64<short> left, Vector64<short> right)
Vector128<long> MultiplyWideningLowerAndAdd(Vector128<long> addend,
Vector64<int> left, Vector64<int> right)
Vector128<short> MultiplyWideningLowerAndAdd(Vector128<short> addend,
Vector64<sbyte> left, Vector64<sbyte> right)
Vector128<uint> MultiplyWideningLowerAndAdd(Vector128<uint> addend,
Vector64<ushort> left, Vector64<ushort> right)
Vector128<ulong> MultiplyWideningLowerAndAdd(Vector128<ulong> addend,
Vector64<uint> left, Vector64<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyWideningLowerAndAddTest(System.Runtime.Intrinsics.Vect
or128`1[UInt16],System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Int
rinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8  -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    umlal  v0.8h, v1.8b, v2.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

226. MultiplyWideningLowerAndSubtract

Vector128<ushort> MultiplyWideningLowerAndSubtract(Vector128<ushort> minuend, Vector64<byte> left, Vector64<byte> right)

This method multiplies the vector elements in the left by the corresponding vector elements of the right vector, and subtracts the results with the vector elements from the minuend vector and return the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input element's size byte.

```
private Vector128<ushort>
MultiplyWideningLowerAndSubtractTest(Vector128<ushort> minuend,
Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.MultiplyWideningLowerAndSubtract(minuend, left, right);
}
// minuend = <11, 12, 13, 14, 15, 16, 17, 18>
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <65316, 65284, 65250, 65214, 65176, 65136, 65094, 65050>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyWideningLowerAndSubtract(Vector128<int> minuend,
Vector64<short> left, Vector64<short> right)
Vector128<long> MultiplyWideningLowerAndSubtract(Vector128<long> minuend,
Vector64<int> left, Vector64<int> right)
Vector128<short> MultiplyWideningLowerAndSubtract(Vector128<short> minuend,
Vector64<sbyte> left, Vector64<sbyte> right)
Vector128<uint> MultiplyWideningLowerAndSubtract(Vector128<uint> minuend,
Vector64<ushort> left, Vector64<ushort> right)
Vector128<ulong> MultiplyWideningLowerAndSubtract(Vector128<ulong> minuend,
Vector64<uint> left, Vector64<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyWideningLowerAndSubtractTest(System.Runtime.Intrinsics
.Vector128`1[UInt16],System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtim
e.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8  -> d1
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 ) simd8  -> d2
HFA(simd8)
;# V03 OutArgs      [V03   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    umlsl  v0.8h, v1.8b, v2.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

227. MultiplyWideningUpper

Vector128<ushort> MultiplyWideningUpper(Vector128<byte> left, Vector128<byte> right)

This method multiplies corresponding vector elements in the upper-half of left and right vector, stores the result in a vector, and returns the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input element's size byte.

```
private Vector128<ushort> MultiplyWideningUpperTest(Vector128<byte> left,
Vector128<byte> right)
{
    return AdvSimd.MultiplyWideningUpper(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// right = <21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36>
// Result = <551, 600, 651, 704, 759, 816, 875, 936>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyWideningUpper(Vector128<short> left, Vector128<short>
right)
Vector128<long> MultiplyWideningUpper(Vector128<int> left, Vector128<int>
right)
Vector128<short> MultiplyWideningUpper(Vector128<sbyte> left,
Vector128<sbyte> right)
Vector128<uint> MultiplyWideningUpper(Vector128<ushort> left,
Vector128<ushort> right)
Vector128<ulong> MultiplyWideningUpper(Vector128<uint> left, Vector128<uint>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyWideningUpperTest(System.Runtime.Intrinsics.Vector128`
1[Byte],System.Runtime.Intrinsics.Vector128`1[Byte]):System.Runtime.Intrinsic
s.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
```

```
umull2 v16.8h, v0.16b, v1.16b
mov     v0.16b, v16.16b
ldp     fp, lr, [sp],#16
ret     lr
```

; Total bytes of code 24, prolog size 8

228. MultiplyWideningUpperAndAdd

Vector128<ushort> MultiplyWideningUpperAndAdd(Vector128<ushort> addend, Vector128<byte> left, Vector128<byte> right)

This method multiplies corresponding vector elements in the upper-half of left and right vector, and accumulates the results with the vector elements of the addend vector and returns the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input element's size byte.

```
private Vector128<ushort> MultiplyWideningUpperAndAddTest(Vector128<ushort>
addend, Vector128<byte> left, Vector128<byte> right)
{
    return AdvSimd.MultiplyWideningUpperAndAdd(addend, left, right);
}
// addend = <11, 12, 13, 14, 15, 16, 17, 18>
// left = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// right = <21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36>
// Result = <562, 612, 664, 718, 774, 832, 892, 954>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyWideningUpperAndAdd(Vector128<int> addend,
Vector128<short> left, Vector128<short> right)
Vector128<long> MultiplyWideningUpperAndAdd(Vector128<long> addend,
Vector128<int> left, Vector128<int> right)
Vector128<short> MultiplyWideningUpperAndAdd(Vector128<short> addend,
Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<uint> MultiplyWideningUpperAndAdd(Vector128<uint> addend,
Vector128<ushort> left, Vector128<ushort> right)
Vector128<ulong> MultiplyWideningUpperAndAdd(Vector128<ulong> addend,
Vector128<uint> left, Vector128<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyWideningUpperAndAddTest(System.Runtime.Intrinsics.Vect
or128`1[UInt16],System.Runtime.Intrinsics.Vector128`1[Byte],System.Runtime.In
trinsics.Vector128`1[Byte]):System.Runtime.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2          [V02,T02] ( 3, 3 ) simd16 -> d2
HFA(simd16)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
```



```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    umlal2 v0.8h, v1.16b, v2.16b
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

229. MultiplyWideningUpperAndSubtract

Vector128<ushort> MultiplyWideningUpperAndSubtract(Vector128<ushort> minuend, Vector128<byte> left, Vector128<byte> right)

This method multiplies the vector elements in the upper-half of left by the corresponding vector elements of the right vector, and subtracts the results with the vector elements from the minuend vector and return the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input element's size byte.

```
private Vector128<ushort>
MultiplyWideningUpperAndSubtractTest(Vector128<ushort> minuend,
Vector128<byte> left, Vector128<byte> right)
{
    return AdvSimd.MultiplyWideningUpperAndSubtract(minuend, left, right);
}
// minuend = <11, 12, 13, 14, 15, 16, 17, 18>
// left = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// right = <21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36>
// Result = <64996, 64948, 64898, 64846, 64792, 64736, 64678, 64618>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> MultiplyWideningUpperAndSubtract(Vector128<int> minuend,
Vector128<short> left, Vector128<short> right)
Vector128<long> MultiplyWideningUpperAndSubtract(Vector128<long> minuend,
Vector128<int> left, Vector128<int> right)
Vector128<short> MultiplyWideningUpperAndSubtract(Vector128<short> minuend,
Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<uint> MultiplyWideningUpperAndSubtract(Vector128<uint> minuend,
Vector128<ushort> left, Vector128<ushort> right)
Vector128<ulong> MultiplyWideningUpperAndSubtract(Vector128<ulong> minuend,
Vector128<uint> left, Vector128<uint> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:MultiplyWideningUpperAndSubtractTest(System.Runtime.Intrinsics
.Vector128`1[UInt16],System.Runtime.Intrinsics.Vector128`1[Byte],System.Runti
me.Intrinsics.Vector128`1[Byte]):System.Runtime.Intrinsics.Vector128`1[UInt16
]
;
; V00 arg0          [V00,T00] (  3,  3  ) simd16 ->  d0
HFA(simd16)
; V01 arg1          [V01,T01] (  3,  3  ) simd16 ->  d1
HFA(simd16)
; V02 arg2          [V02,T02] (  3,  3  ) simd16 ->  d2
HFA(simd16)
```

```

;# V03 OutArgs      [V03      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp      fp, lr, [sp,#-16]!
    mov      fp, sp
    umlsl2   v0.8h, v1.16b, v2.16b
    ldp      fp, lr, [sp],#16
    ret      lr

; Total bytes of code 20, prolog size 8

```

230. Negate

Vector64<short> Negate(Vector64<short> value)

This method negates each element of value vector and returns the result vector.

```
private Vector64<short> NegateTest(Vector64<short> value)
{
    return AdvSimd.Negate(value);
}
// value = <11, 12, 13, 14>
// Result = <-11, -12, -13, -14>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> Negate(Vector64<int> value)
Vector64<sbyte> Negate(Vector64<sbyte> value)
Vector64<float> Negate(Vector64<float> value)
Vector128<short> Negate(Vector128<short> value)
Vector128<int> Negate(Vector128<int> value)
Vector128<sbyte> Negate(Vector128<sbyte> value)
Vector128<float> Negate(Vector128<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Negate(Vector128<double> value)
Vector128<long> Negate(Vector128<long> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:NegateTest(System.Runtime.Intrinsics.Vector64`1[Int16]):System
.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    neg    v16.4h, v0.4h
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

231. NegateSaturate

Vector64<short> NegateSaturate(Vector64<short> value)

This method negates each vector element in the value vector, stores the results in a vector and returns the result vector. All the values in this method are signed integer values. If there is an overflow with the negation, that element result is saturated.

```
private Vector64<short> NegateSaturateTest(Vector64<short> value)
{
    return AdvSimd.NegateSaturate(value);
}
// value = <11, 12, 13, 14>
// Result = <-11, -12, -13, -14>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> NegateSaturate(Vector64<int> value)
Vector64<sbyte> NegateSaturate(Vector64<sbyte> value)
Vector128<short> NegateSaturate(Vector128<short> value)
Vector128<int> NegateSaturate(Vector128<int> value)
Vector128<sbyte> NegateSaturate(Vector128<sbyte> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<long> NegateSaturate(Vector128<long> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.NegateSaturateTest(System.Runtime.Intrinsics.Vector64`1[Int16]
):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqneg   v16.4h, v0.4h
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

232. NegateSaturateScalar

Vector64<short> NegateSaturateScalar(Vector64<short> value)

This method negates 0th vector element in the value vector, stores the result in 0th element of a vector and returns the result vector. All non-zero elements are initialized to 0. All the values in this method are signed integer values. If there is an overflow with the negation, that element result is saturated.

```
private Vector64<short> NegateSaturateScalarTest(Vector64<short> value)
{
    return AdvSimd.Arm64.NegateSaturateScalar(value);
}
// value = <11, 12, 13, 14>
// Result = <-11, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<int> NegateSaturateScalar(Vector64<int> value)
Vector64<long> NegateSaturateScalar(Vector64<long> value)
Vector64<sbyte> NegateSaturateScalar(Vector64<sbyte> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:NegateSaturateScalarTest(System.Runtime.Intrinsics.Vector64`1[
Int16]):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqneg   h16, h0
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

233. NegateScalar

Vector64<double> NegateScalar(Vector64<double> value)

This method negates the elements in the value vector and returns the result.

```
private Vector64<double> NegateScalarTest(Vector64<double> value)
{
    return AdvSimd.NegateScalar(value);
}
// value = <11.5>
// Result = <-11.5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> NegateScalar(Vector64<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<long> NegateScalar(Vector64<long> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:NegateScalarTest(System.Runtime.Intrinsics.Vector64`1[Double])
:System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fneg   d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

234. Not

Vector64<byte> Not(Vector64<byte> value)

This method performs bitwise inverse of each element of the value vector, stores the result in a vector and returns the result vector.

```
private Vector64<byte> NotTest(Vector64<byte> value)
{
    return AdvSimd.Not(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <244, 243, 242, 241, 240, 239, 238, 237>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<double> Not(Vector64<double> value)
Vector64<short> Not(Vector64<short> value)
Vector64<int> Not(Vector64<int> value)
Vector64<long> Not(Vector64<long> value)
Vector64<sbyte> Not(Vector64<sbyte> value)
Vector64<float> Not(Vector64<float> value)
Vector64<ushort> Not(Vector64<ushort> value)
Vector64<uint> Not(Vector64<uint> value)
Vector64<ulong> Not(Vector64<ulong> value)
Vector128<byte> Not(Vector128<byte> value)
Vector128<double> Not(Vector128<double> value)
Vector128<short> Not(Vector128<short> value)
Vector128<int> Not(Vector128<int> value)
Vector128<long> Not(Vector128<long> value)
Vector128<sbyte> Not(Vector128<sbyte> value)
Vector128<float> Not(Vector128<float> value)
Vector128<ushort> Not(Vector128<ushort> value)
Vector128<uint> Not(Vector128<uint> value)
Vector128<ulong> Not(Vector128<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:NotTest(System.Runtime.Intrinsics.Vector64`1[Byte]):System.Run
time.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp      fp, lr, [sp,#-16]!
```



```
mov    fp, sp
mvn    v16.8b, v0.8b
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

235. Or

Vector64<byte> Or(Vector64<byte> left, Vector64<byte> right)

This method performs a bitwise OR between the left and right vectors, and returns the result.

```
private Vector64<byte> OrTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.Or(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <31, 30, 31, 30, 31, 26, 27, 30>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<double> Or(Vector64<double> left, Vector64<double> right)
Vector64<short> Or(Vector64<short> left, Vector64<short> right)
Vector64<int> Or(Vector64<int> left, Vector64<int> right)
Vector64<long> Or(Vector64<long> left, Vector64<long> right)
Vector64<sbyte> Or(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> Or(Vector64<float> left, Vector64<float> right)
Vector64<ushort> Or(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> Or(Vector64<uint> left, Vector64<uint> right)
Vector64<ulong> Or(Vector64<ulong> left, Vector64<ulong> right)
Vector128<byte> Or(Vector128<byte> left, Vector128<byte> right)
Vector128<double> Or(Vector128<double> left, Vector128<double> right)
Vector128<short> Or(Vector128<short> left, Vector128<short> right)
Vector128<int> Or(Vector128<int> left, Vector128<int> right)
Vector128<long> Or(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> Or(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> Or(Vector128<float> left, Vector128<float> right)
Vector128<ushort> Or(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> Or(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> Or(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.OrTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    orr    v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

236. OrNot

Vector64<byte> OrNot(Vector64<byte> left, Vector64<byte> right)

This method performs a bitwise OR NOT between the left and right vectors, and returns the result.

```
private Vector64<byte> OrNotTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.OrNot(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <235, 237, 237, 239, 239, 245, 245, 243>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<double> OrNot(Vector64<double> left, Vector64<double> right)
Vector64<short> OrNot(Vector64<short> left, Vector64<short> right)
Vector64<int> OrNot(Vector64<int> left, Vector64<int> right)
Vector64<long> OrNot(Vector64<long> left, Vector64<long> right)
Vector64<sbyte> OrNot(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> OrNot(Vector64<float> left, Vector64<float> right)
Vector64<ushort> OrNot(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> OrNot(Vector64<uint> left, Vector64<uint> right)
Vector64<ulong> OrNot(Vector64<ulong> left, Vector64<ulong> right)
Vector128<byte> OrNot(Vector128<byte> left, Vector128<byte> right)
Vector128<double> OrNot(Vector128<double> left, Vector128<double> right)
Vector128<short> OrNot(Vector128<short> left, Vector128<short> right)
Vector128<int> OrNot(Vector128<int> left, Vector128<int> right)
Vector128<long> OrNot(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> OrNot(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> OrNot(Vector128<float> left, Vector128<float> right)
Vector128<ushort> OrNot(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> OrNot(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> OrNot(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.OrNotTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Ru
ntime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    orn    v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

237. PolynomialMultiply

Vector64<byte> PolynomialMultiply(Vector64<byte> left, Vector64<byte> right)

This method multiplies corresponding elements in the vectors of the left and right vectors, stores the results in a vector and returns the result vector.

```
private Vector64<byte> PolynomialMultiplyTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.PolynomialMultiply(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <151, 232, 243, 144, 135, 160, 171, 248>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<sbyte> PolynomialMultiply(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector128<byte> PolynomialMultiply(Vector128<byte> left, Vector128<byte>
right)
Vector128<sbyte> PolynomialMultiply(Vector128<sbyte> left, Vector128<sbyte>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:PolynomialMultiplyTest(System.Runtime.Intrinsics.Vector64`1[By
te],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vec
tor64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    pmul   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


238. PolynomialMultiplyWideningLower

Vector128<ushort> PolynomialMultiplyWideningLower(Vector64<byte> left, Vector64<byte> right)

This method multiplies corresponding elements in the left and right vectors, stores the results in a vector and returns the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input element's size byte.

```
private Vector128<ushort> PolynomialMultiplyWideningLowerTest(Vector64<byte>
left, Vector64<byte> right)
{
    return AdvSimd.PolynomialMultiplyWideningLower(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <151, 232, 243, 144, 135, 416, 427, 504>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> PolynomialMultiplyWideningLower(Vector64<sbyte> left,
Vector64<sbyte> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:PolynomialMultiplyWideningLowerTest(System.Runtime.Intrinsics.
Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.I
ntrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    pmull  v16.8h, v0.8b, v1.8b
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

239. PolynomialMultiplyWideningUpper

Vector128<ushort> PolynomialMultiplyWideningUpper(Vector128<byte> left, Vector128<byte> right)

This method multiplies corresponding elements in the upper-half of left with corresponding elements of right vectors, stores the results in a vector and returns the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input element's size byte.

```
private Vector128<ushort> PolynomialMultiplyWideningUpperTest(Vector128<byte>
left, Vector128<byte> right)
{
    return AdvSimd.PolynomialMultiplyWideningUpper(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// right = <21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36>
// Result = <503, 408, 403, 704, 759, 816, 779, 808>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> PolynomialMultiplyWideningUpper(Vector128<sbyte> left,
Vector128<sbyte> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:PolynomialMultiplyWideningUpperTest(System.Runtime.Intrinsics.
Vector128`1[Byte],System.Runtime.Intrinsics.Vector128`1[Byte]):System.Runtime
.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    pmull2 v16.8h, v0.16b, v1.16b
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

240. PopCount

Vector64<byte> PopCount(Vector64<byte> value)

This method counts the number of bits that have a value of one in each element in the value vector, stores the results in a vector and returns the result vector.

```
private Vector64<byte> PopCountTest(Vector64<byte> value)
{
    return AdvSimd.PopCount(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <3, 2, 3, 3, 4, 1, 2, 2>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<sbyte> PopCount(Vector64<sbyte> value)
Vector128<byte> PopCount(Vector128<byte> value)
Vector128<sbyte> PopCount(Vector128<sbyte> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.PopCountTest(System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    cnt    v16.8b, v0.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

241. ReciprocalEstimate

Vector64<float> ReciprocalEstimate(Vector64<float> value)

This method finds an approximate reciprocal estimate for each element in the value vector, stores the results in a vector and returns the result vector.

```
private Vector64<float> ReciprocalEstimateTest(Vector64<float> value)
{
    return AdvSimd.ReciprocalEstimate(value);
}
// value = <11.5, 12.5>
// Result = <0.08691406, 0.079833984>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<uint> ReciprocalEstimate(Vector64<uint> value)
Vector128<float> ReciprocalEstimate(Vector128<float> value)
Vector128<uint> ReciprocalEstimate(Vector128<uint> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> ReciprocalEstimate(Vector128<double> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReciprocalEstimateTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frecpe v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

242. ReciprocalEstimateScalar

Vector64<double> ReciprocalEstimateScalar(Vector64<double> value)

This method finds an approximate reciprocal estimate for each element in the value vector, stores the results in a vector and returns the result vector.

```
private Vector64<double> ReciprocalEstimateScalarTest(Vector64<double> value)
{
    return AdvSimd.Arm64.ReciprocalEstimateScalar(value);
}
// value = <11.5>
// Result = <0.0869140625>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<float> ReciprocalEstimateScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReciprocalEstimateScalarTest(System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frecpe d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

243. ReciprocalExponentScalar

Vector64<double> ReciprocalExponentScalar(Vector64<double> value)

This method finds an approximate reciprocal exponent for each element in the value vector, stores the results in a vector and returns the result vector.

```
private Vector64<double> ReciprocalExponentScalarTest(Vector64<double> value)
{
    return AdvSimd.Arm64.ReciprocalExponentScalar(value);
}
// value = <11.5>
// Result = <0.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<float> ReciprocalExponentScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReciprocalExponentScalarTest(System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frecpd  d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

244. ReciprocalSquareRootEstimate

Vector64<float> ReciprocalSquareRootEstimate(Vector64<float> value)

This method calculates an approximate square root for each element in the value vector, stores the results in a vector and returns the result vector.

```
private Vector64<float> ReciprocalSquareRootEstimateTest(Vector64<float>
value)
{
    return AdvSimd.ReciprocalSquareRootEstimate(value);
}
// value = <11.5, 12.5>
// Result = <0.29492188, 0.28222656>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<uint> ReciprocalSquareRootEstimate(Vector64<uint> value)
Vector128<float> ReciprocalSquareRootEstimate(Vector128<float> value)
Vector128<uint> ReciprocalSquareRootEstimate(Vector128<uint> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> ReciprocalSquareRootEstimate(Vector128<double> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReciprocalSquareRootEstimateTest(System.Runtime.Intrinsics.Vec
tor64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frsq rte v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

245. ReciprocalSquareRootEstimateScalar

Vector64<double> ReciprocalSquareRootEstimateScalar(Vector64<double> value)

This method calculates an approximate square root for each element in the value vector, stores the results in a vector and returns the result vector.

```
private Vector64<double>
ReciprocalSquareRootEstimateScalarTest(Vector64<double> value)
{
    return AdvSimd.Arm64.ReciprocalSquareRootEstimateScalar(value);
}
// value = <11.5>
// Result = <0.294921875>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> ReciprocalSquareRootEstimateScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReciprocalSquareRootEstimateScalarTest(System.Runtime.Intrinsi
cs.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frsq rte d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

246. ReciprocalSquareRootStep

Vector64<float> ReciprocalSquareRootStep(Vector64<float> left, Vector64<float> right)

This method multiplies corresponding floating-point values in the left and right vector, subtracts each of the products from 3.0, divides these results by 2.0, stores the results in a vector and returns the result vector.

```
private Vector64<float> ReciprocalSquareRootStepTest(Vector64<float> left,
Vector64<float> right)
{
    return AdvSimd.ReciprocalSquareRootStep(left, right);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// Result = <-122.125, -139.125>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> ReciprocalSquareRootStep(Vector128<float> left,
Vector128<float> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> ReciprocalSquareRootStep(Vector128<double> left,
Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReciprocalSquareRootStepTest(System.Runtime.Intrinsics.Vector64`1[Single],System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frsqrts v16.2s, v0.2s, v1.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```


; Total bytes of code 24, prolog size 8

247. ReciprocalSquareRootStepScalar

Vector64<double> ReciprocalSquareRootStepScalar(Vector64<double> left, Vector64<double> right)

This method multiplies corresponding floating-point values in the vectors of the left and right vectors, subtracts each of the products from 3.0, divides these results by 2.0, stores the results in a vector and returns the result vector.

```
private Vector64<double> ReciprocalSquareRootStepScalarTest(Vector64<double>
left, Vector64<double> right)
{
    return AdvSimd.Arm64.ReciprocalSquareRootStepScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <-64.625>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> ReciprocalSquareRootStepScalar(Vector64<float> left,
Vector64<float> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReciprocalSquareRootStepScalarTest(System.Runtime.Intrinsics.V
ector64`1[Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtim
e.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frsqrts d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

248. ReciprocalStep

Vector64<float> ReciprocalStep(Vector64<float> left, Vector64<float> right)

This method multiplies the corresponding floating-point values in the left and right vectors, subtracts each of the products from 2.0, stores the results in a vector and returns the result vector.

```
private Vector64<float> ReciprocalStepTest(Vector64<float> left,
Vector64<float> right)
{
    return AdvSimd.ReciprocalStep(left, right);
}
// left = <11.5, 12.5>
// right = <21.5, 22.5>
// Result = <-245.25, -279.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> ReciprocalStep(Vector128<float> left, Vector128<float>
right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> ReciprocalStep(Vector128<double> left, Vector128<double>
right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReciprocalStepTest(System.Runtime.Intrinsics.Vector64`1[Single]
,System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vec
tor64`1[Single]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frecps v16.2s, v0.2s, v1.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


249. ReciprocalStepScalar

Vector64<double> ReciprocalStepScalar(Vector64<double> left, Vector64<double> right)

This method multiplies the corresponding floating-point values in the left and right vectors, subtracts each of the products from 2.0, stores the results in a vector and returns the result vector.

```
private Vector64<double> ReciprocalStepScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.Arm64.ReciprocalStepScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <-130.25>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<float> ReciprocalStepScalar(Vector64<float> left, Vector64<float>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReciprocalStepScalarTest(System.Runtime.Intrinsics.Vector64`1[
Double],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsi
cs.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frecps d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

250. ReverseElement16

Vector64<int> ReverseElement16(Vector64<int> value)

Reverse bytes in each 32-bits value and returns the result.

```
private Vector64<int> ReverseElement16Test(Vector64<int> value)
{
    return AdvSimd.ReverseElement16(value);
}
// value = <11, 12>
// Result = <720896, 786432>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<long> ReverseElement16(Vector64<long> value)
Vector64<uint> ReverseElement16(Vector64<uint> value)
Vector64<ulong> ReverseElement16(Vector64<ulong> value)
Vector128<int> ReverseElement16(Vector128<int> value)
Vector128<long> ReverseElement16(Vector128<long> value)
Vector128<uint> ReverseElement16(Vector128<uint> value)
Vector128<ulong> ReverseElement16(Vector128<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ReverseElement16Test(System.Runtime.Intrinsics.Vector64`1[Int32]):System.Runtime.Intrinsics.Vector64`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    rev32   v16.4h, v0.4h
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

251. ReverseElement32

Vector64<long> ReverseElement32(Vector64<long> value)

Reverse bytes in each 64-bits value and returns the result.

```
private Vector64<long> ReverseElement32Test(Vector64<long> value)
{
    return AdvSimd.ReverseElement32(value);
}
// value = <11>
// Result = <47244640256>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ReverseElement32(Vector64<ulong> value)
Vector128<long> ReverseElement32(Vector128<long> value)
Vector128<ulong> ReverseElement32(Vector128<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ReverseElement32Test(System.Runtime.Intrinsics.Vector64`1[Int6
4]):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    rev64   v16.2s, v0.2s
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

252. ReverseElement8

Vector64<short> ReverseElement8(Vector64<short> value)

Reverse bytes in each 16-bit half word values and returns the result.

```
private Vector64<short> ReverseElement8Test(Vector64<short> value)
{
    return AdvSimd.ReverseElement8(value);
}
// value = <11, 12, 13, 14>
// Result = <2816, 3072, 3328, 3584>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ReverseElement8(Vector64<int> value)
Vector64<long> ReverseElement8(Vector64<long> value)
Vector64<ushort> ReverseElement8(Vector64<ushort> value)
Vector64<uint> ReverseElement8(Vector64<uint> value)
Vector64<ulong> ReverseElement8(Vector64<ulong> value)
Vector128<short> ReverseElement8(Vector128<short> value)
Vector128<int> ReverseElement8(Vector128<int> value)
Vector128<long> ReverseElement8(Vector128<long> value)
Vector128<ushort> ReverseElement8(Vector128<ushort> value)
Vector128<uint> ReverseElement8(Vector128<uint> value)
Vector128<ulong> ReverseElement8(Vector128<ulong> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReverseElement8Test(System.Runtime.Intrinsics.Vector64`1[Int16]
):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    rev16   v16.8b, v0.8b
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

253. ReverseElementBits

Vector64<byte> ReverseElementBits(Vector64<byte> value)

Reverses the bit order of all elements in value vector.

```
private Vector64<byte> ReverseElementBitsTest(Vector64<byte> value)
{
    return AdvSimd.Arm64.ReverseElementBits(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <208, 48, 176, 112, 240, 8, 136, 72>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<sbyte> ReverseElementBits(Vector64<sbyte> value)
Vector128<byte> ReverseElementBits(Vector128<byte> value)
Vector128<sbyte> ReverseElementBits(Vector128<sbyte> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ReverseElementBitsTest(System.Runtime.Intrinsics.Vector64`1[Byte
te]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    rbit   v16.8b, v0.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

254. RoundAwayFromZero

Vector64<float> RoundAwayFromZero(Vector64<float> value)

This method rounds a vector of floating-point values in the value vector to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<float> RoundAwayFromZeroTest(Vector64<float> value)
{
    return AdvSimd.RoundAwayFromZero(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> RoundAwayFromZero(Vector128<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> RoundAwayFromZero(Vector128<double> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:RoundAwayFromZeroTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frinta v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

255. RoundAwayFromZeroScalar

Vector64<double> RoundAwayFromZeroScalar(Vector64<double> value)

This method rounds a floating-point value in the value vector to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and returns the result. As per ARM docs, zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<double> RoundAwayFromZeroScalarTest(Vector64<double> value)
{
    return AdvSimd.RoundAwayFromZeroScalar(value);
}
// value = <11.5>
// Result = <12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> RoundAwayFromZeroScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:RoundAwayFromZeroScalarTest(System.Runtime.Intrinsics.Vector64
`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frinta d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

256. RoundToNearest

Vector64<float> RoundToNearest(Vector64<float> value)

This method rounds a vector of floating-point values in the value vector to integral floating-point values of the same size using the Round to Nearest rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<float> RoundToNearestTest(Vector64<float> value)
{
    return AdvSimd.RoundToNearest(value);
}
// value = <11.4, 12.8>
// Result = <11, 13>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> RoundToNearest(Vector128<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> RoundToNearest(Vector128<double> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:RoundToNearestTest(System.Runtime.Intrinsics.Vector64`1[Single]
):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintn v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

257. RoundToNearestScalar

Vector64<double> RoundToNearestScalar(Vector64<double> value)

This method rounds a vector of floating-point values in the value vector to integral floating-point values of the same size using the Round to Nearest rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<double> RoundToNearestScalarTest(Vector64<double> value)
{
    return AdvSimd.RoundToNearestScalar(value);
}
// value = <11.4>
// Result = <11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> RoundToNearestScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:RoundToNearestScalarTest(System.Runtime.Intrinsics.Vector64`1[
Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintn d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

258. RoundToNegativeInfinity

Vector64<float> RoundToNegativeInfinity(Vector64<float> value)

This method rounds a floating-point value in the value vector to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<float> RoundToNegativeInfinityTest(Vector64<float> value)
{
    return AdvSimd.RoundToNegativeInfinity(value);
}
// value = <11.5, 12.5>
// Result = <11, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> RoundToNegativeInfinity(Vector128<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> RoundToNegativeInfinity(Vector128<double> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:RoundToNegativeInfinityTest(System.Runtime.Intrinsics.Vector64
`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintm v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

259. RoundToNegativeInfinityScalar

Vector64<double> RoundToNegativeInfinityScalar(Vector64<double> value)

This method rounds a floating-point value in the value vector to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<double> RoundToNegativeInfinityScalarTest(Vector64<double>
value)
{
    return AdvSimd.RoundToNegativeInfinityScalar(value);
}
// value = <11.5>
// Result = <11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> RoundToNegativeInfinityScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:RoundToNegativeInfinityScalarTest(System.Runtime.Intrinsics.Ve
ctor64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintm d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

260. RoundToPositiveInfinity

Vector64<float> RoundToPositiveInfinity(Vector64<float> value)

This method rounds a floating-point value in the value vector to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<float> RoundToPositiveInfinityTest(Vector64<float> value)
{
    return AdvSimd.RoundToPositiveInfinity(value);
}
// value = <11.5, 12.5>
// Result = <12, 13>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> RoundToPositiveInfinity(Vector128<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> RoundToPositiveInfinity(Vector128<double> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:RoundToPositiveInfinityTest(System.Runtime.Intrinsics.Vector64
`1[Single]):System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintp v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

261. RoundToPositiveInfinityScalar

Vector64<double> RoundToPositiveInfinityScalar(Vector64<double> value)

This method rounds a floating-point value in the value vector to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<double> RoundToPositiveInfinityScalarTest(Vector64<double>
value)
{
    return AdvSimd.RoundToPositiveInfinityScalar(value);
}
// value = <11.5>
// Result = <12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> RoundToPositiveInfinityScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:RoundToPositiveInfinityScalarTest(System.Runtime.Intrinsics.Ve
ctor64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintp d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

262. RoundToZero

Vector64<float> RoundToZero(Vector64<float> value)

This method rounds a vector of floating-point values in the value vector to integral floating-point values of the same size using the Round towards Zero rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<float> RoundToZeroTest(Vector64<float> value)
{
    return AdvSimd.RoundToZero(value);
}
// value = <11.4, 12.8>
// Result = <11, 12>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<float> RoundToZero(Vector128<float> value)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> RoundToZero(Vector128<double> value)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.RoundToZeroTest(System.Runtime.Intrinsics.Vector64`1[Single]):
System.Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintz v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

263. RoundToZeroScalar

Vector64<double> RoundToZeroScalar(Vector64<double> value)

This method rounds a vector of floating-point values in the value vector to integral floating-point values of the same size using the Round towards Zero rounding mode, and returns the result. As per ARM docs, a zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

```
private Vector64<double> RoundToZeroScalarTest(Vector64<double> value)
{
    return AdvSimd.RoundToZeroScalar(value);
}
// value = <11.4>
// Result = <11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> RoundToZeroScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:RoundToZeroScalarTest(System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    frintz d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

264. ShiftArithmetic

Vector64<short> ShiftArithmetic(Vector64<short> value, Vector64<short> count)

This method performs arithmetic shifts of each signed integer value in the value vector, by the value in corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

```
private Vector64<short> ShiftArithmeticTest(Vector64<short> value,
Vector64<short> count)
{
    return AdvSimd.ShiftArithmetic(value, count);
}
// value = <11, 12, 13, 14>
// count = <18, 2, 3, -2>
// Result = <0, 48, 104, 3>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ShiftArithmetic(Vector64<int> value, Vector64<int> count)
Vector64<sbyte> ShiftArithmetic(Vector64<sbyte> value, Vector64<sbyte> count)
Vector128<short> ShiftArithmetic(Vector128<short> value, Vector128<short>
count)
Vector128<int> ShiftArithmetic(Vector128<int> value, Vector128<int> count)
Vector128<long> ShiftArithmetic(Vector128<long> value, Vector128<long> count)
Vector128<sbyte> ShiftArithmetic(Vector128<sbyte> value, Vector128<sbyte>
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftArithmeticTest(System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sshl   v16.4h, v0.4h, v1.4h
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

265. ShiftArithmeticRounded

Vector64<short> ShiftArithmeticRounded(Vector64<short> value, Vector64<short> count)

This method performs arithmetic shift of each signed integer value in the value vector, by a value in corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

```
private Vector64<short> ShiftArithmeticRoundedTest(Vector64<short> value,
Vector64<short> count)
{
    return AdvSimd.ShiftArithmeticRounded(value, count);
}
// value = <11, 12, 13, 14>
// count = <18, 2, 3, -2>
// Result = <0, 48, 104, 4>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ShiftArithmeticRounded(Vector64<int> value, Vector64<int>
count)
Vector64<sbyte> ShiftArithmeticRounded(Vector64<sbyte> value, Vector64<sbyte>
count)
Vector128<short> ShiftArithmeticRounded(Vector128<short> value,
Vector128<short> count)
Vector128<int> ShiftArithmeticRounded(Vector128<int> value, Vector128<int>
count)
Vector128<long> ShiftArithmeticRounded(Vector128<long> value, Vector128<long>
count)
Vector128<sbyte> ShiftArithmeticRounded(Vector128<sbyte> value,
Vector128<sbyte> count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftArithmeticRoundedTest(System.Runtime.Intrinsics.Vector64`
1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrinsi
cs.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    srshl  v16.4h, v0.4h, v1.4h  
    mov    v0.8b, v16.8b  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 24, prolog size 8

266. ShiftArithmeticRoundedSaturate

Vector64<short> ShiftArithmeticRoundedSaturate(Vector64<short> value, Vector64<short> count)

This method performs arithmetic shift of each vector element in the value vector, by a value of the corresponding vector element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded.

```
private Vector64<short> ShiftArithmeticRoundedSaturateTest(Vector64<short>
value, Vector64<short> count)
{
    return AdvSimd.ShiftArithmeticRoundedSaturate(value, count);
}
// value = <11, 12, 13, 14>
// count = <18, 2, 3, -2>
// Result = <32767, 48, 104, 4>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ShiftArithmeticRoundedSaturate(Vector64<int> value,
Vector64<int> count)
Vector64<sbyte> ShiftArithmeticRoundedSaturate(Vector64<sbyte> value,
Vector64<sbyte> count)
Vector128<short> ShiftArithmeticRoundedSaturate(Vector128<short> value,
Vector128<short> count)
Vector128<int> ShiftArithmeticRoundedSaturate(Vector128<int> value,
Vector128<int> count)
Vector128<long> ShiftArithmeticRoundedSaturate(Vector128<long> value,
Vector128<long> count)
Vector128<sbyte> ShiftArithmeticRoundedSaturate(Vector128<sbyte> value,
Vector128<sbyte> count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftArithmeticRoundedSaturateTest(System.Runtime.Intrinsics.V
ector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.
Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
```



```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    sqrrshl v16.4h, v0.4h, v1.4h  
    mov    v0.8b, v16.8b  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 24, prolog size 8

267. ShiftArithmeticRoundedSaturateScalar

Vector64<long> ShiftArithmeticRoundedSaturateScalar(Vector64<long> value, Vector64<long> count)

This method performs arithmetic shift of 0th element in the value vector, by a value of the corresponding 0th element in the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded.

```
private Vector64<long>
ShiftArithmeticRoundedSaturateScalarTest(Vector64<long> value, Vector64<long>
count)
{
    return AdvSimd.ShiftArithmeticRoundedSaturateScalar(value, count);
}
// value = <11>
// count = <11>
// Result = <22528>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<short> ShiftArithmeticRoundedSaturateScalar(Vector64<short> value,
Vector64<short> count)
Vector64<int> ShiftArithmeticRoundedSaturateScalar(Vector64<int> value,
Vector64<int> count)
Vector64<sbyte> ShiftArithmeticRoundedSaturateScalar(Vector64<sbyte> value,
Vector64<sbyte> count)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftArithmeticRoundedSaturateScalarTest(System.Runtime.Intrin
sics.Vector64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Ru
ntime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqrshl d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

268. ShiftArithmeticRoundedScalar

**Vector64<long> ShiftArithmeticRoundedScalar(Vector64<long> value,
Vector64<long> count)**

This method performs arithmetic shift of each signed integer value in the value vector, by a value of the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

```
private Vector64<long> ShiftArithmeticRoundedScalarTest(Vector64<long> value,  
Vector64<long> count)  
{  
    return AdvSimd.ShiftArithmeticRoundedScalar(value, count);  
}  
// value = <11>  
// count = <11>  
// Result = <22528>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods:ShiftArithmeticRoundedScalarTest(System.Runtime.Intrinsics.Vec  
tor64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Runtime.In  
trinsics.Vector64`1[Int64]  
;  
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0  
HFA(simd8)  
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1  
HFA(simd8)  
;# V02 OutArgs  [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]  
"OutgoingArgSpace"  
; Lcl frame size = 0  
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    srshl  d16, d0, d1  
    mov    v0.8b, v16.8b  
    ldp    fp, lr, [sp],#16  
    ret    lr  
  
; Total bytes of code 24, prolog size 8
```

269. ShiftArithmeticSaturate

Vector64<short> ShiftArithmeticSaturate(Vector64<short> value, Vector64<short> count)

This method performs arithmetic shift of each element in the value vector, by a value of the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated.

```
private Vector64<short> ShiftArithmeticSaturateTest(Vector64<short> value,
Vector64<short> count)
{
    return AdvSimd.ShiftArithmeticSaturate(value, count);
}
// value = <11, 12, 13, 14>
// count = <21, 22, 23, 24>
// Result = <32767, 32767, 32767, 32767>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ShiftArithmeticSaturate(Vector64<int> value, Vector64<int>
count)
Vector64<sbyte> ShiftArithmeticSaturate(Vector64<sbyte> value,
Vector64<sbyte> count)
Vector128<short> ShiftArithmeticSaturate(Vector128<short> value,
Vector128<short> count)
Vector128<int> ShiftArithmeticSaturate(Vector128<int> value, Vector128<int>
count)
Vector128<long> ShiftArithmeticSaturate(Vector128<long> value,
Vector128<long> count)
Vector128<sbyte> ShiftArithmeticSaturate(Vector128<sbyte> value,
Vector128<sbyte> count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftArithmeticSaturateTest(System.Runtime.Intrinsics.Vector64
`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16]):System.Runtime.Intrins
ics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    sqshl  v16.4h, v0.4h, v1.4h  
    mov    v0.8b, v16.8b  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 24, prolog size 8

270. ShiftArithmeticSaturateScalar

Vector64<long> ShiftArithmeticSaturateScalar(Vector64<long> value, Vector64<long> count)

This method performs arithmetic shift of each element in the value vector, by a value of the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated.

```
private Vector64<long> ShiftArithmeticSaturateScalarTest(Vector64<long>
value, Vector64<long> count)
{
    return AdvSimd.ShiftArithmeticSaturateScalar(value, count);
}
// value = <11>
// count = <11>
// Result = <22528>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<short> ShiftArithmeticSaturateScalar(Vector64<short> value,
Vector64<short> count)
Vector64<int> ShiftArithmeticSaturateScalar(Vector64<int> value,
Vector64<int> count)
Vector64<sbyte> ShiftArithmeticSaturateScalar(Vector64<sbyte> value,
Vector64<sbyte> count)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftArithmeticSaturateScalarTest(System.Runtime.Intrinsics.Ve
ctor64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Runtime.I
ntrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqshl  d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

271. ShiftArithmeticScalar

Vector64<long> ShiftArithmeticScalar(Vector64<long> value, Vector64<long> count)

This method performs arithmetic shift of each signed integer value in the value vector, by a value of the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

```
private Vector64<long> ShiftArithmeticScalarTest(Vector64<long> value,
Vector64<long> count)
{
    return AdvSimd.ShiftArithmeticScalar(value, count);
}
// value = <11>
// count = <11>
// Result = <22528>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftArithmeticScalarTest(System.Runtime.Intrinsics.Vector64`1
[Int64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Runtime.Intrinsic
s.Vector64`1[Int64]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sshl   d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

272. ShiftLeftAndInsert

Vector64<byte> ShiftLeftAndInsert(Vector64<byte> left, Vector64<byte> right, byte shift)

This method left shifts each vector element in the right vector, by shift value, and inserts the result into the corresponding vector element in the left vector such that the new zero bits created by the shift are not inserted but retain their existing value as in left vector. Bits shifted out of the left of each vector element in the right are lost.

```
private Vector64<byte> ShiftLeftAndInsertTest(Vector64<byte> left,
Vector64<byte> right, byte shift)
{
    return AdvSimd.ShiftLeftAndInsert(left, right, 1);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <1, 2, 3, 4, 5, 6, 7, 8>
// shift = 1
// Result = <3, 4, 7, 8, 11, 12, 15, 16>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftLeftAndInsert(Vector64<short> left, Vector64<short>
right, byte shift)
Vector64<int> ShiftLeftAndInsert(Vector64<int> left, Vector64<int> right,
byte shift)
Vector64<sbyte> ShiftLeftAndInsert(Vector64<sbyte> left, Vector64<sbyte>
right, byte shift)
Vector64<ushort> ShiftLeftAndInsert(Vector64<ushort> left, Vector64<ushort>
right, byte shift)
Vector64<uint> ShiftLeftAndInsert(Vector64<uint> left, Vector64<uint> right,
byte shift)
Vector128<byte> ShiftLeftAndInsert(Vector128<byte> left, Vector128<byte>
right, byte shift)
Vector128<short> ShiftLeftAndInsert(Vector128<short> left, Vector128<short>
right, byte shift)
Vector128<int> ShiftLeftAndInsert(Vector128<int> left, Vector128<int> right,
byte shift)
Vector128<long> ShiftLeftAndInsert(Vector128<long> left, Vector128<long>
right, byte shift)
Vector128<sbyte> ShiftLeftAndInsert(Vector128<sbyte> left, Vector128<sbyte>
right, byte shift)
Vector128<ushort> ShiftLeftAndInsert(Vector128<ushort> left,
Vector128<ushort> right, byte shift)
Vector128<uint> ShiftLeftAndInsert(Vector128<uint> left, Vector128<uint>
right, byte shift)
Vector128<ulong> ShiftLeftAndInsert(Vector128<ulong> left, Vector128<ulong>
right, byte shift)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftAndInsertTest(System.Runtime.Intrinsics.Vector64`1[Byte]
, System.Runtime.Intrinsics.Vector64`1[Byte], ubyte):System.Runtime.Intrinsi
cs.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02  ] ( 0, 0 )   ubyte  ->  zero-ref
;# V03 OutArgs       [V03  ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sli    v0.8b, v1.8b, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

273. ShiftLeftAndInsertScalar

Vector64<long> ShiftLeftAndInsertScalar(Vector64<long> left, Vector64<long> right, byte shift)

This method left shifts each vector element in the right vector, by shift value, and inserts the result into the corresponding vector element in the left vector such that the new zero bits created by the shift are not inserted but retain their existing value as in left vector. Bits shifted out of the left of each vector element in the right are lost.

```
private Vector64<long> ShiftLeftAndInsertScalarTest(Vector64<long> left,
Vector64<long> right, byte shift)
{
    return AdvSimd.ShiftLeftAndInsertScalar(left, right, 1);
}
// left = <50000>
// right = <60000>
// shift = 1
// Result = <120000>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftLeftAndInsertScalar(Vector64<ulong> left,
Vector64<ulong> right, byte shift)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftAndInsertScalarTest(System.Runtime.Intrinsics.Vector64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64],ubyte):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02  ] ( 0, 0 )  ubyte   ->  zero-ref
;# V03 OutArgs       [V03  ] ( 1, 1 )  lclBlk  ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sli    d0, d1, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```


274. ShiftLeftLogical

Vector64<byte> ShiftLeftLogical(Vector64<byte> value, byte count)

This method left shifts each value from a vector, by count, stores the results in a vector and returns the result vector.

```
private Vector64<byte> ShiftLeftLogicalTest(Vector64<byte> value, byte count)
{
    return AdvSimd.ShiftLeftLogical(value, 1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <22, 24, 26, 28, 30, 32, 34, 36>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftLeftLogical(Vector64<short> value, byte count)
Vector64<int> ShiftLeftLogical(Vector64<int> value, byte count)
Vector64<sbyte> ShiftLeftLogical(Vector64<sbyte> value, byte count)
Vector64<ushort> ShiftLeftLogical(Vector64<ushort> value, byte count)
Vector64<uint> ShiftLeftLogical(Vector64<uint> value, byte count)
Vector128<byte> ShiftLeftLogical(Vector128<byte> value, byte count)
Vector128<short> ShiftLeftLogical(Vector128<short> value, byte count)
Vector128<long> ShiftLeftLogical(Vector128<long> value, byte count)
Vector128<sbyte> ShiftLeftLogical(Vector128<sbyte> value, byte count)
Vector128<ushort> ShiftLeftLogical(Vector128<ushort> value, byte count)
Vector128<uint> ShiftLeftLogical(Vector128<uint> value, byte count)
Vector128<ulong> ShiftLeftLogical(Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftLogicalTest(System.Runtime.Intrinsics.Vector64`1[Byte
],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    shl    v16.8b, v0.8b, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

275. ShiftLeftLogicalSaturate

Vector64<byte> ShiftLeftLogicalSaturate(Vector64<byte> value, byte count)

This method left shifts each element in the value vector, shifts it by count, stores the results in a vector and returns the result vector. The results are truncated.

```
private Vector64<byte> ShiftLeftLogicalSaturateTest(Vector64<byte> value,
byte count)
{
    return AdvSimd.ShiftLeftLogicalSaturate(value, 6);
}
// value = <11, 112, 13, 14, 15, 16, 17, 18>
// count = 6
// Result = <64, 255, 255, 255, 255, 255, 255, 255>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftLeftLogicalSaturate(Vector64<short> value, byte count)
Vector64<int> ShiftLeftLogicalSaturate(Vector64<int> value, byte count)
Vector64<sbyte> ShiftLeftLogicalSaturate(Vector64<sbyte> value, byte count)
Vector64<ushort> ShiftLeftLogicalSaturate(Vector64<ushort> value, byte count)
Vector64<uint> ShiftLeftLogicalSaturate(Vector64<uint> value, byte count)
Vector128<byte> ShiftLeftLogicalSaturate(Vector128<byte> value, byte count)
Vector128<short> ShiftLeftLogicalSaturate(Vector128<short> value, byte count)
Vector128<int> ShiftLeftLogicalSaturate(Vector128<int> value, byte count)
Vector128<long> ShiftLeftLogicalSaturate(Vector128<long> value, byte count)
Vector128<sbyte> ShiftLeftLogicalSaturate(Vector128<sbyte> value, byte count)
Vector128<ushort> ShiftLeftLogicalSaturate(Vector128<ushort> value, byte
count)
Vector128<uint> ShiftLeftLogicalSaturate(Vector128<uint> value, byte count)
Vector128<ulong> ShiftLeftLogicalSaturate(Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftLogicalSaturateTest(System.Runtime.Intrinsics.Vector64`1[Byte],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01  ] ( 0, 0 )  ubyte  ->  zero-ref
;# V02 OutArgs       [V02  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqshl   v16.8b, v0.8b, #6
```



```
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

276. ShiftLeftLogicalSaturateScalar

Vector64<long> ShiftLeftLogicalSaturateScalar(Vector64<long> value, byte count)

This method left shift each element in the value vector, by count, stores the results in a vector and returns the result vector. The results are truncated.

```
private Vector64<long> ShiftLeftLogicalSaturateScalarTest(Vector64<long>
value, byte count)
{
    return AdvSimd.ShiftLeftLogicalSaturateScalar(value, 0);
}
// value = <11>
// count = 0
// Result = <11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftLeftLogicalSaturateScalar(Vector64<ulong> value, byte
count)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<byte> ShiftLeftLogicalSaturateScalar(Vector64<byte> value, byte
count)
Vector64<short> ShiftLeftLogicalSaturateScalar(Vector64<short> value, byte
count)
Vector64<int> ShiftLeftLogicalSaturateScalar(Vector64<int> value, byte count)
Vector64<sbyte> ShiftLeftLogicalSaturateScalar(Vector64<sbyte> value, byte
count)
Vector64<ushort> ShiftLeftLogicalSaturateScalar(Vector64<ushort> value, byte
count)
Vector64<uint> ShiftLeftLogicalSaturateScalar(Vector64<uint> value, byte
count)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftLogicalSaturateScalarTest(System.Runtime.Intrinsics.V
ector64`1[Int64],ubyte):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
                stp    fp, lr, [sp,#-16]!
```

```
mov    fp, sp
sqshl  d16, d0, #0
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

277. ShiftLeftLogicalSaturateUnsigned

Vector64<ushort> ShiftLeftLogicalSaturateUnsigned(Vector64<short> value, byte count)

This method left shifts each signed integer value in the value vector, by count, saturates the shifted result to an unsigned integer value, stores the results in a vector and returns the result vector. The results are truncated.

```
private Vector64<ushort> ShiftLeftLogicalSaturateUnsignedTest(Vector64<short>
value, byte count)
{
    return AdvSimd.ShiftLeftLogicalSaturateUnsigned(value, 1);
}
// value = <11, 12, 13, 14>
// count = 1
// Result = <22, 24, 26, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<uint> ShiftLeftLogicalSaturateUnsigned(Vector64<int> value, byte
count)
Vector64<byte> ShiftLeftLogicalSaturateUnsigned(Vector64<sbyte> value, byte
count)
Vector128<ushort> ShiftLeftLogicalSaturateUnsigned(Vector128<short> value,
byte count)
Vector128<uint> ShiftLeftLogicalSaturateUnsigned(Vector128<int> value, byte
count)
Vector128<ulong> ShiftLeftLogicalSaturateUnsigned(Vector128<long> value, byte
count)
Vector128<byte> ShiftLeftLogicalSaturateUnsigned(Vector128<sbyte> value, byte
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftLogicalSaturateUnsignedTest(System.Runtime.Intrinsics
.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector64`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01  ] ( 0, 0 )  ubyte  ->  zero-ref
;# V02 OutArgs       [V02  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqshlu v16.4h, v0.4h, #1
```

```
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

278. ShiftLeftLogicalSaturateUnsignedScalar

Vector64<ulong> ShiftLeftLogicalSaturateUnsignedScalar(Vector64<long> value, byte count)

This method shifts signed integer value in the value vector, by count, saturates the shifted result to an unsigned integer value, stores the results in a vector and returns the result vector. The results are truncated.

```
private Vector64<ulong>
ShiftLeftLogicalSaturateUnsignedScalarTest(Vector64<long> value, byte count)
{
    return AdvSimd.ShiftLeftLogicalSaturateUnsignedScalar(value, 0);
}
// value = <11>
// count = 0
// Result = <11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<ushort> ShiftLeftLogicalSaturateUnsignedScalar(Vector64<short>
value, byte count)
Vector64<uint> ShiftLeftLogicalSaturateUnsignedScalar(Vector64<int> value,
byte count)
Vector64<byte> ShiftLeftLogicalSaturateUnsignedScalar(Vector64<sbyte> value,
byte count)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftLogicalSaturateUnsignedScalarTest(System.Runtime.Intr
insics.Vector64`1[Int64],ubyte):System.Runtime.Intrinsics.Vector64`1[UInt64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqshlu d16, d0, #0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```


279. ShiftLeftLogicalScalar

Vector64<long> ShiftLeftLogicalScalar(Vector64<long> value, byte count)

This method left shifts each value in value vector, by count, stores the results in a vector and returns the result vector.

```
private Vector64<long> ShiftLeftLogicalScalarTest(Vector64<long> value, byte
count)
{
    return AdvSimd.ShiftLeftLogicalScalar(value, 1);
}
// value = <971324>
// count = 1
// Result = <1942648>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftLeftLogicalScalar(Vector64<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftLogicalScalarTest(System.Runtime.Intrinsics.Vector64`
1[Int64],ubyte):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01  ] ( 0, 0 )  ubyte  ->  zero-ref
;# V02 OutArgs       [V02  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    shl    d16, d0, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

280. ShiftLeftLogicalWideningLower

Vector128<ushort> ShiftLeftLogicalWideningLower(Vector64<byte> value, byte count)

This method left shifts each vector element in the value vector, by the specified number of bits in count, stores the results in a vector and returns the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input vector element's size byte.

```
private Vector128<ushort> ShiftLeftLogicalWideningLowerTest(Vector64<byte>
value, byte count)
{
    return AdvSimd.ShiftLeftLogicalWideningLower(value, 0);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 0
// Result = <11, 12, 13, 14, 15, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ShiftLeftLogicalWideningLower(Vector64<short> value, byte
count)
Vector128<long> ShiftLeftLogicalWideningLower(Vector64<int> value, byte
count)
Vector128<short> ShiftLeftLogicalWideningLower(Vector64<sbyte> value, byte
count)
Vector128<uint> ShiftLeftLogicalWideningLower(Vector64<ushort> value, byte
count)
Vector128<ulong> ShiftLeftLogicalWideningLower(Vector64<uint> value, byte
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftLogicalWideningLowerTest(System.Runtime.Intrinsics.Ve
ctor64`1[Byte],ubyte):System.Runtime.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ushll  v16.8h, v0.8b, #0
    mov    v0.16b, v16.16b
```

```
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

281. ShiftLeftLogicalWideningUpper

Vector128<ushort> ShiftLeftLogicalWideningUpper(Vector128<byte> value, byte count)

This method shifts each vector element in the upper-half of value vector, by the specified number of bits in count, stores the results in a vector and returns the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input vector element's size byte.

```
private Vector128<ushort> ShiftLeftLogicalWideningUpperTest(Vector128<byte>
value, byte count)
{
    return AdvSimd.ShiftLeftLogicalWideningUpper(value, 1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// count = 1
// Result = <38, 40, 42, 44, 46, 48, 50, 52>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ShiftLeftLogicalWideningUpper(Vector128<short> value, byte
count)
Vector128<long> ShiftLeftLogicalWideningUpper(Vector128<int> value, byte
count)
Vector128<short> ShiftLeftLogicalWideningUpper(Vector128<sbyte> value, byte
count)
Vector128<uint> ShiftLeftLogicalWideningUpper(Vector128<ushort> value, byte
count)
Vector128<ulong> ShiftLeftLogicalWideningUpper(Vector128<uint> value, byte
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLeftLogicalWideningUpperTest(System.Runtime.Intrinsics.Ve
ctor128`1[Byte],ubyte):System.Runtime.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ushl12 v16.8h, v0.16b, #1
    mov    v0.16b, v16.16b
```

```
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 24, prolog size 8

282. ShiftLogical

Vector64<byte> ShiftLogical(Vector64<byte> value, Vector64<sbyte> count)

This method shifts each element in the value vector, by the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

```
private Vector64<byte> ShiftLogicalTest(Vector64<byte> value, Vector64<sbyte> count)
{
    return AdvSimd.ShiftLogical(value, count);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = <-3, 2, 3, 5, 6, 7, -7, 0>
// Result = <1, 48, 104, 192, 192, 0, 0, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftLogical(Vector64<short> value, Vector64<short> count)
Vector64<int> ShiftLogical(Vector64<int> value, Vector64<int> count)
Vector64<sbyte> ShiftLogical(Vector64<sbyte> value, Vector64<sbyte> count)
Vector64<ushort> ShiftLogical(Vector64<ushort> value, Vector64<short> count)
Vector64<uint> ShiftLogical(Vector64<uint> value, Vector64<int> count)
Vector128<byte> ShiftLogical(Vector128<byte> value, Vector128<sbyte> count)
Vector128<short> ShiftLogical(Vector128<short> value, Vector128<short> count)
Vector128<int> ShiftLogical(Vector128<int> value, Vector128<int> count)
Vector128<long> ShiftLogical(Vector128<long> value, Vector128<long> count)
Vector128<sbyte> ShiftLogical(Vector128<sbyte> value, Vector128<sbyte> count)
Vector128<ushort> ShiftLogical(Vector128<ushort> value, Vector128<short> count)
Vector128<uint> ShiftLogical(Vector128<uint> value, Vector128<int> count)
Vector128<ulong> ShiftLogical(Vector128<ulong> value, Vector128<long> count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLogicalTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[SByte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ushl   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

283. ShiftLogicalRounded

Vector64<byte> ShiftLogicalRounded(Vector64<byte> value, Vector64<sbyte> count)

This method shifts each element in the value vector, by the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

```
private Vector64<byte> ShiftLogicalRoundedTest(Vector64<byte> value,
Vector64<sbyte> count)
{
    return AdvSimd.ShiftLogicalRounded(value, count);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = <-3, 2, 3, 5, 6, 7, -7, 0>
// Result = <1, 48, 104, 192, 192, 0, 0, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftLogicalRounded(Vector64<short> value, Vector64<short>
count)
Vector64<int> ShiftLogicalRounded(Vector64<int> value, Vector64<int> count)
Vector64<sbyte> ShiftLogicalRounded(Vector64<sbyte> value, Vector64<sbyte>
count)
Vector64<ushort> ShiftLogicalRounded(Vector64<ushort> value, Vector64<short>
count)
Vector64<uint> ShiftLogicalRounded(Vector64<uint> value, Vector64<int> count)
Vector128<byte> ShiftLogicalRounded(Vector128<byte> value, Vector128<sbyte>
count)
Vector128<short> ShiftLogicalRounded(Vector128<short> value, Vector128<short>
count)
Vector128<int> ShiftLogicalRounded(Vector128<int> value, Vector128<int>
count)
Vector128<long> ShiftLogicalRounded(Vector128<long> value, Vector128<long>
count)
Vector128<sbyte> ShiftLogicalRounded(Vector128<sbyte> value, Vector128<sbyte>
count)
Vector128<ushort> ShiftLogicalRounded(Vector128<ushort> value,
Vector128<short> count)
Vector128<uint> ShiftLogicalRounded(Vector128<uint> value, Vector128<int>
count)
Vector128<ulong> ShiftLogicalRounded(Vector128<ulong> value, Vector128<long>
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:ShiftLogicalRoundedTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[SByte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    urshl   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8

```

284. ShiftLogicalRoundedSaturate

Vector64<byte> ShiftLogicalRoundedSaturate(Vector64<byte> value, Vector64<sbyte> count)

This method shifts each vector element of the value vector, by the corresponding vector element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<byte> ShiftLogicalRoundedSaturateTest(Vector64<byte> value,
Vector64<sbyte> count)
{
    return AdvSimd.ShiftLogicalRoundedSaturate(value, count);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <255, 255, 255, 255, 255, 255, 255, 255>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftLogicalRoundedSaturate(Vector64<short> value,
Vector64<short> count)
Vector64<int> ShiftLogicalRoundedSaturate(Vector64<int> value, Vector64<int>
count)
Vector64<sbyte> ShiftLogicalRoundedSaturate(Vector64<sbyte> value,
Vector64<sbyte> count)
Vector64<ushort> ShiftLogicalRoundedSaturate(Vector64<ushort> value,
Vector64<short> count)
Vector64<uint> ShiftLogicalRoundedSaturate(Vector64<uint> value,
Vector64<int> count)
Vector128<byte> ShiftLogicalRoundedSaturate(Vector128<byte> value,
Vector128<sbyte> count)
Vector128<short> ShiftLogicalRoundedSaturate(Vector128<short> value,
Vector128<short> count)
Vector128<int> ShiftLogicalRoundedSaturate(Vector128<int> value,
Vector128<int> count)
Vector128<long> ShiftLogicalRoundedSaturate(Vector128<long> value,
Vector128<long> count)
Vector128<sbyte> ShiftLogicalRoundedSaturate(Vector128<sbyte> value,
Vector128<sbyte> count)
Vector128<ushort> ShiftLogicalRoundedSaturate(Vector128<ushort> value,
Vector128<short> count)
Vector128<uint> ShiftLogicalRoundedSaturate(Vector128<uint> value,
Vector128<int> count)
Vector128<ulong> ShiftLogicalRoundedSaturate(Vector128<ulong> value,
Vector128<long> count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLogicalRoundedSaturateTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[SByte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqrshl v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

285. ShiftLogicalRoundedSaturateScalar

Vector64<long> ShiftLogicalRoundedSaturateScalar(Vector64<long> value, Vector64<long> count)

This method shifts each vector element of the value vector, by the corresponding vector element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<long> ShiftLogicalRoundedSaturateScalarTest(Vector64<long>
value, Vector64<long> count)
{
    return AdvSimd.ShiftLogicalRoundedSaturateScalar(value, count);
}
// value = <11>
// count = <11>
// Result = <22528>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftLogicalRoundedSaturateScalar(Vector64<ulong> value,
Vector64<long> count)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<byte> ShiftLogicalRoundedSaturateScalar(Vector64<byte> value,
Vector64<sbyte> count)
Vector64<short> ShiftLogicalRoundedSaturateScalar(Vector64<short> value,
Vector64<short> count)
Vector64<int> ShiftLogicalRoundedSaturateScalar(Vector64<int> value,
Vector64<int> count)
Vector64<sbyte> ShiftLogicalRoundedSaturateScalar(Vector64<sbyte> value,
Vector64<sbyte> count)
Vector64<ushort> ShiftLogicalRoundedSaturateScalar(Vector64<ushort> value,
Vector64<short> count)
Vector64<uint> ShiftLogicalRoundedSaturateScalar(Vector64<uint> value,
Vector64<int> count)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLogicalRoundedSaturateScalarTest(System.Runtime.Intrinsic
s.Vector64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Runti
me.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] (  3,  3  )  simd8  ->  d1
```

```

HFA(simd8)
;# V02 OutArgs      [V02      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp      fp, lr, [sp,#-16]!
    mov      fp, sp
    uqrshl   d16, d0, d1
    mov      v0.8b, v16.8b
    ldp      fp, lr, [sp],#16
    ret      lr

; Total bytes of code 24, prolog size 8

```

286. ShiftLogicalRoundedScalar

Vector64<long> ShiftLogicalRoundedScalar(Vector64<long> value, Vector64<long> count)

This method shifts each element in the value vector, by the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

```
private Vector64<long> ShiftLogicalRoundedScalarTest(Vector64<long> value,
Vector64<long> count)
{
    return AdvSimd.ShiftLogicalRoundedScalar(value, count);
}
// value = <11>
// count = <11>
// Result = <22528>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftLogicalRoundedScalar(Vector64<ulong> value,
Vector64<long> count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLogicalRoundedScalarTest(System.Runtime.Intrinsics.Vector
64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Runtime.Intri
nsics.Vector64`1[Int64]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02  ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    urshl   d16, d0, d1
    mov     v0.8b, v16.8b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

287. ShiftLogicalSaturate

Vector64<byte> ShiftLogicalSaturate(Vector64<byte> value, Vector64<sbyte> count)

This method shifts each element in the value vector, by the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<byte> ShiftLogicalSaturateTest(Vector64<byte> value,
Vector64<sbyte> count)
{
    return AdvSimd.ShiftLogicalSaturate(value, count);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = <-3, 2, 3, 5, 6, 7, -8, 0>
// Result = <1, 48, 104, 255, 255, 255, 0, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftLogicalSaturate(Vector64<short> value, Vector64<short>
count)
Vector64<int> ShiftLogicalSaturate(Vector64<int> value, Vector64<int> count)
Vector64<sbyte> ShiftLogicalSaturate(Vector64<sbyte> value, Vector64<sbyte>
count)
Vector64<ushort> ShiftLogicalSaturate(Vector64<ushort> value, Vector64<short>
count)
Vector64<uint> ShiftLogicalSaturate(Vector64<uint> value, Vector64<int>
count)
Vector128<byte> ShiftLogicalSaturate(Vector128<byte> value, Vector128<sbyte>
count)
Vector128<short> ShiftLogicalSaturate(Vector128<short> value,
Vector128<short> count)
Vector128<int> ShiftLogicalSaturate(Vector128<int> value, Vector128<int>
count)
Vector128<long> ShiftLogicalSaturate(Vector128<long> value, Vector128<long>
count)
Vector128<sbyte> ShiftLogicalSaturate(Vector128<sbyte> value,
Vector128<sbyte> count)
Vector128<ushort> ShiftLogicalSaturate(Vector128<ushort> value,
Vector128<short> count)
Vector128<uint> ShiftLogicalSaturate(Vector128<uint> value, Vector128<int>
count)
Vector128<ulong> ShiftLogicalSaturate(Vector128<ulong> value, Vector128<long>
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:ShiftLogicalSaturateTest(System.Runtime.Intrinsics.Vector64`1[
Byte],System.Runtime.Intrinsics.Vector64`1[SByte]):System.Runtime.Intrinsics.
Vector64`1[Byte]
;
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] (  3,  3  )  simd8  ->  d1
HFA(simd8)
;# V02 OutArgs      [V02      ] (  1,  1  )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqshl  v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

288. ShiftLogicalSaturateScalar

Vector64<long> ShiftLogicalSaturateScalar(Vector64<long> value, Vector64<long> count)

This method shifts 0th element in the value vector, by the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. If overflow occurs with any of the results, those results are saturated.

```
private Vector64<long> ShiftLogicalSaturateScalarTest(Vector64<long> value,
Vector64<long> count)
{
    return AdvSimd.ShiftLogicalSaturateScalar(value, count);
}
// value = <11>
// count = <11>
// Result = <22528>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftLogicalSaturateScalar(Vector64<ulong> value,
Vector64<long> count)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<byte> ShiftLogicalSaturateScalar(Vector64<byte> value,
Vector64<sbyte> count)
Vector64<short> ShiftLogicalSaturateScalar(Vector64<short> value,
Vector64<short> count)
Vector64<int> ShiftLogicalSaturateScalar(Vector64<int> value, Vector64<int>
count)
Vector64<sbyte> ShiftLogicalSaturateScalar(Vector64<sbyte> value,
Vector64<sbyte> count)
Vector64<ushort> ShiftLogicalSaturateScalar(Vector64<ushort> value,
Vector64<short> count)
Vector64<uint> ShiftLogicalSaturateScalar(Vector64<uint> value, Vector64<int>
count)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLogicalSaturateScalarTest(System.Runtime.Intrinsics.Vecto
r64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Runtime.Intr
insics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] (  3,  3  )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] (  3,  3  )  simd8  ->  d1
```



```

HFA(simd8)
;# V02 OutArgs      [V02      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp    fp, lr, [sp,#-16]!
        mov    fp, sp
        uqshl  d16, d0, d1
        mov    v0.8b, v16.8b
        ldp    fp, lr, [sp],#16
        ret    lr

; Total bytes of code 24, prolog size 8

```

289. ShiftLogicalScalar

Vector64<long> ShiftLogicalScalar(Vector64<long> value, Vector64<long> count)

This method shifts each element in the value vector, by the corresponding element of the count vector, stores the results in a vector and returns the result vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

```
private Vector64<long> ShiftLogicalScalarTest(Vector64<long> value,
Vector64<long> count)
{
    return AdvSimd.ShiftLogicalScalar(value, count);
}
// value = <11>
// count = <11>
// Result = <22528>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftLogicalScalar(Vector64<ulong> value, Vector64<long>
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftLogicalScalarTest(System.Runtime.Intrinsics.Vector64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ushl   d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

290. ShiftRightAndInsert

Vector64<byte> ShiftRightAndInsert(Vector64<byte> left, Vector64<byte> right, byte shift)

This method right shifts each vector element in the right vector, by shift value, and inserts the result into the corresponding vector element in the left vector such that the new zero bits created by the shift are not inserted but retain their existing value as in left vector. Bits shifted out of the left of each vector element in the right are lost.

```
private Vector64<byte> ShiftRightAndInsertTest(Vector64<byte> left,
Vector64<byte> right, byte shift)
{
    return AdvSimd.ShiftRightAndInsert(left, right, 1);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// shift = 1
// Result = <10, 11, 11, 12, 12, 13, 13, 14>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftRightAndInsert(Vector64<short> left, Vector64<short>
right, byte shift)
Vector64<int> ShiftRightAndInsert(Vector64<int> left, Vector64<int> right,
byte shift)
Vector64<sbyte> ShiftRightAndInsert(Vector64<sbyte> left, Vector64<sbyte>
right, byte shift)
Vector64<ushort> ShiftRightAndInsert(Vector64<ushort> left, Vector64<ushort>
right, byte shift)
Vector64<uint> ShiftRightAndInsert(Vector64<uint> left, Vector64<uint> right,
byte shift)
Vector128<byte> ShiftRightAndInsert(Vector128<byte> left, Vector128<byte>
right, byte shift)
Vector128<short> ShiftRightAndInsert(Vector128<short> left, Vector128<short>
right, byte shift)
Vector128<int> ShiftRightAndInsert(Vector128<int> left, Vector128<int> right,
byte shift)
Vector128<long> ShiftRightAndInsert(Vector128<long> left, Vector128<long>
right, byte shift)
Vector128<sbyte> ShiftRightAndInsert(Vector128<sbyte> left, Vector128<sbyte>
right, byte shift)
Vector128<ushort> ShiftRightAndInsert(Vector128<ushort> left,
Vector128<ushort> right, byte shift)
Vector128<uint> ShiftRightAndInsert(Vector128<uint> left, Vector128<uint>
right, byte shift)
Vector128<ulong> ShiftRightAndInsert(Vector128<ulong> left, Vector128<ulong>
right, byte shift)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightAndInsertTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02      ] ( 0, 0 )   ubyte  ->  zero-ref
;# V03 OutArgs       [V03      ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sri    v0.8b, v1.8b, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

291. ShiftRightAndInsertScalar

Vector64<long> ShiftRightAndInsertScalar(Vector64<long> left, Vector64<long> right, byte shift)

This method right shifts each vector element in the right vector, by shift value, and inserts the result into the corresponding vector element in the left vector such that the new zero bits created by the shift are not inserted but retain their existing value as in left vector. Bits shifted out of the left of each vector element in the right are lost.

```
private Vector64<long> ShiftRightAndInsertScalarTest(Vector64<long> left,
Vector64<long> right, byte shift)
{
    return AdvSimd.ShiftRightAndInsertScalar(left, right, 1);
}
// left = <11>
// right = <11>
// shift = 1
// Result = <5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftRightAndInsertScalar(Vector64<ulong> left,
Vector64<ulong> right, byte shift)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightAndInsertScalarTest(System.Runtime.Intrinsics.Vector
64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64],ubyte):System.Runtime
.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02  ] ( 0, 0 )  ubyte   ->  zero-ref
;# V03 OutArgs       [V03  ] ( 1, 1 )  lclBlk  ( 0 ) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sri    d0, d1, #1
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 20, prolog size 8
```


292. ShiftRightArithmetic

Vector64<short> ShiftRightArithmetic(Vector64<short> value, byte count)

This method right shifts each element in the value vector by count, stores the truncated results in a vector and returns the result vector. All the values in this method are signed integer values.

```
private Vector64<short> ShiftRightArithmeticTest(Vector64<short> value, byte
count)
{
    return AdvSimd.ShiftRightArithmetic(value, 1);
}
// value = <11, 12, 13, 14>
// count = 1
// Result = <5, 6, 6, 7>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ShiftRightArithmetic(Vector64<int> value, byte count)
Vector64<sbyte> ShiftRightArithmetic(Vector64<sbyte> value, byte count)
Vector128<short> ShiftRightArithmetic(Vector128<short> value, byte count)
Vector128<int> ShiftRightArithmetic(Vector128<int> value, byte count)
Vector128<long> ShiftRightArithmetic(Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightArithmetic(Vector128<sbyte> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticTest(System.Runtime.Intrinsics.Vector64`1[
Int16],ubyte):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sshr   v16.4h, v0.4h, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

293. ShiftRightArithmeticAdd

Vector64<short> ShiftRightArithmeticAdd(Vector64<short> addend, Vector64<short> value, byte count)

This method right shifts each element in the value vector, by a count, and accumulates the final results with the vector elements of the addend vector and return the accumulated vector. All the values in this method are signed integer values. All results are truncated.

```
private Vector64<short> ShiftRightArithmeticAddTest(Vector64<short> addend,
Vector64<short> value, byte count)
{
    return AdvSimd.ShiftRightArithmeticAdd(addend, value, 1);
}
// addend = <11, 12, 13, 14>
// value = <21, 22, 23, 24>
// count = 1
// Result = <21, 23, 24, 26>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ShiftRightArithmeticAdd(Vector64<int> addend, Vector64<int>
value, byte count)
Vector64<sbyte> ShiftRightArithmeticAdd(Vector64<sbyte> addend,
Vector64<sbyte> value, byte count)
Vector128<short> ShiftRightArithmeticAdd(Vector128<short> addend,
Vector128<short> value, byte count)
Vector128<int> ShiftRightArithmeticAdd(Vector128<int> addend, Vector128<int>
value, byte count)
Vector128<long> ShiftRightArithmeticAdd(Vector128<long> addend,
Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightArithmeticAdd(Vector128<sbyte> addend,
Vector128<sbyte> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticAddTest(System.Runtime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02      ] ( 0, 0 )  ubyte  ->  zero-ref
;# V03 OutArgs       [V03      ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
```



```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ssra   v0.4h, v1.4h, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

294. ShiftRightArithmeticAddScalar

Vector64<long> ShiftRightArithmeticAddScalar(Vector64<long> addend, Vector64<long> value, byte count)

This method right shifts each element in the value vector, by a count, and accumulates the final results with the vector elements of the addend vector and return the accumulated vector. All the values in this method are signed integer values. All results are truncated.

```
private Vector64<long> ShiftRightArithmeticAddScalarTest(Vector64<long>
addend, Vector64<long> value, byte count)
{
    return AdvSimd.ShiftRightArithmeticAddScalar(addend, value, 1);
}
// addend = <11>
// value = <11>
// count = 1
// Result = <16>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticAddScalarTest(System.Runtime.Intrinsics.Ve
ctor64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64],ubyte):System.Run
time.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ssra   d0, d1, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

295. ShiftRightArithmeticNarrowingSaturateLower

Vector64<short> ShiftRightArithmeticNarrowingSaturateLower(Vector128<int> value, byte count)

This method right shifts and truncates each vector element in the value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the result vector. All the values in this method are signed integer values. As seen in below example, the result vector element's size short is half as long as the source vector element's size int.

```
private Vector64<short>
ShiftRightArithmeticNarrowingSaturateLowerTest(Vector128<int> value, byte
count)
{
    return AdvSimd.ShiftRightArithmeticNarrowingSaturateLower(value, 1);
}
// value = <11, 12, 13, 14>
// count = 1
// Result = <5, 6, 6, 7>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ShiftRightArithmeticNarrowingSaturateLower(Vector128<long>
value, byte count)
Vector64<sbyte> ShiftRightArithmeticNarrowingSaturateLower(Vector128<short>
value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticNarrowingSaturateLowerTest(System.Runtime.
Intrinsics.Vector128`1[Int32],ubyte):System.Runtime.Intrinsics.Vector64`1[Int
16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqshrn v16.4h, v0.4s, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

296. ShiftRightArithmeticNarrowingSaturateScalar

Vector64<short> ShiftRightArithmeticNarrowingSaturateScalar(Vector64<int> value, byte count)

This method right shifts and truncates 0th vector element in the value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the 0th element of result vector, other elements being set to 0. All the values in this method are signed integer values. As seen in below example, the result vector element's size short is half as long as the source vector element's size int.

```
private Vector64<short>
ShiftRightArithmeticNarrowingSaturateScalarTest(Vector64<int> value, byte
count)
{
    return AdvSimd.Arm64.ShiftRightArithmeticNarrowingSaturateScalar(value, 1);
}
// value = <11, 12>
// count = 1
// Result = <5, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Adsimd.Arm64
Vector64<int> ShiftRightArithmeticNarrowingSaturateScalar(Vector64<long>
value, byte count)
Vector64<sbyte> ShiftRightArithmeticNarrowingSaturateScalar(Vector64<short>
value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticNarrowingSaturateScalarTest(System.Runtime
.Intrinsics.Vector64`1[Int32],ubyte):System.Runtime.Intrinsics.Vector64`1[Int
16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqshrn h16, s0, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

297. ShiftRightArithmeticNarrowingSaturateUnsignedLower

Vector64<byte>

ShiftRightArithmeticNarrowingSaturateUnsignedLower(Vector128<short> value, byte count)

This method right shifts each signed integer value in the value vector, by count, saturates the result to an unsigned integer value that is half the original width, stores the results in a vector and returns the result vector. The results are truncated.

private Vector64<byte>

ShiftRightArithmeticNarrowingSaturateUnsignedLowerTest(Vector128<short> value, byte count)

```
{
    return AdvSimd.ShiftRightArithmeticNarrowingSaturateUnsignedLower(value,
1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <5, 6, 6, 7, 7, 8, 8, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ushort>
ShiftRightArithmeticNarrowingSaturateUnsignedLower(Vector128<int> value, byte
count)
Vector64<uint>
ShiftRightArithmeticNarrowingSaturateUnsignedLower(Vector128<long> value,
byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticNarrowingSaturateUnsignedLowerTest(System.
Runtime.Intrinsics.Vector128`1[Int16],ubyte):System.Runtime.Intrinsics.Vector
64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqshrun v16.8b, v0.8h, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

298. ShiftRightArithmeticNarrowingSaturateUnsignedScalar

Vector64<byte>

ShiftRightArithmeticNarrowingSaturateUnsignedScalar(Vector64<short> value, byte count)

This method right shifts signed integer value in the value vector at 0th index, by count, saturates the result to an unsigned integer value that is half the original width, stores the results in a vector and returns the result vector, other elements being set to 0. The results are truncated.

```
private Vector64<byte>
ShiftRightArithmeticNarrowingSaturateUnsignedScalarTest(Vector64<short>
value, byte count)
{
    return
AdvSimd.Arm64.ShiftRightArithmeticNarrowingSaturateUnsignedScalar(value, 1);
}
// value = <11, 12, 13, 14>
// count = 1
// Result = <5, 0, 0, 0, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<ushort>
ShiftRightArithmeticNarrowingSaturateUnsignedScalar(Vector64<int> value, byte
count)
Vector64<uint>
ShiftRightArithmeticNarrowingSaturateUnsignedScalar(Vector64<long> value,
byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticNarrowingSaturateUnsignedScalarTest(System
.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector
64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqshrun b16, h0, #1
    mov    v0.8b, v16.8b
```

```
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

299. ShiftRightArithmeticNarrowingSaturateUnsignedUpper

Vector128<byte>

ShiftRightArithmeticNarrowingSaturateUnsignedUpper(Vector64<byte> lower, Vector128<short> value, byte count)

This method right shifts each signed integer value in the upper-half of value vector, by count, saturates the result to an unsigned integer value that is half the original width, stores the final result into a vector, and writes the vector to the upper-half of result vector, the lower-half contains values from lower vector. The results are truncated.

private Vector128<byte>

ShiftRightArithmeticNarrowingSaturateUnsignedUpperTest(Vector64<byte> lower, Vector128<short> value, byte count)

```
{  
    return AdvSimd.ShiftRightArithmeticNarrowingSaturateUnsignedUpper(lower,  
value, 1);  
}
```

```
// lower = <11, 12, 13, 14, 15, 16, 17, 18>
```

```
// value = <11, 12, 13, 14, 15, 16, 17, 18>
```

```
// count = 1
```

```
// Result = <11, 12, 13, 14, 15, 16, 17, 18, 5, 6, 6, 7, 7, 8, 8, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
```

Vector128<ushort>

ShiftRightArithmeticNarrowingSaturateUnsignedUpper(Vector64<ushort> lower, Vector128<int> value, byte count)

Vector128<uint>

ShiftRightArithmeticNarrowingSaturateUnsignedUpper(Vector64<uint> lower, Vector128<long> value, byte count)

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

; Assembly listing for method

AdvSimdMethods:ShiftRightArithmeticNarrowingSaturateUnsignedUpperTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector128`1[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Byte]

;

; V00 arg0 [V00,T00] (3, 3) simd8 -> d0

HFA(simd8)

; V01 arg1 [V01,T01] (3, 3) simd16 -> d1

HFA(simd16)

* V02 arg2 [V02] (0, 0) ubyte -> zero-ref

V03 OutArgs [V03] (1, 1) lclBlk (0) [sp+0x00]

"OutgoingArgSpace"

; Lcl frame size = 0

stp fp, lr, [sp,#-16]!

```
mov    fp, sp
sqshrun2 v0.16b, v1.8h, #1
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 20, prolog size 8

300. ShiftRightArithmeticNarrowingSaturateUpper

Vector128<short> ShiftRightArithmeticNarrowingSaturateUpper(Vector64<short> lower, Vector128<int> value, byte count)

This method right shifts each vector element in the upper-half of value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the upper-half of result vector while lower-half contains lower vector values. All the values in this method are signed integer values. All results are truncated.

```
private Vector128<short>
ShiftRightArithmeticNarrowingSaturateUpperTest(Vector64<short> lower,
Vector128<int> value, byte count)
{
    return AdvSimd.ShiftRightArithmeticNarrowingSaturateUpper(lower, value, 1);
}
// lower = <11, 12, 13, 14>
// value = <11, 12, 13, 14>
// count = 1
// Result = <11, 12, 13, 14, 5, 6, 6, 7>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ShiftRightArithmeticNarrowingSaturateUpper(Vector64<int>
lower, Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightArithmeticNarrowingSaturateUpper(Vector64<sbyte>
lower, Vector128<short> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticNarrowingSaturateUpperTest(System.Runtime.
Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector128`1[Int32],uby
te):System.Runtime.Intrinsics.Vector128`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )  simd16 ->  d1
HFA(simd16)
;* V02 arg2         [V02  ] ( 0, 0 )  ubyte  ->  zero-ref
;# V03 OutArgs      [V03  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqshrn2 v0.8h, v1.4s, #1
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

301. ShiftRightArithmeticRounded

Vector64<short> ShiftRightArithmeticRounded(Vector64<short> value, byte count)

This method right shifts each element in the value vector, by count and then rounded, stores the results in a vector and returns the result vector. All the values in this method are signed integer values.

```
private Vector64<short> ShiftRightArithmeticRoundedTest(Vector64<short>
value, byte count)
{
    return AdvSimd.ShiftRightArithmeticRounded(value, 1);
}
// value = <11, 12, 13, 14>
// count = 1
// Result = <6, 6, 7, 7>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ShiftRightArithmeticRounded(Vector64<int> value, byte count)
Vector64<sbyte> ShiftRightArithmeticRounded(Vector64<sbyte> value, byte
count)
Vector128<short> ShiftRightArithmeticRounded(Vector128<short> value, byte
count)
Vector128<int> ShiftRightArithmeticRounded(Vector128<int> value, byte count)
Vector128<long> ShiftRightArithmeticRounded(Vector128<long> value, byte
count)
Vector128<sbyte> ShiftRightArithmeticRounded(Vector128<sbyte> value, byte
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticRoundedTest(System.Runtime.Intrinsics.Vect
or64`1[Int16],ubyte):System.Runtime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    srshr  v16.4h, v0.4h, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
```

```
ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

302. ShiftRightArithmeticRoundedAdd

Vector64<short> ShiftRightArithmeticRoundedAdd(Vector64<short> addend, Vector64<short> value, byte count)

This method right shifts each element in the value vector, by a count, and accumulates the final results with the vector elements of the addend vector and return the accumulated vector. All the values in this method are signed integer values. All results are rounded.

```
private Vector64<short> ShiftRightArithmeticRoundedAddTest(Vector64<short>
addend, Vector64<short> value, byte count)
{
    return AdvSimd.ShiftRightArithmeticRoundedAdd(addend, value, 1);
}
// addend = <11, 12, 13, 14>
// value = <21, 22, 23, 24>
// count = 1
// Result = <22, 23, 25, 26>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int> ShiftRightArithmeticRoundedAdd(Vector64<int> addend,
Vector64<int> value, byte count)
Vector64<sbyte> ShiftRightArithmeticRoundedAdd(Vector64<sbyte> addend,
Vector64<sbyte> value, byte count)
Vector128<short> ShiftRightArithmeticRoundedAdd(Vector128<short> addend,
Vector128<short> value, byte count)
Vector128<int> ShiftRightArithmeticRoundedAdd(Vector128<int> addend,
Vector128<int> value, byte count)
Vector128<long> ShiftRightArithmeticRoundedAdd(Vector128<long> addend,
Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightArithmeticRoundedAdd(Vector128<sbyte> addend,
Vector128<sbyte> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticRoundedAddTest(System.Runtime.Intrinsics.V
ector64`1[Int16],System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Ru
ntime.Intrinsics.Vector64`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    srsra  v0.4h, v1.4h, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

303. ShiftRightArithmeticRoundedAddScalar

Vector64<long> ShiftRightArithmeticRoundedAddScalar(Vector64<long> addend, Vector64<long> value, byte count)

This method right shifts each element in the value vector, by a count, and accumulates the final results with the vector elements of the addend vector and return the accumulated vector. All the values in this method are signed integer values. All results are rounded.

```
private Vector64<long>
ShiftRightArithmeticRoundedAddScalarTest(Vector64<long> addend,
Vector64<long> value, byte count)
{
    return AdvSimd.ShiftRightArithmeticRoundedAddScalar(addend, value, 1);
}
// addend = <11>
// value = <11>
// count = 1
// Result = <17>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticRoundedAddScalarTest(System.Runtime.Intrin
sics.Vector64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64],ubyte):Sys
tem.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )   simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02  ] ( 0, 0 )   ubyte   ->  zero-ref
;# V03 OutArgs       [V03  ] ( 1, 1 )   lclBlk  ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp     fp, lr, [sp,#-16]!
        mov     fp, sp
        srsra   d0, d1, #1
        ldp     fp, lr, [sp],#16
        ret     lr

; Total bytes of code 20, prolog size 8
```

304. ShiftRightArithmeticRoundedNarrowingSaturateLower

Vector64<short>

ShiftRightArithmeticRoundedNarrowingSaturateLower(Vector128<int> value, byte count)

This method right shifts each vector element in the value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the result vector. All the values in this method are signed integer values. As seen in below example, the result vector element's size short is half as long as the source vector element's size int. The results are rounded.

```
private Vector64<short>
ShiftRightArithmeticRoundedNarrowingSaturateLowerTest(Vector128<int> value,
byte count)
{
    return AdvSimd.ShiftRightArithmeticRoundedNarrowingSaturateLower(value, 1);
}
// value = <11, 12, 13, 14>
// count = 1
// Result = <6, 6, 7, 7>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<int>
ShiftRightArithmeticRoundedNarrowingSaturateLower(Vector128<long> value, byte
count)
Vector64<sbyte>
ShiftRightArithmeticRoundedNarrowingSaturateLower(Vector128<short> value,
byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticRoundedNarrowingSaturateLowerTest(System.R
untime.Intrinsics.Vector128`1[Int32],ubyte):System.Runtime.Intrinsics.Vector6
4`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqrshrn v16.4h, v0.4s, #1
    mov    v0.8b, v16.8b
```

```
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

305. ShiftRightArithmeticRoundedNarrowingSaturateScalar

Vector64<short>

ShiftRightArithmeticRoundedNarrowingSaturateScalar(Vector64<int> value, byte count)

This method right shifts 0th element in the value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into 0th element of vector, and writes the vector to the result vector, other elements being set to 0. All the values in this method are signed integer values. As seen in below example, the result vector element's size short is half as long as the source vector element's size int. The results are rounded.

private Vector64<short>

ShiftRightArithmeticRoundedNarrowingSaturateScalarTest(Vector64<int> value, byte count)

```
{
    return
    AdvSimd.Arm64.ShiftRightArithmeticRoundedNarrowingSaturateScalar(value, 1);
}
// value = <11, 12>
// count = 1
// Result = <6, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<int>
ShiftRightArithmeticRoundedNarrowingSaturateScalar(Vector64<long> value, byte count)
Vector64<sbyte>
ShiftRightArithmeticRoundedNarrowingSaturateScalar(Vector64<short> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticRoundedNarrowingSaturateScalarTest(System.
Runtime.Intrinsics.Vector64`1[Int32],ubyte):System.Runtime.Intrinsics.Vector6
4`1[Int16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp     fp, lr, [sp,#-16]!
        mov     fp, sp
```

```
sqrshrn h16, s0, #1
mov      v0.8b, v16.8b
ldp      fp, lr, [sp],#16
ret      lr
```

; Total bytes of code 24, prolog size 8

306. ShiftRightArithmeticRoundedNarrowingSaturateUnsignedLower

Vector64<byte>

ShiftRightArithmeticRoundedNarrowingSaturateUnsignedLower(Vector128<short> value, byte count)

This method right shifts each signed integer value in the value vector, by count, saturates the result to an unsigned integer value that is half the original width, stores the results in a vector and returns the result vector. The results are rounded.

private Vector64<byte>

ShiftRightArithmeticRoundedNarrowingSaturateUnsignedLowerTest(Vector128<short> value, byte count)

```
{
    return
    AdvSimd.ShiftRightArithmeticRoundedNarrowingSaturateUnsignedLower(value, 1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <6, 6, 7, 7, 8, 8, 9, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ushort>
ShiftRightArithmeticRoundedNarrowingSaturateUnsignedLower(Vector128<int>
value, byte count)
Vector64<uint>
ShiftRightArithmeticRoundedNarrowingSaturateUnsignedLower(Vector128<long>
value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticRoundedNarrowingSaturateUnsignedLowerTest(
System.Runtime.Intrinsics.Vector128`1[Int16],ubyte):System.Runtime.Intrinsics
.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqrshrun v16.8b, v0.8h, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
```



```
ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

307. ShiftRightArithmeticRoundedNarrowingSaturateUnsignedScalar

Vector64<byte>

ShiftRightArithmeticRoundedNarrowingSaturateUnsignedScalar(Vector64<short> value, byte count)

This method right shifts signed integer value in the value vector at 0th index, by count, saturates the result to an unsigned integer value that is half the original width, stores the results in a vector and returns the result vector, other elements being set to 0. The results are rounded.

```
private Vector64<byte>
ShiftRightArithmeticRoundedNarrowingSaturateUnsignedScalarTest(Vector64<short>
> value, byte count)
{
    return
AdvSimd.Arm64.ShiftRightArithmeticRoundedNarrowingSaturateUnsignedScalar(valu
e, 1);
}
// value = <11, 12, 13, 14>
// count = 1
// Result = <6, 0, 0, 0, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<ushort>
ShiftRightArithmeticRoundedNarrowingSaturateUnsignedScalar(Vector64<int>
value, byte count)
Vector64<uint>
ShiftRightArithmeticRoundedNarrowingSaturateUnsignedScalar(Vector64<long>
value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticRoundedNarrowingSaturateUnsignedScalarTest
(System.Runtime.Intrinsics.Vector64`1[Int16],ubyte):System.Runtime.Intrinsics
.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01  ] ( 0, 0 )  ubyte  ->  zero-ref
;# V02 OutArgs       [V02  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sqrshrun b16, h0, #1
```

```
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret     lr
```

; Total bytes of code 24, prolog size 8

308. ShiftRightArithmeticRoundedNarrowingSaturateUnsignedUpper

Vector128<byte>

ShiftRightArithmeticRoundedNarrowingSaturateUnsignedUpper(Vector64<byte> lower, Vector128<short> value, byte count)

This method right shifts each signed integer value in the upper-half of value vector, by count, saturates the result to an unsigned integer value that is half the original width, stores the final result into a vector, and writes the vector to the upper-half of result vector, the lower-half contains values from lower vector. The results are rounded.

```
private Vector128<byte>
ShiftRightArithmeticRoundedNarrowingSaturateUnsignedUpperTest(Vector64<byte>
lower, Vector128<short> value, byte count)
{
    return
AdvSimd.ShiftRightArithmeticRoundedNarrowingSaturateUnsignedUpper(lower,
value, 1);
}
// lower = <11, 12, 13, 14, 15, 16, 17, 18>
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <11, 12, 13, 14, 15, 16, 17, 18, 6, 6, 7, 7, 8, 8, 9, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<ushort>
ShiftRightArithmeticRoundedNarrowingSaturateUnsignedUpper(Vector64<ushort>
lower, Vector128<int> value, byte count)
Vector128<uint>
ShiftRightArithmeticRoundedNarrowingSaturateUnsignedUpper(Vector64<uint>
lower, Vector128<long> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticRoundedNarrowingSaturateUnsignedUpperTest(
System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector128`1[Int16],ubyte):System.Runtime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )  simd16 ->  d1
HFA(simd16)
;* V02 arg2          [V02      ] ( 0, 0 )  ubyte  -> zero-ref
;# V03 OutArgs       [V03      ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    sqrshrun2 v0.16b, v1.8h, #1  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 20, prolog size 8

309. ShiftRightArithmeticRoundedNarrowingSaturateUpper

```
Vector128<short>  
ShiftRightArithmeticRoundedNarrowingSaturateUpper(Vector64<short> lower,  
Vector128<int> value, byte count)
```

This method right shifts each vector element in the upper-half of value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the upper-half of result vector while lower-half contains lower vector values. All the values in this method are signed integer values. All results are rounded.

```
private Vector128<short>  
ShiftRightArithmeticRoundedNarrowingSaturateUpperTest(Vector64<short> lower,  
Vector128<int> value, byte count)  
{  
    return AdvSimd.ShiftRightArithmeticRoundedNarrowingSaturateUpper(lower,  
value, 1);  
}  
// lower = <11, 12, 13, 14>  
// value = <11, 12, 13, 14>  
// count = 1  
// Result = <11, 12, 13, 14, 6, 6, 7, 7>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector128<int>  
ShiftRightArithmeticRoundedNarrowingSaturateUpper(Vector64<int> lower,  
Vector128<long> value, byte count)  
Vector128<sbyte>  
ShiftRightArithmeticRoundedNarrowingSaturateUpper(Vector64<sbyte> lower,  
Vector128<short> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods:ShiftRightArithmeticRoundedNarrowingSaturateUpperTest(System.R  
untime.Intrinsics.Vector64`1[Int16],System.Runtime.Intrinsics.Vector128`1[Int  
32],ubyte):System.Runtime.Intrinsics.Vector128`1[Int16]  
;  
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0  
HFA(simd8)  
; V01 arg1          [V01,T01] ( 3, 3 )  simd16 ->  d1  
HFA(simd16)  
;* V02 arg2          [V02 ] ( 0, 0 )  ubyte  ->  zero-ref  
;# V03 OutArgs       [V03 ] ( 1, 1 )  lclBlk ( 0 ) [sp+0x00]  
"OutgoingArgSpace"  
; Lcl frame size = 0
```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    sqsrshr2 v0.8h, v1.4s, #1  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 20, prolog size 8

310. ShiftRightArithmeticRoundedScalar

Vector64<long> ShiftRightArithmeticRoundedScalar(Vector64<long> value, byte count)

This method right shifts each element in the value vector, by count and then rounded, stores the results in a vector and returns the result vector. All the values in this method are signed integer values.

```
private Vector64<long> ShiftRightArithmeticRoundedScalarTest(Vector64<long>
value, byte count)
{
    return AdvSimd.ShiftRightArithmeticRoundedScalar(value, 1);
}
// value = <11>
// count = 1
// Result = <6>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticRoundedScalarTest(System.Runtime.Intrinsic
s.Vector64`1[Int64],ubyte):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    srshr  d16, d0, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

311. ShiftRightArithmeticScalar

Vector64<long> ShiftRightArithmeticScalar(Vector64<long> value, byte count)

This method right shifts each element in the value vector by count, stores the truncated results in a vector and returns the result vector. All the values in this method are signed integer values.

```
private Vector64<long> ShiftRightArithmeticScalarTest(Vector64<long> value,
byte count)
{
    return AdvSimd.ShiftRightArithmeticScalar(value, 1);
}
// value = <11>
// count = 1
// Result = <5>
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightArithmeticScalarTest(System.Runtime.Intrinsics.Vector64`1[Int64],ubyte):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sshr   d16, d0, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

312. ShiftRightLogical

Vector64<byte> ShiftRightLogical(Vector64<byte> value, byte count)

This method right shifts each element in the value vector, by count, stores the results in a vector and returns the result vector. The results are truncated.

```
private Vector64<byte> ShiftRightLogicalTest(Vector64<byte> value, byte
count)
{
    return AdvSimd.ShiftRightLogical(value, 1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <5, 6, 6, 7, 7, 8, 8, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftRightLogical(Vector64<short> value, byte count)
Vector64<int> ShiftRightLogical(Vector64<int> value, byte count)
Vector64<sbyte> ShiftRightLogical(Vector64<sbyte> value, byte count)
Vector64<ushort> ShiftRightLogical(Vector64<ushort> value, byte count)
Vector64<uint> ShiftRightLogical(Vector64<uint> value, byte count)
Vector128<byte> ShiftRightLogical(Vector128<byte> value, byte count)
Vector128<short> ShiftRightLogical(Vector128<short> value, byte count)
Vector128<int> ShiftRightLogical(Vector128<int> value, byte count)
Vector128<long> ShiftRightLogical(Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightLogical(Vector128<sbyte> value, byte count)
Vector128<ushort> ShiftRightLogical(Vector128<ushort> value, byte count)
Vector128<uint> ShiftRightLogical(Vector128<uint> value, byte count)
Vector128<ulong> ShiftRightLogical(Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalTest(System.Runtime.Intrinsics.Vector64`1[Byte
e],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )   simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01  ] ( 0, 0 )   ubyte   ->  zero-ref
;# V02 OutArgs       [V02  ] ( 1, 1 )   lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ushr   v16.8b, v0.8b, #1
    mov    v0.8b, v16.8b
```

```
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 24, prolog size 8

313. ShiftRightLogicalAdd

Vector64<byte> ShiftRightLogicalAdd(Vector64<byte> addend, Vector64<byte> value, byte count)

This method right shifts each element in the value vector, by count, and accumulates the final results with the vector elements of the addend vector and return the result vector. The results are truncated.

```
private Vector64<byte> ShiftRightLogicalAddTest(Vector64<byte> addend,
Vector64<byte> value, byte count)
{
    return AdvSimd.ShiftRightLogicalAdd(addend, value, 1);
}
// addend = <11, 12, 13, 14, 15, 16, 17, 18>
// value = <21, 22, 23, 24, 25, 26, 27, 28>
// count = 1
// Result = <21, 23, 24, 26, 27, 29, 30, 32>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftRightLogicalAdd(Vector64<short> addend, Vector64<short>
value, byte count)
Vector64<int> ShiftRightLogicalAdd(Vector64<int> addend, Vector64<int> value,
byte count)
Vector64<sbyte> ShiftRightLogicalAdd(Vector64<sbyte> addend, Vector64<sbyte>
value, byte count)
Vector64<ushort> ShiftRightLogicalAdd(Vector64<ushort> addend,
Vector64<ushort> value, byte count)
Vector64<uint> ShiftRightLogicalAdd(Vector64<uint> addend, Vector64<uint>
value, byte count)
Vector128<byte> ShiftRightLogicalAdd(Vector128<byte> addend, Vector128<byte>
value, byte count)
Vector128<short> ShiftRightLogicalAdd(Vector128<short> addend,
Vector128<short> value, byte count)
Vector128<int> ShiftRightLogicalAdd(Vector128<int> addend, Vector128<int>
value, byte count)
Vector128<long> ShiftRightLogicalAdd(Vector128<long> addend, Vector128<long>
value, byte count)
Vector128<sbyte> ShiftRightLogicalAdd(Vector128<sbyte> addend,
Vector128<sbyte> value, byte count)
Vector128<ushort> ShiftRightLogicalAdd(Vector128<ushort> addend,
Vector128<ushort> value, byte count)
Vector128<uint> ShiftRightLogicalAdd(Vector128<uint> addend, Vector128<uint>
value, byte count)
Vector128<ulong> ShiftRightLogicalAdd(Vector128<ulong> addend,
Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalAddTest(System.Runtime.Intrinsics.Vector64`1[
Byte],System.Runtime.Intrinsics.Vector64`1[Byte],ubyte):System.Runtime.Intrin
sics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    usra   v0.8b, v1.8b, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

314. ShiftRightLogicalAddScalar

**Vector64<long> ShiftRightLogicalAddScalar(Vector64<long> addend,
Vector64<long> value, byte count)**

This method right shifts each element in the value vector, by count, and accumulates the final results with the vector elements of the addend vector and return the result vector. The results are truncated.

```
private Vector64<long> ShiftRightLogicalAddScalarTest(Vector64<long> addend,  
Vector64<long> value, byte count)  
{  
    return AdvSimd.ShiftRightLogicalAddScalar(addend, value, 1);  
}  
// addend = <11>  
// value = <11>  
// count = 1  
// Result = <16>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd  
Vector64<ulong> ShiftRightLogicalAddScalar(Vector64<ulong> addend,  
Vector64<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method  
AdvSimdMethods:ShiftRightLogicalAddScalarTest(System.Runtime.Intrinsics.Vector  
r64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64],ubyte):System.Runtim  
e.Intrinsics.Vector64`1[Int64]  
;  
; V00 arg0 [V00,T00] ( 3, 3 ) simd8 -> d0  
HFA(simd8)  
; V01 arg1 [V01,T01] ( 3, 3 ) simd8 -> d1  
HFA(simd8)  
;* V02 arg2 [V02 ] ( 0, 0 ) ubyte -> zero-ref  
;# V03 OutArgs [V03 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]  
"OutgoingArgSpace"  
; Lcl frame size = 0  
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    usra   d0, d1, #1  
    ldp    fp, lr, [sp],#16  
    ret    lr  
  
; Total bytes of code 20, prolog size 8
```

315. ShiftRightLogicalNarrowingLower

Vector64<byte> ShiftRightLogicalNarrowingLower(Vector128<ushort> value, byte count)

This method right shifts each integer value in the value vector, by count, stores the final result into a vector, and writes the vector to the result vector. As seen in below example, the result vector element's size byte is half as long as the source vector element's size ushort. The results are truncated.

```
private Vector64<byte> ShiftRightLogicalNarrowingLowerTest(Vector128<ushort>
value, byte count)
{
    return AdvSimd.ShiftRightLogicalNarrowingLower(value, 1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <5, 6, 6, 7, 7, 8, 8, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftRightLogicalNarrowingLower(Vector128<int> value, byte
count)
Vector64<int> ShiftRightLogicalNarrowingLower(Vector128<long> value, byte
count)
Vector64<sbyte> ShiftRightLogicalNarrowingLower(Vector128<short> value, byte
count)
Vector64<ushort> ShiftRightLogicalNarrowingLower(Vector128<uint> value, byte
count)
Vector64<uint> ShiftRightLogicalNarrowingLower(Vector128<ulong> value, byte
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalNarrowingLowerTest(System.Runtime.Intrinsics.
Vector128`1[UInt16],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1         [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    shrn   v16.8b, v0.8h, #1
    mov    v0.8b, v16.8b
```

```
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 24, prolog size 8

316. ShiftRightLogicalNarrowingSaturateLower

Vector64<byte> ShiftRightLogicalNarrowingSaturateLower(Vector128<ushort> value, byte count)

This method right shifts each element in the value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the result vector. The results are truncated.

```
private Vector64<byte>
ShiftRightLogicalNarrowingSaturateLowerTest(Vector128<ushort> value, byte
count)
{
    return AdvSimd.ShiftRightLogicalNarrowingSaturateLower(value, 1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <5, 6, 6, 7, 7, 8, 8, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftRightLogicalNarrowingSaturateLower(Vector128<int> value,
byte count)
Vector64<int> ShiftRightLogicalNarrowingSaturateLower(Vector128<long> value,
byte count)
Vector64<sbyte> ShiftRightLogicalNarrowingSaturateLower(Vector128<short>
value, byte count)
Vector64<ushort> ShiftRightLogicalNarrowingSaturateLower(Vector128<uint>
value, byte count)
Vector64<uint> ShiftRightLogicalNarrowingSaturateLower(Vector128<ulong>
value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalNarrowingSaturateLowerTest(System.Runtime.Int
rinsics.Vector128`1[UInt16],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqshrn v16.8b, v0.8h, #1
    mov    v0.8b, v16.8b
```

```
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

317. ShiftRightLogicalNarrowingSaturateScalar

Vector64<byte> ShiftRightLogicalNarrowingSaturateScalar(Vector64<ushort> value, byte count)

This method right shifts 0th vector element in the value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the result vector.

```
private Vector64<byte>
ShiftRightLogicalNarrowingSaturateScalarTest(Vector64<ushort> value, byte
count)
{
    return AdvSimd.Arm64.ShiftRightLogicalNarrowingSaturateScalar(value, 1);
}
// value = <11, 12, 13, 14>
// count = 1
// Result = <5, 0, 0, 0, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<short> ShiftRightLogicalNarrowingSaturateScalar(Vector64<int> value,
byte count)
Vector64<int> ShiftRightLogicalNarrowingSaturateScalar(Vector64<long> value,
byte count)
Vector64<sbyte> ShiftRightLogicalNarrowingSaturateScalar(Vector64<short>
value, byte count)
Vector64<ushort> ShiftRightLogicalNarrowingSaturateScalar(Vector64<uint>
value, byte count)
Vector64<uint> ShiftRightLogicalNarrowingSaturateScalar(Vector64<ulong>
value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalNarrowingSaturateScalarTest(System.Runtime.In
trinsics.Vector64`1[UInt16],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqshrn b16, h0, #1
    mov    v0.8b, v16.8b
```

```
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

318. ShiftRightLogicalNarrowingSaturateUpper

Vector128<byte> ShiftRightLogicalNarrowingSaturateUpper(Vector64<byte> lower, Vector128<ushort> value, byte count)

This method right shifts each element in the upper-half of value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the upper-half of result vector, lower-half being values from lower vector. The results are truncated.

```
private Vector128<byte>
ShiftRightLogicalNarrowingSaturateUpperTest(Vector64<byte> lower,
Vector128<ushort> value, byte count)
{
    return AdvSimd.ShiftRightLogicalNarrowingSaturateUpper(lower, value, 1);
}
// lower = <11, 12, 13, 14, 15, 16, 17, 18>
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <11, 12, 13, 14, 15, 16, 17, 18, 5, 6, 6, 7, 7, 8, 8, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> ShiftRightLogicalNarrowingSaturateUpper(Vector64<short>
lower, Vector128<int> value, byte count)
Vector128<int> ShiftRightLogicalNarrowingSaturateUpper(Vector64<int> lower,
Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightLogicalNarrowingSaturateUpper(Vector64<sbyte>
lower, Vector128<short> value, byte count)
Vector128<ushort> ShiftRightLogicalNarrowingSaturateUpper(Vector64<ushort>
lower, Vector128<uint> value, byte count)
Vector128<uint> ShiftRightLogicalNarrowingSaturateUpper(Vector64<uint> lower,
Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalNarrowingSaturateUpperTest(System.Runtime.Int
rinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16],ubyte)
:System.Runtime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqshrn2 v0.16b, v1.8h, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

319. ShiftRightLogicalNarrowingUpper

Vector128<byte> ShiftRightLogicalNarrowingUpper(Vector64<byte> lower, Vector128<ushort> value, byte count)

This method right shifts each integer value from the upper-half of value vector, by count, stores the final result into a vector, and writes the vector to the upper-half of result vector, lower-half being values from lower vector. As seen in below example, the result vector element's size byte is half as long as the input vector element's size ushort. The results are truncated.

```
private Vector128<byte> ShiftRightLogicalNarrowingUpperTest(Vector64<byte>
lower, Vector128<ushort> value, byte count)
{
    return AdvSimd.ShiftRightLogicalNarrowingUpper(lower, value, 1);
}
// lower = <11, 12, 13, 14, 15, 16, 17, 18>
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <11, 12, 13, 14, 15, 16, 17, 18, 5, 6, 6, 7, 7, 8, 8, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> ShiftRightLogicalNarrowingUpper(Vector64<short> lower,
Vector128<int> value, byte count)
Vector128<int> ShiftRightLogicalNarrowingUpper(Vector64<int> lower,
Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightLogicalNarrowingUpper(Vector64<sbyte> lower,
Vector128<short> value, byte count)
Vector128<ushort> ShiftRightLogicalNarrowingUpper(Vector64<ushort> lower,
Vector128<uint> value, byte count)
Vector128<uint> ShiftRightLogicalNarrowingUpper(Vector64<uint> lower,
Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalNarrowingUpperTest(System.Runtime.Intrinsics.
Vector64`1[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16],ubyte):System.
Runtime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    shrn2  v0.16b, v1.8h, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

320. ShiftRightLogicalRounded

Vector64<byte> ShiftRightLogicalRounded(Vector64<byte> value, byte count)

This method right shifts each element in the value vector, by count, stores the results in a vector and returns the result vector. The results are rounded.

```
private Vector64<byte> ShiftRightLogicalRoundedTest(Vector64<byte> value,
byte count)
{
    return AdvSimd.ShiftRightLogicalRounded(value, 1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <6, 6, 7, 7, 8, 8, 9, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftRightLogicalRounded(Vector64<short> value, byte count)
Vector64<int> ShiftRightLogicalRounded(Vector64<int> value, byte count)
Vector64<sbyte> ShiftRightLogicalRounded(Vector64<sbyte> value, byte count)
Vector64<ushort> ShiftRightLogicalRounded(Vector64<ushort> value, byte count)
Vector64<uint> ShiftRightLogicalRounded(Vector64<uint> value, byte count)
Vector128<byte> ShiftRightLogicalRounded(Vector128<byte> value, byte count)
Vector128<short> ShiftRightLogicalRounded(Vector128<short> value, byte count)
Vector128<int> ShiftRightLogicalRounded(Vector128<int> value, byte count)
Vector128<long> ShiftRightLogicalRounded(Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightLogicalRounded(Vector128<sbyte> value, byte count)
Vector128<ushort> ShiftRightLogicalRounded(Vector128<ushort> value, byte
count)
Vector128<uint> ShiftRightLogicalRounded(Vector128<uint> value, byte count)
Vector128<ulong> ShiftRightLogicalRounded(Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalRoundedTest(System.Runtime.Intrinsics.Vector64`1[Byte],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    urshr  v16.8b, v0.8b, #1
```

```
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret     lr
```

; Total bytes of code 24, prolog size 8

321. ShiftRightLogicalRoundedAdd

Vector64<byte> ShiftRightLogicalRoundedAdd(Vector64<byte> addend, Vector64<byte> value, byte count)

This method right shifts each element in the value vector, by count, and accumulates the final results with the vector elements of the addend vector and return the result vector. The results are rounded.

```
private Vector64<byte> ShiftRightLogicalRoundedAddTest(Vector64<byte> addend,
Vector64<byte> value, byte count)
{
    return AdvSimd.ShiftRightLogicalRoundedAdd(addend, value, 1);
}
// addend = <11, 12, 13, 14, 15, 16, 17, 18>
// value = <21, 22, 23, 24, 25, 26, 27, 28>
// count = 1
// Result = <22, 23, 25, 26, 28, 29, 31, 32>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftRightLogicalRoundedAdd(Vector64<short> addend,
Vector64<short> value, byte count)
Vector64<int> ShiftRightLogicalRoundedAdd(Vector64<int> addend, Vector64<int>
value, byte count)
Vector64<sbyte> ShiftRightLogicalRoundedAdd(Vector64<sbyte> addend,
Vector64<sbyte> value, byte count)
Vector64<ushort> ShiftRightLogicalRoundedAdd(Vector64<ushort> addend,
Vector64<ushort> value, byte count)
Vector64<uint> ShiftRightLogicalRoundedAdd(Vector64<uint> addend,
Vector64<uint> value, byte count)
Vector128<byte> ShiftRightLogicalRoundedAdd(Vector128<byte> addend,
Vector128<byte> value, byte count)
Vector128<short> ShiftRightLogicalRoundedAdd(Vector128<short> addend,
Vector128<short> value, byte count)
Vector128<int> ShiftRightLogicalRoundedAdd(Vector128<int> addend,
Vector128<int> value, byte count)
Vector128<long> ShiftRightLogicalRoundedAdd(Vector128<long> addend,
Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightLogicalRoundedAdd(Vector128<sbyte> addend,
Vector128<sbyte> value, byte count)
Vector128<ushort> ShiftRightLogicalRoundedAdd(Vector128<ushort> addend,
Vector128<ushort> value, byte count)
Vector128<uint> ShiftRightLogicalRoundedAdd(Vector128<uint> addend,
Vector128<uint> value, byte count)
Vector128<ulong> ShiftRightLogicalRoundedAdd(Vector128<ulong> addend,
Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalRoundedAddTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 )  simd8  ->  d1
HFA(simd8)
;* V02 arg2          [V02    ] ( 0, 0 )  ubyte  ->  zero-ref
;# V03 OutArgs       [V03    ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ursra  v0.8b, v1.8b, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

322. ShiftRightLogicalRoundedAddScalar

Vector64<long> ShiftRightLogicalRoundedAddScalar(Vector64<long> addend, Vector64<long> value, byte count)

This method right shifts each element in the value vector, by count, and accumulates the final results with the vector elements of the addend vector and return the result vector. The results are rounded.

```
private Vector64<long> ShiftRightLogicalRoundedAddScalarTest(Vector64<long>
addend, Vector64<long> value, byte count)
{
    return AdvSimd.ShiftRightLogicalRoundedAddScalar(addend, value, 1);
}
// addend = <11>
// value = <11>
// count = 1
// Result = <17>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftRightLogicalRoundedAddScalar(Vector64<ulong> addend,
Vector64<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalRoundedAddScalarTest(System.Runtime.Intrinsic
s.Vector64`1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64],ubyte):System
.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ursra  d0, d1, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

323. ShiftRightLogicalRoundedNarrowingLower

Vector64<byte> ShiftRightLogicalRoundedNarrowingLower(Vector128<ushort> value, byte count)

This method right shifts each integer value in the value vector, by count, stores the final result into a vector, and writes the vector to the result vector. As seen in below example, the result vector element's size byte is half as long as the source vector element's size ushort. The results are rounded.

```
private Vector64<byte>
ShiftRightLogicalRoundedNarrowingLowerTest(Vector128<ushort> value, byte
count)
{
    return AdvSimd.ShiftRightLogicalRoundedNarrowingLower(value, 1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <6, 6, 7, 7, 8, 8, 9, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftRightLogicalRoundedNarrowingLower(Vector128<int> value,
byte count)
Vector64<int> ShiftRightLogicalRoundedNarrowingLower(Vector128<long> value,
byte count)
Vector64<sbyte> ShiftRightLogicalRoundedNarrowingLower(Vector128<short>
value, byte count)
Vector64<ushort> ShiftRightLogicalRoundedNarrowingLower(Vector128<uint>
value, byte count)
Vector64<uint> ShiftRightLogicalRoundedNarrowingLower(Vector128<ulong> value,
byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalRoundedNarrowingLowerTest(System.Runtime.Intr
insics.Vector128`1[UInt16],ubyte):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    rshrn  v16.8b, v0.8h, #1
```

```
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

324. ShiftRightLogicalRoundedNarrowingSaturateLower

Vector64<byte>

ShiftRightLogicalRoundedNarrowingSaturateLower(Vector128<ushort> value, byte count)

This method right shifts each element in the value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the result vector. The results are rounded.

private Vector64<byte>

ShiftRightLogicalRoundedNarrowingSaturateLowerTest(Vector128<ushort> value, byte count)

```
{
    return AdvSimd.ShiftRightLogicalRoundedNarrowingSaturateLower(value, 1);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <6, 6, 7, 7, 8, 8, 9, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> ShiftRightLogicalRoundedNarrowingSaturateLower(Vector128<int>
value, byte count)
Vector64<int> ShiftRightLogicalRoundedNarrowingSaturateLower(Vector128<long>
value, byte count)
Vector64<sbyte>
ShiftRightLogicalRoundedNarrowingSaturateLower(Vector128<short> value, byte
count)
Vector64<ushort>
ShiftRightLogicalRoundedNarrowingSaturateLower(Vector128<uint> value, byte
count)
Vector64<uint>
ShiftRightLogicalRoundedNarrowingSaturateLower(Vector128<ulong> value, byte
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalRoundedNarrowingSaturateLowerTest(System.Runt
ime.Intrinsics.Vector128`1[UInt16],ubyte):System.Runtime.Intrinsics.Vector64`
1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
```



```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqrshrn v16.8b, v0.8h, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

325. ShiftRightLogicalRoundedNarrowingSaturateScalar

Vector64<byte>

ShiftRightLogicalRoundedNarrowingSaturateScalar(Vector64<ushort> value, byte count)

This method right shifts 0th vector element in the value vector, by count, saturates each shifted result to a value that is half the original width, stores the final result into a vector, and writes the vector to the result vector.

private Vector64<byte>

ShiftRightLogicalRoundedNarrowingSaturateScalarTest(Vector64<ushort> value, byte count)

```
{
    return AdvSimd.Arm64.ShiftRightLogicalRoundedNarrowingSaturateScalar(value,
1);
}
// value = <11, 12, 13, 14>
// count = 1
// Result = <6, 0, 0, 0, 0, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<short> ShiftRightLogicalRoundedNarrowingSaturateScalar(Vector64<int>
value, byte count)
Vector64<int> ShiftRightLogicalRoundedNarrowingSaturateScalar(Vector64<long>
value, byte count)
Vector64<sbyte>
ShiftRightLogicalRoundedNarrowingSaturateScalar(Vector64<short> value, byte
count)
Vector64<ushort>
ShiftRightLogicalRoundedNarrowingSaturateScalar(Vector64<uint> value, byte
count)
Vector64<uint>
ShiftRightLogicalRoundedNarrowingSaturateScalar(Vector64<ulong> value, byte
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalRoundedNarrowingSaturateScalarTest(System.Run
time.Intrinsics.Vector64`1[UInt16],ubyte):System.Runtime.Intrinsics.Vector64`
1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqrshrn b16, h0, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

326. ShiftRightLogicalRoundedNarrowingSaturateUpper

Vector128<byte> ShiftRightLogicalRoundedNarrowingSaturateUpper(Vector64<byte> lower, Vector128<ushort> value, byte count)

This method right shifts each element in the upper-half of value vector, by count, stores the final result into a vector, and writes the vector to the upper-half of result vector, lower-half being the values from lower vector. The results are rounded. As per ARM docs, if overflow occurs with any of the results, those results are saturated.

```
private Vector128<byte>
ShiftRightLogicalRoundedNarrowingSaturateUpperTest(Vector64<byte> lower,
Vector128<ushort> value, byte count)
{
    return AdvSimd.ShiftRightLogicalRoundedNarrowingSaturateUpper(lower, value,
1);
}
// lower = <11, 12, 13, 14, 15, 16, 17, 18>
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <11, 12, 13, 14, 15, 16, 17, 18, 6, 6, 7, 7, 8, 8, 9, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short>
ShiftRightLogicalRoundedNarrowingSaturateUpper(Vector64<short> lower,
Vector128<int> value, byte count)
Vector128<int> ShiftRightLogicalRoundedNarrowingSaturateUpper(Vector64<int>
lower, Vector128<long> value, byte count)
Vector128<sbyte>
ShiftRightLogicalRoundedNarrowingSaturateUpper(Vector64<sbyte> lower,
Vector128<short> value, byte count)
Vector128<ushort>
ShiftRightLogicalRoundedNarrowingSaturateUpper(Vector64<ushort> lower,
Vector128<uint> value, byte count)
Vector128<uint> ShiftRightLogicalRoundedNarrowingSaturateUpper(Vector64<uint>
lower, Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalRoundedNarrowingSaturateUpperTest(System.Runt
ime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16]
,ubyte):System.Runtime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
```

```

HFA(simd16)
;* V02 arg2      [V02   ] (  0,  0  )  ubyte  ->  zero-ref
;# V03 OutArgs   [V03   ] (  1,  1  )  lclBlk (  0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqrshrn2 v0.16b, v1.8h, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8

```

327. ShiftRightLogicalRoundedNarrowingUpper

Vector128<byte> ShiftRightLogicalRoundedNarrowingUpper(Vector64<byte> lower, Vector128<ushort> value, byte count)

This method right shifts each integer value from the upper-half of value vector, by count, writes the final result to a vector, and writes the vector to the upper-half of result vector, the lower-half being values from lower vector. As seen in below example, the result vector element's size byte is half as long as the input vector element's size ushort. The results are rounded.

```
private Vector128<byte>
ShiftRightLogicalRoundedNarrowingUpperTest(Vector64<byte> lower,
Vector128<ushort> value, byte count)
{
    return AdvSimd.ShiftRightLogicalRoundedNarrowingUpper(lower, value, 1);
}
// lower = <11, 12, 13, 14, 15, 16, 17, 18>
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// count = 1
// Result = <11, 12, 13, 14, 15, 16, 17, 18, 6, 6, 7, 7, 8, 8, 9, 9>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> ShiftRightLogicalRoundedNarrowingUpper(Vector64<short>
lower, Vector128<int> value, byte count)
Vector128<int> ShiftRightLogicalRoundedNarrowingUpper(Vector64<int> lower,
Vector128<long> value, byte count)
Vector128<sbyte> ShiftRightLogicalRoundedNarrowingUpper(Vector64<sbyte>
lower, Vector128<short> value, byte count)
Vector128<ushort> ShiftRightLogicalRoundedNarrowingUpper(Vector64<ushort>
lower, Vector128<uint> value, byte count)
Vector128<uint> ShiftRightLogicalRoundedNarrowingUpper(Vector64<uint> lower,
Vector128<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalRoundedNarrowingUpperTest(System.Runtime.Intr
insics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16],ubyte):
System.Runtime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;* V02 arg2          [V02 ] ( 0, 0 ) ubyte -> zero-ref
;# V03 OutArgs       [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    rshrn2 v0.16b, v1.8h, #1
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

328. ShiftRightLogicalRoundedScalar

Vector64<long> ShiftRightLogicalRoundedScalar(Vector64<long> value, byte count)

This method right shifts each element in the value vector, by count, stores the results in a vector and returns the result vector. The results are rounded.

```
private Vector64<long> ShiftRightLogicalRoundedScalarTest(Vector64<long>
value, byte count)
{
    return AdvSimd.ShiftRightLogicalRoundedScalar(value, 1);
}
// value = <11>
// count = 1
// Result = <6>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftRightLogicalRoundedScalar(Vector64<ulong> value, byte
count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalRoundedScalarTest(System.Runtime.Intrinsics.V
ector64`1[Int64],ubyte):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;* V01 arg1          [V01 ] ( 0, 0 ) ubyte -> zero-ref
;# V02 OutArgs       [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    urshr  d16, d0, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

329. ShiftRightLogicalScalar

Vector64<long> ShiftRightLogicalScalar(Vector64<long> value, byte count)

This method right shifts each vector element in the value vector, by count, stores the results in a vector and returns the result vector. The results are truncated.

```
private Vector64<long> ShiftRightLogicalScalarTest(Vector64<long> value, byte
count)
{
    return AdvSimd.ShiftRightLogicalScalar(value, 1);
}
// value = <11>
// count = 1
// Result = <5>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> ShiftRightLogicalScalar(Vector64<ulong> value, byte count)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ShiftRightLogicalScalarTest(System.Runtime.Intrinsics.Vector64
`1[Int64],ubyte):System.Runtime.Intrinsics.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 )  simd8  ->  d0
HFA(simd8)
;* V01 arg1          [V01  ] ( 0, 0 )  ubyte  ->  zero-ref
;# V02 OutArgs       [V02  ] ( 1, 1 )  lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    ushr   d16, d0, #1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

330. SignExtendWideningLower

Vector128<int> SignExtendWideningLower(Vector64<short> value)

This method duplicates each vector element in the value into a vector, and writes the vector to the result vector. As seen in below example, number of result vector elements are twice as long as the input vector elements. All the values in this method are signed integer values.

```
private Vector128<int> SignExtendWideningLowerTest(Vector64<short> value)
{
    return AdvSimd.SignExtendWideningLower(value);
}
// value = <11, -12, 13, 14>
// Result = <11, -12, 13, 14>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long> SignExtendWideningLower(Vector64<int> value)
Vector128<short> SignExtendWideningLower(Vector64<sbyte> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:SignExtendWideningLowerTest(System.Runtime.Intrinsics.Vector64
`1[Int16]):System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sxtl    v16.4s, v0.4h
    mov     v0.16b, v16.16b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

331. SignExtendWideningUpper

Vector128<int> SignExtendWideningUpper(Vector128<short> value)

This method duplicates each vector element in the upper half of the value into a vector, and writes the vector to the result vector. As seen in below example, the destination vector element's size `int` is twice as long as the input vector element's size `short`. All the values in this method are signed integer values.

```
private Vector128<int> SignExtendWideningUpperTest(Vector128<short> value)
{
    return AdvSimd.SignExtendWideningUpper(value);
}
// value = <11, 12, 13, 14, -15, 16, 17, 18>
// Result = <-15, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<long> SignExtendWideningUpper(Vector128<int> value)
Vector128<short> SignExtendWideningUpper(Vector128<sbyte> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:SignExtendWideningUpperTest(System.Runtime.Intrinsics.Vector128`1[Int16]):System.Runtime.Intrinsics.Vector128`1[Int32]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    sxtl2   v16.4s, v0.8h
    mov     v0.16b, v16.16b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

332. Sqrt

Vector64<float> Sqrt(Vector64<float> value)

This method calculates the square root for each vector element in the value vector, stores the results in a vector and returns the result vector.

```
private Vector64<float> SqrtTest(Vector64<float> value)
{
    return AdvSimd.Arm64.Sqrt(value);
}
// value = <11.5, 12.5>
// Result = <3.391165, 3.535534>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Sqrt(Vector128<double> value)
Vector128<float> Sqrt(Vector128<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SqrtTest(System.Runtime.Intrinsics.Vector64`1[Single]):System.
Runtime.Intrinsics.Vector64`1[Single]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fsqrt  v16.2s, v0.2s
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

333. SqrtScalar

Vector64<double> SqrtScalar(Vector64<double> value)

This method calculates the square root of the value in the value vector and returns the result.

```
private Vector64<double> SqrtScalarTest(Vector64<double> value)
{
    return AdvSimd.SqrtScalar(value);
}
// value = <11.5>
// Result = <3.391164991562634>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<float> SqrtScalar(Vector64<float> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SqrtScalarTest(System.Runtime.Intrinsics.Vector64`1[Double]):S
ystem.Runtime.Intrinsics.Vector64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fsqrt  d16, d0
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

334. Store

void Store(byte* address, Vector64<byte> source)

This method stores the source vector to memory. In below example, it would copy all the elements to the array byte[] as much it can fit. E.g. if byte[] address was of 5 elements, it would just copy 5 elements into it and if there were 10 elements, it would copy the 8 elements from source and keep remaining values in memory untouched.

```
private void StoreTest(byte* address, Vector64<byte> source)
{
    AdvSimd.Store(address, source);
}
// address = Address to byte[] where `source` needs to be stored.
// source = <11, 12, 13, 14, 15, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
void Store(double* address, Vector64<double> source)
void Store(short* address, Vector64<short> source)
void Store(int* address, Vector64<int> source)
void Store(long* address, Vector64<long> source)
void Store(sbyte* address, Vector64<sbyte> source)
void Store(float* address, Vector64<float> source)
void Store(ushort* address, Vector64<ushort> source)
void Store(uint* address, Vector64<uint> source)
void Store(ulong* address, Vector64<ulong> source)
void Store(byte* address, Vector128<byte> source)
void Store(double* address, Vector128<double> source)
void Store(short* address, Vector128<short> source)
void Store(int* address, Vector128<int> source)
void Store(long* address, Vector128<long> source)
void Store(sbyte* address, Vector128<sbyte> source)
void Store(float* address, Vector128<float> source)
void Store(ushort* address, Vector128<ushort> source)
void Store(uint* address, Vector128<uint> source)
void Store(ulong* address, Vector128<ulong> source)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:StoreTest(long,System.Runtime.Intrinsics.Vector64`1[Byte])
;
; V00 arg0      [V00,T00] ( 3, 3 ) long -> x0
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V02 OutArgs  [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
```

```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    st1    {v0.8b}, [x0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

335. StorePair

void StorePair(byte* address, Vector64<byte> value1, Vector64<byte> value2)

Store pair of vectors value1 and value2 in memory whose address is given in address.

```
private void StorePairTest(byte* address, Vector64<byte> value1,
Vector64<byte> value2)
{
    AdvSimd.Arm64.StorePair(address, value1, value2);
}
// address = Address of `byte[]` or Likewise
// value1 = <11, 12, 13, 14, 15, 16, 17, 18>
// value2 = <21, 22, 23, 24, 25, 26, 27, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
void StorePair(double* address, Vector64<double> value1, Vector64<double>
value2)
void StorePair(short* address, Vector64<short> value1, Vector64<short>
value2)
void StorePair(int* address, Vector64<int> value1, Vector64<int> value2)
void StorePair(long* address, Vector64<long> value1, Vector64<long> value2)
void StorePair(sbyte* address, Vector64<sbyte> value1, Vector64<sbyte>
value2)
void StorePair(float* address, Vector64<float> value1, Vector64<float>
value2)
void StorePair(ushort* address, Vector64<ushort> value1, Vector64<ushort>
value2)
void StorePair(uint* address, Vector64<uint> value1, Vector64<uint> value2)
void StorePair(ulong* address, Vector64<ulong> value1, Vector64<ulong>
value2)
void StorePair(byte* address, Vector128<byte> value1, Vector128<byte> value2)
void StorePair(double* address, Vector128<double> value1, Vector128<double>
value2)
void StorePair(short* address, Vector128<short> value1, Vector128<short>
value2)
void StorePair(int* address, Vector128<int> value1, Vector128<int> value2)
void StorePair(long* address, Vector128<long> value1, Vector128<long> value2)
void StorePair(sbyte* address, Vector128<sbyte> value1, Vector128<sbyte>
value2)
void StorePair(float* address, Vector128<float> value1, Vector128<float>
value2)
void StorePair(ushort* address, Vector128<ushort> value1, Vector128<ushort>
value2)
void StorePair(uint* address, Vector128<uint> value1, Vector128<uint> value2)
void StorePair(ulong* address, Vector128<ulong> value1, Vector128<ulong>
value2)
```

See Microsoft docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:StorePairTest(long,System.Runtime.Intrinsics.Vector64`1[Byte],
System.Runtime.Intrinsics.Vector64`1[Byte])
;
; V00 arg0      [V00,T00] ( 3, 3 )    long  ->  x0
; V01 arg1      [V01,T01] ( 3, 3 )    simd8  ->  d0
HFA(simd8)
; V02 arg2      [V02,T02] ( 3, 3 )    simd8  ->  d1
HFA(simd8)
;# V03 OutArgs  [V03    ] ( 1, 1 )    lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    stp    d0, d1, [x0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

336. StorePairNonTemporal

void StorePairNonTemporal(byte* address, Vector64<byte> value1, Vector64<byte> value2)

Store pair of vectors value1 and value2, with non-temporal hint, in memory whose address is given in address.

```
private void StorePairNonTemporalTest(byte* address, Vector64<byte> value1,
Vector64<byte> value2)
{
    AdvSimd.Arm64.StorePairNonTemporal(address, value1, value2);
}
// address = <address>
// value1 = <11, 12, 13, 14, 15, 16, 17, 18>
// value2 = <21, 22, 23, 24, 25, 26, 27, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
void StorePairNonTemporal(double* address, Vector64<double> value1,
Vector64<double> value2)
void StorePairNonTemporal(short* address, Vector64<short> value1,
Vector64<short> value2)
void StorePairNonTemporal(int* address, Vector64<int> value1, Vector64<int>
value2)
void StorePairNonTemporal(long* address, Vector64<long> value1,
Vector64<long> value2)
void StorePairNonTemporal(sbyte* address, Vector64<sbyte> value1,
Vector64<sbyte> value2)
void StorePairNonTemporal(float* address, Vector64<float> value1,
Vector64<float> value2)
void StorePairNonTemporal(ushort* address, Vector64<ushort> value1,
Vector64<ushort> value2)
void StorePairNonTemporal(uint* address, Vector64<uint> value1,
Vector64<uint> value2)
void StorePairNonTemporal(ulong* address, Vector64<ulong> value1,
Vector64<ulong> value2)
void StorePairNonTemporal(byte* address, Vector128<byte> value1,
Vector128<byte> value2)
void StorePairNonTemporal(double* address, Vector128<double> value1,
Vector128<double> value2)
void StorePairNonTemporal(short* address, Vector128<short> value1,
Vector128<short> value2)
void StorePairNonTemporal(int* address, Vector128<int> value1, Vector128<int>
value2)
void StorePairNonTemporal(long* address, Vector128<long> value1,
Vector128<long> value2)
void StorePairNonTemporal(sbyte* address, Vector128<sbyte> value1,
Vector128<sbyte> value2)
void StorePairNonTemporal(float* address, Vector128<float> value1,
```

```

Vector128<float> value2)
void StorePairNonTemporal(ushort* address, Vector128<ushort> value1,
Vector128<ushort> value2)
void StorePairNonTemporal(uint* address, Vector128<uint> value1,
Vector128<uint> value2)
void StorePairNonTemporal(ulong* address, Vector128<ulong> value1,
Vector128<ulong> value2)

```

See Microsoft docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:StorePairNonTemporalTest(long,System.Runtime.Intrinsics.Vector
64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte])
;
; V00 arg0          [V00,T00] ( 3, 3 )    long  ->  x0
; V01 arg1          [V01,T01] ( 3, 3 )    simd8  ->  d0
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 )    simd8  ->  d1
HFA(simd8)
;# V03 OutArgs      [V03      ] ( 1, 1 )    lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp      fp, lr, [sp,#-16]!
        mov      fp, sp
        stnp     d0, d1, [x0]
        ldp      fp, lr, [sp],#16
        ret      lr

; Total bytes of code 20, prolog size 8

```

337. StorePairScalar

void StorePairScalar(int* address, Vector64<int> value1, Vector64<int> value2)

This method stores a pair of value1 and value2 vectors to memory whose address is passed in address. Only the 0th element from each vector is stored in address.

```
private void StorePairScalarTest(int* address, Vector64<int> value1,
Vector64<int> value2)
{
    AdvSimd.Arm64.StorePairScalar(address, value1, value2);
}
// address = <address>
// value1 = <11, 12>
// value2 = <21, 22>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
void StorePairScalar(float* address, Vector64<float> value1, Vector64<float>
value2)
void StorePairScalar(uint* address, Vector64<uint> value1, Vector64<uint>
value2)
```

See Microsoft docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:StorePairScalarTest(long,System.Runtime.Intrinsics.Vector64`1[
Int32],System.Runtime.Intrinsics.Vector64`1[Int32])
;
; V00 arg0          [V00,T00] ( 3, 3 )    long ->  x0
; V01 arg1          [V01,T01] ( 3, 3 )    simd8 ->  d0
HFA(simd8)
; V02 arg2          [V02,T02] ( 3, 3 )    simd8 ->  d1
HFA(simd8)
;# V03 OutArgs      [V03      ] ( 1, 1 )    lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    stp    s0, s1, [x0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

338. StorePairScalarNonTemporal

```
void StorePairScalarNonTemporal(int* address, Vector64<int> value1,
Vector64<int> value2)
```

This method stores a pair of value1 and value2 vectors to memory, issuing a hint to the memory system that the access is non-temporal, whose address is passed in address. Only the 0th element from each vector is stored in address.

```
private void StorePairScalarNonTemporalTest(int* address, Vector64<int>
value1, Vector64<int> value2)
{
    AdvSimd.Arm64.StorePairScalarNonTemporal(address, value1, value2);
}
// address = <address>
// value1 = <11, 12>
// value2 = <21, 22>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
void StorePairScalarNonTemporal(float* address, Vector64<float> value1,
Vector64<float> value2)
void StorePairScalarNonTemporal(uint* address, Vector64<uint> value1,
Vector64<uint> value2)
```

See Microsoft docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:StorePairScalarNonTemporalTest(long,System.Runtime.Intrinsics.
Vector64`1[Int32],System.Runtime.Intrinsics.Vector64`1[Int32])
;
; V00 arg0      [V00,T00] ( 3, 3 ) long -> x0
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V03 OutArgs  [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    stnp   s0, s1, [x0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

339. StoreSelectedScalar

void StoreSelectedScalar(byte* address, Vector64<byte> value, byte index)

This method stores the specified element index of the source vector to memory. In below example, it would copy 14 to 0th element of byte[] whose address is passed.

```
private void StoreSelectedScalarTest(byte* address, Vector64<byte> value,
byte index)
{
    AdvSimd.StoreSelectedScalar(address, value, 3);
}
// address = Address to memory e.g. address of byte[]
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// index = 3
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
void StoreSelectedScalar(short* address, Vector64<short> value, byte index)
void StoreSelectedScalar(int* address, Vector64<int> value, byte index)
void StoreSelectedScalar(sbyte* address, Vector64<sbyte> value, byte index)
void StoreSelectedScalar(float* address, Vector64<float> value, byte index)
void StoreSelectedScalar(ushort* address, Vector64<ushort> value, byte index)
void StoreSelectedScalar(uint* address, Vector64<uint> value, byte index)
void StoreSelectedScalar(byte* address, Vector128<byte> value, byte index)
void StoreSelectedScalar(double* address, Vector128<double> value, byte
index)
void StoreSelectedScalar(short* address, Vector128<short> value, byte index)
void StoreSelectedScalar(int* address, Vector128<int> value, byte index)
void StoreSelectedScalar(long* address, Vector128<long> value, byte index)
void StoreSelectedScalar(sbyte* address, Vector128<sbyte> value, byte index)
void StoreSelectedScalar(float* address, Vector128<float> value, byte index)
void StoreSelectedScalar(ushort* address, Vector128<ushort> value, byte
index)
void StoreSelectedScalar(uint* address, Vector128<uint> value, byte index)
void StoreSelectedScalar(ulong* address, Vector128<ulong> value, byte index)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.StoreSelectedScalarTest(long,System.Runtime.Intrinsics.Vector6
4`1[Byte],ubyte)
;
; V00 arg0          [V00,T00] ( 3, 3 )    long  ->  x0
; V01 arg1          [V01,T01] ( 3, 3 )    simd8  ->  d0
HFA(simd8)
;* V02 arg2          [V02  ] ( 0, 0 )    ubyte  ->  zero-ref
;# V03 OutArgs       [V03  ] ( 1, 1 )    lclBlk ( 0) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    st1    {v0.b}[3], [x0]
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 20, prolog size 8
```

340. Subtract

Vector64<byte> Subtract(Vector64<byte> left, Vector64<byte> right)

This method subtracts each vector element in the right vector from the corresponding vector element in the left vector, stores the results in a vector and returns the result vector.

```
private Vector64<byte> SubtractTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.Subtract(left, right);
}
// left = <21, 22, 23, 24, 25, 26, 27, 18>
// right = <11, 12, 13, 14, 15, 16, 17, 28>
// Result = <10, 10, 10, 10, 10, 10, 10, 246>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> Subtract(Vector64<short> left, Vector64<short> right)
Vector64<int> Subtract(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> Subtract(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> Subtract(Vector64<float> left, Vector64<float> right)
Vector64<ushort> Subtract(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> Subtract(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> Subtract(Vector128<byte> left, Vector128<byte> right)
Vector128<short> Subtract(Vector128<short> left, Vector128<short> right)
Vector128<int> Subtract(Vector128<int> left, Vector128<int> right)
Vector128<long> Subtract(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> Subtract(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> Subtract(Vector128<float> left, Vector128<float> right)
Vector128<ushort> Subtract(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> Subtract(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> Subtract(Vector128<ulong> left, Vector128<ulong> right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<double> Subtract(Vector128<double> left, Vector128<double> right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SubtractTest(System.Runtime.Intrinsics.Vector64`1[Byte],System
.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[By
te]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
```



```

HFA(simd8)
;# V02 OutArgs      [V02      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp    fp, lr, [sp,#-16]!
        mov    fp, sp
        sub    v16.8b, v0.8b, v1.8b
        mov    v0.8b, v16.8b
        ldp    fp, lr, [sp],#16
        ret    lr

; Total bytes of code 24, prolog size 8

```

341. SubtractHighNarrowingLower

Vector64<byte> SubtractHighNarrowingLower(Vector128<ushort> left, Vector128<ushort> right)

This method subtracts each vector element in the right vector from the corresponding vector element in the left vector, stores the most significant half of the result into a vector, and writes the vector to the result vector.

```
private Vector64<byte> SubtractHighNarrowingLowerTest(Vector128<ushort> left,
Vector128<ushort> right)
{
    return AdvSimd.SubtractHighNarrowingLower(left, right);
}
// left = <1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000>
// right = <10, 20, 30, 40, 50, 60, 70, 80>
// Result = <3, 7, 11, 15, 19, 23, 27, 30>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> SubtractHighNarrowingLower(Vector128<int> left,
Vector128<int> right)
Vector64<int> SubtractHighNarrowingLower(Vector128<long> left,
Vector128<long> right)
Vector64<sbyte> SubtractHighNarrowingLower(Vector128<short> left,
Vector128<short> right)
Vector64<ushort> SubtractHighNarrowingLower(Vector128<uint> left,
Vector128<uint> right)
Vector64<uint> SubtractHighNarrowingLower(Vector128<ulong> left,
Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SubtractHighNarrowingLowerTest(System.Runtime.Intrinsics.Vector128`1[UInt16],System.Runtime.Intrinsics.Vector128`1[UInt16]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    subhn  v16.8b, v0.8h, v1.8h
```

```
mov    v0.8b, v16.8b
ldp    fp, lr, [sp],#16
ret    lr
```

; Total bytes of code 24, prolog size 8

342. SubtractHighNarrowingUpper

Vector128<byte> SubtractHighNarrowingUpper(Vector64<byte> lower, Vector128<ushort> left, Vector128<ushort> right)

This method subtracts each vector element in the right vector from the corresponding vector element in the left vector, stores the most significant half of the result into a vector, and writes the vector to the upper half of the result vector.

```
private Vector128<byte> SubtractHighNarrowingUpperTest(Vector64<byte> lower,
Vector128<ushort> left, Vector128<ushort> right)
{
    return AdvSimd.SubtractHighNarrowingUpper(lower, left, right);
}
// lower = <5, 5, 5, 5, 5, 5, 5, 5>
// left = <1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000>
// right = <10, 20, 30, 40, 50, 60, 70, 80>
// Result = <5, 5, 5, 5, 5, 5, 5, 5, 3, 7, 11, 15, 19, 23, 27, 30>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> SubtractHighNarrowingUpper(Vector64<short> lower,
Vector128<int> left, Vector128<int> right)
Vector128<int> SubtractHighNarrowingUpper(Vector64<int> lower,
Vector128<long> left, Vector128<long> right)
Vector128<sbyte> SubtractHighNarrowingUpper(Vector64<sbyte> lower,
Vector128<short> left, Vector128<short> right)
Vector128<ushort> SubtractHighNarrowingUpper(Vector64<ushort> lower,
Vector128<uint> left, Vector128<uint> right)
Vector128<uint> SubtractHighNarrowingUpper(Vector64<uint> lower,
Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SubtractHighNarrowingUpperTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16],System.Runtime.Intrinsics.Vector128`1[UInt16]):System.Runtime.Intrinsics.Vector128`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2          [V02,T02] ( 3, 3 ) simd16 -> d2
HFA(simd16)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    subhn2 v0.16b, v1.8h, v2.8h  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 20, prolog size 8

343. SubtractRoundedHighNarrowingLower

Vector64<byte> SubtractRoundedHighNarrowingLower(Vector128<ushort> left, Vector128<ushort> right)

This method subtracts each vector element of the right vector from the corresponding vector element of the left vector, stores the most significant half of the result into a vector, and writes the vector to the result vector. The results are rounded.

```
private Vector64<byte>
SubtractRoundedHighNarrowingLowerTest(Vector128<ushort> left,
Vector128<ushort> right)
{
    return AdvSimd.SubtractRoundedHighNarrowingLower(left, right);
}
// left = <1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000>
// right = <10, 20, 30, 40, 50, 60, 70, 80>
// Result = <4, 8, 12, 15, 19, 23, 27, 31>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> SubtractRoundedHighNarrowingLower(Vector128<int> left,
Vector128<int> right)
Vector64<int> SubtractRoundedHighNarrowingLower(Vector128<long> left,
Vector128<long> right)
Vector64<sbyte> SubtractRoundedHighNarrowingLower(Vector128<short> left,
Vector128<short> right)
Vector64<ushort> SubtractRoundedHighNarrowingLower(Vector128<uint> left,
Vector128<uint> right)
Vector64<uint> SubtractRoundedHighNarrowingLower(Vector128<ulong> left,
Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SubtractRoundedHighNarrowingLowerTest(System.Runtime.Intrinsic
s.Vector128`1[UInt16],System.Runtime.Intrinsics.Vector128`1[UInt16]):System.R
untime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
```

```
    rsubhn    v16.8b, v0.8h, v1.8h
    mov       v0.8b, v16.8b
    ldp       fp, lr, [sp],#16
    ret       lr
```

; Total bytes of code 24, prolog size 8

344. SubtractRoundedHighNarrowingUpper

Vector128<byte> SubtractRoundedHighNarrowingUpper(Vector64<byte> lower, Vector128<ushort> left, Vector128<ushort> right)

This method subtracts each vector element in the right vector from the corresponding vector element in the left vector, stores the most significant half of the result into a vector, and writes the vector to the upper half of the result vector. The results are rounded.

```
private Vector128<byte> SubtractRoundedHighNarrowingUpperTest(Vector64<byte>
lower, Vector128<ushort> left, Vector128<ushort> right)
{
    return AdvSimd.SubtractRoundedHighNarrowingUpper(lower, left, right);
}
// lower = <5, 5, 5, 5, 5, 5, 5, 5>
// left = <1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000>
// right = <10, 20, 30, 40, 50, 60, 70, 80>
// Result = <5, 5, 5, 5, 5, 5, 5, 5, 4, 8, 12, 15, 19, 23, 27, 31>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<short> SubtractRoundedHighNarrowingUpper(Vector64<short> lower,
Vector128<int> left, Vector128<int> right)
Vector128<int> SubtractRoundedHighNarrowingUpper(Vector64<int> lower,
Vector128<long> left, Vector128<long> right)
Vector128<sbyte> SubtractRoundedHighNarrowingUpper(Vector64<sbyte> lower,
Vector128<short> left, Vector128<short> right)
Vector128<ushort> SubtractRoundedHighNarrowingUpper(Vector64<ushort> lower,
Vector128<uint> left, Vector128<uint> right)
Vector128<uint> SubtractRoundedHighNarrowingUpper(Vector64<uint> lower,
Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:SubtractRoundedHighNarrowingUpperTest(System.Runtime.Intrinsic
s.Vector64`1[Byte],System.Runtime.Intrinsics.Vector128`1[UInt16],System.Runti
me.Intrinsics.Vector128`1[UInt16]):System.Runtime.Intrinsics.Vector128`1[Byte
]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2          [V02,T02] ( 3, 3 ) simd16 -> d2
HFA(simd16)
;# V03 OutArgs      [V03 ] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00]
"OutgoingArgSpace"
```



```
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    rsubhn2 v0.16b, v1.8h, v2.8h
    ldp    fp, lr, [sp],#16
    ret    lr
```

```
; Total bytes of code 20, prolog size 8
```

345. SubtractSaturate

Vector64<byte> SubtractSaturate(Vector64<byte> left, Vector64<byte> right)

This method subtracts the element values of the right vector from the corresponding element values of the left vector, stores the results in a vector and returns the result vector. As per ARM docs, if overflow occurs with any of the results, those results are saturated.

```
private Vector64<byte> SubtractSaturateTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.SubtractSaturate(left, right);
}
// left = <250, 240, 230, 220, 210, 200, 190, 180>
// right = <10, 20, 30, 40, 50, 255, 250, 250>
// Result = <240, 220, 200, 180, 160, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<short> SubtractSaturate(Vector64<short> left, Vector64<short> right)
Vector64<int> SubtractSaturate(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> SubtractSaturate(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<ushort> SubtractSaturate(Vector64<ushort> left, Vector64<ushort>
right)
Vector64<uint> SubtractSaturate(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> SubtractSaturate(Vector128<byte> left, Vector128<byte> right)
Vector128<short> SubtractSaturate(Vector128<short> left, Vector128<short>
right)
Vector128<int> SubtractSaturate(Vector128<int> left, Vector128<int> right)
Vector128<long> SubtractSaturate(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> SubtractSaturate(Vector128<sbyte> left, Vector128<sbyte>
right)
Vector128<ushort> SubtractSaturate(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> SubtractSaturate(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> SubtractSaturate(Vector128<ulong> left, Vector128<ulong>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SubtractSaturateTest(System.Runtime.Intrinsics.Vector64`1[Byte
],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vecto
r64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
```

```

; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uqsub  v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

346. SubtractSaturateScalar

Vector64<long> SubtractSaturateScalar(Vector64<long> left, Vector64<long> right)

This method subtracts the element values of the right vector from the corresponding element values of the left vector, stores the results in a vector and returns the result vector. As per ARM docs, if overflow occurs with any of the results, those results are saturated.

```
private Vector64<long> SubtractSaturateScalarTest(Vector64<long> left,
Vector64<long> right)
{
    return AdvSimd.SubtractSaturateScalar(left, right);
}
// left = <500>
// right = <-200>
// Result = <700>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<ulong> SubtractSaturateScalar(Vector64<ulong> left, Vector64<ulong>
right)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<byte> SubtractSaturateScalar(Vector64<byte> left, Vector64<byte>
right)
Vector64<short> SubtractSaturateScalar(Vector64<short> left, Vector64<short>
right)
Vector64<int> SubtractSaturateScalar(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> SubtractSaturateScalar(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector64<ushort> SubtractSaturateScalar(Vector64<ushort> left,
Vector64<ushort> right)
Vector64<uint> SubtractSaturateScalar(Vector64<uint> left, Vector64<uint>
right)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SubtractSaturateScalarTest(System.Runtime.Intrinsics.Vector64`
1[Int64],System.Runtime.Intrinsics.Vector64`1[Int64]):System.Runtime.Intrinsi
cs.Vector64`1[Int64]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
```

```

;# V02 OutArgs      [V02      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp      fp, lr, [sp,#-16]!
    mov      fp, sp
    sqsub    d16, d0, d1
    mov      v0.8b, v16.8b
    ldp      fp, lr, [sp],#16
    ret      lr

; Total bytes of code 24, prolog size 8

```

347. SubtractScalar

Vector64<double> SubtractScalar(Vector64<double> left, Vector64<double> right)

This method subtracts the floating-point value of the right vector from the floating-point value of the left vector and returns the result.

```
private Vector64<double> SubtractScalarTest(Vector64<double> left,
Vector64<double> right)
{
    return AdvSimd.SubtractScalar(left, right);
}
// left = <11.5>
// right = <11.5>
// Result = <0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<long> SubtractScalar(Vector64<long> left, Vector64<long> right)
Vector64<float> SubtractScalar(Vector64<float> left, Vector64<float> right)
Vector64<ulong> SubtractScalar(Vector64<ulong> left, Vector64<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SubtractScalarTest(System.Runtime.Intrinsics.Vector64`1[Double
],System.Runtime.Intrinsics.Vector64`1[Double]):System.Runtime.Intrinsics.Vec
tor64`1[Double]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    fsub    d16, d0, d1
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

348. SubtractWideningLower

Vector128<ushort> SubtractWideningLower(Vector64<byte> left, Vector64<byte> right)

This method subtracts each vector element in the right from the corresponding vector element of the left vector, stores the results in a vector and returns the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input vector element's size byte.

```
private Vector128<ushort> SubtractWideningLowerTest(Vector64<byte> left,
Vector64<byte> right)
{
    return AdvSimd.SubtractWideningLower(left, right);
}
// left = <250, 240, 230, 220, 210, 200, 190, 180>
// right = <10, 20, 30, 40, 50, 255, 250, 250>
// Result = <240, 220, 200, 180, 160, 65481, 65476, 65466>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> SubtractWideningLower(Vector64<short> left, Vector64<short>
right)
Vector128<long> SubtractWideningLower(Vector64<int> left, Vector64<int>
right)
Vector128<short> SubtractWideningLower(Vector64<sbyte> left, Vector64<sbyte>
right)
Vector128<uint> SubtractWideningLower(Vector64<ushort> left, Vector64<ushort>
right)
Vector128<ulong> SubtractWideningLower(Vector64<uint> left, Vector64<uint>
right)
Vector128<short> SubtractWideningLower(Vector128<short> left, Vector64<sbyte>
right)
Vector128<int> SubtractWideningLower(Vector128<int> left, Vector64<short>
right)
Vector128<long> SubtractWideningLower(Vector128<long> left, Vector64<int>
right)
Vector128<ushort> SubtractWideningLower(Vector128<ushort> left,
Vector64<byte> right)
Vector128<uint> SubtractWideningLower(Vector128<uint> left, Vector64<ushort>
right)
Vector128<ulong> SubtractWideningLower(Vector128<ulong> left, Vector64<uint>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.SubtractWideningLowerTest(System.Runtime.Intrinsics.Vector64`1
```

```
[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.  
Vector128`1[UInt16]
```

```
;  
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0  
HFA(simd8)  
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1  
HFA(simd8)  
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]  
"OutgoingArgSpace"  
; Lcl frame size = 0  
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    usubl  v16.8h, v0.8b, v1.8b  
    mov    v0.16b, v16.16b  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

```
; Total bytes of code 24, prolog size 8
```

349. SubtractWideningUpper

Vector128<ushort> SubtractWideningUpper(Vector128<byte> left, Vector128<byte> right)

This method subtracts each vector element in the upper-half of *right* from the corresponding vector element of the *left* vector, stores the results in a vector and returns the result vector. As seen in below example, the result vector element's size *ushort* is twice as long as the input vector element's size *byte*.

```
private Vector128<ushort> SubtractWideningUpperTest(Vector128<byte> left,
Vector128<byte> right)
{
    return AdvSimd.SubtractWideningUpper(left, right);
}
// left = <250, 240, 230, 220, 210, 200, 190, 180, 100, 100, 100, 100, 100, 100, 100>
// right = <10, 20, 30, 40, 50, 255, 250, 250, 50, 50, 50, 50, 200, 200, 200, 200>
// Result = <50, 50, 50, 50, 65436, 65436, 65436, 65436>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> SubtractWideningUpper(Vector128<short> left, Vector128<short>
right)
Vector128<short> SubtractWideningUpper(Vector128<short> left,
Vector128<sbyte> right)
Vector128<int> SubtractWideningUpper(Vector128<int> left, Vector128<short>
right)
Vector128<long> SubtractWideningUpper(Vector128<int> left, Vector128<int>
right)
Vector128<long> SubtractWideningUpper(Vector128<long> left, Vector128<int>
right)
Vector128<short> SubtractWideningUpper(Vector128<sbyte> left,
Vector128<sbyte> right)
Vector128<ushort> SubtractWideningUpper(Vector128<ushort> left,
Vector128<byte> right)
Vector128<uint> SubtractWideningUpper(Vector128<ushort> left,
Vector128<ushort> right)
Vector128<uint> SubtractWideningUpper(Vector128<uint> left, Vector128<ushort>
right)
Vector128<ulong> SubtractWideningUpper(Vector128<uint> left, Vector128<uint>
right)
Vector128<ulong> SubtractWideningUpper(Vector128<ulong> left, Vector128<uint>
right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```

; Assembly listing for method
AdvSimdMethods:SubtractWideningUpperTest(System.Runtime.Intrinsics.Vector128`
1[Byte],System.Runtime.Intrinsics.Vector128`1[Byte]):System.Runtime.Intrinsic
s.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
;# V02 OutArgs      [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    usubl2 v16.8h, v0.16b, v1.16b
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

350. TransposeEven

Vector64<byte> TransposeEven(Vector64<byte> left, Vector64<byte> right)

This method reads corresponding even-numbered vector elements from `left` and `right`, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the result vector. Vector elements from the `left` vector are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the `right` vector are placed into odd-numbered elements of the destination vector. By using this method with `TransposeOdd()`, a 2 x 2 matrix can be transposed.

```
private Vector64<byte> TransposeEvenTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.Arm64.TransposeEven(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <11, 21, 13, 23, 15, 25, 17, 27>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<short> TransposeEven(Vector64<short> left, Vector64<short> right)
Vector64<int> TransposeEven(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> TransposeEven(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> TransposeEven(Vector64<float> left, Vector64<float> right)
Vector64<ushort> TransposeEven(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> TransposeEven(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> TransposeEven(Vector128<byte> left, Vector128<byte> right)
Vector128<double> TransposeEven(Vector128<double> left, Vector128<double>
right)
Vector128<short> TransposeEven(Vector128<short> left, Vector128<short> right)
Vector128<int> TransposeEven(Vector128<int> left, Vector128<int> right)
Vector128<long> TransposeEven(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> TransposeEven(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> TransposeEven(Vector128<float> left, Vector128<float> right)
Vector128<ushort> TransposeEven(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> TransposeEven(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> TransposeEven(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:TransposeEvenTest(System.Runtime.Intrinsics.Vector64`1[Byte],S
ystem.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64
`1[Byte]
;
```

```

; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    trn1   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

351. TransposeOdd

Vector64<byte> TransposeOdd(Vector64<byte> left, Vector64<byte> right)

This method reads corresponding odd-numbered vector elements from the left and right vectors, places each result into consecutive elements of a vector, and writes the vector to the result vector. Vector elements from the left vector are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the right vector are placed into odd-numbered elements of the destination vector. By using this method with TransposeEven(), a 2 x 2 matrix can be transposed.

```
private Vector64<byte> TransposeOddTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.Arm64.TransposeOdd(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <12, 22, 14, 24, 16, 26, 18, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector64<short> TransposeOdd(Vector64<short> left, Vector64<short> right)
Vector64<int> TransposeOdd(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> TransposeOdd(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> TransposeOdd(Vector64<float> left, Vector64<float> right)
Vector64<ushort> TransposeOdd(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> TransposeOdd(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> TransposeOdd(Vector128<byte> left, Vector128<byte> right)
Vector128<double> TransposeOdd(Vector128<double> left, Vector128<double>
right)
Vector128<short> TransposeOdd(Vector128<short> left, Vector128<short> right)
Vector128<int> TransposeOdd(Vector128<int> left, Vector128<int> right)
Vector128<long> TransposeOdd(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> TransposeOdd(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> TransposeOdd(Vector128<float> left, Vector128<float> right)
Vector128<ushort> TransposeOdd(Vector128<ushort> left, Vector128<ushort>
right)
Vector128<uint> TransposeOdd(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> TransposeOdd(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:TransposeOddTest(System.Runtime.Intrinsics.Vector64`1[Byte],Sy
stem.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`
1[Byte]
;
```

```

; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs  [V02   ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    trn2   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

```

```

; Total bytes of code 24, prolog size 8

```

352. UnzipEven

Vector64<byte> UnzipEven(Vector64<byte> left, Vector64<byte> right)

This method reads corresponding even-numbered vector elements from the left and right vectors, starting at zero, places the result from the left vector into consecutive elements in the lower half of a vector, and the result from the right vector into consecutive elements in the upper half of a vector, and writes the vector to the result vector. This method can be used with `UnzipOdd()` to de-interleave two vectors.

```
private Vector64<byte> UnzipEvenTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.Arm64.UnzipEven(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <11, 13, 15, 17, 21, 23, 25, 27>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<short> UnzipEven(Vector64<short> left, Vector64<short> right)
Vector64<int> UnzipEven(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> UnzipEven(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> UnzipEven(Vector64<float> left, Vector64<float> right)
Vector64<ushort> UnzipEven(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> UnzipEven(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> UnzipEven(Vector128<byte> left, Vector128<byte> right)
Vector128<double> UnzipEven(Vector128<double> left, Vector128<double> right)
Vector128<short> UnzipEven(Vector128<short> left, Vector128<short> right)
Vector128<int> UnzipEven(Vector128<int> left, Vector128<int> right)
Vector128<long> UnzipEven(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> UnzipEven(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> UnzipEven(Vector128<float> left, Vector128<float> right)
Vector128<ushort> UnzipEven(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> UnzipEven(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> UnzipEven(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:UnzipEvenTest(System.Runtime.Intrinsics.Vector64`1[Byte],Syste
m.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[B
yte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
```

```

HFA(simd8)
;# V02 OutArgs      [V02      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp    fp, lr, [sp,#-16]!
        mov    fp, sp
        uzp1   v16.8b, v0.8b, v1.8b
        mov    v0.8b, v16.8b
        ldp    fp, lr, [sp],#16
        ret    lr

; Total bytes of code 24, prolog size 8

```

353. UnzipOdd

Vector64<byte> UnzipOdd(Vector64<byte> left, Vector64<byte> right)

This method reads corresponding odd-numbered vector elements from the left and right vectors, places the result from the left vector into consecutive elements in the lower half of a vector, and the result from the right vector into consecutive elements in the upper half of a vector, and writes the vector to the result vector. This method can be used with `UnzipEven()` to de-interleave two vectors.

```
private Vector64<byte> UnzipOddTest(Vector64<byte> left, Vector64<byte>
right)
{
    return AdvSimd.Arm64.UnzipOdd(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <12, 14, 16, 18, 22, 24, 26, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<short> UnzipOdd(Vector64<short> left, Vector64<short> right)
Vector64<int> UnzipOdd(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> UnzipOdd(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> UnzipOdd(Vector64<float> left, Vector64<float> right)
Vector64<ushort> UnzipOdd(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> UnzipOdd(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> UnzipOdd(Vector128<byte> left, Vector128<byte> right)
Vector128<double> UnzipOdd(Vector128<double> left, Vector128<double> right)
Vector128<short> UnzipOdd(Vector128<short> left, Vector128<short> right)
Vector128<int> UnzipOdd(Vector128<int> left, Vector128<int> right)
Vector128<long> UnzipOdd(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> UnzipOdd(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> UnzipOdd(Vector128<float> left, Vector128<float> right)
Vector128<ushort> UnzipOdd(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> UnzipOdd(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> UnzipOdd(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:UnzipOddTest(System.Runtime.Intrinsics.Vector64`1[Byte],System
.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[By
te]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
```

```

HFA(simd8)
;# V02 OutArgs      [V02      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
        stp    fp, lr, [sp,#-16]!
        mov    fp, sp
        uzp2   v16.8b, v0.8b, v1.8b
        mov    v0.8b, v16.8b
        ldp    fp, lr, [sp],#16
        ret    lr

; Total bytes of code 24, prolog size 8

```

354. VectorTableLookup

Vector64<byte> VectorTableLookup(Vector128<byte> table, Vector64<byte> byteIndexes)

This method reads each value from the vector elements in the byteIndexes vector, uses each result as an index to perform a lookup in a table of bytes that is described by table vector, places the lookup result in a vector, and writes the vector to the result vector. If an index is out of range for the table, the result for that lookup is 0.

```
private Vector64<byte> VectorTableLookupTest(Vector128<byte> table,
Vector64<byte> byteIndexes)
{
    return AdvSimd.VectorTableLookup(table, byteIndexes);
}
// table = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// byteIndexes = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <22, 23, 24, 25, 26, 0, 0, 0>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<sbyte> VectorTableLookup(Vector128<sbyte> table, Vector64<sbyte>
byteIndexes)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<byte> VectorTableLookup(Vector128<byte> table, Vector128<byte>
byteIndexes)
Vector128<sbyte> VectorTableLookup(Vector128<sbyte> table, Vector128<sbyte>
byteIndexes)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:VectorTableLookupTest(System.Runtime.Intrinsics.Vector128`1[By
te],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vec
tor64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    tbl    v16.8b, {v0.16b}, v1.8b
    mov    v0.8b, v16.8b
```

```
    ldp    fp, lr, [sp],#16
    ret    lr
```

; Total bytes of code 24, prolog size 8

355. VectorTableLookupExtension

Vector64<byte> VectorTableLookupExtension(Vector64<byte> defaultValues, Vector128<byte> table, Vector64<byte> byteIndexes)

This method reads each value from the vector elements in the byteIndexes vector, uses each result as an index to perform a lookup in a table of bytes that is described by table vector, places the lookup result in a vector, and writes the vector to the result vector. If an index is out of range for the table, the value from defaultValue is picked.

```
private Vector64<byte> VectorTableLookupExtensionTest(Vector64<byte>
defaultValues, Vector128<byte> table, Vector64<byte> byteIndexes)
{
    return AdvSimd.VectorTableLookupExtension(defaultValues, table,
byteIndexes);
}
// defaultValues = <5, 5, 5, 5, 5, 5, 5, 5>
// table = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// byteIndexes = <1, 15, 4, 3, 8, 19, 1, 0>
// Result = <12, 26, 15, 14, 19, 5, 12, 11>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<sbyte> VectorTableLookupExtension(Vector64<sbyte> defaultValues,
Vector128<sbyte> table, Vector64<sbyte> byteIndexes)

// class System.Runtime.Intrinsics.AdvSimd.Arm64
Vector128<byte> VectorTableLookupExtension(Vector128<byte> defaultValues,
Vector128<byte> table, Vector128<byte> byteIndexes)
Vector128<sbyte> VectorTableLookupExtension(Vector128<sbyte> defaultValues,
Vector128<sbyte> table, Vector128<sbyte> byteIndexes)
```

See Microsoft docs [here](#) and [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:VectorTableLookupExtensionTest(System.Runtime.Intrinsics.Vecto
r64`1[Byte],System.Runtime.Intrinsics.Vector128`1[Byte],System.Runtime.Intrin
sics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0      [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1      [V01,T01] ( 3, 3 ) simd16 -> d1
HFA(simd16)
; V02 arg2      [V02,T02] ( 3, 3 ) simd8 -> d2
HFA(simd8)
;# V03 OutArgs  [V03 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
```

```
    stp    fp, lr, [sp,#-16]!  
    mov    fp, sp  
    tbx    v0.8b, {v1.16b}, v2.8b  
    ldp    fp, lr, [sp],#16  
    ret    lr
```

; Total bytes of code 20, prolog size 8

356. Xor

Vector64<byte> Xor(Vector64<byte> left, Vector64<byte> right)

This method performs a bitwise Exclusive OR operation between the left and right vector, and stores the results in a vector and returns the result vector.

```
private Vector64<byte> XorTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.Xor(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <30, 26, 26, 22, 22, 10, 10, 14>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector64<double> Xor(Vector64<double> left, Vector64<double> right)
Vector64<short> Xor(Vector64<short> left, Vector64<short> right)
Vector64<int> Xor(Vector64<int> left, Vector64<int> right)
Vector64<long> Xor(Vector64<long> left, Vector64<long> right)
Vector64<sbyte> Xor(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> Xor(Vector64<float> left, Vector64<float> right)
Vector64<ushort> Xor(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> Xor(Vector64<uint> left, Vector64<uint> right)
Vector64<ulong> Xor(Vector64<ulong> left, Vector64<ulong> right)
Vector128<byte> Xor(Vector128<byte> left, Vector128<byte> right)
Vector128<double> Xor(Vector128<double> left, Vector128<double> right)
Vector128<short> Xor(Vector128<short> left, Vector128<short> right)
Vector128<int> Xor(Vector128<int> left, Vector128<int> right)
Vector128<long> Xor(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> Xor(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> Xor(Vector128<float> left, Vector128<float> right)
Vector128<ushort> Xor(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> Xor(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> Xor(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:XorTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byte]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
;# V02 OutArgs      [V02 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
```

```
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    eor    v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8
```

357. ZeroExtendWideningLower

Vector128<ushort> ZeroExtendWideningLower(Vector64<byte> value)

This method copies each vector element from the value vector into a vector, and writes the vector to the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input vector element's size byte.

```
private Vector128<ushort> ZeroExtendWideningLowerTest(Vector64<byte> value)
{
    return AdvSimd.ZeroExtendWideningLower(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18>
// Result = <11, 12, 13, 14, 15, 16, 17, 18>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ZeroExtendWideningLower(Vector64<short> value)
Vector128<long> ZeroExtendWideningLower(Vector64<int> value)
Vector128<short> ZeroExtendWideningLower(Vector64<sbyte> value)
Vector128<uint> ZeroExtendWideningLower(Vector64<ushort> value)
Vector128<ulong> ZeroExtendWideningLower(Vector64<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ZeroExtendWideningLowerTest(System.Runtime.Intrinsics.Vector64
`1[Byte]):System.Runtime.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uxtl    v16.8h, v0.8b
    mov    v0.16b, v16.16b
    ldp    fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

358. ZeroExtendWideningUpper

Vector128<ushort> ZeroExtendWideningUpper(Vector128<byte> value)

This method copies each vector element from the upper-half of value vector into a vector, and writes the vector to the result vector. As seen in below example, the result vector element's size ushort is twice as long as the input vector element's size byte.

```
private Vector128<ushort> ZeroExtendWideningUpperTest(Vector128<byte> value)
{
    return AdvSimd.ZeroExtendWideningUpper(value);
}
// value = <11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26>
// Result = <19, 20, 21, 22, 23, 24, 25, 26>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.AdvSimd
Vector128<int> ZeroExtendWideningUpper(Vector128<short> value)
Vector128<long> ZeroExtendWideningUpper(Vector128<int> value)
Vector128<short> ZeroExtendWideningUpper(Vector128<sbyte> value)
Vector128<uint> ZeroExtendWideningUpper(Vector128<ushort> value)
Vector128<ulong> ZeroExtendWideningUpper(Vector128<uint> value)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods:ZeroExtendWideningUpperTest(System.Runtime.Intrinsics.Vector128`1[Byte]):System.Runtime.Intrinsics.Vector128`1[UInt16]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd16 -> d0
HFA(simd16)
;# V01 OutArgs      [V01 ] ( 1, 1 ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    uxtl2   v16.8h, v0.16b
    mov     v0.16b, v16.16b
    ldp     fp, lr, [sp],#16
    ret     lr

; Total bytes of code 24, prolog size 8
```

359. ZipHigh

Vector64<byte> ZipHigh(Vector64<byte> left, Vector64<byte> right)

This method reads adjacent vector elements from the lower half of left and right vector as pairs, interleaves the pairs and stores the results in a vector and returns the result vector. The first pair from the left vector is placed into the two lowest vector elements, with subsequent pairs taken alternately from each argument vector. This method can be used with ZipLow() to interleave two vectors.

```
private Vector64<byte> ZipHighTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.Arm64.ZipHigh(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <15, 25, 16, 26, 17, 27, 18, 28>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<short> ZipHigh(Vector64<short> left, Vector64<short> right)
Vector64<int> ZipHigh(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> ZipHigh(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> ZipHigh(Vector64<float> left, Vector64<float> right)
Vector64<ushort> ZipHigh(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> ZipHigh(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> ZipHigh(Vector128<byte> left, Vector128<byte> right)
Vector128<double> ZipHigh(Vector128<double> left, Vector128<double> right)
Vector128<short> ZipHigh(Vector128<short> left, Vector128<short> right)
Vector128<int> ZipHigh(Vector128<int> left, Vector128<int> right)
Vector128<long> ZipHigh(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> ZipHigh(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> ZipHigh(Vector128<float> left, Vector128<float> right)
Vector128<ushort> ZipHigh(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> ZipHigh(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> ZipHigh(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
; Assembly listing for method
AdvSimdMethods.ZipHighTest(System.Runtime.Intrinsics.Vector64`1[Byte],System.
Runtime.Intrinsics.Vector64`1[Byte]):System.Runtime.Intrinsics.Vector64`1[Byt
e]
;
; V00 arg0          [V00,T00] ( 3, 3 ) simd8 -> d0
HFA(simd8)
; V01 arg1          [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8)
```

```

;# V02 OutArgs      [V02      ] (  1,  1  ) lclBlk ( 0) [sp+0x00]
"OutgoingArgSpace"
; Lcl frame size = 0
    stp    fp, lr, [sp,#-16]!
    mov    fp, sp
    zip2   v16.8b, v0.8b, v1.8b
    mov    v0.8b, v16.8b
    ldp    fp, lr, [sp],#16
    ret    lr

; Total bytes of code 24, prolog size 8

```

360. ZipLow

Vector64<byte> ZipLow(Vector64<byte> left, Vector64<byte> right)

This method reads adjacent vector elements from the upper half of left and right vector as pairs, interleaves the pairs and stores the results in a vector and returns the result vector. The first pair from the left vector is placed into the two lowest vector elements, with subsequent pairs taken alternately from each argument vectors. This method can be used with ZipHigh() to interleave two vectors.

```
private Vector64<byte> ZipLowTest(Vector64<byte> left, Vector64<byte> right)
{
    return AdvSimd.Arm64.ZipLow(left, right);
}
// left = <11, 12, 13, 14, 15, 16, 17, 18>
// right = <21, 22, 23, 24, 25, 26, 27, 28>
// Result = <11, 21, 12, 22, 13, 23, 14, 24>
```

Similar APIs that operate on different sizes:

```
// class System.Runtime.Intrinsics.Arm64
Vector64<short> ZipLow(Vector64<short> left, Vector64<short> right)
Vector64<int> ZipLow(Vector64<int> left, Vector64<int> right)
Vector64<sbyte> ZipLow(Vector64<sbyte> left, Vector64<sbyte> right)
Vector64<float> ZipLow(Vector64<float> left, Vector64<float> right)
Vector64<ushort> ZipLow(Vector64<ushort> left, Vector64<ushort> right)
Vector64<uint> ZipLow(Vector64<uint> left, Vector64<uint> right)
Vector128<byte> ZipLow(Vector128<byte> left, Vector128<byte> right)
Vector128<double> ZipLow(Vector128<double> left, Vector128<double> right)
Vector128<short> ZipLow(Vector128<short> left, Vector128<short> right)
Vector128<int> ZipLow(Vector128<int> left, Vector128<int> right)
Vector128<long> ZipLow(Vector128<long> left, Vector128<long> right)
Vector128<sbyte> ZipLow(Vector128<sbyte> left, Vector128<sbyte> right)
Vector128<float> ZipLow(Vector128<float> left, Vector128<float> right)
Vector128<ushort> ZipLow(Vector128<ushort> left, Vector128<ushort> right)
Vector128<uint> ZipLow(Vector128<uint> left, Vector128<uint> right)
Vector128<ulong> ZipLow(Vector128<ulong> left, Vector128<ulong> right)
```

See Microsoft docs [here](#), ARM docs [here](#).

Assembly generated:

```
``armasm ; Assembly listing for method
AdvSimdMethods.ZipLowTest(System.Runtime.Intrinsics.Vector64<Byte>,System.Run
time.Intrinsics.Vector64<Byte>):System.Runtime.Intrinsics.Vector64<Byte>; V00
arg0 [V00,T00] ( 3, 3 ) simd8 -> d0 HFA(simd8); V01 arg1 [V01,T01] ( 3, 3 ) simd8 -> d1
HFA(simd8); # V02 OutArgs [V02] ( 1, 1 ) lclBlk ( 0 ) [sp+0x00] "OutgoingArgSpace"; Lcl
frame size = 0 stp fp, lr, [sp,#-16]! mov fp, sp zip1 v16.8b, v0.8b, v1.8b mov v0.8b, v16.8b
ldp fp, lr, [sp],#16 ret lr
```

; Total bytes of code 24, prolog size 8