

*Table-Of-Contents.txt lines 1 to 60*

|          |                       |        |      |   |      |
|----------|-----------------------|--------|------|---|------|
| Page 1:  | Table-Of-Contents.txt | Lines: | 1    | - | 60   |
| Page 2:  | Table-Of-Contents.txt | Lines: | 61   | - | 120  |
| Page 3:  | Table-Of-Contents.txt | Lines: | 121  | - | 180  |
| Page 4:  | Table-Of-Contents.txt | Lines: | 181  | - | 240  |
| Page 5:  | Table-Of-Contents.txt | Lines: | 241  | - | 300  |
| Page 6:  | Table-Of-Contents.txt | Lines: | 301  | - | 350  |
| Page 7:  | README.txt            | Lines: | 1    | - | 15   |
| Page 8:  | License.txt           | Lines: | 1    | - | 32   |
| Page 9:  | src/gf.c              | Lines: | 1    | - | 60   |
| Page 10: | src/gf.c              | Lines: | 61   | - | 120  |
| Page 11: | src/gf.c              | Lines: | 121  | - | 180  |
| Page 12: | src/gf.c              | Lines: | 181  | - | 240  |
| Page 13: | src/gf.c              | Lines: | 241  | - | 300  |
| Page 14: | src/gf.c              | Lines: | 301  | - | 360  |
| Page 15: | src/gf.c              | Lines: | 361  | - | 420  |
| Page 16: | src/gf.c              | Lines: | 421  | - | 480  |
| Page 17: | src/gf.c              | Lines: | 481  | - | 540  |
| Page 18: | src/gf.c              | Lines: | 541  | - | 600  |
| Page 19: | src/gf.c              | Lines: | 601  | - | 660  |
| Page 20: | src/gf.c              | Lines: | 661  | - | 720  |
| Page 21: | src/gf.c              | Lines: | 721  | - | 780  |
| Page 22: | src/gf.c              | Lines: | 781  | - | 840  |
| Page 23: | src/gf.c              | Lines: | 841  | - | 900  |
| Page 24: | src/gf.c              | Lines: | 901  | - | 960  |
| Page 25: | src/gf.c              | Lines: | 961  | - | 1020 |
| Page 26: | src/gf.c              | Lines: | 1021 | - | 1036 |
| Page 27: | src/gf_general.c      | Lines: | 1    | - | 60   |
| Page 28: | src/gf_general.c      | Lines: | 61   | - | 120  |
| Page 29: | src/gf_general.c      | Lines: | 121  | - | 180  |
| Page 30: | src/gf_general.c      | Lines: | 181  | - | 240  |
| Page 31: | src/gf_general.c      | Lines: | 241  | - | 300  |
| Page 32: | src/gf_general.c      | Lines: | 301  | - | 360  |
| Page 33: | src/gf_general.c      | Lines: | 361  | - | 420  |
| Page 34: | src/gf_general.c      | Lines: | 421  | - | 480  |
| Page 35: | src/gf_general.c      | Lines: | 481  | - | 539  |
| Page 36: | src/gf_method.c       | Lines: | 1    | - | 60   |
| Page 37: | src/gf_method.c       | Lines: | 61   | - | 120  |
| Page 38: | src/gf_method.c       | Lines: | 121  | - | 180  |
| Page 39: | src/gf_method.c       | Lines: | 181  | - | 188  |
| Page 40: | src/gf_rand.c         | Lines: | 1    | - | 60   |
| Page 41: | src/gf_rand.c         | Lines: | 61   | - | 80   |
| Page 42: | src/gf_w128.c         | Lines: | 1    | - | 60   |
| Page 43: | src/gf_w128.c         | Lines: | 61   | - | 120  |
| Page 44: | src/gf_w128.c         | Lines: | 121  | - | 180  |
| Page 45: | src/gf_w128.c         | Lines: | 181  | - | 240  |
| Page 46: | src/gf_w128.c         | Lines: | 241  | - | 300  |
| Page 47: | src/gf_w128.c         | Lines: | 301  | - | 360  |
| Page 48: | src/gf_w128.c         | Lines: | 361  | - | 420  |
| Page 49: | src/gf_w128.c         | Lines: | 421  | - | 480  |
| Page 50: | src/gf_w128.c         | Lines: | 481  | - | 540  |
| Page 51: | src/gf_w128.c         | Lines: | 541  | - | 600  |
| Page 52: | src/gf_w128.c         | Lines: | 601  | - | 660  |
| Page 53: | src/gf_w128.c         | Lines: | 661  | - | 720  |
| Page 54: | src/gf_w128.c         | Lines: | 721  | - | 780  |
| Page 55: | src/gf_w128.c         | Lines: | 781  | - | 840  |
| Page 56: | src/gf_w128.c         | Lines: | 841  | - | 900  |
| Page 57: | src/gf_w128.c         | Lines: | 901  | - | 960  |
| Page 58: | src/gf_w128.c         | Lines: | 961  | - | 1020 |
| Page 59: | src/gf_w128.c         | Lines: | 1021 | - | 1080 |
| Page 60: | src/gf_w128.c         | Lines: | 1081 | - | 1140 |



*Table-Of-Contents.txt lines 61 to 120*

|           |               |        |      |   |      |
|-----------|---------------|--------|------|---|------|
| Page 61:  | src/gf_w128.c | Lines: | 1141 | - | 1200 |
| Page 62:  | src/gf_w128.c | Lines: | 1201 | - | 1260 |
| Page 63:  | src/gf_w128.c | Lines: | 1261 | - | 1320 |
| Page 64:  | src/gf_w128.c | Lines: | 1321 | - | 1380 |
| Page 65:  | src/gf_w128.c | Lines: | 1381 | - | 1440 |
| Page 66:  | src/gf_w128.c | Lines: | 1441 | - | 1500 |
| Page 67:  | src/gf_w128.c | Lines: | 1501 | - | 1560 |
| Page 68:  | src/gf_w128.c | Lines: | 1561 | - | 1620 |
| Page 69:  | src/gf_w128.c | Lines: | 1621 | - | 1680 |
| Page 70:  | src/gf_w128.c | Lines: | 1681 | - | 1740 |
| Page 71:  | src/gf_w128.c | Lines: | 1741 | - | 1794 |
| Page 72:  | src/gf_w16.c  | Lines: | 1    | - | 60   |
| Page 73:  | src/gf_w16.c  | Lines: | 61   | - | 120  |
| Page 74:  | src/gf_w16.c  | Lines: | 121  | - | 180  |
| Page 75:  | src/gf_w16.c  | Lines: | 181  | - | 240  |
| Page 76:  | src/gf_w16.c  | Lines: | 241  | - | 300  |
| Page 77:  | src/gf_w16.c  | Lines: | 301  | - | 360  |
| Page 78:  | src/gf_w16.c  | Lines: | 361  | - | 420  |
| Page 79:  | src/gf_w16.c  | Lines: | 421  | - | 480  |
| Page 80:  | src/gf_w16.c  | Lines: | 481  | - | 540  |
| Page 81:  | src/gf_w16.c  | Lines: | 541  | - | 600  |
| Page 82:  | src/gf_w16.c  | Lines: | 601  | - | 660  |
| Page 83:  | src/gf_w16.c  | Lines: | 661  | - | 720  |
| Page 84:  | src/gf_w16.c  | Lines: | 721  | - | 780  |
| Page 85:  | src/gf_w16.c  | Lines: | 781  | - | 840  |
| Page 86:  | src/gf_w16.c  | Lines: | 841  | - | 900  |
| Page 87:  | src/gf_w16.c  | Lines: | 901  | - | 960  |
| Page 88:  | src/gf_w16.c  | Lines: | 961  | - | 1020 |
| Page 89:  | src/gf_w16.c  | Lines: | 1021 | - | 1080 |
| Page 90:  | src/gf_w16.c  | Lines: | 1081 | - | 1140 |
| Page 91:  | src/gf_w16.c  | Lines: | 1141 | - | 1200 |
| Page 92:  | src/gf_w16.c  | Lines: | 1201 | - | 1260 |
| Page 93:  | src/gf_w16.c  | Lines: | 1261 | - | 1320 |
| Page 94:  | src/gf_w16.c  | Lines: | 1321 | - | 1380 |
| Page 95:  | src/gf_w16.c  | Lines: | 1381 | - | 1440 |
| Page 96:  | src/gf_w16.c  | Lines: | 1441 | - | 1500 |
| Page 97:  | src/gf_w16.c  | Lines: | 1501 | - | 1560 |
| Page 98:  | src/gf_w16.c  | Lines: | 1561 | - | 1620 |
| Page 99:  | src/gf_w16.c  | Lines: | 1621 | - | 1680 |
| Page 100: | src/gf_w16.c  | Lines: | 1681 | - | 1740 |
| Page 101: | src/gf_w16.c  | Lines: | 1741 | - | 1800 |
| Page 102: | src/gf_w16.c  | Lines: | 1801 | - | 1860 |
| Page 103: | src/gf_w16.c  | Lines: | 1861 | - | 1920 |
| Page 104: | src/gf_w16.c  | Lines: | 1921 | - | 1980 |
| Page 105: | src/gf_w16.c  | Lines: | 1981 | - | 2040 |
| Page 106: | src/gf_w16.c  | Lines: | 2041 | - | 2100 |
| Page 107: | src/gf_w16.c  | Lines: | 2101 | - | 2160 |
| Page 108: | src/gf_w16.c  | Lines: | 2161 | - | 2220 |
| Page 109: | src/gf_w16.c  | Lines: | 2221 | - | 2280 |
| Page 110: | src/gf_w16.c  | Lines: | 2281 | - | 2340 |
| Page 111: | src/gf_w16.c  | Lines: | 2341 | - | 2400 |
| Page 112: | src/gf_w16.c  | Lines: | 2401 | - | 2460 |
| Page 113: | src/gf_w16.c  | Lines: | 2461 | - | 2502 |
| Page 114: | src/gf_w32.c  | Lines: | 1    | - | 60   |
| Page 115: | src/gf_w32.c  | Lines: | 61   | - | 120  |
| Page 116: | src/gf_w32.c  | Lines: | 121  | - | 180  |
| Page 117: | src/gf_w32.c  | Lines: | 181  | - | 240  |
| Page 118: | src/gf_w32.c  | Lines: | 241  | - | 300  |
| Page 119: | src/gf_w32.c  | Lines: | 301  | - | 360  |
| Page 120: | src/gf_w32.c  | Lines: | 361  | - | 420  |



*Table-Of-Contents.txt lines 121 to 180*

|           |              |        |      |   |      |
|-----------|--------------|--------|------|---|------|
| Page 121: | src/gf_w32.c | Lines: | 421  | – | 480  |
| Page 122: | src/gf_w32.c | Lines: | 481  | – | 540  |
| Page 123: | src/gf_w32.c | Lines: | 541  | – | 600  |
| Page 124: | src/gf_w32.c | Lines: | 601  | – | 660  |
| Page 125: | src/gf_w32.c | Lines: | 661  | – | 720  |
| Page 126: | src/gf_w32.c | Lines: | 721  | – | 780  |
| Page 127: | src/gf_w32.c | Lines: | 781  | – | 840  |
| Page 128: | src/gf_w32.c | Lines: | 841  | – | 900  |
| Page 129: | src/gf_w32.c | Lines: | 901  | – | 960  |
| Page 130: | src/gf_w32.c | Lines: | 961  | – | 1020 |
| Page 131: | src/gf_w32.c | Lines: | 1021 | – | 1080 |
| Page 132: | src/gf_w32.c | Lines: | 1081 | – | 1140 |
| Page 133: | src/gf_w32.c | Lines: | 1141 | – | 1200 |
| Page 134: | src/gf_w32.c | Lines: | 1201 | – | 1260 |
| Page 135: | src/gf_w32.c | Lines: | 1261 | – | 1320 |
| Page 136: | src/gf_w32.c | Lines: | 1321 | – | 1380 |
| Page 137: | src/gf_w32.c | Lines: | 1381 | – | 1440 |
| Page 138: | src/gf_w32.c | Lines: | 1441 | – | 1500 |
| Page 139: | src/gf_w32.c | Lines: | 1501 | – | 1560 |
| Page 140: | src/gf_w32.c | Lines: | 1561 | – | 1620 |
| Page 141: | src/gf_w32.c | Lines: | 1621 | – | 1680 |
| Page 142: | src/gf_w32.c | Lines: | 1681 | – | 1740 |
| Page 143: | src/gf_w32.c | Lines: | 1741 | – | 1800 |
| Page 144: | src/gf_w32.c | Lines: | 1801 | – | 1860 |
| Page 145: | src/gf_w32.c | Lines: | 1861 | – | 1920 |
| Page 146: | src/gf_w32.c | Lines: | 1921 | – | 1980 |
| Page 147: | src/gf_w32.c | Lines: | 1981 | – | 2040 |
| Page 148: | src/gf_w32.c | Lines: | 2041 | – | 2100 |
| Page 149: | src/gf_w32.c | Lines: | 2101 | – | 2160 |
| Page 150: | src/gf_w32.c | Lines: | 2161 | – | 2220 |
| Page 151: | src/gf_w32.c | Lines: | 2221 | – | 2280 |
| Page 152: | src/gf_w32.c | Lines: | 2281 | – | 2340 |
| Page 153: | src/gf_w32.c | Lines: | 2341 | – | 2400 |
| Page 154: | src/gf_w32.c | Lines: | 2401 | – | 2460 |
| Page 155: | src/gf_w32.c | Lines: | 2461 | – | 2520 |
| Page 156: | src/gf_w32.c | Lines: | 2521 | – | 2580 |
| Page 157: | src/gf_w32.c | Lines: | 2581 | – | 2640 |
| Page 158: | src/gf_w32.c | Lines: | 2641 | – | 2700 |
| Page 159: | src/gf_w32.c | Lines: | 2701 | – | 2760 |
| Page 160: | src/gf_w32.c | Lines: | 2761 | – | 2820 |
| Page 161: | src/gf_w32.c | Lines: | 2821 | – | 2877 |
| Page 162: | src/gf_w4.c  | Lines: | 1    | – | 60   |
| Page 163: | src/gf_w4.c  | Lines: | 61   | – | 120  |
| Page 164: | src/gf_w4.c  | Lines: | 121  | – | 180  |
| Page 165: | src/gf_w4.c  | Lines: | 181  | – | 240  |
| Page 166: | src/gf_w4.c  | Lines: | 241  | – | 300  |
| Page 167: | src/gf_w4.c  | Lines: | 301  | – | 360  |
| Page 168: | src/gf_w4.c  | Lines: | 361  | – | 420  |
| Page 169: | src/gf_w4.c  | Lines: | 421  | – | 480  |
| Page 170: | src/gf_w4.c  | Lines: | 481  | – | 540  |
| Page 171: | src/gf_w4.c  | Lines: | 541  | – | 600  |
| Page 172: | src/gf_w4.c  | Lines: | 601  | – | 660  |
| Page 173: | src/gf_w4.c  | Lines: | 661  | – | 720  |
| Page 174: | src/gf_w4.c  | Lines: | 721  | – | 780  |
| Page 175: | src/gf_w4.c  | Lines: | 781  | – | 840  |
| Page 176: | src/gf_w4.c  | Lines: | 841  | – | 900  |
| Page 177: | src/gf_w4.c  | Lines: | 901  | – | 960  |
| Page 178: | src/gf_w4.c  | Lines: | 961  | – | 1020 |
| Page 179: | src/gf_w4.c  | Lines: | 1021 | – | 1080 |
| Page 180: | src/gf_w4.c  | Lines: | 1081 | – | 1140 |



*Table-Of-Contents.txt lines 181 to 240*

|           |              |        |      |   |      |
|-----------|--------------|--------|------|---|------|
| Page 181: | src/gf_w4.c  | Lines: | 1141 | – | 1200 |
| Page 182: | src/gf_w4.c  | Lines: | 1201 | – | 1260 |
| Page 183: | src/gf_w4.c  | Lines: | 1261 | – | 1320 |
| Page 184: | src/gf_w4.c  | Lines: | 1321 | – | 1380 |
| Page 185: | src/gf_w4.c  | Lines: | 1381 | – | 1440 |
| Page 186: | src/gf_w4.c  | Lines: | 1441 | – | 1500 |
| Page 187: | src/gf_w4.c  | Lines: | 1501 | – | 1560 |
| Page 188: | src/gf_w4.c  | Lines: | 1561 | – | 1620 |
| Page 189: | src/gf_w4.c  | Lines: | 1621 | – | 1680 |
| Page 190: | src/gf_w4.c  | Lines: | 1681 | – | 1740 |
| Page 191: | src/gf_w4.c  | Lines: | 1741 | – | 1800 |
| Page 192: | src/gf_w4.c  | Lines: | 1801 | – | 1860 |
| Page 193: | src/gf_w4.c  | Lines: | 1861 | – | 1920 |
| Page 194: | src/gf_w4.c  | Lines: | 1921 | – | 1980 |
| Page 195: | src/gf_w4.c  | Lines: | 1981 | – | 2040 |
| Page 196: | src/gf_w4.c  | Lines: | 2041 | – | 2086 |
| Page 197: | src/gf_w64.c | Lines: | 1    | – | 60   |
| Page 198: | src/gf_w64.c | Lines: | 61   | – | 120  |
| Page 199: | src/gf_w64.c | Lines: | 121  | – | 180  |
| Page 200: | src/gf_w64.c | Lines: | 181  | – | 240  |
| Page 201: | src/gf_w64.c | Lines: | 241  | – | 300  |
| Page 202: | src/gf_w64.c | Lines: | 301  | – | 360  |
| Page 203: | src/gf_w64.c | Lines: | 361  | – | 420  |
| Page 204: | src/gf_w64.c | Lines: | 421  | – | 480  |
| Page 205: | src/gf_w64.c | Lines: | 481  | – | 540  |
| Page 206: | src/gf_w64.c | Lines: | 541  | – | 600  |
| Page 207: | src/gf_w64.c | Lines: | 601  | – | 660  |
| Page 208: | src/gf_w64.c | Lines: | 661  | – | 720  |
| Page 209: | src/gf_w64.c | Lines: | 721  | – | 780  |
| Page 210: | src/gf_w64.c | Lines: | 781  | – | 840  |
| Page 211: | src/gf_w64.c | Lines: | 841  | – | 900  |
| Page 212: | src/gf_w64.c | Lines: | 901  | – | 960  |
| Page 213: | src/gf_w64.c | Lines: | 961  | – | 1020 |
| Page 214: | src/gf_w64.c | Lines: | 1021 | – | 1080 |
| Page 215: | src/gf_w64.c | Lines: | 1081 | – | 1140 |
| Page 216: | src/gf_w64.c | Lines: | 1141 | – | 1200 |
| Page 217: | src/gf_w64.c | Lines: | 1201 | – | 1260 |
| Page 218: | src/gf_w64.c | Lines: | 1261 | – | 1320 |
| Page 219: | src/gf_w64.c | Lines: | 1321 | – | 1380 |
| Page 220: | src/gf_w64.c | Lines: | 1381 | – | 1440 |
| Page 221: | src/gf_w64.c | Lines: | 1441 | – | 1500 |
| Page 222: | src/gf_w64.c | Lines: | 1501 | – | 1560 |
| Page 223: | src/gf_w64.c | Lines: | 1561 | – | 1620 |
| Page 224: | src/gf_w64.c | Lines: | 1621 | – | 1680 |
| Page 225: | src/gf_w64.c | Lines: | 1681 | – | 1740 |
| Page 226: | src/gf_w64.c | Lines: | 1741 | – | 1800 |
| Page 227: | src/gf_w64.c | Lines: | 1801 | – | 1860 |
| Page 228: | src/gf_w64.c | Lines: | 1861 | – | 1920 |
| Page 229: | src/gf_w64.c | Lines: | 1921 | – | 1980 |
| Page 230: | src/gf_w64.c | Lines: | 1981 | – | 2040 |
| Page 231: | src/gf_w64.c | Lines: | 2041 | – | 2100 |
| Page 232: | src/gf_w64.c | Lines: | 2101 | – | 2160 |
| Page 233: | src/gf_w64.c | Lines: | 2161 | – | 2220 |
| Page 234: | src/gf_w64.c | Lines: | 2221 | – | 2249 |
| Page 235: | src/gf_w8.c  | Lines: | 1    | – | 60   |
| Page 236: | src/gf_w8.c  | Lines: | 61   | – | 120  |
| Page 237: | src/gf_w8.c  | Lines: | 121  | – | 180  |
| Page 238: | src/gf_w8.c  | Lines: | 181  | – | 240  |
| Page 239: | src/gf_w8.c  | Lines: | 241  | – | 300  |
| Page 240: | src/gf_w8.c  | Lines: | 301  | – | 360  |



*Table-Of-Contents.txt lines 241 to 300*

|           |                       |        |      |   |      |
|-----------|-----------------------|--------|------|---|------|
| Page 241: | src/gf_w8.c           | Lines: | 361  | - | 420  |
| Page 242: | src/gf_w8.c           | Lines: | 421  | - | 480  |
| Page 243: | src/gf_w8.c           | Lines: | 481  | - | 540  |
| Page 244: | src/gf_w8.c           | Lines: | 541  | - | 600  |
| Page 245: | src/gf_w8.c           | Lines: | 601  | - | 660  |
| Page 246: | src/gf_w8.c           | Lines: | 661  | - | 720  |
| Page 247: | src/gf_w8.c           | Lines: | 721  | - | 780  |
| Page 248: | src/gf_w8.c           | Lines: | 781  | - | 840  |
| Page 249: | src/gf_w8.c           | Lines: | 841  | - | 900  |
| Page 250: | src/gf_w8.c           | Lines: | 901  | - | 960  |
| Page 251: | src/gf_w8.c           | Lines: | 961  | - | 1020 |
| Page 252: | src/gf_w8.c           | Lines: | 1021 | - | 1080 |
| Page 253: | src/gf_w8.c           | Lines: | 1081 | - | 1140 |
| Page 254: | src/gf_w8.c           | Lines: | 1141 | - | 1200 |
| Page 255: | src/gf_w8.c           | Lines: | 1201 | - | 1260 |
| Page 256: | src/gf_w8.c           | Lines: | 1261 | - | 1320 |
| Page 257: | src/gf_w8.c           | Lines: | 1321 | - | 1380 |
| Page 258: | src/gf_w8.c           | Lines: | 1381 | - | 1440 |
| Page 259: | src/gf_w8.c           | Lines: | 1441 | - | 1500 |
| Page 260: | src/gf_w8.c           | Lines: | 1501 | - | 1560 |
| Page 261: | src/gf_w8.c           | Lines: | 1561 | - | 1620 |
| Page 262: | src/gf_w8.c           | Lines: | 1621 | - | 1680 |
| Page 263: | src/gf_w8.c           | Lines: | 1681 | - | 1740 |
| Page 264: | src/gf_w8.c           | Lines: | 1741 | - | 1800 |
| Page 265: | src/gf_w8.c           | Lines: | 1801 | - | 1860 |
| Page 266: | src/gf_w8.c           | Lines: | 1861 | - | 1920 |
| Page 267: | src/gf_w8.c           | Lines: | 1921 | - | 1980 |
| Page 268: | src/gf_w8.c           | Lines: | 1981 | - | 2040 |
| Page 269: | src/gf_w8.c           | Lines: | 2041 | - | 2100 |
| Page 270: | src/gf_w8.c           | Lines: | 2101 | - | 2160 |
| Page 271: | src/gf_w8.c           | Lines: | 2161 | - | 2220 |
| Page 272: | src/gf_w8.c           | Lines: | 2221 | - | 2280 |
| Page 273: | src/gf_w8.c           | Lines: | 2281 | - | 2340 |
| Page 274: | src/gf_w8.c           | Lines: | 2341 | - | 2400 |
| Page 275: | src/gf_w8.c           | Lines: | 2401 | - | 2460 |
| Page 276: | src/gf_w8.c           | Lines: | 2461 | - | 2467 |
| Page 277: | src/gf_wgen.c         | Lines: | 1    | - | 60   |
| Page 278: | src/gf_wgen.c         | Lines: | 61   | - | 120  |
| Page 279: | src/gf_wgen.c         | Lines: | 121  | - | 180  |
| Page 280: | src/gf_wgen.c         | Lines: | 181  | - | 240  |
| Page 281: | src/gf_wgen.c         | Lines: | 241  | - | 300  |
| Page 282: | src/gf_wgen.c         | Lines: | 301  | - | 360  |
| Page 283: | src/gf_wgen.c         | Lines: | 361  | - | 420  |
| Page 284: | src/gf_wgen.c         | Lines: | 421  | - | 480  |
| Page 285: | src/gf_wgen.c         | Lines: | 481  | - | 540  |
| Page 286: | src/gf_wgen.c         | Lines: | 541  | - | 600  |
| Page 287: | src/gf_wgen.c         | Lines: | 601  | - | 660  |
| Page 288: | src/gf_wgen.c         | Lines: | 661  | - | 720  |
| Page 289: | src/gf_wgen.c         | Lines: | 721  | - | 780  |
| Page 290: | src/gf_wgen.c         | Lines: | 781  | - | 840  |
| Page 291: | src/gf_wgen.c         | Lines: | 841  | - | 900  |
| Page 292: | src/gf_wgen.c         | Lines: | 901  | - | 960  |
| Page 293: | src/gf_wgen.c         | Lines: | 961  | - | 1019 |
| Page 294: | include/config.h      | Lines: | 1    | - | 60   |
| Page 295: | include/config.h      | Lines: | 61   | - | 93   |
| Page 296: | include/gf_complete.h | Lines: | 1    | - | 60   |
| Page 297: | include/gf_complete.h | Lines: | 61   | - | 120  |
| Page 298: | include/gf_complete.h | Lines: | 121  | - | 180  |
| Page 299: | include/gf_complete.h | Lines: | 181  | - | 193  |
| Page 300: | include/gf_general.h  | Lines: | 1    | - | 60   |



*Table-Of-Contents.txt lines 301 to 350*

|           |                         |        |     |   |     |
|-----------|-------------------------|--------|-----|---|-----|
| Page 301: | include/gf_general.h    | Lines: | 61  | - | 61  |
| Page 302: | include/gf_int.h        | Lines: | 1   | - | 60  |
| Page 303: | include/gf_int.h        | Lines: | 61  | - | 120 |
| Page 304: | include/gf_int.h        | Lines: | 121 | - | 180 |
| Page 305: | include/gf_int.h        | Lines: | 181 | - | 200 |
| Page 306: | include/gf_method.h     | Lines: | 1   | - | 20  |
| Page 307: | include/gf_rand.h       | Lines: | 1   | - | 22  |
| Page 308: | examples/gf_example_1.c | Lines: | 1   | - | 57  |
| Page 309: | examples/gf_example_2.c | Lines: | 1   | - | 60  |
| Page 310: | examples/gf_example_2.c | Lines: | 61  | - | 107 |
| Page 311: | examples/gf_example_3.c | Lines: | 1   | - | 60  |
| Page 312: | examples/gf_example_3.c | Lines: | 61  | - | 76  |
| Page 313: | examples/gf_example_4.c | Lines: | 1   | - | 60  |
| Page 314: | examples/gf_example_4.c | Lines: | 61  | - | 69  |
| Page 315: | examples/gf_example_5.c | Lines: | 1   | - | 60  |
| Page 316: | examples/gf_example_5.c | Lines: | 61  | - | 78  |
| Page 317: | examples/gf_example_6.c | Lines: | 1   | - | 60  |
| Page 318: | examples/gf_example_6.c | Lines: | 61  | - | 84  |
| Page 319: | examples/gf_example_7.c | Lines: | 1   | - | 60  |
| Page 320: | examples/gf_example_7.c | Lines: | 61  | - | 75  |
| Page 321: | tools/gf_add.c          | Lines: | 1   | - | 60  |
| Page 322: | tools/gf_add.c          | Lines: | 61  | - | 114 |
| Page 323: | tools/gf_div.c          | Lines: | 1   | - | 60  |
| Page 324: | tools/gf_div.c          | Lines: | 61  | - | 68  |
| Page 325: | tools/gf_inline_time.c  | Lines: | 1   | - | 60  |
| Page 326: | tools/gf_inline_time.c  | Lines: | 61  | - | 120 |
| Page 327: | tools/gf_inline_time.c  | Lines: | 121 | - | 170 |
| Page 328: | tools/gf_methods.c      | Lines: | 1   | - | 60  |
| Page 329: | tools/gf_methods.c      | Lines: | 61  | - | 120 |
| Page 330: | tools/gf_methods.c      | Lines: | 121 | - | 180 |
| Page 331: | tools/gf_methods.c      | Lines: | 181 | - | 228 |
| Page 332: | tools/gf_mult.c         | Lines: | 1   | - | 60  |
| Page 333: | tools/gf_mult.c         | Lines: | 61  | - | 68  |
| Page 334: | tools/gf_poly.c         | Lines: | 1   | - | 60  |
| Page 335: | tools/gf_poly.c         | Lines: | 61  | - | 120 |
| Page 336: | tools/gf_poly.c         | Lines: | 121 | - | 180 |
| Page 337: | tools/gf_poly.c         | Lines: | 181 | - | 240 |
| Page 338: | tools/gf_poly.c         | Lines: | 241 | - | 275 |
| Page 339: | tools/gf_time.c         | Lines: | 1   | - | 60  |
| Page 340: | tools/gf_time.c         | Lines: | 61  | - | 120 |
| Page 341: | tools/gf_time.c         | Lines: | 121 | - | 180 |
| Page 342: | tools/gf_time.c         | Lines: | 181 | - | 212 |
| Page 343: | test/gf_unit.c          | Lines: | 1   | - | 60  |
| Page 344: | test/gf_unit.c          | Lines: | 61  | - | 120 |
| Page 345: | test/gf_unit.c          | Lines: | 121 | - | 180 |
| Page 346: | test/gf_unit.c          | Lines: | 181 | - | 240 |
| Page 347: | test/gf_unit.c          | Lines: | 241 | - | 300 |
| Page 348: | test/gf_unit.c          | Lines: | 301 | - | 360 |
| Page 349: | test/gf_unit.c          | Lines: | 361 | - | 420 |
| Page 350: | test/gf_unit.c          | Lines: | 421 | - | 432 |



# ***README.txt lines 1 to 15***

This is GF-Complete, Revision c25310f, June 16, 2014.

Authors: James S. Plank (University of Tennessee)  
Ethan L. Miller (UC Santa Cruz)  
Kevin M. Greenan (Box)  
Benjamin A. Arnold (University of Tennessee)  
John A. Burnum (University of Tennessee)  
Adam W. Disney (University of Tennessee,  
Allen C. McBride (University of Tennessee)

If you want to cite GF-Complete in a paper, I suggest citing the technical report version. The precise citation information for that is in <http://www.cs.utk.edu/~plank/plank/papers/CS-13-716.html>.



## *License.txt lines 1 to 32*

Copyright (c) 2013, James S. Plank, Ethan L. Miller, Kevin M. Greenan, Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the University of Tennessee nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



# src/gf.c lines 1 to 60

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf.c
 *
 * Generic routines for Galois fields
 */

#include "gf_int.h"
#include <stdio.h>
#include <stdlib.h>

int _gf_errno = GF_E_DEFAULT;

void gf_error()
{
    char *s;

    switch(_gf_errno) {
        case GF_E_DEFAULT: s = "No Error."; break;
        case GF_E_TWOMULT: s = "Cannot specify two -m's."; break;
        case GF_E_TWO_DIV: s = "Cannot specify two -d's."; break;
        case GF_E_POLYSPC: s = "-p needs to be followed by a number in hex (0x optional)."; break;
        case GF_E_GROUPAR: s = "Ran out of arguments in -m GROUP."; break;
        case GF_E_GROUPNU: s = "In -m GROUP g_s g_r -- g_s and g_r need to be numbers."; break;
        case GF_E_SPLITAR: s = "Ran out of arguments in -m SPLIT."; break;
        case GF_E_SPLITNU: s = "In -m SPLIT w_a w_b -- w_a and w_b need to be numbers."; break;
        case GF_E_FEWARGS: s = "Not enough arguments (Perhaps end with '-'?)"; break;
        case GF_E_CFM__W: s = "-m CARRY_FREE, w must be 4, 8, 16, 32, 64 or 128."; break;
        case GF_E_COMPPXP: s =
            "-m COMPOSITE, No poly specified, and we don't have a default for the given sub-field.";
            break;
        case GF_E_BASE__W: s = "-m COMPOSITE and the base field is not for w/2."; break;
        case GF_E_CFM4POL: s = "-m CARRY_FREE, w=4. (Prim-poly & 0xc) must equal 0."; break;
        case GF_E_CFM8POL: s = "-m CARRY_FREE, w=8. (Prim-poly & 0x80) must equal 0."; break;
        case GF_E_CF16POL: s = "-m CARRY_FREE, w=16. (Prim-poly & 0xe000) must equal 0."; break;
        case GF_E_CF32POL: s = "-m CARRY_FREE, w=32. (Prim-poly & 0xfe000000) must equal 0."; break;
        case GF_E_CF64POL: s = "-m CARRY_FREE, w=64. (Prim-poly & 0xfffe000000000000ULL) must equal 0."; break;
        case GF_E_MDEFDIV: s = "If multiplication method == default, can't change division."; break;
        case GF_E_MDEFREG: s = "If multiplication method == default, can't change region."; break;
        case GF_E_MDEFARG: s = "If multiplication method == default, can't use arg1/arg2."; break;
        case GF_E_DIVCOMP: s = "Cannot change the division technique with -m COMPOSITE."; break;
        case GF_E_DOUQUAD: s = "Cannot specify -r DOUBLE and -r QUAD."; break;
        case GF_E_SSE__NO: s = "Cannot specify -r SSE and -r NOSSE."; break;
        case GF_E_CAUCHYB: s = "Cannot specify -r CAUCHY and any other -r."; break;
        case GF_E_CAUCOMP: s = "Cannot specify -m COMPOSITE and -r CAUCHY."; break;
        case GF_E_CAUGT32: s = "Cannot specify -r CAUCHY with w > 32."; break;
        case GF_E_ARG1SET: s = "Only use arg1 with SPLIT, GROUP or COMPOSITE."; break;
        case GF_E_ARG2SET: s = "Only use arg2 with SPLIT or GROUP."; break;
        case GF_E_MATRIXW: s = "Cannot specify -d MATRIX with w > 32."; break;
        case GF_E_BAD__W: s = "W must be 1-32, 64 or 128."; break;
        case GF_E_DOUBLET: s = "Can only specify -r DOUBLE with -m TABLE."; break;
        case GF_E_DOUBLEW: s = "Can only specify -r DOUBLE w = 4 or w = 8."; break;
        case GF_E_DOUBLEJ: s = "Cannot specify -r DOUBLE with -r ALTMAP|SSE|NOSSE."; break;
        case GF_E_DOUBLEL: s = "Can only specify -r DOUBLE -r LAZY with w = 8"; break;
        case GF_E_QUAD__T: s = "Can only specify -r QUAD with -m TABLE."; break;
        case GF_E_QUAD__W: s = "Can only specify -r QUAD w = 4."; break;
        case GF_E_QUAD__J: s = "Cannot specify -r QUAD with -r ALTMAP|SSE|NOSSE."; break;
```



*src/gf.c lines 61 to 120*

```
case GF_E_BADPOLY: s = "Bad primitive polynomial (high bits set)."; break;
case GF_E_COMP_PP: s = "Bad primitive polynomial -- bigger than sub-field."; break;
case GF_E_LAZY__X: s = "If -r LAZY, then -r must be DOUBLE or QUAD."; break;
case GF_E_ALTSHIF: s = "Cannot specify -m SHIFT and -r ALTMAP."; break;
case GF_E_SSESHIF: s = "Cannot specify -m SHIFT and -r SSE|NOSSE."; break;
case GF_E_ALT_CFM: s = "Cannot specify -m CARRY_FREE and -r ALTMAP."; break;
case GF_E_SSE_CFM: s = "Cannot specify -m CARRY_FREE and -r SSE|NOSSE."; break;
case GF_E_PCLMULX: s = "Specified -m CARRY_FREE, but PCLMUL is not supported."; break;
case GF_E_ALT_BY2: s = "Cannot specify -m BYTWO_x and -r ALTMAP."; break;
case GF_E_BY2_SSE: s = "Specified -m BYTWO_x -r SSE, but SSE2 is not supported."; break;
case GF_E_LOGBADW: s = "With Log Tables, w must be <= 27."; break;
case GF_E_LOG__J: s = "Cannot use Log tables with -r ALTMAP|SSE|NOSSE."; break;
case GF_E_LOGPOLY: s = "Cannot use Log tables because the polynomial is not primitive."; break;
case GF_E_ZERBADW: s = "With -m LOG_ZERO, w must be 8 or 16."; break;
case GF_E_ZEXBADW: s = "With -m LOG_ZERO_EXT, w must be 8."; break;
case GF_E_GR_ARGX: s = "With -m GROUP, arg1 and arg2 must be >= 0."; break;
case GF_E_GR_W_48: s = "With -m GROUP, w cannot be 4 or 8."; break;
case GF_E_GR_W_16: s = "With -m GROUP, w == 16, arg1 and arg2 must be 4."; break;
case GF_E_GR_128A: s = "With -m GROUP, w == 128, arg1 must be 4, and arg2 in { 4,8,16 }."; break;
case GF_E_GR_A_27: s = "With -m GROUP, arg1 and arg2 must be <= 27."; break;
case GF_E_GR_AR_W: s = "With -m GROUP, arg1 and arg2 must be <= w."; break;
case GF_E_GR__J: s = "Cannot use GROUP with -r ALTMAP|SSE|NOSSE."; break;
case GF_E_TABLE_W: s = "With -m TABLE, w must be < 15, or == 16."; break;
case GF_E_TAB_SSE: s = "With -m TABLE, SSE|NOSSE only applies to w=4."; break;
case GF_E_TABSSE3: s = "With -m TABLE, -r SSE, you need SSSE3 supported."; break;
case GF_E_TAB_ALT: s = "With -m TABLE, you cannot use ALTMAP."; break;
case GF_E_SP128AR: s = "With -m SPLIT, w=128, bad arg1/arg2."; break;
case GF_E_SP128AL: s = "With -m SPLIT, w=128, -r SSE requires -r ALTMAP."; break;
case GF_E_SP128AS: s = "With -m SPLIT, w=128, ALTMAP needs SSSE3 supported."; break;
case GF_E_SP128_A: s = "With -m SPLIT, w=128, -r ALTMAP only with arg1/arg2 = 4/128."; break;
case GF_E_SP128_S: s = "With -m SPLIT, w=128, -r SSE|NOSSE only with arg1/arg2 = 4/128."; break;
case GF_E_SPLIT_W: s = "With -m SPLIT, w must be in {8, 16, 32, 64, 128}."; break;
case GF_E_SP_16AR: s = "With -m SPLIT, w=16, Bad arg1/arg2."; break;
case GF_E_SP_16_A: s = "With -m SPLIT, w=16, -r ALTMAP only with arg1/arg2 = 4/16."; break;
case GF_E_SP_16_S: s = "With -m SPLIT, w=16, -r SSE|NOSSE only with arg1/arg2 = 4/16."; break;
case GF_E_SP_32AR: s = "With -m SPLIT, w=32, Bad arg1/arg2."; break;
case GF_E_SP_32AS: s = "With -m SPLIT, w=32, -r ALTMAP needs SSSE3 supported."; break;
case GF_E_SP_32_A: s = "With -m SPLIT, w=32, -r ALTMAP only with arg1/arg2 = 4/32."; break;
case GF_E_SP_32_S: s = "With -m SPLIT, w=32, -r SSE|NOSSE only with arg1/arg2 = 4/32."; break;
case GF_E_SP_64AR: s = "With -m SPLIT, w=64, Bad arg1/arg2."; break;
case GF_E_SP_64AS: s = "With -m SPLIT, w=64, -r ALTMAP needs SSSE3 supported."; break;
case GF_E_SP_64_A: s = "With -m SPLIT, w=64, -r ALTMAP only with arg1/arg2 = 4/64."; break;
case GF_E_SP_64_S: s = "With -m SPLIT, w=64, -r SSE|NOSSE only with arg1/arg2 = 4/64."; break;
case GF_E_SP_8_AR: s = "With -m SPLIT, w=8, Bad arg1/arg2."; break;
case GF_E_SP_8__A: s = "With -m SPLIT, w=8, Can't have -r ALTMAP."; break;
case GF_E_SP_SSE3: s = "With -m SPLIT, Need SSSE3 support for SSE."; break;
case GF_E_COMP_A2: s = "With -m COMPOSITE, arg1 must equal 2."; break;
case GF_E_COMP_SS: s = "With -m COMPOSITE, -r SSE and -r NOSSE do not apply."; break;
case GF_E_COMP__W: s = "With -m COMPOSITE, w must be 8, 16, 32, 64 or 128."; break;
case GF_E_UNKFLAG: s = "Unknown method flag - should be -m, -d, -r or -p."; break;
case GF_E_UNKNOWN: s = "Unknown multiplication type."; break;
case GF_E_UNK_REG: s = "Unknown region type."; break;
case GF_E_UNK_DIV: s = "Unknown division type."; break;
default: s = "Undefined error.";
}

fprintf(stderr, "%s\n", s);
}

uint64_t gf_composite_get_default_poly(gf_t *base)
```



*src/gf.c lines 121 to 180*

```
{
    gf_internal_t *h;
    int rv;

    h = (gf_internal_t *) base->scratch;
    if (h->w == 4) {
        if (h->mult_type == GF_MULT_COMPOSITE) return 0;
        if (h->prim_poly == 0x13) return 2;
        return 0;
    }
    if (h->w == 8) {
        if (h->mult_type == GF_MULT_COMPOSITE) return 0;
        if (h->prim_poly == 0x11d) return 3;
        return 0;
    }
    if (h->w == 16) {
        if (h->mult_type == GF_MULT_COMPOSITE) {
            rv = gf_composite_get_default_poly(h->base_gf);
            if (rv != h->prim_poly) return 0;
            if (rv == 3) return 0x105;
            return 0;
        } else {
            if (h->prim_poly == 0x1100b) return 2;
            if (h->prim_poly == 0x1002d) return 7;
            return 0;
        }
    }
    if (h->w == 32) {
        if (h->mult_type == GF_MULT_COMPOSITE) {
            rv = gf_composite_get_default_poly(h->base_gf);
            if (rv != h->prim_poly) return 0;
            if (rv == 2) return 0x10005;
            if (rv == 7) return 0x10008;
            if (rv == 0x105) return 0x10002;
            return 0;
        } else {
            if (h->prim_poly == 0x400007) return 2;
            if (h->prim_poly == 0xc5) return 3;
            return 0;
        }
    }
    if (h->w == 64) {
        if (h->mult_type == GF_MULT_COMPOSITE) {
            rv = gf_composite_get_default_poly(h->base_gf);
            if (rv != h->prim_poly) return 0;
            if (rv == 3) return 0x100000009ULL;
            if (rv == 2) return 0x100000004ULL;
            if (rv == 0x10005) return 0x100000003ULL;
            if (rv == 0x10002) return 0x100000005ULL;
            if (rv == 0x10008) return 0x100000006ULL; /* JSP: (0x0x100000003 works too,
                                                         but I want to differentiate cases). */

            return 0;
        } else {
            if (h->prim_poly == 0x1bULL) return 2;
            return 0;
        }
    }
}
return 0;
}
```



*src/gf.c lines 181 to 240*

```
int gf_error_check(int w, int mult_type, int region_type, int divide_type,
                  int arg1, int arg2, uint64_t poly, gf_t *base)
{
    int sse3 = 0;
    int sse2 = 0;
    int pclmul = 0;
    int rdouble, rquad, rlazy, rsse, rnosse, raltmap, rcauchy, tmp;
    gf_internal_t *sub;

    rdouble = (region_type & GF_REGION_DOUBLE_TABLE);
    rquad = (region_type & GF_REGION_QUAD_TABLE);
    rlazy = (region_type & GF_REGION_LAZY);
    rsse = (region_type & GF_REGION_SSE);
    rnosse = (region_type & GF_REGION_NOSSE);
    raltmap = (region_type & GF_REGION_ALTMAP);
    rcauchy = (region_type & GF_REGION_CAUCHY);

    if (divide_type != GF_DIVIDE_DEFAULT &&
        divide_type != GF_DIVIDE_MATRIX &&
        divide_type != GF_DIVIDE_EUCLID) {
        _gf_errno = GF_E_UNK_DIV;
        return 0;
    }

    tmp = ( GF_REGION_DOUBLE_TABLE | GF_REGION_QUAD_TABLE | GF_REGION_LAZY |
            GF_REGION_SSE | GF_REGION_NOSSE | GF_REGION_ALTMAP | GF_REGION_CAUCHY );
    if (region_type & (~tmp)) { _gf_errno = GF_E_UNK_REG; return 0; }

#ifdef INTEL_SSE2
    sse2 = 1;
#endif

#ifdef INTEL_SSSE3
    sse3 = 1;
#endif

#ifdef INTEL_SSE4_PCLMUL
    pclmul = 1;
#endif

    if (w < 1 || (w > 32 && w != 64 && w != 128)) { _gf_errno = GF_E_BAD___W; return 0; }

    if (mult_type != GF_MULT_COMPOSITE && w < 64) {
        if ((poly >> (w+1)) != 0) { _gf_errno = GF_E_BADPOLY; return 0; }
    }

    if (mult_type == GF_MULT_DEFAULT) {
        if (divide_type != GF_DIVIDE_DEFAULT) { _gf_errno = GF_E_MDEFDIV; return 0; }
        if (region_type != GF_REGION_DEFAULT) { _gf_errno = GF_E_MDEFREG; return 0; }
        if (arg1 != 0 || arg2 != 0) { _gf_errno = GF_E_MDEFARG; return 0; }
        return 1;
    }

    if (rsse && rnosse) { _gf_errno = GF_E_SSE___NO; return 0; }
    if (rcauchy && w > 32) { _gf_errno = GF_E_CAUGT32; return 0; }
    if (rcauchy && region_type != GF_REGION_CAUCHY) { _gf_errno = GF_E_CAUCHYB; return 0; }
    if (rcauchy && mult_type == GF_MULT_COMPOSITE) { _gf_errno = GF_E_CAUCOMP; return 0; }

    if (arg1 != 0 && mult_type != GF_MULT_COMPOSITE &&
```



*src/gf.c lines 241 to 300*

```
    mult_type != GF_MULT_SPLIT_TABLE && mult_type != GF_MULT_GROUP) {
    _gf_errno = GF_E_ARG1SET;
    return 0;
}

if (arg2 != 0 && mult_type != GF_MULT_SPLIT_TABLE && mult_type != GF_MULT_GROUP) {
    _gf_errno = GF_E_ARG2SET;
    return 0;
}

if (divide_type == GF_DIVIDE_MATRIX && w > 32) { _gf_errno = GF_E_MATRIXW; return 0; }

if (rdouble) {
    if (rquad)
        { _gf_errno = GF_E_DOUQUAD; return 0; }
    if (mult_type != GF_MULT_TABLE) { _gf_errno = GF_E_DOUBLET; return 0; }
    if (w != 4 && w != 8) { _gf_errno = GF_E_DOUBLEW; return 0; }
    if (rsse || rnosse || raltmap) { _gf_errno = GF_E_DOUBLEJ; return 0; }
    if (rlazy && w == 4) { _gf_errno = GF_E_DOUBLEL; return 0; }
    return 1;
}

if (rquad) {
    if (mult_type != GF_MULT_TABLE) { _gf_errno = GF_E_QUAD__T; return 0; }
    if (w != 4) { _gf_errno = GF_E_QUAD__W; return 0; }
    if (rsse || rnosse || raltmap) { _gf_errno = GF_E_QUAD__J; return 0; }
    return 1;
}

if (rlazy) { _gf_errno = GF_E_LAZY__X; return 0; }

if (mult_type == GF_MULT_SHIFT) {
    if (raltmap) { _gf_errno = GF_E_ALTSHIF; return 0; }
    if (rsse || rnosse) { _gf_errno = GF_E_SSESHIF; return 0; }
    return 1;
}

if (mult_type == GF_MULT_CARRY_FREE) {
    if (w != 4 && w != 8 && w != 16 &&
        w != 32 && w != 64 && w != 128) { _gf_errno = GF_E_CFM__W; return 0; }
    if (w == 4 && (poly & 0xc)) { _gf_errno = GF_E_CFM4POL; return 0; }
    if (w == 8 && (poly & 0x80)) { _gf_errno = GF_E_CFM8POL; return 0; }
    if (w == 16 && (poly & 0xe000)) { _gf_errno = GF_E_CF16POL; return 0; }
    if (w == 32 && (poly & 0xfe000000)) { _gf_errno = GF_E_CF32POL; return 0; }
    if (w == 64 && (poly & 0xfffe000000000000ULL)) { _gf_errno = GF_E_CF64POL; return 0; }
    if (raltmap) { _gf_errno = GF_E_ALT_CFM; return 0; }
    if (rsse || rnosse) { _gf_errno = GF_E_SSE_CFM; return 0; }
    if (!pclmul) { _gf_errno = GF_E_PCLMULX; return 0; }
    return 1;
}

if (mult_type == GF_MULT_CARRY_FREE_GK) {
    if (w != 4 && w != 8 && w != 16 &&
        w != 32 && w != 64 && w != 128) { _gf_errno = GF_E_CFM__W; return 0; }
    if (raltmap) { _gf_errno = GF_E_ALT_CFM; return 0; }
    if (rsse || rnosse) { _gf_errno = GF_E_SSE_CFM; return 0; }
    if (!pclmul) { _gf_errno = GF_E_PCLMULX; return 0; }
    return 1;
}

if (mult_type == GF_MULT_BYTWO_p || mult_type == GF_MULT_BYTWO_b) {
```



*src/gf.c lines 301 to 360*

```
    if (raltmap) { _gf_errno = GF_E_ALT_BY2; return 0; }
    if (rsse && !sse2) { _gf_errno = GF_E_BY2_SSE; return 0; }
    return 1;
}

if (mult_type == GF_MULT_LOG_TABLE || mult_type == GF_MULT_LOG_ZERO
    || mult_type == GF_MULT_LOG_ZERO_EXT ) {
    if (w > 27) { _gf_errno = GF_E_LOGBADW; return 0; }
    if (raltmap || rsse || rnosse) { _gf_errno = GF_E_LOG___J; return 0; }

    if (mult_type == GF_MULT_LOG_TABLE) return 1;

    if (w != 8 && w != 16) { _gf_errno = GF_E_ZERBADW; return 0; }

    if (mult_type == GF_MULT_LOG_ZERO) return 1;

    if (w != 8) { _gf_errno = GF_E_ZEXBADW; return 0; }
    return 1;
}

if (mult_type == GF_MULT_GROUP) {
    if (arg1 <= 0 || arg2 <= 0) { _gf_errno = GF_E_GR_ARGX; return 0; }
    if (w == 4 || w == 8) { _gf_errno = GF_E_GR_W_48; return 0; }
    if (w == 16 && (arg1 != 4 || arg2 != 4)) { _gf_errno = GF_E_GR_W_16; return 0; }
    if (w == 128 && (arg1 != 4 ||
        (arg2 != 4 && arg2 != 8 && arg2 != 16))) { _gf_errno = GF_E_GR_128A; return 0; }
    if (arg1 > 27 || arg2 > 27) { _gf_errno = GF_E_GR_A_27; return 0; }
    if (arg1 > w || arg2 > w) { _gf_errno = GF_E_GR_AR_W; return 0; }
    if (raltmap || rsse || rnosse) { _gf_errno = GF_E_GR___J; return 0; }
    return 1;
}

if (mult_type == GF_MULT_TABLE) {
    if (w != 16 && w >= 15) { _gf_errno = GF_E_TABLE_W; return 0; }
    if (w != 4 && (rsse || rnosse)) { _gf_errno = GF_E_TAB_SSE; return 0; }
    if (rsse && !sse3) { _gf_errno = GF_E_TABSSE3; return 0; }
    if (raltmap) { _gf_errno = GF_E_TAB_ALT; return 0; }
    return 1;
}

if (mult_type == GF_MULT_SPLIT_TABLE) {
    if (arg1 > arg2) {
        tmp = arg1;
        arg1 = arg2;
        arg2 = tmp;
    }
    if (w == 8) {
        if (arg1 != 4 || arg2 != 8) { _gf_errno = GF_E_SP_8_AR; return 0; }
        if (rsse && !sse3) { _gf_errno = GF_E_SP_SSE3; return 0; }
        if (raltmap) { _gf_errno = GF_E_SP_8___A; return 0; }
    }
    else if (w == 16) {
        if ((arg1 == 8 && arg2 == 8) ||
            (arg1 == 8 && arg2 == 16)) {
            if (rsse || rnosse) { _gf_errno = GF_E_SP_16_S; return 0; }
            if (raltmap) { _gf_errno = GF_E_SP_16_A; return 0; }
        }
        else if (arg1 == 4 && arg2 == 16) {
            if (rsse && !sse3) { _gf_errno = GF_E_SP_SSE3; return 0; }
        }
        else {
            _gf_errno = GF_E_SP_16AR; return 0; }
    }
    else if (w == 32) {
        if ((arg1 == 8 && arg2 == 8) ||
```



src/gf.c lines 361 to 420

```
        (arg1 == 8 && arg2 == 32) ||
        (arg1 == 16 && arg2 == 32)) {
    if (rsse || rnosse)
    if (raltmap)
} else if (arg1 == 4 && arg2 == 32) {
    if (rsse && !sse3)
    if (raltmap && !sse3)
    if (raltmap && rnosse)
} else
} else if (w == 64) {
    if ((arg1 == 8 && arg2 == 8) ||
        (arg1 == 8 && arg2 == 64) ||
        (arg1 == 16 && arg2 == 64)) {
        if (rsse || rnosse)
        if (raltmap)
    } else if (arg1 == 4 && arg2 == 64) {
        if (rsse && !sse3)
        if (raltmap && !sse3)
        if (raltmap && rnosse)
    } else
} else if (w == 128) {
    if (arg1 == 8 && arg2 == 128) {
        if (rsse || rnosse)
        if (raltmap)
    } else if (arg1 == 4 && arg2 == 128) {
        if (rsse && !sse3)
        if (raltmap && !sse3)
        if (raltmap && rnosse)
    } else
} else
return 1;
}

if (mult_type == GF_MULT_COMPOSITE) {
    if (w != 8 && w != 16 && w != 32
        && w != 64 && w != 128)
    if ((poly >> (w/2)) != 0)
    if (divide_type != GF_DIVIDE_DEFAULT)
    if (arg1 != 2)
    if (rsse || rnosse)
    if (base != NULL) {
        sub = (gf_internal_t *) base->scratch;
        if (sub->w != w/2)
        if (poly == 0) {
            if (gf_composite_get_default_poly(base) == 0) {
        }
    }
    return 1;
}

_gf_errno = GF_E_UNKNOWN;
return 0;
}

int gf_scratch_size(int w,
                    int mult_type,
                    int region_type,
                    int divide_type,
                    int arg1,
                    int arg2)
```

```
{ _gf_errno = GF_E_SP_32_S; return 0; }
{ _gf_errno = GF_E_SP_32_A; return 0; }

{ _gf_errno = GF_E_SP_SSE3; return 0; }
{ _gf_errno = GF_E_SP_32AS; return 0; }
{ _gf_errno = GF_E_SP_32AS; return 0; }
{ _gf_errno = GF_E_SP_32AR; return 0; }

{ _gf_errno = GF_E_SP_64_S; return 0; }
{ _gf_errno = GF_E_SP_64_A; return 0; }

{ _gf_errno = GF_E_SP_SSE3; return 0; }
{ _gf_errno = GF_E_SP_64AS; return 0; }
{ _gf_errno = GF_E_SP_64AS; return 0; }
{ _gf_errno = GF_E_SP_64AR; return 0; }

{ _gf_errno = GF_E_SP128_S; return 0; }
{ _gf_errno = GF_E_SP128_A; return 0; }

{ _gf_errno = GF_E_SP_SSE3; return 0; }
{ _gf_errno = GF_E_SP128AS; return 0; }
{ _gf_errno = GF_E_SP128AS; return 0; }
{ _gf_errno = GF_E_SP128AR; return 0; }
{ _gf_errno = GF_E_SPLIT_W; return 0; }
```

```
{ _gf_errno = GF_E_COMP__W; return 0; }
{ _gf_errno = GF_E_COMP_PP; return 0; }
{ _gf_errno = GF_E_DIVCOMP; return 0; }
{ _gf_errno = GF_E_COMP_A2; return 0; }
{ _gf_errno = GF_E_COMP_SS; return 0; }

{ _gf_errno = GF_E_BASE__W; return 0; }
{ _gf_errno = GF_E_COMPPXP; return 0; }
```



*src/gf.c lines 421 to 480*

```
{
    if (gf_error_check(w, mult_type, region_type, divide_type, arg1, arg2, 0, NULL) == 0) return 0;

    switch(w) {
        case 4: return gf_w4_scratch_size(mult_type, region_type, divide_type, arg1, arg2);
        case 8: return gf_w8_scratch_size(mult_type, region_type, divide_type, arg1, arg2);
        case 16: return gf_w16_scratch_size(mult_type, region_type, divide_type, arg1, arg2);
        case 32: return gf_w32_scratch_size(mult_type, region_type, divide_type, arg1, arg2);
        case 64: return gf_w64_scratch_size(mult_type, region_type, divide_type, arg1, arg2);
        case 128: return gf_w128_scratch_size(mult_type, region_type, divide_type, arg1, arg2);
        default: return gf_wgen_scratch_size(w, mult_type, region_type, divide_type, arg1, arg2);
    }
}
```

```
extern int gf_size(gf_t *gf)
{
    gf_internal_t *h;
    int s;

    s = sizeof(gf_t);
    h = (gf_internal_t *) gf->scratch;
    s += gf_scratch_size(h->w, h->mult_type, h->region_type, h->divide_type, h->arg1, h->arg2);
    if (h->mult_type == GF_MULT_COMPOSITE) s += gf_size(h->base_gf);
    return s;
}
```

```
int gf_init_easy(gf_t *gf, int w)
{
    return gf_init_hard(gf, w, GF_MULT_DEFAULT, GF_REGION_DEFAULT, GF_DIVIDE_DEFAULT,
                        0, 0, 0, NULL, NULL);
}
```

/\* Allen: What's going on here is this function is putting info into the  
scratch mem of gf, and then calling the relevant REAL init  
func for the word size. Probably done this way to consolidate  
those aspects of initialization that don't rely on word size,  
and then take care of word-size-specific stuff. \*/

```
int gf_init_hard(gf_t *gf, int w, int mult_type,
                 int region_type,
                 int divide_type,
                 uint64_t prim_poly,
                 int arg1, int arg2,
                 gf_t *base_gf,
                 void *scratch_memory)
{
    int sz;
    gf_internal_t *h;

    if (gf_error_check(w, mult_type, region_type, divide_type,
                      arg1, arg2, prim_poly, base_gf) == 0) return 0;

    sz = gf_scratch_size(w, mult_type, region_type, divide_type, arg1, arg2);
    if (sz <= 0) return 0; /* This shouldn't happen, as all errors should get caught
                           in gf_error_check() */

    if (scratch_memory == NULL) {
        h = (gf_internal_t *) malloc(sz);
        h->free_me = 1;
    }
}
```



*src/gf.c lines 481 to 540*

```
    } else {
        h = scratch_memory;
        h->free_me = 0;
    }
    gf->scratch = (void *) h;
    h->mult_type = mult_type;
    h->region_type = region_type;
    h->divide_type = divide_type;
    h->w = w;
    h->prim_poly = prim_poly;
    h->arg1 = arg1;
    h->arg2 = arg2;
    h->base_gf = base_gf;
    h->private = (void *) gf->scratch;
    h->private = (uint8_t *)h->private + (sizeof(gf_internal_t));
    gf->extract_word.w32 = NULL;

    switch(w) {
        case 4: return gf_w4_init(gf);
        case 8: return gf_w8_init(gf);
        case 16: return gf_w16_init(gf);
        case 32: return gf_w32_init(gf);
        case 64: return gf_w64_init(gf);
        case 128: return gf_w128_init(gf);
        default: return gf_wgen_init(gf);
    }
}

int gf_free(gf_t *gf, int recursive)
{
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    if (recursive && h->base_gf != NULL) {
        gf_free(h->base_gf, 1);
        free(h->base_gf);
    }
    if (h->free_me) free(h);
    return 0; /* Making compiler happy */
}

void gf_alignment_error(char *s, int a)
{
    fprintf(stderr, "Alignment error in %s:\n", s);
    fprintf(stderr, "    The source and destination buffers must be aligned to each other,\n");
    fprintf(stderr, "    and they must be aligned to a %d-byte address.\n", a);
    exit(1);
}

static
void gf_invert_binary_matrix(uint32_t *mat, uint32_t *inv, int rows) {
    int cols, i, j;
    uint32_t tmp;

    cols = rows;

    for (i = 0; i < rows; i++) inv[i] = (1 << i);

    /* First -- convert into upper triangular */
}
```



*src/gf.c lines 541 to 600*

```
for (i = 0; i < cols; i++) {

    /* Swap rows if we ave a zero i,i element.  If we can't swap, then the
       matrix was not invertible */

    if ((mat[i] & (1 << i)) == 0) {
        for (j = i+1; j < rows && (mat[j] & (1 << i)) == 0; j++) ;
        if (j == rows) {
            fprintf(stderr, "galois_invert_matrix: Matrix not invertible!!\n");
            exit(1);
        }
        tmp = mat[i]; mat[i] = mat[j]; mat[j] = tmp;
        tmp = inv[i]; inv[i] = inv[j]; inv[j] = tmp;
    }

    /* Now for each j>i, add A_ji*Ai to Aj */
    for (j = i+1; j != rows; j++) {
        if ((mat[j] & (1 << i)) != 0) {
            mat[j] ^= mat[i];
            inv[j] ^= inv[i];
        }
    }
}

/* Now the matrix is upper triangular.  Start at the top and multiply down */

for (i = rows-1; i >= 0; i--) {
    for (j = 0; j < i; j++) {
        if (mat[j] & (1 << i)) {
            /* mat[j] ^= mat[i]; */
            inv[j] ^= inv[i];
        }
    }
}

uint32_t gf_bitmatrix_inverse(uint32_t y, int w, uint32_t pp)
{
    uint32_t mat[32], inv[32], mask;
    int i;

    mask = (w == 32) ? 0xffffffff : (1 << w) - 1;
    for (i = 0; i < w; i++) {
        mat[i] = y;

        if (y & (1 << (w-1))) {
            y = y << 1;
            y = ((y ^ pp) & mask);
        } else {
            y = y << 1;
        }
    }

    gf_invert_binary_matrix(mat, inv, w);
    return inv[0];
}

void gf_two_byte_region_table_multiply(gf_region_data *rd, uint16_t *base)
{
    uint64_t a, prod;
```



*src/gf.c lines 601 to 660*

```
int xor;
uint64_t *s64, *d64, *top;

s64 = rd->s_start;
d64 = rd->d_start;
top = rd->d_top;
xor = rd->xor;

if (xor) {
    while (d64 != top) {
        a = *s64;
        prod = base[a >> 48];
        a <<= 16;
        prod <<= 16;
        prod ^= base[a >> 48];
        a <<= 16;
        prod <<= 16;
        prod ^= base[a >> 48];
        a <<= 16;
        prod <<= 16;
        prod ^= base[a >> 48];
        prod ^= *d64;
        *d64 = prod;
        s64++;
        d64++;
    }
} else {
    while (d64 != top) {
        a = *s64;
        prod = base[a >> 48];
        a <<= 16;
        prod <<= 16;
        prod ^= base[a >> 48];
        a <<= 16;
        prod <<= 16;
        prod ^= base[a >> 48];
        a <<= 16;
        prod <<= 16;
        prod ^= base[a >> 48];
        *d64 = prod;
        s64++;
        d64++;
    }
}
}

static void gf_slow_multiply_region(gf_region_data *rd, void *src, void *dest, void *s_top)
{
    uint8_t *s8, *d8;
    uint16_t *s16, *d16;
    uint32_t *s32, *d32;
    uint64_t *s64, *d64;
    gf_internal_t *h;
    int wb;
    uint32_t p, a;

    h = rd->gf->scratch;
    wb = (h->w)/8;
    if (wb == 0) wb = 1;
```



*src/gf.c lines 661 to 720*

```
while (src < s_top) {
    switch (h->w) {
        case 8:
            s8 = (uint8_t *) src;
            d8 = (uint8_t *) dest;
            *d8 = (rd->xor) ? (*d8 ^ rd->gf->multiply.w32(rd->gf, rd->val, *s8)) :
                           rd->gf->multiply.w32(rd->gf, rd->val, *s8);

            break;
        case 4:
            s8 = (uint8_t *) src;
            d8 = (uint8_t *) dest;
            a = *s8;
            p = rd->gf->multiply.w32(rd->gf, rd->val, a&0xf);
            p |= (rd->gf->multiply.w32(rd->gf, rd->val, a >> 4) << 4);
            if (rd->xor) p ^= *d8;
            *d8 = p;
            break;
        case 16:
            s16 = (uint16_t *) src;
            d16 = (uint16_t *) dest;
            *d16 = (rd->xor) ? (*d16 ^ rd->gf->multiply.w32(rd->gf, rd->val, *s16)) :
                             rd->gf->multiply.w32(rd->gf, rd->val, *s16);

            break;
        case 32:
            s32 = (uint32_t *) src;
            d32 = (uint32_t *) dest;
            *d32 = (rd->xor) ? (*d32 ^ rd->gf->multiply.w32(rd->gf, rd->val, *s32)) :
                             rd->gf->multiply.w32(rd->gf, rd->val, *s32);

            break;
        case 64:
            s64 = (uint64_t *) src;
            d64 = (uint64_t *) dest;
            *d64 = (rd->xor) ? (*d64 ^ rd->gf->multiply.w64(rd->gf, rd->val, *s64)) :
                             rd->gf->multiply.w64(rd->gf, rd->val, *s64);

            break;
        default:
            fprintf(stderr, "Error: gf_slow_multiply_region: w=%d not implemented.\n", h->w);
            exit(1);
    }
    src = (uint8_t *)src + wb;
    dest = (uint8_t *)dest + wb;
}
}
```

/\* JSP - The purpose of this procedure is to error check alignment,  
and to set up the region operation so that it can best leverage  
large words.

It stores its information in rd.

Assuming you're not doing Cauchy coding, (see below for that),  
then w will be 4, 8, 16, 32 or 64. It can't be 128 (probably  
should change that).

src and dest must then be aligned on ceil(w/8)-byte boundaries.  
Moreover, bytes must be a multiple of ceil(w/8). If the variable  
align is equal to ceil(w/8), then we will set s\_start = src,  
d\_start = dest, s\_top to (src+bytes) and d\_top to (dest+bytes).  
And we return -- the implementation will go ahead and do the  
multiplication on individual words (e.g. using discrete logs).



If align is greater than  $\text{ceil}(w/8)$ , then the implementation needs to work on groups of "align" bytes. For example, suppose you are implementing BYTWO, without SSE. Then you will be doing the region multiplication in units of 8 bytes, so align = 8. Or, suppose you are doing a Quad table in  $\text{GF}(2^4)$ . You will be doing the region multiplication in units of 2 bytes, so align = 2. Or, suppose you are doing split multiplication with SSE operations in  $\text{GF}(2^8)$ . Then align = 16. Worse yet, suppose you are doing split multiplication with SSE operations in  $\text{GF}(2^{16})$ , with or without ALTMAP. Then, you will be doing the multiplication on 256 bits at a time. So align = 32.

When align does not equal  $\text{ceil}(w/8)$ , we split the region multiplication into three parts. We are going to make s\_start be the first address greater than or equal to src that is a multiple of align. s\_top is going to be the largest address  $\geq \text{src} + \text{bytes}$  such that  $(\text{s\_top} - \text{s\_start})$  is a multiple of align. We do the same with d\_start and d\_top. When we say that "src and dest must be aligned with respect to each other, we mean that  $\text{s\_start} - \text{src}$  must equal  $\text{d\_start} - \text{dest}$ .

Now, the region multiplication is done in three parts -- the part between src and s\_start must be done using single words. Similarly, the part between s\_top and  $\text{src} + \text{bytes}$  must also be done using single words. The part between s\_start and s\_top will be done in chunks of "align" bytes.

One final thing -- if align > 16, then s\_start and d\_start will be aligned on a 16 byte boundary. Perhaps we should have two variables: align and chunksize. Then we'd have s\_start & d\_start aligned to "align", and have  $\text{s\_top} - \text{s\_start}$  be a multiple of chunksize. That may be less confusing, but it would be a big change.

Finally, if align = -1, then we are doing Cauchy multiplication, using only XOR's. In this case, we're not going to care about alignment because we are just doing XOR's. Instead, the only thing we care about is that bytes must be a multiple of w.

This is not to say that alignment doesn't matter in performance with XOR's. See that discussion in `gf_multby_one()`.

After you call `gf_set_region_data()`, the procedure `gf_do_initial_region_alignment()` calls `gf->multiply.w32()` on everything between src and s\_start. The procedure `gf_do_final_region_alignment()` calls `gf->multiply.w32()` on everything between s\_top and  $\text{src} + \text{bytes}$ .  
\*/

```
void gf_set_region_data(gf_region_data *rd,
    gf_t *gf,
    void *src,
    void *dest,
    int bytes,
    uint64_t val,
    int xor,
    int align)
{
    gf_internal_t *h = NULL;
```



*src/gf.c lines 781 to 840*

```
int wb;
uint32_t a;
unsigned long uls, uld;

if (gf == NULL) { /* JSP - Can be NULL if you're just doing XOR's */
    wb = 1;
} else {
    h = gf->scratch;
    wb = (h->w)/8;
    if (wb == 0) wb = 1;
}

rd->gf = gf;
rd->src = src;
rd->dest = dest;
rd->bytes = bytes;
rd->val = val;
rd->xor = xor;
rd->align = align;

uls = (unsigned long) src;
uld = (unsigned long) dest;

a = (align <= 16) ? align : 16;

if (align == -1) { /* JSP: This is cauchy. Error check bytes, then set up the pointers
                    so that there are no alignment regions. */
    if (h != NULL && bytes % h->w != 0) {
        fprintf(stderr, "Error in region multiply operation.\n");
        fprintf(stderr, "The size must be a multiple of %d bytes.\n", h->w);
        exit(1);
    }

    rd->s_start = src;
    rd->d_start = dest;
    rd->s_top = (uint8_t *)src + bytes;
    rd->d_top = (uint8_t *)src + bytes;
    return;
}

if (uls % a != uld % a) {
    fprintf(stderr, "Error in region multiply operation.\n");
    fprintf(stderr, "The source & destination pointers must be aligned with respect\n");
    fprintf(stderr, "to each other along a %d byte boundary.\n", a);
    fprintf(stderr, "Src = 0x%lx. Dest = 0x%lx\n", (unsigned long) src,
        (unsigned long) dest);
    exit(1);
}

if (uls % wb != 0) {
    fprintf(stderr, "Error in region multiply operation.\n");
    fprintf(stderr, "The pointers must be aligned along a %d byte boundary.\n", wb);
    fprintf(stderr, "Src = 0x%lx. Dest = 0x%lx\n", (unsigned long) src,
        (unsigned long) dest);
    exit(1);
}

if (bytes % wb != 0) {
    fprintf(stderr, "Error in region multiply operation.\n");
    fprintf(stderr, "The size must be a multiple of %d bytes.\n", wb);
}
```



*src/gf.c lines 841 to 900*

```
    exit(1);
}

uls %= a;
if (uls != 0) uls = (a-uls);
rd->s_start = (uint8_t *)rd->src + uls;
rd->d_start = (uint8_t *)rd->dest + uls;
bytes -= uls;
bytes -= (bytes % align);
rd->s_top = (uint8_t *)rd->s_start + bytes;
rd->d_top = (uint8_t *)rd->d_start + bytes;
}

void gf_do_initial_region_alignment(gf_region_data *rd)
{
    gf_slow_multiply_region(rd, rd->src, rd->dest, rd->s_start);
}

void gf_do_final_region_alignment(gf_region_data *rd)
{
    gf_slow_multiply_region(rd, rd->s_top, rd->d_top, (uint8_t *)rd->src+rd->bytes);
}

void gf_multby_zero(void *dest, int bytes, int xor)
{
    if (xor) return;
    bzero(dest, bytes);
    return;
}

/* JSP - gf_multby_one tries to do this in the most efficient way
   possible.  If xor = 0, then simply call memcpy() since that
   should be optimized by the system.  Otherwise, try to do the xor
   in the following order:

   If src and dest are aligned with respect to each other on 16-byte
   boundaries and you have SSE instructions, then use aligned SSE
   instructions.

   If they aren't but you still have SSE instructions, use unaligned
   SSE instructions.

   If there are no SSE instructions, but they are aligned with
   respect to each other on 8-byte boundaries, then do them with
   uint64_t's.

   Otherwise, call gf_unaligned_xor(), which does the following:
   align a destination pointer along an 8-byte boundary, and then
   memcpy 32 bytes at a time from the src pointer to an array of
   doubles.  I'm not sure if that's the best -- probably needs
   testing, but this seems like it could be a black hole.
*/

static void gf_unaligned_xor(void *src, void *dest, int bytes);

void gf_multby_one(void *src, void *dest, int bytes, int xor)
{
#ifdef INTEL_SSE2
    __m128i ms, md;
```



*src/gf.c lines 901 to 960*

```
#endif
unsigned long uls, uld;
uint8_t *s8, *d8;
uint64_t *s64, *d64, *dtop64;
gf_region_data rd;

if (!xor) {
    memcpy(dest, src, bytes);
    return;
}
uls = (unsigned long) src;
uld = (unsigned long) dest;

#ifdef INTEL_SSE2
int abytes;
s8 = (uint8_t *) src;
d8 = (uint8_t *) dest;
if (uls % 16 == uld % 16) {
    gf_set_region_data(&rd, NULL, src, dest, bytes, 1, xor, 16);
    while (s8 != rd.s_start) {
        *d8 ^= *s8;
        d8++;
        s8++;
    }
    while (s8 < (uint8_t *) rd.s_top) {
        ms = _mm_load_si128 ((__m128i *) (s8));
        md = _mm_load_si128 ((__m128i *) (d8));
        md = _mm_xor_si128(md, ms);
        _mm_store_si128((__m128i *) (d8), md);
        s8 += 16;
        d8 += 16;
    }
    while (s8 != (uint8_t *) src + bytes) {
        *d8 ^= *s8;
        d8++;
        s8++;
    }
    return;
}

abytes = (bytes & 0xffffffff0);

while (d8 < (uint8_t *) dest + abytes) {
    ms = _mm_loadu_si128 ((__m128i *) (s8));
    md = _mm_loadu_si128 ((__m128i *) (d8));
    md = _mm_xor_si128(md, ms);
    _mm_storeu_si128((__m128i *) (d8), md);
    s8 += 16;
    d8 += 16;
}
while (d8 != (uint8_t *) dest + bytes) {
    *d8 ^= *s8;
    d8++;
    s8++;
}
return;
#endif

if (uls % 8 != uld % 8) {
    gf_unaligned_xor(src, dest, bytes);
}
```



*src/gf.c lines 961 to 1020*

```
    return;
}

gf_set_region_data(&rd, NULL, src, dest, bytes, 1, xor, 8);
s8 = (uint8_t *) src;
d8 = (uint8_t *) dest;
while (d8 != rd.d_start) {
    *d8 ^= *s8;
    d8++;
    s8++;
}
dtop64 = (uint64_t *) rd.d_top;

d64 = (uint64_t *) rd.d_start;
s64 = (uint64_t *) rd.s_start;

while (d64 < dtop64) {
    *d64 ^= *s64;
    d64++;
    s64++;
}

s8 = (uint8_t *) rd.s_top;
d8 = (uint8_t *) rd.d_top;

while (d8 != (uint8_t *) dest+bytes) {
    *d8 ^= *s8;
    d8++;
    s8++;
}
return;
}

#define UNALIGNED_BUFSIZE (8)

static void gf_unaligned_xor(void *src, void *dest, int bytes)
{
    uint64_t scopy[UNALIGNED_BUFSIZE], *d64;
    int i;
    gf_region_data rd;
    uint8_t *s8, *d8;

    /* JSP - call gf_set_region_data(), but use dest in both places. This is
       because I only want to set up dest. If I used src, gf_set_region_data()
       would fail because src and dest are not aligned to each other wrt
       8-byte pointers. I know this will actually align d_start to 16 bytes.
       If I change gf_set_region_data() to split alignment & chunksize, then
       I could do this correctly. */

    gf_set_region_data(&rd, NULL, dest, dest, bytes, 1, 1, 8*UNALIGNED_BUFSIZE);
    s8 = (uint8_t *) src;
    d8 = (uint8_t *) dest;

    while (d8 < (uint8_t *) rd.d_start) {
        *d8 ^= *s8;
        d8++;
        s8++;
    }

    d64 = (uint64_t *) d8;
```



*src/gf.c lines 1021 to 1036*

```
while (d64 < (uint64_t *) rd.d_top) {
    memcpy(scopy, s8, 8*UNALIGNED_BUFSIZE);
    s8 += 8*UNALIGNED_BUFSIZE;
    for (i = 0; i < UNALIGNED_BUFSIZE; i++) {
        *d64 ^= scopy[i];
        d64++;
    }
}

d8 = (uint8_t *) d64;
while (d8 < (uint8_t *) ((uint8_t *)dest+bytes)) {
    *d8 ^= *s8;
    d8++;
    s8++;
}
}
```



*src/gf\_general.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_general.c
 *
 * This file has helper routines for doing basic GF operations with any
 * legal value of w. The problem is that w <= 32, w=64 and w=128 all have
 * different data types, which is a pain. The procedures in this file try
 * to alleviate that pain. They are used in gf_unit and gf_time.
 */
```

```
#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
```

```
#include "gf_complete.h"
#include "gf_int.h"
#include "gf_method.h"
#include "gf_rand.h"
#include "gf_general.h"
```

```
void gf_general_set_zero(gf_general_t *v, int w)
{
    if (w <= 32) {
        v->w32 = 0;
    } else if (w <= 64) {
        v->w64 = 0;
    } else {
        v->w128[0] = 0;
        v->w128[1] = 0;
    }
}
```

```
void gf_general_set_one(gf_general_t *v, int w)
{
    if (w <= 32) {
        v->w32 = 1;
    } else if (w <= 64) {
        v->w64 = 1;
    } else {
        v->w128[0] = 0;
        v->w128[1] = 1;
    }
}
```

```
void gf_general_set_two(gf_general_t *v, int w)
{
    if (w <= 32) {
        v->w32 = 2;
    } else if (w <= 64) {
        v->w64 = 2;
    } else {
        v->w128[0] = 0;
        v->w128[1] = 2;
    }
}
```



*src/gf\_general.c lines 61 to 120*

```
}

int gf_general_is_zero(gf_general_t *v, int w)
{
    if (w <= 32) {
        return (v->w32 == 0);
    } else if (w <= 64) {
        return (v->w64 == 0);
    } else {
        return (v->w128[0] == 0 && v->w128[1] == 0);
    }
}

int gf_general_is_one(gf_general_t *v, int w)
{
    if (w <= 32) {
        return (v->w32 == 1);
    } else if (w <= 64) {
        return (v->w64 == 1);
    } else {
        return (v->w128[0] == 0 && v->w128[1] == 1);
    }
}

void gf_general_set_random(gf_general_t *v, int w, int zero_ok)
{
    if (w <= 32) {
        v->w32 = MOA_Random_W(w, zero_ok);
    } else if (w <= 64) {
        while (1) {
            v->w64 = MOA_Random_64();
            if (v->w64 != 0 || zero_ok) return;
        }
    } else {
        while (1) {
            MOA_Random_128(v->w128);
            if (v->w128[0] != 0 || v->w128[1] != 0 || zero_ok) return;
        }
    }
}

void gf_general_val_to_s(gf_general_t *v, int w, char *s, int hex)
{
    if (w <= 32) {
        if (hex) {
            sprintf(s, "%x", v->w32);
        } else {
            sprintf(s, "%u", v->w32);
        }
    } else if (w <= 64) {
        if (hex) {
            sprintf(s, "%llx", (long long unsigned int) v->w64);
        } else {
            sprintf(s, "%lld", (long long unsigned int) v->w64);
        }
    } else {
        if (v->w128[0] == 0) {
            sprintf(s, "%llx", (long long unsigned int) v->w128[1]);
        } else {
            sprintf(s, "%llx%016llx", (long long unsigned int) v->w128[0],
```



*src/gf\_general.c lines 121 to 180*

```
                (long long unsigned int) v->w128[1]));
    }
}

int gf_general_s_to_val(gf_general_t *v, int w, char *s, int hex)
{
    int l;
    int save;

    if (w <= 32) {
        if (hex) {
            if (sscanf(s, "%x", &(v->w32)) == 0) return 0;
        } else {
            if (sscanf(s, "%u", &(v->w32)) == 0) return 0;
        }
        if (w == 32) return 1;
        if (w == 31) {
            if (v->w32 & (1 << 31)) return 0;
            return 1;
        }
        if (v->w32 & ~((1 << w)-1)) return 0;
        return 1;
    } else if (w <= 64) {
        if (hex) return (sscanf(s, "%llx", (long long unsigned int *) (&(v->w64))) == 1);
        return (sscanf(s, "%lld", (long long int *) (&(v->w64))) == 1);
    } else {
        if (!hex) return 0;
        l = strlen(s);
        if (l <= 16) {
            v->w128[0] = 0;
            return (sscanf(s, "%llx", (long long unsigned int *) (&(v->w128[1]))) == 1);
        } else {
            if (l > 32) return 0;
            save = s[l-16];
            s[l-16] = '\\0';
            if (sscanf(s, "%llx", (long long unsigned int *) (&(v->w128[0]))) == 0) {
                s[l-16] = save;
                return 0;
            }
            return (sscanf(s+(l-16), "%llx", (long long unsigned int *) (&(v->w128[1]))) == 1);
        }
    }
}

void gf_general_add(gf_t *gf, gf_general_t *a, gf_general_t *b, gf_general_t *c)
{
    gf_internal_t *h;
    int w;

    h = (gf_internal_t *) gf->scratch;
    w = h->w;

    if (w <= 32) {
        c->w32 = a->w32 ^ b->w32;
    } else if (w <= 64) {
        c->w64 = a->w64 ^ b->w64;
    } else {
        c->w128[0] = a->w128[0] ^ b->w128[0];
        c->w128[1] = a->w128[1] ^ b->w128[1];
    }
}
```



*src/gf\_general.c lines 181 to 240*

```
    }  
}  
  
void gf_general_multiply(gf_t *gf, gf_general_t *a, gf_general_t *b, gf_general_t *c)  
{  
    gf_internal_t *h;  
    int w;  
  
    h = (gf_internal_t *) gf->scratch;  
    w = h->w;  
  
    if (w <= 32) {  
        c->w32 = gf->multiply.w32(gf, a->w32, b->w32);  
    } else if (w <= 64) {  
        c->w64 = gf->multiply.w64(gf, a->w64, b->w64);  
    } else {  
        gf->multiply.w128(gf, a->w128, b->w128, c->w128);  
    }  
}  
  
void gf_general_divide(gf_t *gf, gf_general_t *a, gf_general_t *b, gf_general_t *c)  
{  
    gf_internal_t *h;  
    int w;  
  
    h = (gf_internal_t *) gf->scratch;  
    w = h->w;  
  
    if (w <= 32) {  
        c->w32 = gf->divide.w32(gf, a->w32, b->w32);  
    } else if (w <= 64) {  
        c->w64 = gf->divide.w64(gf, a->w64, b->w64);  
    } else {  
        gf->divide.w128(gf, a->w128, b->w128, c->w128);  
    }  
}  
  
void gf_general_inverse(gf_t *gf, gf_general_t *a, gf_general_t *b)  
{  
    gf_internal_t *h;  
    int w;  
  
    h = (gf_internal_t *) gf->scratch;  
    w = h->w;  
  
    if (w <= 32) {  
        b->w32 = gf->inverse.w32(gf, a->w32);  
    } else if (w <= 64) {  
        b->w64 = gf->inverse.w64(gf, a->w64);  
    } else {  
        gf->inverse.w128(gf, a->w128, b->w128);  
    }  
}  
  
int gf_general_are_equal(gf_general_t *v1, gf_general_t *v2, int w)  
{  
    if (w <= 32) {  
        return (v1->w32 == v2->w32);  
    } else if (w <= 64) {  
        return (v1->w64 == v2->w64);  
    }  
}
```



*src/gf\_general.c lines 241 to 300*

```
    } else {  
        return (v1->w128[0] == v2->w128[0] &&  
                v1->w128[1] == v2->w128[1]);  
    }  
}
```

```
void gf_general_do_region_multiply(gf_t *gf, gf_general_t *a, void *ra, void *rb, int bytes, int xor)
```

```
{  
    gf_internal_t *h;  
    int w;  
  
    h = (gf_internal_t *) gf->scratch;  
    w = h->w;  
  
    if (w <= 32) {  
        gf->multiply_region.w32(gf, ra, rb, a->w32, bytes, xor);  
    } else if (w <= 64) {  
        gf->multiply_region.w64(gf, ra, rb, a->w64, bytes, xor);  
    } else {  
        gf->multiply_region.w128(gf, ra, rb, a->w128, bytes, xor);  
    }  
}
```

```
void gf_general_do_region_check(gf_t *gf, gf_general_t *a, void *orig_a,  
                               void *orig_target, void *final_target, int bytes, int xor)
```

```
{  
    gf_internal_t *h;  
    int w, words, i;  
    gf_general_t oa, ot, ft, sb;  
    char sa[50], soa[50], sot[50], sft[50], ssb[50];  
  
    h = (gf_internal_t *) gf->scratch;  
    w = h->w;  
  
    words = (bytes * 8) / w;  
    for (i = 0; i < words; i++) {  
        if (w <= 32) {  
            oa.w32 = gf->extract_word.w32(gf, orig_a, bytes, i);  
            ot.w32 = gf->extract_word.w32(gf, orig_target, bytes, i);  
            ft.w32 = gf->extract_word.w32(gf, final_target, bytes, i);  
            sb.w32 = gf->multiply.w32(gf, a->w32, oa.w32);  
            if (xor) sb.w32 ^= ot.w32;  
        } else if (w <= 64) {  
            oa.w64 = gf->extract_word.w64(gf, orig_a, bytes, i);  
            ot.w64 = gf->extract_word.w64(gf, orig_target, bytes, i);  
            ft.w64 = gf->extract_word.w64(gf, final_target, bytes, i);  
            sb.w64 = gf->multiply.w64(gf, a->w64, oa.w64);  
            if (xor) sb.w64 ^= ot.w64;  
        } else {  
            gf->extract_word.w128(gf, orig_a, bytes, i, oa.w128);  
            gf->extract_word.w128(gf, orig_target, bytes, i, ot.w128);  
            gf->extract_word.w128(gf, final_target, bytes, i, ft.w128);  
            gf->multiply.w128(gf, a->w128, oa.w128, sb.w128);  
            if (xor) {  
                sb.w128[0] ^= ot.w128[0];  
                sb.w128[1] ^= ot.w128[1];  
            }  
        }  
    }  
  
    if (!gf_general_are_equal(&ft, &sb, w)) {
```



*src/gf\_general.c lines 301 to 360*

```
    fprintf(stderr, "Problem with region multiply (all values in hex):\n");
    fprintf(stderr, "    Target address base: 0x%x. Word 0x%x of 0x%x. Xor: %d\n",
        (unsigned long) final_target, i, words, xor);
    gf_general_val_to_s(a, w, sa, 1);
    gf_general_val_to_s(&oa, w, soa, 1);
    gf_general_val_to_s(&ot, w, sot, 1);
    gf_general_val_to_s(&ft, w, sft, 1);
    gf_general_val_to_s(&sb, w, ssb, 1);
    fprintf(stderr, "    Value: %s\n", sa);
    fprintf(stderr, "    Original source word: %s\n", soa);
    if (xor) fprintf(stderr, "    XOR with target word: %s\n", sot);
    fprintf(stderr, "    Product word: %s\n", sft);
    fprintf(stderr, "    It should be: %s\n", ssb);
    exit(0);
}
}
```

```
void gf_general_set_up_single_timing_test(int w, void *ra, void *rb, int size)
{
```

```
    void *top;
    gf_general_t g;
    uint8_t *r8, *r8a;
    uint16_t *r16;
    uint32_t *r32;
    uint64_t *r64;
    int i;
```

```
    top = (uint8_t *)rb+size;
```

```
    /* If w is 8, 16, 32, 64 or 128, fill the regions with random bytes.
       However, don't allow for zeros in rb, because that will screw up
       division.
```

```
       When w is 4, you fill the regions with random 4-bit words in each byte.
```

```
       Otherwise, treat every four bytes as an uint32_t
       and fill it with a random value mod (1 << w).
    */
```

```
    if (w == 8 || w == 16 || w == 32 || w == 64 || w == 128) {
        MOA_Fill_Random_Region (ra, size);
        while (rb < top) {
            gf_general_set_random(&g, w, 0);
            switch (w) {
                case 8:
                    r8 = (uint8_t *) rb;
                    *r8 = g.w32;
                    break;
                case 16:
                    r16 = (uint16_t *) rb;
                    *r16 = g.w32;
                    break;
                case 32:
                    r32 = (uint32_t *) rb;
                    *r32 = g.w32;
                    break;
                case 64:
                    r64 = (uint64_t *) rb;
```



*src/gf\_general.c lines 361 to 420*

```
        *r64 = g.w64;
        break;
    case 128:
        r64 = (uint64_t *) rb;
        r64[0] = g.w128[0];
        r64[1] = g.w128[1];
        break;
    }
    rb = (uint8_t *)rb + (w/8);
}
} else if (w == 4) {
    r8a = (uint8_t *) ra;
    r8 = (uint8_t *) rb;
    while (r8 < (uint8_t *) top) {
        gf_general_set_random(&g, w, 1);
        *r8a = g.w32;
        gf_general_set_random(&g, w, 0);
        *r8 = g.w32;
        r8a++;
        r8++;
    }
} else {
    r32 = (uint32_t *) ra;
    for (i = 0; i < size/4; i++) r32[i] = MOA_Random_W(w, 1);
    r32 = (uint32_t *) rb;
    for (i = 0; i < size/4; i++) r32[i] = MOA_Random_W(w, 0);
}
}
```

/\* This sucks, but in order to time, you really need to avoid putting ifs in the inner loops. So, I'm doing a separate timing test for each w: (4 & 8), 16, 32, 64, 128 and everything else. Fortunately, the "everything else" tests can be equivalent to w=32.

I'm also putting the results back into ra, because otherwise, the optimizer might figure out that we're not really doing anything in the inner loops and it will chuck that. \*/

```
int gf_general_do_single_timing_test(gf_t *gf, void *ra, void *rb, int size, char test)
{
    gf_internal_t *h;
    void *top;
    uint8_t *r8a, *r8b, *top8;
    uint16_t *r16a, *r16b, *top16;
    uint32_t *r32a, *r32b, *top32;
    uint64_t *r64a, *r64b, *top64, *r64c;
    int w, rv;

    h = (gf_internal_t *) gf->scratch;
    w = h->w;
    top = (uint8_t *)ra + size;

    if (w == 8 || w == 4) {
        r8a = (uint8_t *) ra;
        r8b = (uint8_t *) rb;
        top8 = (uint8_t *) top;
        if (test == 'M') {
            while (r8a < top8) {
                *r8a = gf->multiply.w32(gf, *r8a, *r8b);
                r8a++;
            }
        }
    }
}
```



*src/gf\_general.c lines 421 to 480*

```
        r8b++;
    }
} else if (test == 'D') {
    while (r8a < top8) {
        *r8a = gf->divide.w32(gf, *r8a, *r8b);
        r8a++;
        r8b++;
    }
} else if (test == 'I') {
    while (r8a < top8) {
        *r8a = gf->inverse.w32(gf, *r8a);
        r8a++;
    }
}
return (top8 - (uint8_t *) ra);
}

if (w == 16) {
    r16a = (uint16_t *) ra;
    r16b = (uint16_t *) rb;
    top16 = (uint16_t *) top;
    if (test == 'M') {
        while (r16a < top16) {
            *r16a = gf->multiply.w32(gf, *r16a, *r16b);
            r16a++;
            r16b++;
        }
    } else if (test == 'D') {
        while (r16a < top16) {
            *r16a = gf->divide.w32(gf, *r16a, *r16b);
            r16a++;
            r16b++;
        }
    } else if (test == 'I') {
        while (r16a < top16) {
            *r16a = gf->inverse.w32(gf, *r16a);
            r16a++;
        }
    }
    return (top16 - (uint16_t *) ra);
}
if (w <= 32) {
    r32a = (uint32_t *) ra;
    r32b = (uint32_t *) rb;
    top32 = (uint32_t *) ra + (size/4); /* This is for the "everything else" */

    if (test == 'M') {
        while (r32a < top32) {
            *r32a = gf->multiply.w32(gf, *r32a, *r32b);
            r32a++;
            r32b++;
        }
    } else if (test == 'D') {
        while (r32a < top32) {
            *r32a = gf->divide.w32(gf, *r32a, *r32b);
            r32a++;
            r32b++;
        }
    } else if (test == 'I') {
        while (r32a < top32) {
```



*src/gf\_general.c lines 481 to 539*

```
        *r32a = gf->inverse.w32(gf, *r32a);
        r32a++;
    }
}
return (top32 - (uint32_t *) ra);
}
if (w == 64) {
    r64a = (uint64_t *) ra;
    r64b = (uint64_t *) rb;
    top64 = (uint64_t *) top;
    if (test == 'M') {
        while (r64a < top64) {
            *r64a = gf->multiply.w64(gf, *r64a, *r64b);
            r64a++;
            r64b++;
        }
    } else if (test == 'D') {
        while (r64a < top64) {
            *r64a = gf->divide.w64(gf, *r64a, *r64b);
            r64a++;
            r64b++;
        }
    } else if (test == 'I') {
        while (r64a < top64) {
            *r64a = gf->inverse.w64(gf, *r64a);
            r64a++;
        }
    }
    return (top64 - (uint64_t *) ra);
}
if (w == 128) {
    r64a = (uint64_t *) ra;
    r64c = r64a;
    r64a += 2;
    r64b = (uint64_t *) rb;
    top64 = (uint64_t *) top;
    rv = (top64 - r64a)/2;
    if (test == 'M') {
        while (r64a < top64) {
            gf->multiply.w128(gf, r64a, r64b, r64c);
            r64a += 2;
            r64b += 2;
        }
    } else if (test == 'D') {
        while (r64a < top64) {
            gf->divide.w128(gf, r64a, r64b, r64c);
            r64a += 2;
            r64b += 2;
        }
    } else if (test == 'I') {
        while (r64a < top64) {
            gf->inverse.w128(gf, r64a, r64c);
            r64a += 2;
        }
    }
    return rv;
}
return 0;
}
```



*src/gf\_method.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_method.c
 *
 * Parses argv to figure out the mult_type and arguments.  Returns the gf.
 */
```

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
```

```
#include "gf_complete.h"
#include "gf_int.h"
#include "gf_method.h"
```

```
int create_gf_from_argv(gf_t *gf, int w, int argc, char **argv, int starting)
{
```

```
    int mult_type, divide_type, region_type;
    int arg1, arg2;
    uint64_t prim_poly;
    gf_t *base;
```

```
    mult_type = GF_MULT_DEFAULT;
    region_type = GF_REGION_DEFAULT;
    divide_type = GF_DIVIDE_DEFAULT;
    prim_poly = 0;
    base = NULL;
```

```
    arg1 = 0;
    arg2 = 0;
```

```
    while (1) {
```

```
        if (argc > starting) {
```

```
            if (strcmp(argv[starting], "-m") == 0) {
                starting++;
```

```
                if (mult_type != GF_MULT_DEFAULT) {
                    if (base != NULL) gf_free(base, 1);
                    _gf_errno = GF_E_TWOMULT;
                    return 0;
                }
```

```
                if (strcmp(argv[starting], "SHIFT") == 0) {
                    mult_type = GF_MULT_SHIFT;
                    starting++;
```

```
                } else if (strcmp(argv[starting], "CARRY_FREE") == 0) {
                    mult_type = GF_MULT_CARRY_FREE;
                    starting++;
```

```
                } else if (strcmp(argv[starting], "CARRY_FREE_GK") == 0) {
                    mult_type = GF_MULT_CARRY_FREE_GK;
                    starting++;
```

```
                } else if (strcmp(argv[starting], "GROUP") == 0) {
                    mult_type = GF_MULT_GROUP;
                    if (argc < starting + 3) {
                        _gf_errno = GF_E_GROUPPAR;
                        return 0;
                    }
```

```
                    if (sscanf(argv[starting+1], "%d", &arg1) == 0 ||
                        sscanf(argv[starting+2], "%d", &arg2) == 0) {
```



*src/gf\_method.c lines 61 to 120*

```
    _gf_errno = GF_E_GROUPNU;
    return 0;
}
starting += 3;
} else if (strcmp(argv[starting], "BYTWO_p") == 0) {
    mult_type = GF_MULT_BYTWO_p;
    starting++;
} else if (strcmp(argv[starting], "BYTWO_b") == 0) {
    mult_type = GF_MULT_BYTWO_b;
    starting++;
} else if (strcmp(argv[starting], "TABLE") == 0) {
    mult_type = GF_MULT_TABLE;
    starting++;
} else if (strcmp(argv[starting], "LOG") == 0) {
    mult_type = GF_MULT_LOG_TABLE;
    starting++;
} else if (strcmp(argv[starting], "LOG_ZERO") == 0) {
    mult_type = GF_MULT_LOG_ZERO;
    starting++;
} else if (strcmp(argv[starting], "LOG_ZERO_EXT") == 0) {
    mult_type = GF_MULT_LOG_ZERO_EXT;
    starting++;
} else if (strcmp(argv[starting], "SPLIT") == 0) {
    mult_type = GF_MULT_SPLIT_TABLE;
    if (argc < starting + 3) {
        _gf_errno = GF_E_SPLITAR;
        return 0;
    }
    if (sscanf(argv[starting+1], "%d", &arg1) == 0 ||
        sscanf(argv[starting+2], "%d", &arg2) == 0) {
        _gf_errno = GF_E_SPLITNU;
        return 0;
    }
    starting += 3;
} else if (strcmp(argv[starting], "COMPOSITE") == 0) {
    mult_type = GF_MULT_COMPOSITE;
    if (argc < starting + 2) { _gf_errno = GF_E_FEWARGS; return 0; }
    if (sscanf(argv[starting+1], "%d", &arg1) == 0) {
        _gf_errno = GF_E_COMP_A2;
        return 0;
    }
    starting += 2;
    base = (gf_t *) malloc(sizeof(gf_t));
    starting = create_gf_from_argv(base, w/arg1, argc, argv, starting);
    if (starting == 0) {
        free(base);
        return 0;
    }
} else {
    if (base != NULL) gf_free(base, 1);
    _gf_errno = GF_E_UNKNOWN;
    return 0;
}
} else if (strcmp(argv[starting], "-r") == 0) {
    starting++;
    if (strcmp(argv[starting], "DOUBLE") == 0) {
        region_type |= GF_REGION_DOUBLE_TABLE;
        starting++;
    } else if (strcmp(argv[starting], "QUAD") == 0) {
        region_type |= GF_REGION_QUAD_TABLE;
```



*src/gf\_method.c lines 121 to 180*

```
    starting++;
} else if (strcmp(argv[starting], "LAZY") == 0) {
    region_type |= GF_REGION_LAZY;
    starting++;
} else if (strcmp(argv[starting], "SSE") == 0) {
    region_type |= GF_REGION_SSE;
    starting++;
} else if (strcmp(argv[starting], "NOSSE") == 0) {
    region_type |= GF_REGION_NOSSE;
    starting++;
} else if (strcmp(argv[starting], "CAUCHY") == 0) {
    region_type |= GF_REGION_CAUCHY;
    starting++;
} else if (strcmp(argv[starting], "ALTMAP") == 0) {
    region_type |= GF_REGION_ALTMAP;
    starting++;
} else {
    if (base != NULL) gf_free(base, 1);
    _gf_errno = GF_E_UNK_REG;
    return 0;
}
} else if (strcmp(argv[starting], "-p") == 0) {
    starting++;
    if (sscanf(argv[starting], "%llx", (long long unsigned int *)(&prim_poly)) == 0) {
        if (base != NULL) gf_free(base, 1);
        _gf_errno = GF_E_POLYSPC;
        return 0;
    }
    starting++;
} else if (strcmp(argv[starting], "-d") == 0) {
    starting++;
    if (divide_type != GF_DIVIDE_DEFAULT) {
        if (base != NULL) gf_free(base, 1);
        _gf_errno = GF_E_TWO_DIV;
        return 0;
    } else if (strcmp(argv[starting], "EUCLID") == 0) {
        divide_type = GF_DIVIDE_EUCLID;
        starting++;
    } else if (strcmp(argv[starting], "MATRIX") == 0) {
        divide_type = GF_DIVIDE_MATRIX;
        starting++;
    } else {
        _gf_errno = GF_E_UNK_DIV;
        return 0;
    }
} else if (strcmp(argv[starting], "-") == 0) {
    /*
    printf("Scratch size: %d\n", gf_scratch_size(w,
                                                mult_type, region_type, divide_type, arg1, arg2));
    */
    if (gf_init_hard(gf, w, mult_type, region_type, divide_type,
                    prim_poly, arg1, arg2, base, NULL) == 0) {
        if (base != NULL) gf_free(base, 1);
        return 0;
    } else
        return starting + 1;
} else {
    if (base != NULL) gf_free(base, 1);
    _gf_errno = GF_E_UNKFLAG;
    return 0;
}
```



*src/gf\_method.c lines 181 to 188*

```
    }  
  } else {  
    if (base != NULL) gf_free(base, 1);  
    _gf_errno = GF_E_FEWARGS;  
    return 0;  
  }  
}  
}
```



*src/gf\_rand.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_rand.c -- Random number generator.
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "gf_rand.h"

/* Lifted the "Mother of All" random number generator from http://www.agner.org/random/ */

static uint32_t MOA_X[5];

uint32_t MOA_Random_32() {
    uint64_t sum;
    sum = (uint64_t)2111111111UL * (uint64_t)MOA_X[3] +
        (uint64_t)1492 * (uint64_t)(MOA_X[2]) +
        (uint64_t)1776 * (uint64_t)(MOA_X[1]) +
        (uint64_t)5115 * (uint64_t)(MOA_X[0]) +
        (uint64_t)MOA_X[4];
    MOA_X[3] = MOA_X[2]; MOA_X[2] = MOA_X[1]; MOA_X[1] = MOA_X[0];
    MOA_X[4] = (uint32_t)(sum >> 32);
    MOA_X[0] = (uint32_t)sum;
    return MOA_X[0];
}

uint64_t MOA_Random_64() {
    uint64_t sum;

    sum = MOA_Random_32();
    sum <<= 32;
    sum |= MOA_Random_32();
    return sum;
}

void MOA_Random_128(uint64_t *x) {
    x[0] = MOA_Random_64();
    x[1] = MOA_Random_64();
    return;
}

uint32_t MOA_Random_W(int w, int zero_ok)
{
    uint32_t b;

    do {
        b = MOA_Random_32();
        if (w == 31) b &= 0x7fffffff;
        if (w < 31) b %= (1 << w);
    } while (!zero_ok && b == 0);
    return b;
}

void MOA_Seed(uint32_t seed) {
    int i;
    uint32_t s = seed;
```



*src/gf\_rand.c lines 61 to 80*

```
    for (i = 0; i < 5; i++) {  
        s = s * 29943829 - 1;  
        MOA_X[i] = s;  
    }  
    for (i=0; i<19; i++) MOA_Random_32();  
}
```

```
void MOA_Fill_Random_Region (void *reg, int size)  
{  
    uint32_t *r32;  
    uint8_t *r8;  
    int i;  
  
    r32 = (uint32_t *) reg;  
    r8 = (uint8_t *) reg;  
    for (i = 0; i < size/4; i++) r32[i] = MOA_Random_32();  
    for (i *= 4; i < size; i++) r8[i] = MOA_Random_W(8, 1);  
}
```



*src/gf\_wl28.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_wl28.c
 *
 * Routines for 128-bit Galois fields
 */

#include "gf_int.h"
#include <stdio.h>
#include <stdlib.h>

#define GF_FIELD_WIDTH (128)

#define two_x(a) {\
    a[0] <=& 1; \
    if (a[1] & 1ULL << 63) a[0] ^= 1; \
    a[1] <=& 1; }

#define a_get_b(a, i, b, j) {\
    a[i] = b[j]; \
    a[i + 1] = b[j + 1];}

#define set_zero(a, i) {\
    a[i] = 0; \
    a[i + 1] = 0;}

struct gf_wl28_split_4_128_data {
    uint64_t last_value[2];
    uint64_t tables[2][32][16];
};

struct gf_wl28_split_8_128_data {
    uint64_t last_value[2];
    uint64_t tables[2][16][256];
};

typedef struct gf_group_tables_s {
    gf_val_128_t m_table;
    gf_val_128_t r_table;
} gf_group_tables_t;

#define MM_PRINT8(s, r) { uint8_t blah[16], ii; printf("%-12s", s); \
    _mm_storeu_si128((__m128i *)blah, r); \
    for (ii = 0; ii < 16; ii += 1) \
        printf("%s%02x", (ii%4==0) ? "    " : " ", blah[15-ii]); printf("\n"); }

static
void
gf_wl28_multiply_region_from_single(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes,
int xor)
{
    int i;
    gf_val_128_t s128;
    gf_val_128_t d128;
    uint64_t c128[2];
    gf_region_data rd;
```



*src/gf\_w128.c lines 61 to 120*

```
/* We only do this to check on alignment. */
gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 8);

if (val[0] == 0) {
    if (val[1] == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val[1] == 1) { gf_multby_one(src, dest, bytes, xor); return; }
}

set_zero(c128, 0);

s128 = (gf_val_128_t) src;
d128 = (gf_val_128_t) dest;

if (xor) {
    for (i = 0; i < bytes/sizeof(gf_val_64_t); i += 2) {
        gf->multiply.w128(gf, &s128[i], val, c128);
        d128[i] ^= c128[0];
        d128[i+1] ^= c128[1];
    }
} else {
    for (i = 0; i < bytes/sizeof(gf_val_64_t); i += 2) {
        gf->multiply.w128(gf, &s128[i], val, &d128[i]);
    }
}
}

#if defined(INTEL_SSE4_PCLMUL)
static
void
gf_w128_clm_multiply_region_from_single(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes,
int xor)
{
    int i;
    gf_val_128_t s128;
    gf_val_128_t d128;
    gf_region_data rd;
    __m128i a,b;
    __m128i result0,result1;
    __m128i prim_poly;
    __m128i c,d,e,f;
    gf_internal_t *h = gf->scratch;
    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)h->prim_poly);
    /* We only do this to check on alignment. */
    gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 8);

    if (val[0] == 0) {
        if (val[1] == 0) { gf_multby_zero(dest, bytes, xor); return; }
        if (val[1] == 1) { gf_multby_one(src, dest, bytes, xor); return; }
    }

    s128 = (gf_val_128_t) src;
    d128 = (gf_val_128_t) dest;

    if (xor) {
        for (i = 0; i < bytes/sizeof(gf_val_64_t); i += 2) {
            a = _mm_insert_epi64 (_mm_setzero_si128(), s128[i+1], 0);
            b = _mm_insert_epi64 (a, val[1], 0);
            a = _mm_insert_epi64 (a, s128[i], 1);
            b = _mm_insert_epi64 (b, val[0], 1);
        }
    }
}
```



*src/gf\_wl28.c lines 121 to 180*

```
c = _mm_clmulepi64_si128 (a, b, 0x00); /*low-low*/
f = _mm_clmulepi64_si128 (a, b, 0x01); /*high-low*/
e = _mm_clmulepi64_si128 (a, b, 0x10); /*low-high*/
d = _mm_clmulepi64_si128 (a, b, 0x11); /*high-high*/

/* now reusing a and b as temporary variables*/
result0 = _mm_setzero_si128();
result1 = result0;

result0 = _mm_xor_si128 (result0, _mm_insert_epi64 (d, 0, 0));
a = _mm_xor_si128 (_mm_srli_si128 (e, 8), _mm_insert_epi64 (d, 0, 1));
result0 = _mm_xor_si128 (result0, _mm_xor_si128 (_mm_srli_si128 (f, 8), a));

a = _mm_xor_si128 (_mm_slli_si128 (e, 8), _mm_insert_epi64 (c, 0, 0));
result1 = _mm_xor_si128 (result1, _mm_xor_si128 (_mm_slli_si128 (f, 8), a));
result1 = _mm_xor_si128 (result1, _mm_insert_epi64 (c, 0, 1));
/* now we have constructed our 'result' with result0 being the carry bits, and we have to reduce. */

a = _mm_srli_si128 (result0, 8);
b = _mm_clmulepi64_si128 (a, prim_poly, 0x00);
result0 = _mm_xor_si128 (result0, _mm_srli_si128 (b, 8));
result1 = _mm_xor_si128 (result1, _mm_slli_si128 (b, 8));

a = _mm_insert_epi64 (result0, 0, 1);
b = _mm_clmulepi64_si128 (a, prim_poly, 0x00);
result1 = _mm_xor_si128 (result1, b);
d128[i] ^= (uint64_t) _mm_extract_epi64 (result1, 1);
d128[i+1] ^= (uint64_t) _mm_extract_epi64 (result1, 0);
}
} else {
for (i = 0; i < bytes/sizeof(gf_val_64_t); i += 2) {
a = _mm_insert_epi64 (_mm_setzero_si128(), s128[i+1], 0);
b = _mm_insert_epi64 (a, val[1], 0);
a = _mm_insert_epi64 (a, s128[i], 1);
b = _mm_insert_epi64 (b, val[0], 1);

c = _mm_clmulepi64_si128 (a, b, 0x00); /*low-low*/
f = _mm_clmulepi64_si128 (a, b, 0x01); /*high-low*/
e = _mm_clmulepi64_si128 (a, b, 0x10); /*low-high*/
d = _mm_clmulepi64_si128 (a, b, 0x11); /*high-high*/

/* now reusing a and b as temporary variables*/
result0 = _mm_setzero_si128();
result1 = result0;

result0 = _mm_xor_si128 (result0, _mm_insert_epi64 (d, 0, 0));
a = _mm_xor_si128 (_mm_srli_si128 (e, 8), _mm_insert_epi64 (d, 0, 1));
result0 = _mm_xor_si128 (result0, _mm_xor_si128 (_mm_srli_si128 (f, 8), a));

a = _mm_xor_si128 (_mm_slli_si128 (e, 8), _mm_insert_epi64 (c, 0, 0));
result1 = _mm_xor_si128 (result1, _mm_xor_si128 (_mm_slli_si128 (f, 8), a));
result1 = _mm_xor_si128 (result1, _mm_insert_epi64 (c, 0, 1));
/* now we have constructed our 'result' with result0 being the carry bits, and we have to reduce.*/

a = _mm_srli_si128 (result0, 8);
b = _mm_clmulepi64_si128 (a, prim_poly, 0x00);
result0 = _mm_xor_si128 (result0, _mm_srli_si128 (b, 8));
result1 = _mm_xor_si128 (result1, _mm_slli_si128 (b, 8));

a = _mm_insert_epi64 (result0, 0, 1);
```



*src/gf\_w128.c lines 181 to 240*

```
        b = _mm_clmulepi64_si128 (a, prim_poly, 0x00);
        result1 = _mm_xor_si128 (result1, b);
        d128[i] = (uint64_t) _mm_extract_epi64(result1,1);
        d128[i+1] = (uint64_t) _mm_extract_epi64(result1,0);
    }
}
#endif

/*
 * Some w128 notes:
 * --Big Endian
 * --return values allocated beforehand
 */

#define GF_W128_IS_ZERO(val) (val[0] == 0 && val[1] == 0)

void
gf_w128_shift_multiply(gf_t *gf, gf_val_128_t a128, gf_val_128_t b128, gf_val_128_t c128)
{
    /* ordered highest bit to lowest l[0] l[1] r[0] r[1] */
    uint64_t pl[2], pr[2], ppl[2], ppr[2], i, a[2], bl[2], br[2], one, lbit;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    if (GF_W128_IS_ZERO(a128) || GF_W128_IS_ZERO(b128)) {
        set_zero(c128, 0);
        return;
    }

    a_get_b(a, 0, a128, 0);
    a_get_b(br, 0, b128, 0);
    set_zero(bl, 0);

    one = 1;
    lbit = (one << 63);

    set_zero(pl, 0);
    set_zero(pr, 0);

    /* Allen: a*b for right half of a */
    for (i = 0; i < GF_FIELD_WIDTH/2; i++) {
        if (a[1] & (one << i)) {
            pl[1] ^= bl[1];
            pr[0] ^= br[0];
            pr[1] ^= br[1];
        }
        bl[1] <=< 1;
        if (br[0] & lbit) bl[1] ^= 1;
        br[0] <=< 1;
        if (br[1] & lbit) br[0] ^= 1;
        br[1] <=< 1;
    }

    /* Allen: a*b for left half of a */
    for (i = 0; i < GF_FIELD_WIDTH/2; i++) {
        if (a[0] & (one << i)) {
            pl[0] ^= bl[0];
            pl[1] ^= bl[1];
        }
    }
}
```



*src/gf\_w128.c lines 241 to 300*

```
    pr[0] ^= br[0];
}
bl[0] <<= 1;
if (bl[1] & lbit) bl[0] ^= 1;
bl[1] <<= 1;
if (br[0] & lbit) bl[1] ^= 1;
br[0] <<= 1;
}

/* Allen: do first half of reduction (based on left quarter of initial product) */
one = lbit >> 1;
ppl[0] = one; /* Allen: introduce leading one of primitive polynomial */
ppl[1] = h->prim_poly >> 2;
ppr[0] = h->prim_poly << (GF_FIELD_WIDTH/2-2);
ppr[1] = 0;
while (one != 0) {
    if (pl[0] & one) {
        pl[0] ^= ppl[0];
        pl[1] ^= ppl[1];
        pr[0] ^= ppr[0];
        pr[1] ^= ppr[1];
    }
    one >>= 1;
    ppr[1] >>= 1;
    if (ppr[0] & 1) ppr[1] ^= lbit;
    ppr[0] >>= 1;
    if (ppl[1] & 1) ppr[0] ^= lbit;
    ppl[1] >>= 1;
    if (ppl[0] & 1) ppl[1] ^= lbit;
    ppl[0] >>= 1;
}

/* Allen: final half of reduction */
one = lbit;
while (one != 0) {
    if (pl[1] & one) {
        pl[1] ^= ppl[1];
        pr[0] ^= ppr[0];
        pr[1] ^= ppr[1];
    }
    one >>= 1;
    ppr[1] >>= 1;
    if (ppr[0] & 1) ppr[1] ^= lbit;
    ppr[0] >>= 1;
    if (ppl[1] & 1) ppr[0] ^= lbit;
    ppl[1] >>= 1;
}

/* Allen: if we really want to optimize this we can just be using c128 instead of pr all along */
c128[0] = pr[0];
c128[1] = pr[1];

return;
}
```

```
void
gf_w128_clm_multiply(gf_t *gf, gf_val_128_t a128, gf_val_128_t b128, gf_val_128_t c128)
{
#ifdef INTEL_SSE4_PCLMUL
```



*src/gf\_w128.c lines 301 to 360*

```
__m128i      a,b;
__m128i      result0,result1;
__m128i      prim_poly;
__m128i      c,d,e,f;
gf_internal_t *h = gf->scratch;

a = _mm_insert_epi64 (_mm_setzero_si128(), a128[1], 0);
b = _mm_insert_epi64 (a, b128[1], 0);
a = _mm_insert_epi64 (a, a128[0], 1);
b = _mm_insert_epi64 (b, b128[0], 1);

prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)h->prim_poly);

/* we need to test algorithm 2 later*/
c = _mm_clmulepi64_si128 (a, b, 0x00); /*low-low*/
f = _mm_clmulepi64_si128 (a, b, 0x01); /*high-low*/
e = _mm_clmulepi64_si128 (a, b, 0x10); /*low-high*/
d = _mm_clmulepi64_si128 (a, b, 0x11); /*high-high*/

/* now reusing a and b as temporary variables*/
result0 = _mm_setzero_si128();
result1 = result0;

result0 = _mm_xor_si128 (result0, _mm_insert_epi64 (d, 0, 0));
a = _mm_xor_si128 (_mm_srli_si128 (e, 8), _mm_insert_epi64 (d, 0, 1));
result0 = _mm_xor_si128 (result0, _mm_xor_si128 (_mm_srli_si128 (f, 8), a));

a = _mm_xor_si128 (_mm_slli_si128 (e, 8), _mm_insert_epi64 (c, 0, 0));
result1 = _mm_xor_si128 (result1, _mm_xor_si128 (_mm_slli_si128 (f, 8), a));
result1 = _mm_xor_si128 (result1, _mm_insert_epi64 (c, 0, 1));
/* now we have constructed our 'result' with result0 being the carry bits, and we have to reduce.*/

a = _mm_srli_si128 (result0, 8);
b = _mm_clmulepi64_si128 (a, prim_poly, 0x00);
result0 = _mm_xor_si128 (result0, _mm_srli_si128 (b, 8));
result1 = _mm_xor_si128 (result1, _mm_slli_si128 (b, 8));

a = _mm_insert_epi64 (result0, 0, 1);
b = _mm_clmulepi64_si128 (a, prim_poly, 0x00);
result1 = _mm_xor_si128 (result1, b);

c128[0] = (uint64_t)_mm_extract_epi64(result1,1);
c128[1] = (uint64_t)_mm_extract_epi64(result1,0);
#endif
return;
}

void
gf_w128_bytwo_p_multiply(gf_t *gf, gf_val_128_t a128, gf_val_128_t b128, gf_val_128_t c128)
{
    uint64_t amask[2], pmask, pp, prod[2]; /*John: pmask is always the highest bit set,
                                           and the rest zeros. amask changes, it's a countdown.*/
    uint64_t topbit; /* this is used as a boolean value */
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;
    prod[0] = 0;
    prod[1] = 0;
    pmask = 0x8000000000000000ULL;
```



*src/gf\_w128.c lines 361 to 420*

```
amask[0] = 0x8000000000000000ULL;
amask[1] = 0;

while (amask[1] != 0 || amask[0] != 0) {
    topbit = (prod[0] & pmask);
    prod[0] <<= 1;
    if (prod[1] & pmask) prod[0] ^= 1;
    prod[1] <<= 1;
    if (topbit) prod[1] ^= pp;
    if ((a128[0] & amask[0]) || (a128[1] & amask[1])) {
        prod[0] ^= b128[0];
        prod[1] ^= b128[1];
    }
    amask[1] >>= 1;
    if (amask[0] & 1) amask[1] ^= pmask;
    amask[0] >>= 1;
}
c128[0] = prod [0];
c128[1] = prod [1];
return;
}

void
gf_w128_sse_bytwo_p_multiply(gf_t *gf, gf_val_128_t a128, gf_val_128_t b128, gf_val_128_t c128)
{
#ifdef INTEL_SSE4
    int i;
    __m128i a, b, pp, prod, amask, u_middle_one;
    /*John: pmask is always the highest bit set, and the rest zeros. amask changes, it's a countdown.*/
    uint32_t topbit, middlebit, pmask; /* this is used as a boolean value */
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = _mm_set_epi32(0, 0, 0, (uint32_t)h->prim_poly);
    prod = _mm_setzero_si128();
    a = _mm_insert_epi64(prod, a128[1], 0x0);
    a = _mm_insert_epi64(a, a128[0], 0x1);
    b = _mm_insert_epi64(prod, b128[1], 0x0);
    b = _mm_insert_epi64(b, b128[0], 0x1);
    pmask = 0x80000000;
    amask = _mm_insert_epi32(prod, 0x80000000, 0x3);
    u_middle_one = _mm_insert_epi32(prod, 1, 0x2);

    for (i = 0; i < 64; i++) {
        topbit = (_mm_extract_epi32(prod, 0x3) & pmask);
        middlebit = (_mm_extract_epi32(prod, 0x1) & pmask);
        prod = _mm_slli_epi64(prod, 1); /* this instruction loses the middle bit */
        if (middlebit) {
            prod = _mm_xor_si128(prod, u_middle_one);
        }
        if (topbit) {
            prod = _mm_xor_si128(prod, pp);
        }
        if (((uint64_t)_mm_extract_epi64(_mm_and_si128(a, amask), 1))) {
            prod = _mm_xor_si128(prod, b);
        }
        amask = _mm_srli_epi64(amask, 1); /*so does this one, but we can just replace after loop*/
    }
    amask = _mm_insert_epi32(amask, 1 << 31, 0x1);
#endif
}
```



*src/gf\_w128.c lines 421 to 480*

```
for (i = 64; i < 128; i++) {
    topbit = (_mm_extract_epi32(prod, 0x3) & pmask);
    middlebit = (_mm_extract_epi32(prod, 0x1) & pmask);
    prod = _mm_slli_epi64(prod, 1);
    if (middlebit) prod = _mm_xor_si128(prod, u_middle_one);
    if (topbit) prod = _mm_xor_si128(prod, pp);
    if (((uint64_t)_mm_extract_epi64(_mm_and_si128(a, amask), 0))) {
        prod = _mm_xor_si128(prod, b);
    }
    amask = _mm_srli_epi64(amask, 1);
}
c128[0] = (uint64_t)_mm_extract_epi64(prod, 1);
c128[1] = (uint64_t)_mm_extract_epi64(prod, 0);
#endif
return;
}
```

/\* Ben: This slow function implements sse instructions for bytwo\_b because why not \*/

```
void
gf_w128_sse_bytwo_b_multiply(gf_t *gf, gf_val_128_t a128, gf_val_128_t b128, gf_val_128_t c128)
{
#ifdef INTEL_SSE4
    __m128i a, b, lmask, hmask, pp, c, middle_one;
    gf_internal_t *h;
    uint64_t topbit, middlebit;

    h = (gf_internal_t *) gf->scratch;

    c = _mm_setzero_si128();
    lmask = _mm_insert_epi64(c, 1ULL << 63, 0);
    hmask = _mm_insert_epi64(c, 1ULL << 63, 1);
    b = _mm_insert_epi64(c, a128[0], 1);
    b = _mm_insert_epi64(b, a128[1], 0);
    a = _mm_insert_epi64(c, b128[0], 1);
    a = _mm_insert_epi64(a, b128[1], 0);
    pp = _mm_insert_epi64(c, h->prim_poly, 0);
    middle_one = _mm_insert_epi64(c, 1, 0x1);

    while (1) {
        if (_mm_extract_epi32(a, 0x0) & 1) {
            c = _mm_xor_si128(c, b);
        }
        middlebit = (_mm_extract_epi32(a, 0x2) & 1);
        a = _mm_srli_epi64(a, 1);
        if (middlebit) a = _mm_xor_si128(a, lmask);
        if ((_mm_extract_epi64(a, 0x1) == 0ULL) && (_mm_extract_epi64(a, 0x0) == 0ULL)){
            c128[0] = _mm_extract_epi64(c, 0x1);
            c128[1] = _mm_extract_epi64(c, 0x0);
            return;
        }
        topbit = (_mm_extract_epi64(_mm_and_si128(b, hmask), 1));
        middlebit = (_mm_extract_epi64(_mm_and_si128(b, lmask), 0));
        b = _mm_slli_epi64(b, 1);
        if (middlebit) b = _mm_xor_si128(b, middle_one);
        if (topbit) b = _mm_xor_si128(b, pp);
    }
#endif
}
```



*src/gf\_w128.c lines 481 to 540*

```
void
gf_w128_bytwo_b_multiply(gf_t *gf, gf_val_128_t a128, gf_val_128_t b128, gf_val_128_t c128)
{
    uint64_t bmask, pp;
    gf_internal_t *h;
    uint64_t a[2], b[2], c[2];

    h = (gf_internal_t *) gf->scratch;

    bmask = (1ULL << 63);
    set_zero(c, 0);
    b[0] = a128[0];
    b[1] = a128[1];
    a[0] = b128[0];
    a[1] = b128[1];

    while (1) {
        if (a[1] & 1) {
            c[0] ^= b[0];
            c[1] ^= b[1];
        }
        a[1] >>= 1;
        if (a[0] & 1) a[1] ^= bmask;
        a[0] >>= 1;
        if (a[1] == 0 && a[0] == 0) {
            c128[0] = c[0];
            c128[1] = c[1];
            return;
        }
        pp = (b[0] & bmask);
        b[0] <<= 1;
        if (b[1] & bmask) b[0] ^= 1;
        b[1] <<= 1;
        if (pp) b[1] ^= h->prim_poly;
    }
}

static
void
gf_w128_split_4_128_multiply_region(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes, int xor)
{
    int i, j, k;
    uint64_t pp;
    gf_internal_t *h;
    uint64_t *s64, *d64, *top;
    gf_region_data rd;
    uint64_t v[2], s;
    struct gf_w128_split_4_128_data *ld;

    /* We only do this to check on alignment. */
    gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 8);

    if (val[0] == 0) {
        if (val[1] == 0) { gf_multby_zero(dest, bytes, xor); return; }
        if (val[1] == 1) { gf_multby_one(src, dest, bytes, xor); return; }
    }

    h = (gf_internal_t *) gf->scratch;
    ld = (struct gf_w128_split_4_128_data *) h->private;
```



*src/gf\_w128.c lines 541 to 600*

```
s64 = (uint64_t *) rd.s_start;
d64 = (uint64_t *) rd.d_start;
top = (uint64_t *) rd.d_top;

if (val[0] != ld->last_value[0] || val[1] != ld->last_value[1]) {
    v[0] = val[0];
    v[1] = val[1];
    for (i = 0; i < 32; i++) {
        ld->tables[0][i][0] = 0;
        ld->tables[1][i][0] = 0;
        for (j = 1; j < 16; j <= 1) {
            for (k = 0; k < j; k++) {
                ld->tables[0][i][k^j] = (v[0] ^ ld->tables[0][i][k]);
                ld->tables[1][i][k^j] = (v[1] ^ ld->tables[1][i][k]);
            }
            pp = (v[0] & (1ULL << 63));
            v[0] <= 1;
            if (v[1] & (1ULL << 63)) v[0] ^= 1;
            v[1] <= 1;
            if (pp) v[1] ^= h->prim_poly;
        }
    }
    ld->last_value[0] = val[0];
    ld->last_value[1] = val[1];
}

/*
for (i = 0; i < 32; i++) {
    for (j = 0; j < 16; j++) {
        printf("%2d %2d %016llx %016llx\n", i, j, ld->tables[0][i][j], ld->tables[1][i][j]);
    }
    printf("\n");
}
*/
while (d64 < top) {
    v[0] = (xor) ? d64[0] : 0;
    v[1] = (xor) ? d64[1] : 0;
    s = s64[1];
    i = 0;
    while (s != 0) {
        v[0] ^= ld->tables[0][i][s&0xf];
        v[1] ^= ld->tables[1][i][s&0xf];
        s >>= 4;
        i++;
    }
    s = s64[0];
    i = 16;
    while (s != 0) {
        v[0] ^= ld->tables[0][i][s&0xf];
        v[1] ^= ld->tables[1][i][s&0xf];
        s >>= 4;
        i++;
    }
    d64[0] = v[0];
    d64[1] = v[1];
    s64 += 2;
    d64 += 2;
}
}
```



*src/gf\_wl28.c lines 601 to 660*

```
#if defined(INTEL_SSSE3) && defined(INTEL_SSE4)
static
void
gf_wl28_split_4_128_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes, int xor)
{
    gf_internal_t *h;
    int i, j, k;
    uint64_t pp, v[2], s, *s64, *d64, *top;
    __m128i p, tables[32][16];
    struct gf_wl28_split_4_128_data *ld;
    gf_region_data rd;

    if (val[0] == 0) {
        if (val[1] == 0) { gf_multby_zero(dest, bytes, xor); return; }
        if (val[1] == 1) { gf_multby_one(src, dest, bytes, xor); return; }
    }

    h = (gf_internal_t *) gf->scratch;

    /* We only do this to check on alignment. */
    gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 16);

    /* Doing this instead of gf_do_initial_region_alignment() because that doesn't hold 128-bit vals */
    gf_wl28_multiply_region_from_single(gf, src, dest, val, ((uint8_t *)rd.s_start-(uint8_t *)src), xor);

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    top = (uint64_t *) rd.d_top;

    ld = (struct gf_wl28_split_4_128_data *) h->private;

    if (val[0] != ld->last_value[0] || val[1] != ld->last_value[1]) {
        v[0] = val[0];
        v[1] = val[1];
        for (i = 0; i < 32; i++) {
            ld->tables[0][i][0] = 0;
            ld->tables[1][i][0] = 0;
            for (j = 1; j < 16; j <= 1) {
                for (k = 0; k < j; k++) {
                    ld->tables[0][i][k^j] = (v[0] ^ ld->tables[0][i][k]);
                    ld->tables[1][i][k^j] = (v[1] ^ ld->tables[1][i][k]);
                }
                pp = (v[0] & (1ULL << 63));
                v[0] <= 1;
                if (v[1] & (1ULL << 63)) v[0] ^= 1;
                v[1] <= 1;
                if (pp) v[1] ^= h->prim_poly;
            }
        }
    }

    ld->last_value[0] = val[0];
    ld->last_value[1] = val[1];

    for (i = 0; i < 32; i++) {
        for (j = 0; j < 16; j++) {
            v[0] = ld->tables[0][i][j];
            v[1] = ld->tables[1][i][j];
            tables[i][j] = __mm_loadu_si128((__m128i *) v);
        }
    }
}
```



*src/gf\_w128.c lines 661 to 720*

```
/*
    printf("%2d %2d: ", i, j);
    MM_PRINT8("", tables[i][j]); */
}
}

while (d64 != top) {
    if (xor) {
        p = _mm_load_si128 ((__m128i *) d64);
    } else {
        p = _mm_setzero_si128();
    }
    s = *s64;
    s64++;
    for (i = 0; i < 16; i++) {
        j = (s&0xf);
        s >>= 4;
        p = _mm_xor_si128(p, tables[16+i][j]);
    }
    s = *s64;
    s64++;
    for (i = 0; i < 16; i++) {
        j = (s&0xf);
        s >>= 4;
        p = _mm_xor_si128(p, tables[i][j]);
    }
    _mm_store_si128((__m128i *) d64, p);
    d64 += 2;
}

/* Doing this instead of gf_do_final_region_alignment() because that doesn't hold 128-bit vals */
gf_w128_multiply_region_from_single(gf, rd.s_top, rd.d_top, val, ((uint8_t *)src+bytes)-(uint8_t *)rd.s_top, xor);
}
#endif

#if defined(INTEL_SSSE3) && defined(INTEL_SSE4)
static
void
gf_w128_split_4_128_sse_altmap_multiply_region(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes, int xor)
{
    gf_internal_t *h;
    int i, j, k;
    uint64_t pp, v[2], *s64, *d64, *top;
    __m128i si, tables[32][16], p[16], v0, mask1;
    struct gf_w128_split_4_128_data *ld;
    uint8_t btable[16];
    gf_region_data rd;

    if (val[0] == 0) {
        if (val[1] == 0) { gf_multby_zero(dest, bytes, xor); return; }
        if (val[1] == 1) { gf_multby_one(src, dest, bytes, xor); return; }
    }

    h = (gf_internal_t *) gf->scratch;

    /* We only do this to check on alignment. */
    gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 256);
```



*src/gf\_wl28.c lines 721 to 780*

```
/* Doing this instead of gf_do_initial_region_alignment() because that doesn't hold 128-bit vals */
gf_wl28_multiply_region_from_single(gf, src, dest, val, ((uint8_t *)rd.s_start-(uint8_t *)src), xor);

s64 = (uint64_t *) rd.s_start;
d64 = (uint64_t *) rd.d_start;
top = (uint64_t *) rd.d_top;

ld = (struct gf_wl28_split_4_128_data *) h->private;

if (val[0] != ld->last_value[0] || val[1] != ld->last_value[1]) {
    v[0] = val[0];
    v[1] = val[1];
    for (i = 0; i < 32; i++) {
        ld->tables[0][i][0] = 0;
        ld->tables[1][i][0] = 0;
        for (j = 1; j < 16; j <= 1) {
            for (k = 0; k < j; k++) {
                ld->tables[0][i][k^j] = (v[0] ^ ld->tables[0][i][k]);
                ld->tables[1][i][k^j] = (v[1] ^ ld->tables[1][i][k]);
            }
            pp = (v[0] & (1ULL << 63));
            v[0] <= 1;
            if (v[1] & (1ULL << 63)) v[0] ^= 1;
            v[1] <= 1;
            if (pp) v[1] ^= h->prim_poly;
        }
    }
}

ld->last_value[0] = val[0];
ld->last_value[1] = val[1];

for (i = 0; i < 32; i++) {
    for (j = 0; j < 16; j++) {
        for (k = 0; k < 16; k++) {
            btable[k] = (uint8_t) ld->tables[1-(j/8)][i][k];
            ld->tables[1-(j/8)][i][k] >= 8;
        }
        tables[i][j] = _mm_loadu_si128((__m128i *) btable);
    }
}

/*
    printf("%2d %2d: ", i, j);
    MM_PRINT8("", tables[i][j]);
*/
}
}

mask1 = _mm_set1_epi8(0xf);

while (d64 != top) {
    if (xor) {
        for (i = 0; i < 16; i++) p[i] = _mm_load_si128 ((__m128i *) (d64+i*2));
    } else {
        for (i = 0; i < 16; i++) p[i] = _mm_setzero_si128();
    }
    i = 0;
    for (k = 0; k < 16; k++) {
```



*src/gf\_wl28.c lines 781 to 840*

```
    v0 = _mm_load_si128((__m128i *) s64);
    s64 += 2;

    si = _mm_and_si128(v0, mask1);

    for (j = 0; j < 16; j++) {
        p[j] = _mm_xor_si128(p[j], _mm_shuffle_epi8(tables[i][j], si));
    }
    i++;
    v0 = _mm_srli_epi32(v0, 4);
    si = _mm_and_si128(v0, mask1);
    for (j = 0; j < 16; j++) {
        p[j] = _mm_xor_si128(p[j], _mm_shuffle_epi8(tables[i][j], si));
    }
    i++;
}
for (i = 0; i < 16; i++) {
    _mm_store_si128((__m128i *) d64, p[i]);
    d64 += 2;
}
}
/* Doing this instead of gf_do_final_region_alignment() because that doesn't hold 128-bit vals */
gf_wl28_multiply_region_from_single(gf, rd.s_top, rd.d_top, val, ((uint8_t *)src+bytes)-(uint8_t *)rd.s_top, xor);
}
#endif

static
void
gf_wl28_split_8_128_multiply_region(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes, int xor)
{
    int i, j, k;
    uint64_t pp;
    gf_internal_t *h;
    uint64_t *s64, *d64, *top;
    gf_region_data rd;
    uint64_t v[2], s;
    struct gf_wl28_split_8_128_data *ld;

    /* Check on alignment. Ignore it otherwise. */
    gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 8);

    if (val[0] == 0) {
        if (val[1] == 0) { gf_multby_zero(dest, bytes, xor); return; }
        if (val[1] == 1) { gf_multby_one(src, dest, bytes, xor); return; }
    }

    h = (gf_internal_t *) gf->scratch;
    ld = (struct gf_wl28_split_8_128_data *) h->private;

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    top = (uint64_t *) rd.d_top;

    if (val[0] != ld->last_value[0] || val[1] != ld->last_value[1]) {
        v[0] = val[0];
        v[1] = val[1];
        for (i = 0; i < 16; i++) {
            ld->tables[0][i][0] = 0;
            ld->tables[1][i][0] = 0;
        }
    }
}
```

*src/gf\_wl28.c lines 841 to 900*

```
    for (j = 1; j < (1 << 8); j <= 1) {
        for (k = 0; k < j; k++) {
            ld->tables[0][i][k^j] = (v[0] ^ ld->tables[0][i][k]);
            ld->tables[1][i][k^j] = (v[1] ^ ld->tables[1][i][k]);
        }
        pp = (v[0] & (1ULL << 63));
        v[0] <= 1;
        if (v[1] & (1ULL << 63)) v[0] ^= 1;
        v[1] <= 1;
        if (pp) v[1] ^= h->prim_poly;
    }
}
ld->last_value[0] = val[0];
ld->last_value[1] = val[1];

while (d64 < top) {
    v[0] = (xor) ? d64[0] : 0;
    v[1] = (xor) ? d64[1] : 0;
    s = s64[1];
    i = 0;
    while (s != 0) {
        v[0] ^= ld->tables[0][i][s&0xff];
        v[1] ^= ld->tables[1][i][s&0xff];
        s >>= 8;
        i++;
    }
    s = s64[0];
    i = 8;
    while (s != 0) {
        v[0] ^= ld->tables[0][i][s&0xff];
        v[1] ^= ld->tables[1][i][s&0xff];
        s >>= 8;
        i++;
    }
    d64[0] = v[0];
    d64[1] = v[1];
    s64 += 2;
    d64 += 2;
}

void
gf_wl28_bytwo_b_multiply_region(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes, int xor)
{
    uint64_t bmask, pp;
    gf_internal_t *h;
    uint64_t a[2], c[2], b[2], *s64, *d64, *top;
    gf_region_data rd;

    /* We only do this to check on alignment. */
    gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 8);

    if (val[0] == 0) {
        if (val[1] == 0) { gf_multby_zero(dest, bytes, xor); return; }
        if (val[1] == 1) { gf_multby_one(src, dest, bytes, xor); return; }
    }

    h = (gf_internal_t *) gf->scratch;
    s64 = (uint64_t *) rd.s_start;
```



*src/gf\_w128.c lines 901 to 960*

```
d64 = (uint64_t *) rd.d_start;
top = (uint64_t *) rd.d_top;
bmask = (1ULL << 63);

while (d64 < top) {
    set_zero(c, 0);
    b[0] = s64[0];
    b[1] = s64[1];
    a[0] = val[0];
    a[1] = val[1];

    while (a[0] != 0) {
        if (a[1] & 1) {
            c[0] ^= b[0];
            c[1] ^= b[1];
        }
        a[1] >>= 1;
        if (a[0] & 1) a[1] ^= bmask;
        a[0] >>= 1;
        pp = (b[0] & bmask);
        b[0] <<= 1;
        if (b[1] & bmask) b[0] ^= 1;
        b[1] <<= 1;
        if (pp) b[1] ^= h->prim_poly;
    }
    while (1) {
        if (a[1] & 1) {
            c[0] ^= b[0];
            c[1] ^= b[1];
        }
        a[1] >>= 1;
        if (a[1] == 0) break;
        pp = (b[0] & bmask);
        b[0] <<= 1;
        if (b[1] & bmask) b[0] ^= 1;
        b[1] <<= 1;
        if (pp) b[1] ^= h->prim_poly;
    }
    if (xor) {
        d64[0] ^= c[0];
        d64[1] ^= c[1];
    } else {
        d64[0] = c[0];
        d64[1] = c[1];
    }
    s64 += 2;
    d64 += 2;
}
}
```

```
static
void gf_w128_group_m_init(gf_t *gf, gf_val_128_t b128)
{
    int i, j;
    int g_m;
    uint64_t prim_poly, lbit;
    gf_internal_t *scratch;
    gf_group_tables_t *gt;
    uint64_t a128[2];
    scratch = (gf_internal_t *) gf->scratch;
```

*src/gf\_w128.c lines 961 to 1020*

```
gt = scratch->private;
g_m = scratch->arg1;
prim_poly = scratch->prim_poly;

set_zero(gt->m_table, 0);
a_get_b(gt->m_table, 2, b128, 0);
lbit = 1;
lbit <=< 63;

for (i = 2; i < (1 << g_m); i <=< 1) {
    a_get_b(a128, 0, gt->m_table, 2 * (i >> 1));
    two_x(a128);
    a_get_b(gt->m_table, 2 * i, a128, 0);
    if (gt->m_table[2 * (i >> 1)] & lbit) gt->m_table[(2 * i) + 1] ^= prim_poly;
    for (j = 0; j < i; j++) {
        gt->m_table[(2 * i) + (2 * j)] = gt->m_table[(2 * i)] ^ gt->m_table[(2 * j)];
        gt->m_table[(2 * i) + (2 * j) + 1] = gt->m_table[(2 * i) + 1] ^ gt->m_table[(2 * j) + 1];
    }
}
return;
}
```

```
void
gf_w128_group_multiply(GFP gf, gf_val_128_t a128, gf_val_128_t b128, gf_val_128_t c128)
{
```

```
    int i;
    /* index_r, index_m, total_m (if g_r > g_m) */
    int i_r, i_m, t_m;
    int mask_m, mask_r;
    int g_m, g_r;
    uint64_t p_i[2], a[2];
    gf_internal_t *scratch;
    gf_group_tables_t *gt;
```

```
    scratch = (gf_internal_t *) gf->scratch;
    gt = scratch->private;
    g_m = scratch->arg1;
    g_r = scratch->arg2;
```

```
    mask_m = (1 << g_m) - 1;
    mask_r = (1 << g_r) - 1;
```

```
    if (b128[0] != gt->m_table[2] || b128[1] != gt->m_table[3]) {
        gf_w128_group_m_init(gf, b128);
    }
```

```
    p_i[0] = 0;
    p_i[1] = 0;
    a[0] = a128[0];
    a[1] = a128[1];
```

```
    t_m = 0;
    i_r = 0;
```

```
    /* Top 64 bits */
    for (i = ((GF_FIELD_WIDTH / 2) / g_m) - 1; i >= 0; i--) {
        i_m = (a[0] >> (i * g_m)) & mask_m;
        i_r ^= (p_i[0] >> (64 - g_m)) & mask_r;
        p_i[0] <=<= g_m;
```



*src/gf\_wl28.c lines 1021 to 1080*

```
    p_i[0] ^= (p_i[1] >> (64-g_m));
    p_i[1] <<= g_m;
    p_i[0] ^= gt->m_table[2 * i_m];
    p_i[1] ^= gt->m_table[(2 * i_m) + 1];
    t_m += g_m;
    if (t_m == g_r) {
        p_i[1] ^= gt->r_table[i_r];
        t_m = 0;
        i_r = 0;
    } else {
        i_r <<= g_m;
    }
}

for (i = ((GF_FIELD_WIDTH / 2) / g_m) - 1; i >= 0; i--) {
    i_m = (a[1] >> (i * g_m)) & mask_m;
    i_r ^= (p_i[0] >> (64 - g_m)) & mask_r;
    p_i[0] <<= g_m;
    p_i[0] ^= (p_i[1] >> (64-g_m));
    p_i[1] <<= g_m;
    p_i[0] ^= gt->m_table[2 * i_m];
    p_i[1] ^= gt->m_table[(2 * i_m) + 1];
    t_m += g_m;
    if (t_m == g_r) {
        p_i[1] ^= gt->r_table[i_r];
        t_m = 0;
        i_r = 0;
    } else {
        i_r <<= g_m;
    }
}
c128[0] = p_i[0];
c128[1] = p_i[1];
}
```

```
static
void
gf_wl28_group_multiply_region(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes, int xor)
{
    int i;
    int i_r, i_m, t_m;
    int mask_m, mask_r;
    int g_m, g_r;
    uint64_t p_i[2], a[2];
    gf_internal_t *scratch;
    gf_group_tables_t *gt;
    gf_region_data rd;
    uint64_t *a128, *c128, *top;

    /* We only do this to check on alignment. */
    gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 8);

    if (val[0] == 0) {
        if (val[1] == 0) { gf_multby_zero(dest, bytes, xor); return; }
        if (val[1] == 1) { gf_multby_one(src, dest, bytes, xor); return; }
    }

    scratch = (gf_internal_t *) gf->scratch;
    gt = scratch->private;
    g_m = scratch->arg1;
```

*src/gf\_wl28.c lines 1081 to 1140*

```
g_r = scratch->arg2;

mask_m = (1 << g_m) - 1;
mask_r = (1 << g_r) - 1;

if (val[0] != gt->m_table[2] || val[1] != gt->m_table[3]) {
    gf_wl28_group_m_init(gf, val);
}

a128 = (uint64_t *) src;
c128 = (uint64_t *) dest;
top = (uint64_t *) rd.d_top;

while (c128 < top) {
    p_i[0] = 0;
    p_i[1] = 0;
    a[0] = a128[0];
    a[1] = a128[1];

    t_m = 0;
    i_r = 0;

    /* Top 64 bits */
    for (i = ((GF_FIELD_WIDTH / 2) / g_m) - 1; i >= 0; i--) {
        i_m = (a[0] >> (i * g_m)) & mask_m;
        i_r ^= (p_i[0] >> (64 - g_m)) & mask_r;
        p_i[0] <<= g_m;
        p_i[0] ^= (p_i[1] >> (64 - g_m));
        p_i[1] <<= g_m;

        p_i[0] ^= gt->m_table[2 * i_m];
        p_i[1] ^= gt->m_table[(2 * i_m) + 1];
        t_m += g_m;
        if (t_m == g_r) {
            p_i[1] ^= gt->r_table[i_r];
            t_m = 0;
            i_r = 0;
        } else {
            i_r <<= g_m;
        }
    }
    for (i = ((GF_FIELD_WIDTH / 2) / g_m) - 1; i >= 0; i--) {
        i_m = (a[1] >> (i * g_m)) & mask_m;
        i_r ^= (p_i[0] >> (64 - g_m)) & mask_r;
        p_i[0] <<= g_m;
        p_i[0] ^= (p_i[1] >> (64 - g_m));
        p_i[1] <<= g_m;
        p_i[0] ^= gt->m_table[2 * i_m];
        p_i[1] ^= gt->m_table[(2 * i_m) + 1];
        t_m += g_m;
        if (t_m == g_r) {
            p_i[1] ^= gt->r_table[i_r];
            t_m = 0;
            i_r = 0;
        } else {
            i_r <<= g_m;
        }
    }
}

if (xor) {
```



*src/gf\_w128.c lines 1141 to 1200*

```
        c128[0] ^= p_i[0];
        c128[1] ^= p_i[1];
    } else {
        c128[0] = p_i[0];
        c128[1] = p_i[1];
    }
    a128 += 2;
    c128 += 2;
}

/* a^-1 -> b */
void
gf_w128_euclid(GFP gf, gf_val_128_t a128, gf_val_128_t b128)
{
    uint64_t e_i[2], e_im1[2], e_ip1[2];
    uint64_t d_i, d_im1, d_ip1;
    uint64_t y_i[2], y_im1[2], y_ip1[2];
    uint64_t c_i[2];
    uint64_t *b;
    uint64_t one = 1;

    /* This needs to return some sort of error (in b128?) */
    if (a128[0] == 0 && a128[1] == 0) return;

    b = (uint64_t *) b128;

    e_im1[0] = 0;
    e_im1[1] = ((gf_internal_t *) (gf->scratch))->prim_poly;
    e_i[0] = a128[0];
    e_i[1] = a128[1];
    d_im1 = 128;

    //Allen: I think d_i starts at 63 here, and checks each bit of a, starting at MSB, looking for the first nonzero bit
    // so d_i should be 0 if this half of a is all 0s, otherwise it should
    // be the position from right of the first-from-left zero bit of this
    // half of a. BUT if d_i is 0 at end we won't know yet if the
    // rightmost bit of this half is 1 or not

    for (d_i = (d_im1-1) % 64; ((one << d_i) & e_i[0]) == 0 && d_i > 0; d_i--) ;

    //Allen: this is testing just the first half of the stop condition
    //         above, so if it holds we know we did not find a nonzero
    //         bit yet

    if (!(one << d_i) & e_i[0]) {
        //Allen: this is doing the same thing on the other half of a. In
        //         other words, we're still searching for a nonzero bit of
        //         a. but not bothering to test if d_i hits zero, which is
        //         fine because we've already tested for a=0.

        for (d_i = (d_im1-1) % 64; ((one << d_i) & e_i[1]) == 0; d_i--) ;
    } else {

        //Allen: if a 1 was found in more-significant half of a, make d_i
        //         the ACTUAL index of the first nonzero bit in the entire a.

        d_i += 64;
    }
}
```

*src/gf\_wl28.c lines 1201 to 1260*

```
}
y_i[0] = 0;
y_i[1] = 1;
y_im1[0] = 0;
y_im1[1] = 0;

while (!(e_i[0] == 0 && e_i[1] == 1)) {

    e_ip1[0] = e_im1[0];
    e_ip1[1] = e_im1[1];
    d_ip1 = d_im1;
    c_i[0] = 0;
    c_i[1] = 0;

    while (d_ip1 >= d_i) {
        if ((d_ip1 - d_i) >= 64) {
            c_i[0] ^= (one << ((d_ip1 - d_i) - 64));
            e_ip1[0] ^= (e_i[1] << ((d_ip1 - d_i) - 64));
        } else {
            c_i[1] ^= (one << (d_ip1 - d_i));
            e_ip1[0] ^= (e_i[0] << (d_ip1 - d_i));
            if (d_ip1 - d_i > 0) e_ip1[0] ^= (e_i[1] >> (64 - (d_ip1 - d_i)));
            e_ip1[1] ^= (e_i[1] << (d_ip1 - d_i));
        }
        d_ip1--;
        if (e_ip1[0] == 0 && e_ip1[1] == 0) { b[0] = 0; b[1] = 0; return; }
        while (d_ip1 >= 64 && (e_ip1[0] & (one << (d_ip1 - 64))) == 0) d_ip1--;
        while (d_ip1 < 64 && (e_ip1[1] & (one << d_ip1)) == 0) d_ip1--;
    }
    gf->multiply.wl28(gf, c_i, y_i, y_ip1);
    y_ip1[0] ^= y_im1[0];
    y_ip1[1] ^= y_im1[1];

    y_im1[0] = y_i[0];
    y_im1[1] = y_i[1];

    y_i[0] = y_ip1[0];
    y_i[1] = y_ip1[1];

    e_im1[0] = e_i[0];
    e_im1[1] = e_i[1];
    d_im1 = d_i;
    e_i[0] = e_ip1[0];
    e_i[1] = e_ip1[1];
    d_i = d_ip1;
}

b[0] = y_i[0];
b[1] = y_i[1];
return;
}

void
gf_wl28_divide_from_inverse(GFP gf, gf_val_128_t a128, gf_val_128_t b128, gf_val_128_t c128)
{
    uint64_t d[2];
    gf->inverse.wl28(gf, b128, d);
    gf->multiply.wl28(gf, a128, d, c128);
    return;
}
```



*src/gf\_w128.c lines 1261 to 1320*

```
void
gf_w128_inverse_from_divide(GFP gf, gf_val_128_t a128, gf_val_128_t b128)
{
    uint64_t one128[2];
    one128[0] = 0;
    one128[1] = 1;
    gf->divide.w128(gf, one128, a128, b128);
    return;
}
```

```
static
void
gf_w128_composite_inverse(gf_t *gf, gf_val_128_t a, gf_val_128_t inv)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint64_t a0 = a[1];
    uint64_t a1 = a[0];
    uint64_t c0, c1, d, tmp;
    uint64_t a0inv, a1inv;

    if (a0 == 0) {
        a1inv = base_gf->inverse.w64(base_gf, a1);
        c0 = base_gf->multiply.w64(base_gf, a1inv, h->prim_poly);
        c1 = a1inv;
    } else if (a1 == 0) {
        c0 = base_gf->inverse.w64(base_gf, a0);
        c1 = 0;
    } else {
        a1inv = base_gf->inverse.w64(base_gf, a1);
        a0inv = base_gf->inverse.w64(base_gf, a0);

        d = base_gf->multiply.w64(base_gf, a1, a0inv);

        tmp = (base_gf->multiply.w64(base_gf, a1, a0inv) ^ base_gf->multiply.w64(base_gf, a0, a1inv) ^ h->prim_poly);
        tmp = base_gf->inverse.w64(base_gf, tmp);

        d = base_gf->multiply.w64(base_gf, d, tmp);

        c0 = base_gf->multiply.w64(base_gf, (d^1), a0inv);
        c1 = base_gf->multiply.w64(base_gf, d, a1inv);
    }
    inv[0] = c1;
    inv[1] = c0;
}
```

```
static
void
gf_w128_composite_multiply(gf_t *gf, gf_val_128_t a, gf_val_128_t b, gf_val_128_t rv)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint64_t b0 = b[1];
    uint64_t b1 = b[0];
    uint64_t a0 = a[1];
    uint64_t a1 = a[0];
    uint64_t alb1;
```

*src/gf\_wl28.c lines 1321 to 1380*

```
    alb1 = base_gf->multiply.w64(base_gf, a1, b1);

    rv[1] = (base_gf->multiply.w64(base_gf, a0, b0) ^ alb1);
    rv[0] = base_gf->multiply.w64(base_gf, a1, b0) ^
        base_gf->multiply.w64(base_gf, a0, b1) ^
        base_gf->multiply.w64(base_gf, alb1, h->prim_poly);
}

static
void
gf_wl28_composite_multiply_region(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes, int xor)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint64_t b0 = val[1];
    uint64_t b1 = val[0];
    uint64_t *s64, *d64;
    uint64_t *top;
    uint64_t a0, a1, alb1;
    gf_region_data rd;

    if (val[0] == 0 && val[1] == 0) { gf_multby_zero(dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 8);

    s64 = rd.s_start;
    d64 = rd.d_start;
    top = rd.d_top;

    if (xor) {
        while (d64 < top) {
            a1 = s64[0];
            a0 = s64[1];
            alb1 = base_gf->multiply.w64(base_gf, a1, b1);

            d64[1] ^= (base_gf->multiply.w64(base_gf, a0, b0) ^ alb1);
            d64[0] ^= (base_gf->multiply.w64(base_gf, a1, b0) ^
                base_gf->multiply.w64(base_gf, a0, b1) ^
                base_gf->multiply.w64(base_gf, alb1, h->prim_poly));
            s64 += 2;
            d64 += 2;
        }
    } else {
        while (d64 < top) {
            a1 = s64[0];
            a0 = s64[1];
            alb1 = base_gf->multiply.w64(base_gf, a1, b1);

            d64[1] = (base_gf->multiply.w64(base_gf, a0, b0) ^ alb1);
            d64[0] = (base_gf->multiply.w64(base_gf, a1, b0) ^
                base_gf->multiply.w64(base_gf, a0, b1) ^
                base_gf->multiply.w64(base_gf, alb1, h->prim_poly));
            s64 += 2;
            d64 += 2;
        }
    }
}

static
void
```



*src/gf\_wl28.c lines 1381 to 1440*

```
gf_wl28_composite_multiply_region_alt(gf_t *gf, void *src, void *dest, gf_val_128_t val, int bytes, int
    xor)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;  gf_t *base_gf = h->base_gf;
    gf_val_64_t val0 = val[1];
    gf_val_64_t val1 = val[0];
    uint8_t *slow, *shigh;
    uint8_t *dlow, *dhigh, *top;
    int sub_reg_size;
    gf_region_data rd;

    gf_set_region_data(&rd, gf, src, dest, bytes, 0, xor, 64);
    gf_wl28_multiply_region_from_single(gf, src, dest, val, ((uint8_t *)rd.s_start-(uint8_t *)src), xor);

    slow = (uint8_t *) rd.s_start;
    dlow = (uint8_t *) rd.d_start;
    top = (uint8_t*) rd.d_top;
    sub_reg_size = (top - dlow)/2;
    shigh = slow + sub_reg_size;
    dhigh = dlow + sub_reg_size;

    base_gf->multiply_region.w64(base_gf, slow, dlow, val0, sub_reg_size, xor);
    base_gf->multiply_region.w64(base_gf, shigh, dlow, val1, sub_reg_size, 1);
    base_gf->multiply_region.w64(base_gf, slow, dhigh, val1, sub_reg_size, xor);
    base_gf->multiply_region.w64(base_gf, shigh, dhigh, val0, sub_reg_size, 1);
    base_gf->multiply_region.w64(base_gf, shigh, dhigh, base_gf->multiply.w64(base_gf, h->prim_poly, val1
        ), sub_reg_size, 1);

    gf_wl28_multiply_region_from_single(gf, rd.s_top, rd.d_top, val, ((uint8_t *)src+bytes)-(uint8_t *)rd.s_top, xor);
}

static
int gf_wl28_composite_init(gf_t *gf)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;

    if (h->region_type & GF_REGION_ALTMAP) {
        gf->multiply_region.wl28 = gf_wl28_composite_multiply_region_alt;
    } else {
        gf->multiply_region.wl28 = gf_wl28_composite_multiply_region;
    }

    gf->multiply.wl28 = gf_wl28_composite_multiply;
    gf->divide.wl28 = gf_wl28_divide_from_inverse;
    gf->inverse.wl28 = gf_wl28_composite_inverse;

    return 1;
}

static
int gf_wl28_cfm_init(gf_t *gf)
{
#ifdef INTEL_SSE4_PCLMUL
    gf->inverse.wl28 = gf_wl28_euclid;
    gf->multiply.wl28 = gf_wl28_clm_multiply;
    gf->multiply_region.wl28 = gf_wl28_clm_multiply_region_from_single;
    return 1;
#endif
}
```

*src/gf\_w128.c lines 1441 to 1500*

```
    return 0;
}

static
int gf_w128_shift_init(gf_t *gf)
{
    gf->multiply.w128 = gf_w128_shift_multiply;
    gf->inverse.w128 = gf_w128_euclid;
    gf->multiply_region.w128 = gf_w128_multiply_region_from_single;
    return 1;
}

static
int gf_w128_bytwo_init(gf_t *gf)
{
    gf_internal_t *h;
    h = (gf_internal_t *) gf->scratch;

    if (h->mult_type == GF_MULT_BYTWO_p) {
        gf->multiply.w128 = gf_w128_bytwo_p_multiply;
        /*gf->multiply.w128 = gf_w128_sse_bytwo_p_multiply;*/
        /* John: the sse function is slower.*/
    } else {
        gf->multiply.w128 = gf_w128_bytwo_b_multiply;
        /*gf->multiply.w128 = gf_w128_sse_bytwo_b_multiply;
Ben: This sse function is also slower. */
    }
    gf->inverse.w128 = gf_w128_euclid;
    gf->multiply_region.w128 = gf_w128_bytwo_b_multiply_region;
    return 1;
}

/*
 * Because the prim poly is only 8 bits and we are limiting g_r to 16, I do not need the high 64
 * bits in all of these numbers.
 */
static
void gf_w128_group_r_init(gf_t *gf)
{
    int i, j;
    int g_r;
    uint64_t pp;
    gf_internal_t *scratch;
    gf_group_tables_t *gt;
    scratch = (gf_internal_t *) gf->scratch;
    gt = scratch->private;
    g_r = scratch->arg2;
    pp = scratch->prim_poly;

    gt->r_table[0] = 0;
    for (i = 1; i < (1 << g_r); i++) {
        gt->r_table[i] = 0;
        for (j = 0; j < g_r; j++) {
            if (i & (1 << j)) {
                gt->r_table[i] ^= (pp << j);
            }
        }
    }
    return;
}
```



*src/gf\_wl28.c lines 1501 to 1560*

```
#if 0 // defined(INTEL_SSE4)
    static
void gf_wl28_group_r_sse_init(gf_t *gf)
{
    int i, j;
    int g_r;
    uint64_t pp;
    gf_internal_t *scratch;
    gf_group_tables_t *gt;
    scratch = (gf_internal_t *) gf->scratch;
    gt = scratch->private;
    __m128i zero = _mm_setzero_si128();
    __m128i *table = (__m128i *) (gt->r_table);
    g_r = scratch->arg2;
    pp = scratch->prim_poly;
    table[0] = zero;
    for (i = 1; i < (1 << g_r); i++) {
        table[i] = zero;
        for (j = 0; j < g_r; j++) {
            if (i & (1 << j)) {
                table[i] = _mm_xor_si128(table[i], _mm_insert_epi64(zero, pp << j, 0));
            }
        }
    }
    return;
}
#endif

    static
int gf_wl28_split_init(gf_t *gf)
{
    struct gf_wl28_split_4_128_data *sd4;
    struct gf_wl28_split_8_128_data *sd8;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    gf->multiply.wl28 = gf_wl28_bytwo_p_multiply;
#if defined(INTEL_SSE4_PCLMUL)
    if (!(h->region_type & GF_REGION_NOSSE)) {
        gf->multiply.wl28 = gf_wl28_clm_multiply;
    }
#endif

    gf->inverse.wl28 = gf_wl28_euclid;

    if ((h->arg1 != 4 && h->arg2 != 4) || h->mult_type == GF_MULT_DEFAULT) {
        sd8 = (struct gf_wl28_split_8_128_data *) h->private;
        sd8->last_value[0] = 0;
        sd8->last_value[1] = 0;
        gf->multiply_region.wl28 = gf_wl28_split_8_128_multiply_region;
    } else {
        sd4 = (struct gf_wl28_split_4_128_data *) h->private;
        sd4->last_value[0] = 0;
        sd4->last_value[1] = 0;
        if ((h->region_type & GF_REGION_ALTMAP))
        {
            #ifdef INTEL_SSE4
                if (!(h->region_type & GF_REGION_NOSSE))
            
```

*src/gf\_wl28.c lines 1561 to 1620*

```
        gf->multiply_region.wl28 = gf_wl28_split_4_128_sse_altmap_multiply_region;
    else
        return 0;
    #else
        return 0;
    #endif
}
else {
    #ifdef INTEL_SSE4
        if(!(h->region_type & GF_REGION_NOSSE))
            gf->multiply_region.wl28 = gf_wl28_split_4_128_sse_multiply_region;
        else
            gf->multiply_region.wl28 = gf_wl28_split_4_128_multiply_region;
        #else
            gf->multiply_region.wl28 = gf_wl28_split_4_128_multiply_region;
        #endif
    }
}
return 1;
}
```

```
static
int gf_wl28_group_init(gf_t *gf)
{
    gf_internal_t *scratch;
    gf_group_tables_t *gt;
    int g_r, size_r;

    scratch = (gf_internal_t *) gf->scratch;
    gt = scratch->private;
    g_r = scratch->arg2;
    size_r = (1 << g_r);

    gt->r_table = (gf_val_128_t)((uint8_t *)scratch->private + (2 * sizeof(uint64_t *)));
    gt->m_table = gt->r_table + size_r;
    gt->m_table[2] = 0;
    gt->m_table[3] = 0;

    gf->multiply.wl28 = gf_wl28_group_multiply;
    gf->inverse.wl28 = gf_wl28_euclid;
    gf->multiply_region.wl28 = gf_wl28_group_multiply_region;

    gf_wl28_group_r_init(gf);

    return 1;
}
```

```
void gf_wl28_extract_word(gf_t *gf, void *start, int bytes, int index, gf_val_128_t rv)
{
    gf_val_128_t s;

    s = (gf_val_128_t) start;
    s += (index * 2);
    memcpy(rv, s, 16);
}
```

```
static void gf_wl28_split_extract_word(gf_t *gf, void *start, int bytes, int index, gf_val_128_t rv)
{
    int i, blocks;
```



*src/gf\_wl28.c lines 1621 to 1680*

```
uint64_t *r64, tmp;
uint8_t *r8;
gf_region_data rd;

gf_set_region_data(&rd, gf, start, start, bytes, 0, 0, 256);
r64 = (uint64_t *) start;
if ((r64 + index*2 < (uint64_t *) rd.d_start) ||
    (r64 + index*2 >= (uint64_t *) rd.d_top)) {
    memcpy(rv, r64+(index*2), 16);
    return;
}

index -= (((uint64_t *) rd.d_start) - r64)/2;
r64 = (uint64_t *) rd.d_start;

blocks = index/16;
r64 += (blocks*32);
index %= 16;
r8 = (uint8_t *) r64;
r8 += index;
rv[0] = 0;
rv[1] = 0;

for (i = 0; i < 8; i++) {
    tmp = *r8;
    rv[1] |= (tmp << (i*8));
    r8 += 16;
}

for (i = 0; i < 8; i++) {
    tmp = *r8;
    rv[0] |= (tmp << (i*8));
    r8 += 16;
}
return;
}

static
void gf_wl28_composite_extract_word(gf_t *gf, void *start, int bytes, int index, gf_val_128_t rv)
{
    int sub_size;
    gf_internal_t *h;
    uint8_t *r8, *top;
    uint64_t *r64;
    gf_region_data rd;

    h = (gf_internal_t *) gf->scratch;
    gf_set_region_data(&rd, gf, start, start, bytes, 0, 0, 64);
    r64 = (uint64_t *) start;
    if ((r64 + index*2 < (uint64_t *) rd.d_start) ||
        (r64 + index*2 >= (uint64_t *) rd.d_top)) {
        memcpy(rv, r64+(index*2), 16);
        return;
    }
    index -= (((uint64_t *) rd.d_start) - r64)/2;
    r8 = (uint8_t *) rd.d_start;
    top = (uint8_t *) rd.d_top;
    sub_size = (top-r8)/2;

    rv[1] = h->base_gf->extract_word.w64(h->base_gf, r8, sub_size, index);
```

*src/gf\_wl28.c lines 1681 to 1740*

```
    rv[0] = h->base_gf->extract_word.w64(h->base_gf, r8+sub_size, sub_size, index);

    return;
}

int gf_wl28_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2)
{
    int size_m, size_r;
    if (divide_type==GF_DIVIDE_MATRIX) return 0;

    switch(mult_type)
    {
        case GF_MULT_CARRY_FREE:
            return sizeof(gf_internal_t);
            break;
        case GF_MULT_SHIFT:
            return sizeof(gf_internal_t);
            break;
        case GF_MULT_BYTWO_p:
        case GF_MULT_BYTWO_b:
            return sizeof(gf_internal_t);
            break;
        case GF_MULT_DEFAULT:
        case GF_MULT_SPLIT_TABLE:
            if ((arg1 == 4 && arg2 == 128) || (arg1 == 128 && arg2 == 4)) {
                return sizeof(gf_internal_t) + sizeof(struct gf_wl28_split_4_128_data) + 64;
            } else if ((arg1 == 8 && arg2 == 128) || (arg1 == 128 && arg2 == 8) || mult_type == GF_MULT_DEFAULT) {
                return sizeof(gf_internal_t) + sizeof(struct gf_wl28_split_8_128_data) + 64;
            }
            return 0;
            break;
        case GF_MULT_GROUP:
            /* JSP We've already error checked the arguments. */
            size_m = (1 << arg1) * 2 * sizeof(uint64_t);
            size_r = (1 << arg2) * 2 * sizeof(uint64_t);
            /*
             * two pointers prepend the table data for structure
             * because the tables are of dynamic size
             */
            return sizeof(gf_internal_t) + size_m + size_r + 4 * sizeof(uint64_t *);
            break;
        case GF_MULT_COMPOSITE:
            if (arg1 == 2) {
                return sizeof(gf_internal_t) + 4;
            } else {
                return 0;
            }
            break;

        default:
            return 0;
    }
}

int gf_wl28_init(gf_t *gf)
{
    gf_internal_t *h;
    int no_default_flag = 0;

    h = (gf_internal_t *) gf->scratch;
```



*src/gf\_w128.c lines 1741 to 1794*

```
/* Allen: set default primitive polynomial / irreducible polynomial if needed */

if (h->prim_poly == 0) {
    if (h->mult_type == GF_MULT_COMPOSITE) {
        h->prim_poly = gf_composite_get_default_poly(h->base_gf);
        if (h->prim_poly == 0) return 0; /* This shouldn't happen */
    } else {
        h->prim_poly = 0x87; /* Omitting the leftmost 1 as in w=32 */
    }
    if (no_default_flag == 1) {
        fprintf(stderr, "Code contains no default irreducible polynomial for given base field\n");
        return 0;
    }
}

gf->multiply.w128 = NULL;
gf->divide.w128 = NULL;
gf->inverse.w128 = NULL;
gf->multiply_region.w128 = NULL;
switch(h->mult_type) {
    case GF_MULT_BYTWO_p:
    case GF_MULT_BYTWO_b:      if (gf_w128_bytwo_init(gf) == 0) return 0; break;
    case GF_MULT_CARRY_FREE:   if (gf_w128_cfm_init(gf) == 0) return 0; break;
    case GF_MULT_SHIFT:        if (gf_w128_shift_init(gf) == 0) return 0; break;
    case GF_MULT_GROUP:        if (gf_w128_group_init(gf) == 0) return 0; break;
    case GF_MULT_DEFAULT:
    case GF_MULT_SPLIT_TABLE:   if (gf_w128_split_init(gf) == 0) return 0; break;
    case GF_MULT_COMPOSITE:     if (gf_w128_composite_init(gf) == 0) return 0; break;
    default: return 0;
}

/* Ben: Used to be h->region_type == GF_REGION_ALTMAP, but failed since there
   are multiple flags in h->region_type */
if (h->mult_type == GF_MULT_SPLIT_TABLE && (h->region_type & GF_REGION_ALTMAP)) {
    gf->extract_word.w128 = gf_w128_split_extract_word;
} else if (h->mult_type == GF_MULT_COMPOSITE && h->region_type == GF_REGION_ALTMAP) {
    gf->extract_word.w128 = gf_w128_composite_extract_word;
} else {
    gf->extract_word.w128 = gf_w128_extract_word;
}

if (h->divide_type == GF_DIVIDE_EUCLID) {
    gf->divide.w128 = gf_w128_divide_from_inverse;
}

if (gf->inverse.w128 != NULL && gf->divide.w128 == NULL) {
    gf->divide.w128 = gf_w128_divide_from_inverse;
}
if (gf->inverse.w128 == NULL && gf->divide.w128 != NULL) {
    gf->inverse.w128 = gf_w128_inverse_from_divide;
}
return 1;
}
```

*src/gf\_w16.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_w16.c
 *
 * Routines for 16-bit Galois fields
 */

#include "gf_int.h"
#include <stdio.h>
#include <stdlib.h>

#define GF_FIELD_WIDTH (16)
#define GF_FIELD_SIZE (1 << GF_FIELD_WIDTH)
#define GF_MULT_GROUP_SIZE GF_FIELD_SIZE-1

#define GF_BASE_FIELD_WIDTH (8)
#define GF_BASE_FIELD_SIZE (1 << GF_BASE_FIELD_WIDTH)

struct gf_w16_logtable_data {
    uint16_t log_tbl[GF_FIELD_SIZE];
    uint16_t antilog_tbl[GF_FIELD_SIZE * 2];
    uint16_t inv_tbl[GF_FIELD_SIZE];
    uint16_t *d_antilog;
};

struct gf_w16_zero_logtable_data {
    int log_tbl[GF_FIELD_SIZE];
    uint16_t antilog_tbl[GF_FIELD_SIZE * 4];
    uint16_t *antilog_tbl;
    uint16_t inv_tbl[GF_FIELD_SIZE];
};

struct gf_w16_lazytable_data {
    uint16_t log_tbl[GF_FIELD_SIZE];
    uint16_t antilog_tbl[GF_FIELD_SIZE * 2];
    uint16_t inv_tbl[GF_FIELD_SIZE];
    uint16_t *d_antilog;
    uint16_t lazytable[GF_FIELD_SIZE];
};

struct gf_w16_bytwo_data {
    uint64_t prim_poly;
    uint64_t mask1;
    uint64_t mask2;
};

struct gf_w16_split_8_8_data {
    uint16_t tables[3][256][256];
};

struct gf_w16_group_4_4_data {
    uint16_t reduce[16];
    uint16_t shift[16];
};

struct gf_w16_composite_data {
    uint8_t *mult_table;
```



*src/gf\_w16.c lines 61 to 120*

```
};

#define AB2(ip, am1, am2, b, t1, t2) {\
    t1 = (b << 1) & am1;\
    t2 = b & am2;\
    t2 = ((t2 << 1) - (t2 >> (GF_FIELD_WIDTH-1))); \
    b = (t1 ^ (t2 & ip));}

#define SSE_AB2(pp, m1, m2, va, t1, t2) {\
    t1 = _mm_and_si128(_mm_slli_epi64(va, 1), m1);\
    t2 = _mm_and_si128(va, m2);\
    t2 = _mm_sub_epi64(_mm_slli_epi64(t2, 1), _mm_srli_epi64(t2, (GF_FIELD_WIDTH-1))); \
    va = _mm_xor_si128(t1, _mm_and_si128(t2, pp)); }

#define MM_PRINT(s, r) { uint8_t blah[16], ii; printf("%-12s", s); \
    _mm_storeu_si128((__m128i *)blah, r); \
    for (ii = 0; ii < 16; ii += 2) \
        printf("    %02x %02x", blah[15-ii], blah[14-ii]); printf("\n"); }

#define GF_FIRST_BIT (1 << 15)
#define GF_MULTBY_TWO(p) (((p) & GF_FIRST_BIT) ? ((p) << 1) ^ h->prim_poly) : (p) << 1)

static
inline
gf_val_32_t gf_w16_inverse_from_divide (gf_t *gf, gf_val_32_t a)
{
    return gf->divide.w32(gf, 1, a);
}

static
inline
gf_val_32_t gf_w16_divide_from_inverse (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    b = gf->inverse.w32(gf, b);
    return gf->multiply.w32(gf, a, b);
}

static
void
gf_w16_multiply_region_from_single(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    gf_region_data rd;
    uint16_t *s16;
    uint16_t *d16;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 2);
    gf_do_initial_region_alignment(&rd);

    s16 = (uint16_t *) rd.s_start;
    d16 = (uint16_t *) rd.d_start;

    if (xor) {
        while (d16 < ((uint16_t *) rd.d_top)) {
            *d16 ^= gf->multiply.w32(gf, val, *s16);
            d16++;
            s16++;
        }
    }
}
```

*src/gf\_w16.c lines 121 to 180*

```
    } else {
        while (d16 < ((uint16_t *) rd.d_top)) {
            *d16 = gf->multiply.w32(gf, val, *s16);
            d16++;
            s16++;
        }
    }
    gf_do_final_region_alignment(&rd);
}

#if defined(INTEL_SSE4_PCLMUL)
static
void
gf_w16_clm_multiply_region_from_single_2(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    gf_region_data rd;
    uint16_t *s16;
    uint16_t *d16;
    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      w;
    gf_internal_t * h = gf->scratch;
    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1ffffULL));

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 2);
    gf_do_initial_region_alignment(&rd);

    a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);

    s16 = (uint16_t *) rd.s_start;
    d16 = (uint16_t *) rd.d_start;

    if (xor) {
        while (d16 < ((uint16_t *) rd.d_top)) {

            /* see gf_w16_clm_multiply() to see explanation of method */

            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s16), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
            result = _mm_xor_si128 (result, w);

            *d16 ^= ((gf_val_32_t)_mm_extract_epi32(result, 0));
            d16++;
            s16++;
        }
    } else {
        while (d16 < ((uint16_t *) rd.d_top)) {

            /* see gf_w16_clm_multiply() to see explanation of method */

            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s16), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
```



*src/gf\_wl6.c lines 181 to 240*

```
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);

    *d16 = ((gf_val_32_t) _mm_extract_epi32(result, 0));
    d16++;
    s16++;
}
}
gf_do_final_region_alignment(&rd);
}
#endif

#if defined(INTEL_SSE4_PCLMUL)
static
void
gf_wl6_clm_multiply_region_from_single_3(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    gf_region_data rd;
    uint16_t *s16;
    uint16_t *d16;

    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      w;
    gf_internal_t *h = gf->scratch;
    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1ffffULL));

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 2);
    gf_do_initial_region_alignment(&rd);

    s16 = (uint16_t *) rd.s_start;
    d16 = (uint16_t *) rd.d_start;

    if (xor) {
        while (d16 < ((uint16_t *) rd.d_top)) {

            /* see gf_wl6_clm_multiply() to see explanation of method */

            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s16), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
            result = _mm_xor_si128 (result, w);

            *d16 ^= ((gf_val_32_t) _mm_extract_epi32(result, 0));
            d16++;
            s16++;
        }
    } else {
        while (d16 < ((uint16_t *) rd.d_top)) {
```

*src/gf\_w16.c lines 241 to 300*

```
    /* see gf_w16_clm_multiply() to see explanation of method */
    b = _mm_insert_epi32 (a, (gf_val_32_t)(*s16), 0);
    result = _mm_clmulepi64_si128 (a, b, 0);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);

    *d16 = ((gf_val_32_t)_mm_extract_epi32(result, 0));
    d16++;
    s16++;
}
}
gf_do_final_region_alignment(&rd);
}
#endif

#if defined(INTEL_SSE4_PCLMUL)
static
void
gf_w16_clm_multiply_region_from_single_4(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    gf_region_data rd;
    uint16_t *s16;
    uint16_t *d16;

    __m128i a, b;
    __m128i result;
    __m128i prim_poly;
    __m128i w;
    gf_internal_t *h = gf->scratch;
    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1ffffULL));

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 2);
    gf_do_initial_region_alignment(&rd);

    a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);

    s16 = (uint16_t *) rd.s_start;
    d16 = (uint16_t *) rd.d_start;

    if (xor) {
        while (d16 < ((uint16_t *) rd.d_top)) {

            /* see gf_w16_clm_multiply() to see explanation of method */

            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s16), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
```



*src/gf\_w16.c lines 301 to 360*

```
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);

    *d16 ^= ((gf_val_32_t)_mm_extract_epi32(result, 0));
    d16++;
    s16++;
}
} else {
    while (d16 < ((uint16_t *) rd.d_top)) {

        /* see gf_w16_clm_multiply() to see explanation of method */

        b = _mm_insert_epi32 (a, (gf_val_32_t)(*s16), 0);
        result = _mm_clmulepi64_si128 (a, b, 0);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
        result = _mm_xor_si128 (result, w);

        *d16 = ((gf_val_32_t)_mm_extract_epi32(result, 0));
        d16++;
        s16++;
    }
}
gf_do_final_region_alignment(&rd);
}
#endif

static
inline
gf_val_32_t gf_w16_euclid (gf_t *gf, gf_val_32_t b)
{
    gf_val_32_t e_i, e_im1, e_ip1;
    gf_val_32_t d_i, d_im1, d_ip1;
    gf_val_32_t y_i, y_im1, y_ip1;
    gf_val_32_t c_i;

    if (b == 0) return -1;
    e_im1 = ((gf_internal_t *) (gf->scratch))->prim_poly;
    e_i = b;
    d_im1 = 16;
    for (d_i = d_im1; ((1 << d_i) & e_i) == 0; d_i--) ;
    y_i = 1;
    y_im1 = 0;

    while (e_i != 1) {

        e_ip1 = e_im1;
        d_ip1 = d_im1;
        c_i = 0;

        while (d_ip1 >= d_i) {
            c_i ^= (1 << (d_ip1 - d_i));
            e_ip1 ^= (e_i << (d_ip1 - d_i));
            if (e_ip1 == 0) return 0;
        }
    }
}
```

*src/gf\_w16.c lines 361 to 420*

```
    while ((e_ip1 & (1 << d_ip1)) == 0) d_ip1--;
}

y_ip1 = y_im1 ^ gf->multiply.w32(gf, c_i, y_i);
y_im1 = y_i;
y_i = y_ip1;

e_im1 = e_i;
d_im1 = d_i;
e_i = e_ip1;
d_i = d_ip1;
}

return y_i;
}

static
gf_val_32_t gf_w16_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    uint16_t *r16, rv;

    r16 = (uint16_t *) start;
    rv = r16[index];
    return rv;
}

static
gf_val_32_t gf_w16_composite_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    int sub_size;
    gf_internal_t *h;
    uint8_t *r8, *top;
    uint16_t a, b, *r16;
    gf_region_data rd;

    h = (gf_internal_t *) gf->scratch;
    gf_set_region_data(&rd, gf, start, start, bytes, 0, 0, 32);
    r16 = (uint16_t *) start;
    if (r16 + index < (uint16_t *) rd.d_start) return r16[index];
    if (r16 + index >= (uint16_t *) rd.d_top) return r16[index];
    index -= ((uint16_t *) rd.d_start) - r16;
    r8 = (uint8_t *) rd.d_start;
    top = (uint8_t *) rd.d_top;
    sub_size = (top-r8)/2;

    a = h->base_gf->extract_word.w32(h->base_gf, r8, sub_size, index);
    b = h->base_gf->extract_word.w32(h->base_gf, r8+sub_size, sub_size, index);
    return (a | (b << 8));
}

static
gf_val_32_t gf_w16_split_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    uint16_t *r16, rv;
    uint8_t *r8;
    gf_region_data rd;

    gf_set_region_data(&rd, gf, start, start, bytes, 0, 0, 32);
    r16 = (uint16_t *) start;
    if (r16 + index < (uint16_t *) rd.d_start) return r16[index];
```



*src/gf\_w16.c lines 421 to 480*

```
    if (r16 + index >= (uint16_t *) rd.d_top) return r16[index];
    index -= (((uint16_t *) rd.d_start) - r16);
    r8 = (uint8_t *) rd.d_start;
    r8 += ((index & 0xffffffff0)*2);
    r8 += (index & 0xf);
    rv = (*r8 << 8);
    r8 += 16;
    rv |= *r8;
    return rv;
}

static
inline
gf_val_32_t gf_w16_matrix (gf_t *gf, gf_val_32_t b)
{
    return gf_bitmatrix_inverse(b, 16, ((gf_internal_t *) (gf->scratch))->prim_poly);
}

/* JSP: GF_MULT_SHIFT: The world's dumbest multiplication algorithm. I only
   include it for completeness. It does have the feature that it requires no
   extra memory.
*/

static
inline
gf_val_32_t
gf_w16_clm_multiply_2 (gf_t *gf, gf_val_32_t a16, gf_val_32_t b16)
{
    gf_val_32_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL

    __m128i a, b;
    __m128i result;
    __m128i prim_poly;
    __m128i w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi32 (_mm_setzero_si128(), a16, 0);
    b = _mm_insert_epi32 (a, b16, 0);

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1ffffULL));

    /* Do the initial multiply */

    result = _mm_clmulepi64_si128 (a, b, 0);

    /* Ben: Do prim_poly reduction twice. We are guaranteed that we will only
       have to do the reduction at most twice, because (w-2)/z == 2. Where
       z is equal to the number of zeros after the leading 1

       _mm_clmulepi64_si128 is the carryless multiply operation. Here
       _mm_srli_si128 shifts the result to the right by 2 bytes. This allows
       us to multiply the prim_poly by the leading bits of the result. We
       then xor the result of that operation back with the result.*/

    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);
```

*src/gf\_w16.c lines 481 to 540*

```
/* Extracts 32 bit value from result. */
rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));

#endif
return rv;
}

static
inline
gf_val_32_t
gf_w16_clm_multiply_3 (gf_t *gf, gf_val_32_t a16, gf_val_32_t b16)
{
    gf_val_32_t rv = 0;

#if defined(INTEL_SSE4_PCLMUL)

    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi32 (_mm_setzero_si128(), a16, 0);
    b = _mm_insert_epi32 (a, b16, 0);

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1ffffULL));

    /* Do the initial multiply */
    result = _mm_clmulepi64_si128 (a, b, 0);

    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
    result = _mm_xor_si128 (result, w);

    /* Extracts 32 bit value from result. */
    rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));

#endif
return rv;
}

static
inline
gf_val_32_t
gf_w16_clm_multiply_4 (gf_t *gf, gf_val_32_t a16, gf_val_32_t b16)
{
    gf_val_32_t rv = 0;

#if defined(INTEL_SSE4_PCLMUL)

    __m128i      a, b;
```



*src/gf\_w16.c lines 541 to 600*

```
__m128i      result;
__m128i      prim_poly;
__m128i      w;
gf_internal_t * h = gf->scratch;

a = _mm_insert_epi32 (_mm_setzero_si128(), a16, 0);
b = _mm_insert_epi32 (a, b16, 0);

prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1ffffULL));

/* Do the initial multiply */

result = _mm_clmulepi64_si128 (a, b, 0);

w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
result = _mm_xor_si128 (result, w);
w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
result = _mm_xor_si128 (result, w);
w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
result = _mm_xor_si128 (result, w);
w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 2), 0);
result = _mm_xor_si128 (result, w);

/* Extracts 32 bit value from result. */

rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));
```

```
#endif
return rv;
}
```

```
static
inline
gf_val_32_t
gf_w16_shift_multiply (gf_t *gf, gf_val_32_t a16, gf_val_32_t b16)
{
    gf_val_32_t product, i, pp, a, b;
    gf_internal_t *h;

    a = a16;
    b = b16;
    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    product = 0;

    for (i = 0; i < GF_FIELD_WIDTH; i++) {
        if (a & (1 << i)) product ^= (b << i);
    }
    for (i = (GF_FIELD_WIDTH*2-2); i >= GF_FIELD_WIDTH; i--) {
        if (product & (1 << i)) product ^= (pp << (i-GF_FIELD_WIDTH));
    }
    return product;
}
```

```
static
int gf_w16_shift_init(gf_t *gf)
{

```

*src/gf\_w16.c lines 601 to 660*

```
gf->multiply.w32 = gf_w16_shift_multiply;
return 1;
}

static
int gf_w16_cfm_init(gf_t *gf)
{
#ifdef INTEL_SSE4_PCLMUL
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    /*Ben: Determining how many reductions to do */

    if ((0xfe00 & h->prim_poly) == 0) {
        gf->multiply.w32 = gf_w16_clm_multiply_2;
        gf->multiply_region.w32 = gf_w16_clm_multiply_region_from_single_2;
    } else if ((0xf000 & h->prim_poly) == 0) {
        gf->multiply.w32 = gf_w16_clm_multiply_3;
        gf->multiply_region.w32 = gf_w16_clm_multiply_region_from_single_3;
    } else if ((0xe000 & h->prim_poly) == 0) {
        gf->multiply.w32 = gf_w16_clm_multiply_4;
        gf->multiply_region.w32 = gf_w16_clm_multiply_region_from_single_4;
    } else {
        return 0;
    }
    return 1;
#endif
}

return 0;
}

/* KMG: GF_MULT_LOGTABLE: */

static
void
gf_w16_log_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint16_t *s16, *d16;
    int lv;
    struct gf_w16_logtable_data *ltd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 2);
    gf_do_initial_region_alignment(&rd);

    ltd = (struct gf_w16_logtable_data *) ((gf_internal_t *) gf->scratch)->private;
    s16 = (uint16_t *) rd.s_start;
    d16 = (uint16_t *) rd.d_start;

    lv = ltd->log_tbl[val];

    if (xor) {
        while (d16 < (uint16_t *) rd.d_top) {
            *d16 ^= (*s16 == 0 ? 0 : ltd->antilog_tbl[lv + ltd->log_tbl[*s16]]);
            d16++;
            s16++;
        }
    }
}
```



*src/gf\_w16.c lines 661 to 720*

```
    }
} else {
    while (d16 < (uint16_t *) rd.d_top) {
        *d16 = (*s16 == 0 ? 0 : ltd->antilog_tbl[lv + ltd->log_tbl[*s16]]);
        d16++;
        s16++;
    }
}
gf_do_final_region_alignment(&rd);
}
```

```
static
inline
gf_val_32_t
gf_w16_log_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w16_logtable_data *ltd;

    ltd = (struct gf_w16_logtable_data *) ((gf_internal_t *) gf->scratch)->private;
    return (a == 0 || b == 0) ? 0 : ltd->antilog_tbl[(int) ltd->log_tbl[a] + (int) ltd->log_tbl[b]];
}
```

```
static
inline
gf_val_32_t
gf_w16_log_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    int log_sum = 0;
    struct gf_w16_logtable_data *ltd;

    if (a == 0 || b == 0) return 0;
    ltd = (struct gf_w16_logtable_data *) ((gf_internal_t *) gf->scratch)->private;

    log_sum = (int) ltd->log_tbl[a] - (int) ltd->log_tbl[b];
    return (ltd->d_antilog[log_sum]);
}
```

```
static
gf_val_32_t
gf_w16_log_inverse(gf_t *gf, gf_val_32_t a)
{
    struct gf_w16_logtable_data *ltd;

    ltd = (struct gf_w16_logtable_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ltd->inv_tbl[a]);
}
```

```
static
int gf_w16_log_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_w16_logtable_data *ltd;
    int i, b;
    int check = 0;

    h = (gf_internal_t *) gf->scratch;
    ltd = h->private;

    for (i = 0; i < GF_MULT_GROUP_SIZE+1; i++)
        ltd->log_tbl[i] = 0;
```

*src/gf\_w16.c lines 721 to 780*

```
    ltd->d_antilog = ltd->antilog_tbl + GF_MULT_GROUP_SIZE;

    b = 1;
    for (i = 0; i < GF_MULT_GROUP_SIZE; i++) {
        if (ltd->log_tbl[b] != 0) check = 1;
        ltd->log_tbl[b] = i;
        ltd->antilog_tbl[i] = b;
        ltd->antilog_tbl[i+GF_MULT_GROUP_SIZE] = b;
        b <<= 1;
        if (b & GF_FIELD_SIZE) {
            b = b ^ h->prim_poly;
        }
    }

    /* If you can't construct the log table, there's a problem. This code is used for
       some other implementations (e.g. in SPLIT), so if the log table doesn't work in
       that instance, use CARRY_FREE / SHIFT instead. */

    if (check) {
        if (h->mult_type != GF_MULT_LOG_TABLE) {
#ifdef INTEL_SSE4_PCLMUL
            return gf_w16_cfm_init(gf);
#elseif
            return gf_w16_shift_init(gf);
        } else {
            _gf_errno = GF_E_LOGPOLY;
            return 0;
        }
    }

    ltd->inv_tbl[0] = 0; /* Not really, but we need to fill it with something */
    ltd->inv_tbl[1] = 1;
    for (i = 2; i < GF_FIELD_SIZE; i++) {
        ltd->inv_tbl[i] = ltd->antilog_tbl[GF_MULT_GROUP_SIZE-ltd->log_tbl[i]];
    }

    gf->inverse.w32 = gf_w16_log_inverse;
    gf->divide.w32 = gf_w16_log_divide;
    gf->multiply.w32 = gf_w16_log_multiply;
    gf->multiply_region.w32 = gf_w16_log_multiply_region;

    return 1;
}

/* JSP: GF_MULT_SPLIT_TABLE: Using 8 multiplication tables to leverage SSE instructions.
*/

/* Ben: Does alternate mapping multiplication using a split table in the
   lazy method without sse instructions*/

static
void
gf_w16_split_4_16_lazy_nosse_altmap_multiply_region(gf_t *gf,
    void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t i, j, c, prod;
    uint8_t *s8, *d8, *top;
    uint16_t table[4][16];
```



*src/gf\_wl6.c lines 781 to 840*

```
gf_region_data rd;

if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 32);
gf_do_initial_region_alignment(&rd);

/*Ben: Constructs lazy multiplication table*/

for (j = 0; j < 16; j++) {
    for (i = 0; i < 4; i++) {
        c = (j << (i*4));
        table[i][j] = gf->multiply.w32(gf, c, val);
    }
}

/*Ben: s8 is the start of source, d8 is the start of dest, top is end of dest region. */

s8 = (uint8_t *) rd.s_start;
d8 = (uint8_t *) rd.d_start;
top = (uint8_t *) rd.d_top;

while (d8 < top) {
    /*Ben: Multiplies across 16 two byte quantities using alternate mapping
       high bits are on the left, low bits are on the right. */

    for (j=0; j<16; j++) {
        /*Ben: If the xor flag is set, the product should include what is in dest */
        prod = (xor) ? ((uint16_t) (*d8)<<8) ^ *(d8+16) : 0;

        /*Ben: xors all 4 table lookups into the product variable*/

        prod ^= ((table[0][*(s8+16)&0xf]) ^
            (table[1][(*(s8+16)&0xf0]>>4]) ^
            (table[2][*(s8)&0xf]) ^
            (table[3][*(s8)&0xf0]>>4]));

        /*Ben: Stores product in the destination and moves on*/

        *d8 = (uint8_t) (prod >> 8);
        *(d8+16) = (uint8_t) (prod & 0x00ff);
        s8++;
        d8++;
    }
    s8+=16;
    d8+=16;
}
gf_do_final_region_alignment(&rd);
}

static
void
gf_wl6_split_4_16_lazy_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t i, j, a, c, prod;
    uint16_t *s16, *d16, *top;
```

*src/gf\_wl6.c lines 841 to 900*

```
uint16_t table[4][16];
gf_region_data rd;

if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 2);
gf_do_initial_region_alignment(&rd);

for (j = 0; j < 16; j++) {
    for (i = 0; i < 4; i++) {
        c = (j << (i*4));
        table[i][j] = gf->multiply.w32(gf, c, val);
    }
}

s16 = (uint16_t *) rd.s_start;
d16 = (uint16_t *) rd.d_start;
top = (uint16_t *) rd.d_top;

while (d16 < top) {
    a = *s16;
    prod = (xor) ? *d16 : 0;
    for (i = 0; i < 4; i++) {
        prod ^= table[i][a&0xf];
        a >>= 4;
    }
    *d16 = prod;
    s16++;
    d16++;
}

static
void
gf_wl6_split_8_16_lazy_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t j, k, v, a, prod, *s64, *d64, *top64;
    gf_internal_t *h;
    uint64_t htable[256], ltable[256];
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
    gf_do_initial_region_alignment(&rd);

    h = (gf_internal_t *) gf->scratch;

    v = val;
    ltable[0] = 0;
    for (j = 1; j < 256; j <= 1) {
        for (k = 0; k < j; k++) ltable[k^j] = (v ^ ltable[k]);
        v = GF_MULTBY_TWO(v);
    }
    htable[0] = 0;
    for (j = 1; j < 256; j <= 1) {
        for (k = 0; k < j; k++) htable[k^j] = (v ^ htable[k]);
        v = GF_MULTBY_TWO(v);
    }
}
```



*src/gf\_wl6.c lines 901 to 960*

```
    }

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    top64 = (uint64_t *) rd.d_top;

/* Does Unrolling Matter?  -- Doesn't seem to.
   while (d64 != top64) {
       a = *s64;

       prod = htable[a >> 56];
       a <<= 8;
       prod ^= ltable[a >> 56];
       a <<= 8;
       prod <<= 16;

       prod ^= htable[a >> 56];
       a <<= 8;
       prod ^= ltable[a >> 56];
       a <<= 8;
       prod <<= 16;

       prod ^= htable[a >> 56];
       a <<= 8;
       prod ^= ltable[a >> 56];
       a <<= 8;
       prod <<= 16;

       prod ^= htable[a >> 56];
       a <<= 8;
       prod ^= ltable[a >> 56];
       prod ^= ((xor) ? *d64 : 0);
       *d64 = prod;
       s64++;
       d64++;
   }
*/

   while (d64 != top64) {
       a = *s64;

       prod = 0;
       for (j = 0; j < 4; j++) {
           prod <<= 16;
           prod ^= htable[a >> 56];
           a <<= 8;
           prod ^= ltable[a >> 56];
           a <<= 8;
       }

       /* JSP: We can move the conditional outside the while loop,
          but we need to fully test it to understand which is better. */

       prod ^= ((xor) ? *d64 : 0);
       *d64 = prod;
       s64++;
       d64++;
   }
   gf_do_final_region_alignment(&rd);
}
```

*src/gf\_w16.c lines 961 to 1020*

```
static void
gf_w16_table_lazy_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t c;
    gf_internal_t *h;
    struct gf_w16_lazytable_data *ltd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
    gf_do_initial_region_alignment(&rd);

    h = (gf_internal_t *) gf->scratch;
    ltd = (struct gf_w16_lazytable_data *) h->private;

    ltd->lazytable[0] = 0;

    /*
    a = val;
    c = 1;
    pp = h->prim_poly;

    do {
        ltd->lazytable[c] = a;
        c <<= 1;
        if (c & (1 << GF_FIELD_WIDTH)) c ^= pp;
        a <<= 1;
        if (a & (1 << GF_FIELD_WIDTH)) a ^= pp;
    } while (c != 1);
    */

    for (c = 1; c < GF_FIELD_SIZE; c++) {
        ltd->lazytable[c] = gf_w16_shift_multiply(gf, c, val);
    }

    gf_two_byte_region_table_multiply(&rd, ltd->lazytable);
    gf_do_final_region_alignment(&rd);
}

static
void
gf_w16_split_4_16_lazy_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
#ifdef INTEL_SSSE3
    uint64_t i, j, *s64, *d64, *top64;;
    uint64_t c, prod;
    uint8_t low[4][16];
    uint8_t high[4][16];
    gf_region_data rd;

    __m128i mask, ta, tb, ti, tpl, tph, tlow[4], thigh[4], tta, ttb, lmask;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 32);
    gf_do_initial_region_alignment(&rd);
#endif
}
```



*src/gf\_w16.c lines 1021 to 1080*

```
for (j = 0; j < 16; j++) {
    for (i = 0; i < 4; i++) {
        c = (j << (i*4));
        prod = gf->multiply.w32(gf, c, val);
        low[i][j] = (prod & 0xff);
        high[i][j] = (prod >> 8);
    }
}

for (i = 0; i < 4; i++) {
    tlow[i] = _mm_loadu_si128((__m128i *)low[i]);
    thigh[i] = _mm_loadu_si128((__m128i *)high[i]);
}

s64 = (uint64_t *) rd.s_start;
d64 = (uint64_t *) rd.d_start;
top64 = (uint64_t *) rd.d_top;

mask = _mm_set1_epi8 (0x0f);
lmask = _mm_set1_epi16 (0xff);

if (xor) {
    while (d64 != top64) {

        ta = _mm_load_si128((__m128i *) s64);
        tb = _mm_load_si128((__m128i *) (s64+2));

        tta = _mm_srli_epi16(ta, 8);
        ttb = _mm_srli_epi16(tb, 8);
        tpl = _mm_and_si128(tb, lmask);
        tph = _mm_and_si128(ta, lmask);

        tb = _mm_packus_epi16(tpl, tph);
        ta = _mm_packus_epi16(ttb, tta);

        ti = _mm_and_si128 (mask, tb);
        tph = _mm_shuffle_epi8 (thigh[0], ti);
        tpl = _mm_shuffle_epi8 (tlow[0], ti);

        tb = _mm_srli_epi16(tb, 4);
        ti = _mm_and_si128 (mask, tb);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[1], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[1], ti), tph);

        ti = _mm_and_si128 (mask, ta);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[2], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[2], ti), tph);

        ta = _mm_srli_epi16(ta, 4);
        ti = _mm_and_si128 (mask, ta);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[3], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[3], ti), tph);

        ta = _mm_unpackhi_epi8(tpl, tph);
        tb = _mm_unpacklo_epi8(tpl, tph);

        tta = _mm_load_si128((__m128i *) d64);
        ta = _mm_xor_si128(ta, tta);
        ttb = _mm_load_si128((__m128i *) (d64+2));
```

*src/gf\_wl6.c lines 1081 to 1140*

```
    tb = _mm_xor_si128(tb, ttb);
    _mm_store_si128 ((__m128i *)d64, ta);
    _mm_store_si128 ((__m128i *) (d64+2), tb);

    d64 += 4;
    s64 += 4;
}
} else {
    while (d64 != top64) {

        ta = _mm_load_si128((__m128i *) s64);
        tb = _mm_load_si128((__m128i *) (s64+2));

        tta = _mm_srli_epi16(ta, 8);
        ttb = _mm_srli_epi16(tb, 8);
        tpl = _mm_and_si128(tb, lmask);
        tph = _mm_and_si128(ta, lmask);

        tb = _mm_packus_epi16(tpl, tph);
        ta = _mm_packus_epi16(ttb, tta);

        ti = _mm_and_si128 (mask, tb);
        tph = _mm_shuffle_epi8 (thigh[0], ti);
        tpl = _mm_shuffle_epi8 (tlow[0], ti);

        tb = _mm_srli_epi16(tb, 4);
        ti = _mm_and_si128 (mask, tb);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[1], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[1], ti), tph);

        ti = _mm_and_si128 (mask, ta);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[2], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[2], ti), tph);

        ta = _mm_srli_epi16(ta, 4);
        ti = _mm_and_si128 (mask, ta);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[3], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[3], ti), tph);

        ta = _mm_unpackhi_epi8(tpl, tph);
        tb = _mm_unpacklo_epi8(tpl, tph);

        _mm_store_si128 ((__m128i *)d64, ta);
        _mm_store_si128 ((__m128i *) (d64+2), tb);

        d64 += 4;
        s64 += 4;
    }
}

gf_do_final_region_alignment(&rd);
#endif
}

static
void
gf_wl6_split_4_16_lazy_sse_altmap_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
#ifdef INTEL_SSSE3
```



*src/gf\_wl6.c lines 1141 to 1200*

```
uint64_t i, j, *s64, *d64, *top64;;
uint64_t c, prod;
uint8_t low[4][16];
uint8_t high[4][16];
gf_region_data rd;
__m128i mask, ta, tb, ti, tpl, tph, tlow[4], thigh[4];

if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 32);
gf_do_initial_region_alignment(&rd);

for (j = 0; j < 16; j++) {
    for (i = 0; i < 4; i++) {
        c = (j << (i*4));
        prod = gf->multiply.w32(gf, c, val);
        low[i][j] = (prod & 0xff);
        high[i][j] = (prod >> 8);
    }
}

for (i = 0; i < 4; i++) {
    tlow[i] = _mm_loadu_si128((__m128i *)low[i]);
    thigh[i] = _mm_loadu_si128((__m128i *)high[i]);
}

s64 = (uint64_t *) rd.s_start;
d64 = (uint64_t *) rd.d_start;
top64 = (uint64_t *) rd.d_top;

mask = _mm_set1_epi8 (0x0f);

if (xor) {
    while (d64 != top64) {

        ta = _mm_load_si128((__m128i *) s64);
        tb = _mm_load_si128((__m128i *) (s64+2));

        ti = _mm_and_si128 (mask, tb);
        tph = _mm_shuffle_epi8 (thigh[0], ti);
        tpl = _mm_shuffle_epi8 (tlow[0], ti);

        tb = _mm_srli_epi16(tb, 4);
        ti = _mm_and_si128 (mask, tb);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[1], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[1], ti), tph);

        ti = _mm_and_si128 (mask, ta);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[2], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[2], ti), tph);

        ta = _mm_srli_epi16(ta, 4);
        ti = _mm_and_si128 (mask, ta);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[3], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[3], ti), tph);

        ta = _mm_load_si128((__m128i *) d64);
        tph = _mm_xor_si128(tph, ta);
        _mm_store_si128 ((__m128i *)d64, tph);
    }
}
```

*src/gf\_w16.c lines 1201 to 1260*

```
    tb = _mm_load_si128((__m128i *) (d64+2));
    tpl = _mm_xor_si128(tpl, tb);
    _mm_store_si128 ((__m128i *) (d64+2), tpl);

    d64 += 4;
    s64 += 4;
}
} else {
    while (d64 != top64) {

        ta = _mm_load_si128((__m128i *) s64);
        tb = _mm_load_si128((__m128i *) (s64+2));

        ti = _mm_and_si128 (mask, tb);
        tph = _mm_shuffle_epi8 (thigh[0], ti);
        tpl = _mm_shuffle_epi8 (tlow[0], ti);

        tb = _mm_srli_epi16(tb, 4);
        ti = _mm_and_si128 (mask, tb);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[1], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[1], ti), tph);

        ti = _mm_and_si128 (mask, ta);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[2], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[2], ti), tph);

        ta = _mm_srli_epi16(ta, 4);
        ti = _mm_and_si128 (mask, ta);
        tpl = _mm_xor_si128(_mm_shuffle_epi8 (tlow[3], ti), tpl);
        tph = _mm_xor_si128(_mm_shuffle_epi8 (thigh[3], ti), tph);

        _mm_store_si128 ((__m128i *) d64, tph);
        _mm_store_si128 ((__m128i *) (d64+2), tpl);

        d64 += 4;
        s64 += 4;

    }
}
gf_do_final_region_alignment(&rd);
```

```
#endif
}

uint32_t
gf_w16_split_8_8_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint32_t alow, blow;
    struct gf_w16_split_8_8_data *d8;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    d8 = (struct gf_w16_split_8_8_data *) h->private;

    alow = a & 0xff;
    blow = b & 0xff;
    a >>= 8;
    b >>= 8;

    return d8->tables[0][alow][blow] ^
```



*src/gf\_w16.c lines 1261 to 1320*

```
        d8->tables[1][alow][b] ^
        d8->tables[1][a][blow] ^
        d8->tables[2][a][b];
}

static
int gf_w16_split_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_w16_split_8_8_data *d8;
    int i, j, exp, issse3;
    uint32_t p, basep;

    h = (gf_internal_t *) gf->scratch;

#ifdef INTEL_SSSE3
    issse3 = 1;
#else
    issse3 = 0;
#endif

    if (h->arg1 == 8 && h->arg2 == 8) {
        d8 = (struct gf_w16_split_8_8_data *) h->private;
        basep = 1;
        for (exp = 0; exp < 3; exp++) {
            for (j = 0; j < 256; j++) d8->tables[exp][0][j] = 0;
            for (i = 0; i < 256; i++) d8->tables[exp][i][0] = 0;
            d8->tables[exp][1][1] = basep;
            for (i = 2; i < 256; i++) {
                if (i&1) {
                    p = d8->tables[exp][i^1][1];
                    d8->tables[exp][i][1] = p ^ basep;
                } else {
                    p = d8->tables[exp][i>>1][1];
                    d8->tables[exp][i][1] = GF_MULTBY_TWO(p);
                }
            }
            for (i = 1; i < 256; i++) {
                p = d8->tables[exp][i][1];
                for (j = 1; j < 256; j++) {
                    if (j&1) {
                        d8->tables[exp][i][j] = d8->tables[exp][i][j^1] ^ p;
                    } else {
                        d8->tables[exp][i][j] = GF_MULTBY_TWO(d8->tables[exp][i][j>>1]);
                    }
                }
            }
            for (i = 0; i < 8; i++) basep = GF_MULTBY_TWO(basep);
        }
        gf->multiply.w32 = gf_w16_split_8_8_multiply;
        gf->multiply_region.w32 = gf_w16_split_8_16_lazy_multiply_region;
        return 1;
    }

    /* We'll be using LOG for multiplication, unless the pp isn't primitive.
       In that case, we'll be using SHIFT. */

    gf_w16_log_init(gf);
```

*src/gf\_wl6.c lines 1321 to 1380*

```
/* Defaults */

if (issse3) {
    gf->multiply_region.w32 = gf_wl6_split_4_16_lazy_sse_multiply_region;
} else {
    gf->multiply_region.w32 = gf_wl6_split_8_16_lazy_multiply_region;
}

if ((h->arg1 == 8 && h->arg2 == 16) || (h->arg2 == 8 && h->arg1 == 16)) {
    gf->multiply_region.w32 = gf_wl6_split_8_16_lazy_multiply_region;
} else if ((h->arg1 == 4 && h->arg2 == 16) || (h->arg2 == 4 && h->arg1 == 16)) {
    if (issse3) {
        if(h->region_type & GF_REGION_ALTMAP && h->region_type & GF_REGION_NOSSE)
            gf->multiply_region.w32 = gf_wl6_split_4_16_lazy_nosse_altmap_multiply_region;
        else if(h->region_type & GF_REGION_NOSSE)
            gf->multiply_region.w32 = gf_wl6_split_4_16_lazy_multiply_region;
        else if(h->region_type & GF_REGION_ALTMAP)
            gf->multiply_region.w32 = gf_wl6_split_4_16_lazy_sse_altmap_multiply_region;
    } else {
        if(h->region_type & GF_REGION_SSE)
            return 0;
        else if(h->region_type & GF_REGION_ALTMAP)
            gf->multiply_region.w32 = gf_wl6_split_4_16_lazy_nosse_altmap_multiply_region;
        else
            gf->multiply_region.w32 = gf_wl6_split_4_16_lazy_multiply_region;
    }
}

return 1;
}

static
int gf_wl6_table_init(gf_t *gf)
{
    gf_wl6_log_init(gf);

    gf->multiply_region.w32 = gf_wl6_table_lazy_multiply_region;
    return 1;
}

static
void
gf_wl6_log_zero_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint16_t lv;
    int i;
    uint16_t *s16, *d16, *top16;
    struct gf_wl6_zero_logtable_data *ltd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 2);
    gf_do_initial_region_alignment(&rd);

    ltd = (struct gf_wl6_zero_logtable_data*) ((gf_internal_t *) gf->scratch)->private;
    s16 = (uint16_t *) rd.s_start;
```



*src/gf\_w16.c lines 1381 to 1440*

```
    d16 = (uint16_t *) rd.d_start;
    top16 = (uint16_t *) rd.d_top;
    bytes = top16 - d16;

    lv = ltd->log_tbl[val];

    if (xor) {
        for (i = 0; i < bytes; i++) {
            d16[i] ^= (ltd->antilog_tbl[lv + ltd->log_tbl[s16[i]]]);
        }
    } else {
        for (i = 0; i < bytes; i++) {
            d16[i] = (ltd->antilog_tbl[lv + ltd->log_tbl[s16[i]]]);
        }
    }

    /* This isn't necessary. */

    gf_do_final_region_alignment(&rd);
}

/* Here -- double-check Kevin */

static
inline
gf_val_32_t
gf_w16_log_zero_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w16_zero_logtable_data *ltd;

    ltd = (struct gf_w16_zero_logtable_data *) ((gf_internal_t *) gf->scratch)->private;
    return ltd->antilog_tbl[ltd->log_tbl[a] + ltd->log_tbl[b]];
}

static
inline
gf_val_32_t
gf_w16_log_zero_divide (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    int log_sum = 0;
    struct gf_w16_zero_logtable_data *ltd;

    if (a == 0 || b == 0) return 0;
    ltd = (struct gf_w16_zero_logtable_data *) ((gf_internal_t *) gf->scratch)->private;

    log_sum = ltd->log_tbl[a] - ltd->log_tbl[b] + (GF_MULT_GROUP_SIZE);
    return (ltd->antilog_tbl[log_sum]);
}

static
gf_val_32_t
gf_w16_log_zero_inverse (gf_t *gf, gf_val_32_t a)
{
    struct gf_w16_zero_logtable_data *ltd;

    ltd = (struct gf_w16_zero_logtable_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ltd->inv_tbl[a]);
}

static
```

*src/gf\_wl6.c lines 1441 to 1500*

```
inline
gf_val_32_t
gf_wl6_bytwo_p_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint32_t prod, pp, pmask, amask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;
    pmask = 0x8000;
    amask = 0x8000;

    while (amask != 0) {
        if (prod & pmask) {
            prod = ((prod << 1) ^ pp);
        } else {
            prod <<= 1;
        }
        if (a & amask) prod ^= b;
        amask >>= 1;
    }
    return prod;
}
```

```
static
inline
gf_val_32_t
gf_wl6_bytwo_b_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint32_t prod, pp, bmask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;
    bmask = 0x8000;

    while (1) {
        if (a & 1) prod ^= b;
        a >>= 1;
        if (a == 0) return prod;
        if (b & bmask) {
            b = ((b << 1) ^ pp);
        } else {
            b <<= 1;
        }
    }
}
```

```
static
void
gf_wl6_bytwo_p_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t *s64, *d64, t1, t2, ta, prod, amask;
    gf_region_data rd;
    struct gf_wl6_bytwo_data *btd;
```



*src/gf\_wl6.c lines 1501 to 1560*

```
if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

btd = (struct gf_wl6_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
gf_do_initial_region_alignment(&rd);

s64 = (uint64_t *) rd.s_start;
d64 = (uint64_t *) rd.d_start;

if (xor) {
    while (s64 < (uint64_t *) rd.s_top) {
        prod = 0;
        amask = 0x8000;
        ta = *s64;
        while (amask != 0) {
            AB2(btd->prim_poly, btd->mask1, btd->mask2, prod, t1, t2);
            if (val & amask) prod ^= ta;
            amask >>= 1;
        }
        *d64 ^= prod;
        d64++;
        s64++;
    }
} else {
    while (s64 < (uint64_t *) rd.s_top) {
        prod = 0;
        amask = 0x8000;
        ta = *s64;
        while (amask != 0) {
            AB2(btd->prim_poly, btd->mask1, btd->mask2, prod, t1, t2);
            if (val & amask) prod ^= ta;
            amask >>= 1;
        }
        *d64 = prod;
        d64++;
        s64++;
    }
}
gf_do_final_region_alignment(&rd);
}

#define BYTWO_P_ONESTEP {\
    SSE_AB2(pp, m1, m2, prod, t1, t2); \
    t1 = _mm_and_si128(v, one); \
    t1 = _mm_sub_epi16(t1, one); \
    t1 = _mm_and_si128(t1, ta); \
    prod = _mm_xor_si128(prod, t1); \
    v = _mm_srli_epi64(v, 1); }

#ifdef INTEL_SSE2
static
void
gf_wl6_bytwo_p_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int i;
    uint8_t *s8, *d8;
    uint32_t vrev;
```





*src/gf\_w16.c lines 1621 to 1680*

```
uint8_t *d8, *s8;
__m128i pp, m1, m2, t1, t2, va;

s8 = (uint8_t *) rd->s_start;
d8 = (uint8_t *) rd->d_start;

pp = _mm_set1_epi16(btd->prim_poly&0xffff);
m1 = _mm_set1_epi16((btd->mask1)&0xffff);
m2 = _mm_set1_epi16((btd->mask2)&0xffff);

while (d8 < (uint8_t *) rd->d_top) {
    va = _mm_load_si128 ((__m128i *) (s8));
    SSE_AB2(pp, m1, m2, va, t1, t2);
    _mm_store_si128((__m128i *) d8, va);
    d8 += 16;
    s8 += 16;
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w16_bytwo_b_sse_region_2_xor(gf_region_data *rd, struct gf_w16_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi16(btd->prim_poly&0xffff);
    m1 = _mm_set1_epi16((btd->mask1)&0xffff);
    m2 = _mm_set1_epi16((btd->mask2)&0xffff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        SSE_AB2(pp, m1, m2, va, t1, t2);
        vb = _mm_load_si128 ((__m128i *) (d8));
        vb = _mm_xor_si128(vb, va);
        _mm_store_si128((__m128i *) d8, vb);
        d8 += 16;
        s8 += 16;
    }
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w16_bytwo_b_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int itb;
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va, vb;
    struct gf_w16_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
```

*src/gf\_w16.c lines 1681 to 1740*

```
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    btd = (struct gf_w16_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

    if (val == 2) {
        if (xor) {
            gf_w16_bytwo_b_sse_region_2_xor(&rd, btd);
        } else {
            gf_w16_bytwo_b_sse_region_2_noxor(&rd, btd);
        }
        gf_do_final_region_alignment(&rd);
        return;
    }

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    pp = _mm_set1_epi16(btd->prim_poly&0xffff);
    m1 = _mm_set1_epi16((btd->mask1)&0xffff);
    m2 = _mm_set1_epi16((btd->mask2)&0xffff);

    while (d8 < (uint8_t *) rd.d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        vb = (!xor) ? _mm_setzero_si128() : _mm_load_si128 ((__m128i *) (d8));
        itb = val;
        while (1) {
            if (itb & 1) vb = _mm_xor_si128(vb, va);
            itb >>= 1;
            if (itb == 0) break;
            SSE_AB2(pp, m1, m2, va, t1, t2);
        }
        _mm_store_si128((__m128i *) d8, vb);
        d8 += 16;
        s8 += 16;
    }

    gf_do_final_region_alignment(&rd);
}
#endif

static
void
gf_w16_bytwo_b_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t *s64, *d64, t1, t2, ta, tb, prod;
    struct gf_w16_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    btd = (struct gf_w16_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;
    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
```



*src/gf\_wl6.c lines 1741 to 1800*

```
switch (val) {
case 2:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= ta;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta;
            d64++;
            s64++;
        }
    }
    break;
case 3:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
            d64++;
            s64++;
        }
    }
    break;
case 4:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= ta;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta;
            d64++;
            s64++;
        }
    }
}
```

*src/gf\_w16.c lines 1801 to 1860*

```
    }
}
break;
case 5:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta ^ prod;
            d64++;
            s64++;
        }
    }
break;
default:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            prod = *d64 ;
            ta = *s64;
            tb = val;
            while (1) {
                if (tb & 1) prod ^= ta;
                tb >>= 1;
                if (tb == 0) break;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            }
            *d64 = prod;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            prod = 0 ;
            ta = *s64;
            tb = val;
            while (1) {
                if (tb & 1) prod ^= ta;
                tb >>= 1;
                if (tb == 0) break;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            }
            *d64 = prod;
            d64++;
            s64++;
        }
    }
break;
}
```



*src/gf\_w16.c lines 1861 to 1920*

```
    gf_do_final_region_alignment(&rd);
}

static
int gf_w16_bytwo_init(gf_t *gf)
{
    gf_internal_t *h;
    uint64_t ip, m1, m2;
    struct gf_w16_bytwo_data *btd;

    h = (gf_internal_t *) gf->scratch;
    btd = (struct gf_w16_bytwo_data *) (h->private);
    ip = h->prim_poly & 0xffff;
    m1 = 0xfffe;
    m2 = 0x8000;
    btd->prim_poly = 0;
    btd->mask1 = 0;
    btd->mask2 = 0;

    while (ip != 0) {
        btd->prim_poly |= ip;
        btd->mask1 |= m1;
        btd->mask2 |= m2;
        ip <<= GF_FIELD_WIDTH;
        m1 <<= GF_FIELD_WIDTH;
        m2 <<= GF_FIELD_WIDTH;
    }

    if (h->mult_type == GF_MULT_BYTWO_p) {
        gf->multiply.w32 = gf_w16_bytwo_p_multiply;
#ifdef INTEL_SSE2
        if (h->region_type & GF_REGION_NOSSE)
            gf->multiply_region.w32 = gf_w16_bytwo_p_nosse_multiply_region;
        else
            gf->multiply_region.w32 = gf_w16_bytwo_p_sse_multiply_region;
#else
        gf->multiply_region.w32 = gf_w16_bytwo_p_nosse_multiply_region;
        if(h->region_type & GF_REGION_SSE)
            return 0;
#endif
    } else {
        gf->multiply.w32 = gf_w16_bytwo_b_multiply;
#ifdef INTEL_SSE2
        if (h->region_type & GF_REGION_NOSSE)
            gf->multiply_region.w32 = gf_w16_bytwo_b_nosse_multiply_region;
        else
            gf->multiply_region.w32 = gf_w16_bytwo_b_sse_multiply_region;
#else
        gf->multiply_region.w32 = gf_w16_bytwo_b_nosse_multiply_region;
        if(h->region_type & GF_REGION_SSE)
            return 0;
#endif
    }

    return 1;
}
```

```
static
int gf_w16_log_zero_init(gf_t *gf)
{

```

*src/gf\_w16.c lines 1921 to 1980*

```
gf_internal_t *h;
struct gf_w16_zero_logtable_data *ltd;
int i, b;

h = (gf_internal_t *) gf->scratch;
ltd = h->private;

ltd->log_tbl[0] = (-GF_MULT_GROUP_SIZE) + 1;

bzero(&(ltd->_antilog_tbl[0]), sizeof(ltd->_antilog_tbl));

ltd->antilog_tbl = &(ltd->_antilog_tbl[GF_FIELD_SIZE * 2]);

b = 1;
for (i = 0; i < GF_MULT_GROUP_SIZE; i++) {
    ltd->log_tbl[b] = (uint16_t)i;
    ltd->antilog_tbl[i] = (uint16_t)b;
    ltd->antilog_tbl[i+GF_MULT_GROUP_SIZE] = (uint16_t)b;
    b <= 1;
    if (b & GF_FIELD_SIZE) {
        b = b ^ h->prim_poly;
    }
}
ltd->inv_tbl[0] = 0; /* Not really, but we need to fill it with something */
ltd->inv_tbl[1] = 1;
for (i = 2; i < GF_FIELD_SIZE; i++) {
    ltd->inv_tbl[i] = ltd->antilog_tbl[GF_MULT_GROUP_SIZE-ltd->log_tbl[i]];
}

gf->inverse.w32 = gf_w16_log_zero_inverse;
gf->divide.w32 = gf_w16_log_zero_divide;
gf->multiply.w32 = gf_w16_log_zero_multiply;
gf->multiply_region.w32 = gf_w16_log_zero_multiply_region;
return 1;
}
```

```
static
gf_val_32_t
gf_w16_composite_multiply_recursive(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint8_t b0 = b & 0x00ff;
    uint8_t b1 = (b & 0xff00) >> 8;
    uint8_t a0 = a & 0x00ff;
    uint8_t a1 = (a & 0xff00) >> 8;
    uint8_t albl;
    uint16_t rv;

    albl = base_gf->multiply.w32(base_gf, a1, b1);

    rv = ((base_gf->multiply.w32(base_gf, a0, b0) ^ albl) |
          ((base_gf->multiply.w32(base_gf, a1, b0) ^
            base_gf->multiply.w32(base_gf, a0, b1) ^
            base_gf->multiply.w32(base_gf, albl, h->prim_poly)) << 8));
    return rv;
}
```

```
static
gf_val_32_t
```



*src/gf\_w16.c lines 1981 to 2040*

```
gf_w16_composite_multiply_inline(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    uint8_t b0 = b & 0x00ff;
    uint8_t b1 = (b & 0xff00) >> 8;
    uint8_t a0 = a & 0x00ff;
    uint8_t a1 = (a & 0xff00) >> 8;
    uint8_t alb1, *mt;
    uint16_t rv;
    struct gf_w16_composite_data *cd;

    cd = (struct gf_w16_composite_data *) h->private;
    mt = cd->mult_table;

    alb1 = GF_W8_INLINE_MULTDIV(mt, a1, b1);

    rv = ((GF_W8_INLINE_MULTDIV(mt, a0, b0) ^ alb1) |
          ((GF_W8_INLINE_MULTDIV(mt, a1, b0) ^
            GF_W8_INLINE_MULTDIV(mt, a0, b1) ^
            GF_W8_INLINE_MULTDIV(mt, alb1, h->prim_poly)) << 8));
    return rv;
}
```

```
/*
 * Composite field division trick (explained in 2007 tech report)
 *
 * Compute  $a / b = a \cdot b^{-1}$ , where  $p(x) = x^2 + sx + 1$ 
 *
 * let  $c = b^{-1}$ 
 *
 *  $c \cdot b = (s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1)x + (b_1 c_1 + b_0 c_0)$ 
 *
 * want  $(s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1) = 0$  and  $(b_1 c_1 + b_0 c_0) = 1$ 
 *
 * let  $d = b_1 c_1$  and  $d+1 = b_0 c_0$ 
 *
 * solve  $s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1 = 0$ 
 *
 * solution:  $d = (b_1 b_0^{-1}) (b_1 b_0^{-1} + b_0 b_1^{-1} + s)^{-1}$ 
 *
 *  $c_0 = (d+1) b_0^{-1}$ 
 *  $c_1 = d \cdot b_1^{-1}$ 
 *
 *  $a / b = a * c$ 
 */
```

```
static
gf_val_32_t
gf_w16_composite_inverse(gf_t *gf, gf_val_32_t a)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint8_t a0 = a & 0x00ff;
    uint8_t a1 = (a & 0xff00) >> 8;
    uint8_t c0, c1, d, tmp;
    uint16_t c;
    uint8_t a0inv, a1inv;

    if (a0 == 0) {
        a1inv = base_gf->inverse.w32(base_gf, a1);
```

*src/gf\_w16.c lines 2041 to 2100*

```
    c0 = base_gf->multiply.w32(base_gf, a1inv, h->prim_poly);
    c1 = a1inv;
} else if (a1 == 0) {
    c0 = base_gf->inverse.w32(base_gf, a0);
    c1 = 0;
} else {
    a1inv = base_gf->inverse.w32(base_gf, a1);
    a0inv = base_gf->inverse.w32(base_gf, a0);

    d = base_gf->multiply.w32(base_gf, a1, a0inv);

    tmp = (base_gf->multiply.w32(base_gf, a1, a0inv) ^ base_gf->multiply.w32(base_gf, a0, a1inv) ^ h->prim_poly);
    tmp = base_gf->inverse.w32(base_gf, tmp);

    d = base_gf->multiply.w32(base_gf, d, tmp);

    c0 = base_gf->multiply.w32(base_gf, (d^1), a0inv);
    c1 = base_gf->multiply.w32(base_gf, d, a1inv);
}

c = c0 | (c1 << 8);

return c;
}
```

```
static
void
gf_w16_composite_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
```

```
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint8_t b0 = val & 0x00ff;
    uint8_t b1 = (val & 0xff00) >> 8;
    uint16_t *s16, *d16, *top;
    uint8_t a0, a1, alb1, *mt;
    gf_region_data rd;
    struct gf_w16_composite_data *cd;
```

```
    cd = (struct gf_w16_composite_data *) h->private;
    mt = cd->mult_table;
```

```
    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 2);
```

```
    s16 = rd.s_start;
    d16 = rd.d_start;
    top = rd.d_top;
```

```
    if (mt == NULL) {
        if (xor) {
            while (d16 < top) {
                a0 = (*s16) & 0x00ff;
                a1 = ((*s16) & 0xff00) >> 8;
                alb1 = base_gf->multiply.w32(base_gf, a1, b1);

                (*d16) ^= ((base_gf->multiply.w32(base_gf, a0, b0) ^ alb1) |
                           ((base_gf->multiply.w32(base_gf, a1, b0) ^
                              base_gf->multiply.w32(base_gf, a0, b1) ^
                              base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 8));

                s16++;
```



*src/gf\_w16.c lines 2101 to 2160*

```
        d16++;
    }
} else {
    while (d16 < top) {
        a0 = (*s16) & 0x00ff;
        a1 = ((*s16) & 0xff00) >> 8;
        alb1 = base_gf->multiply.w32(base_gf, a1, b1);

        (*d16) = ((base_gf->multiply.w32(base_gf, a0, b0) ^ alb1) |
                  ((base_gf->multiply.w32(base_gf, a1, b0) ^
                    base_gf->multiply.w32(base_gf, a0, b1) ^
                    base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 8));

        s16++;
        d16++;
    }
}
} else {
    if (xor) {
        while (d16 < top) {
            a0 = (*s16) & 0x00ff;
            a1 = ((*s16) & 0xff00) >> 8;
            alb1 = GF_W8_INLINE_MULTDIV(mt, a1, b1);

            (*d16) ^= ((GF_W8_INLINE_MULTDIV(mt, a0, b0) ^ alb1) |
                      ((GF_W8_INLINE_MULTDIV(mt, a1, b0) ^
                        GF_W8_INLINE_MULTDIV(mt, a0, b1) ^
                        GF_W8_INLINE_MULTDIV(mt, alb1, h->prim_poly)) << 8));

            s16++;
            d16++;
        }
    } else {
        while (d16 < top) {
            a0 = (*s16) & 0x00ff;
            a1 = ((*s16) & 0xff00) >> 8;
            alb1 = GF_W8_INLINE_MULTDIV(mt, a1, b1);

            (*d16) = ((GF_W8_INLINE_MULTDIV(mt, a0, b0) ^ alb1) |
                      ((GF_W8_INLINE_MULTDIV(mt, a1, b0) ^
                        GF_W8_INLINE_MULTDIV(mt, a0, b1) ^
                        GF_W8_INLINE_MULTDIV(mt, alb1, h->prim_poly)) << 8));

            s16++;
            d16++;
        }
    }
}
}
}
```

```
static
void
gf_w16_composite_multiply_region_alt(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint8_t val0 = val & 0x00ff;
    uint8_t val1 = (val & 0xff00) >> 8;
    gf_region_data rd;
    int sub_reg_size;
    uint8_t *slow, *shigh;
    uint8_t *dlow, *dhigh, *top;;
```

*src/gf\_wl6.c lines 2161 to 2220*

```
/* JSP: I want the two pointers aligned wrt each other on 16 byte
   boundaries. So I'm going to make sure that the area on
   which the two operate is a multiple of 32. Of course, that
   junks up the mapping, but so be it -- that's why we have extract_word.... */

gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 32);
gf_do_initial_region_alignment(&rd);

slow = (uint8_t *) rd.s_start;
dlow = (uint8_t *) rd.d_start;
top = (uint8_t *) rd.d_top;
sub_reg_size = (top - dlow)/2;
shigh = slow + sub_reg_size;
dhigh = dlow + sub_reg_size;

base_gf->multiply_region.w32(base_gf, slow, dlow, val0, sub_reg_size, xor);
base_gf->multiply_region.w32(base_gf, shigh, dlow, val1, sub_reg_size, 1);
base_gf->multiply_region.w32(base_gf, slow, dhigh, val1, sub_reg_size, xor);
base_gf->multiply_region.w32(base_gf, shigh, dhigh, val0, sub_reg_size, 1);
base_gf->multiply_region.w32(base_gf, shigh, dhigh,
    base_gf->multiply.w32(base_gf, h->prim_poly, val1), sub_reg_size, 1);

gf_do_final_region_alignment(&rd);
}

static
int gf_wl6_composite_init(gf_t *gf)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    struct gf_wl6_composite_data *cd;

    if (h->base_gf == NULL) return 0;

    cd = (struct gf_wl6_composite_data *) h->private;
    cd->mult_table = gf_w8_get_mult_table(h->base_gf);

    if (h->region_type & GF_REGION_ALTMAP) {
        gf->multiply_region.w32 = gf_wl6_composite_multiply_region_alt;
    } else {
        gf->multiply_region.w32 = gf_wl6_composite_multiply_region;
    }

    if (cd->mult_table == NULL) {
        gf->multiply.w32 = gf_wl6_composite_multiply_recursive;
    } else {
        gf->multiply.w32 = gf_wl6_composite_multiply_inline;
    }
    gf->divide.w32 = NULL;
    gf->inverse.w32 = gf_wl6_composite_inverse;

    return 1;
}

static
void
gf_wl6_group_4_set_shift_tables(uint16_t *shift, uint16_t val, gf_internal_t *h)
{
    int i, j;

    shift[0] = 0;
```



*src/gf\_w16.c lines 2221 to 2280*

```
for (i = 0; i < 16; i += 2) {
    j = (shift[i>>1] << 1);
    if (j & (1 << 16)) j ^= h->prim_poly;
    shift[i] = j;
    shift[i^1] = j^val;
}
}
```

```
static
inline
gf_val_32_t
gf_w16_group_4_4_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint16_t p, l, ind, r, a16;

    struct gf_w16_group_4_4_data *d44;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;

    d44 = (struct gf_w16_group_4_4_data *) h->private;
    gf_w16_group_4_set_shift_tables(d44->shift, b, h);

    a16 = a;
    ind = a16 >> 12;
    a16 <<= 4;
    p = d44->shift[ind];
    r = p & 0xffff;
    l = p >> 12;
    ind = a16 >> 12;
    a16 <<= 4;
    p = (d44->shift[ind] ^ d44->reduce[l] ^ (r << 4));
    r = p & 0xffff;
    l = p >> 12;
    ind = a16 >> 12;
    a16 <<= 4;
    p = (d44->shift[ind] ^ d44->reduce[l] ^ (r << 4));
    r = p & 0xffff;
    l = p >> 12;
    ind = a16 >> 12;
    p = (d44->shift[ind] ^ d44->reduce[l] ^ (r << 4));
    return p;
}
```

```
static
void gf_w16_group_4_4_region_multiply(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint16_t p, l, ind, r, a16, p16;
    struct gf_w16_group_4_4_data *d44;
    gf_region_data rd;
    uint16_t *s16, *d16, *top;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    d44 = (struct gf_w16_group_4_4_data *) h->private;
    gf_w16_group_4_set_shift_tables(d44->shift, val, h);

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 2);
    gf_do_initial_region_alignment(&rd);
}
```

*src/gf\_w16.c lines 2281 to 2340*

```
s16 = (uint16_t *) rd.s_start;
d16 = (uint16_t *) rd.d_start;
top = (uint16_t *) rd.d_top;

while (d16 < top) {
    a16 = *s16;
    p16 = (xor) ? *d16 : 0;
    ind = a16 >> 12;
    a16 <=< 4;
    p = d44->shift[ind];
    r = p & 0xffff;
    l = p >> 12;
    ind = a16 >> 12;
    a16 <=< 4;
    p = (d44->shift[ind] ^ d44->reduce[l] ^ (r << 4));
    r = p & 0xffff;
    l = p >> 12;
    ind = a16 >> 12;
    a16 <=< 4;
    p = (d44->shift[ind] ^ d44->reduce[l] ^ (r << 4));
    r = p & 0xffff;
    l = p >> 12;
    ind = a16 >> 12;
    p = (d44->shift[ind] ^ d44->reduce[l] ^ (r << 4));
    p ^= p16;
    *d16 = p;
    d16++;
    s16++;
}
gf_do_final_region_alignment(&rd);
}

static
int gf_w16_group_init(gf_t *gf)
{
    int i, j, p;
    struct gf_w16_group_4_4_data *d44;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;

    d44 = (struct gf_w16_group_4_4_data *) h->private;
    d44->reduce[0] = 0;
    for (i = 0; i < 16; i++) {
        p = 0;
        for (j = 0; j < 4; j++) {
            if (i & (1 << j)) p ^= (h->prim_poly << j);
        }
        d44->reduce[p>>16] = (p&0xfffff);
    }

    gf->multiply.w32 = gf_w16_group_4_4_multiply;
    gf->divide.w32 = NULL;
    gf->inverse.w32 = NULL;
    gf->multiply_region.w32 = gf_w16_group_4_4_region_multiply;

    return 1;
}

int gf_w16_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2)
{
    switch(mult_type)
```



*src/gf\_w16.c lines 2341 to 2400*

```
{
    case GF_MULT_TABLE:
        return sizeof(gf_internal_t) + sizeof(struct gf_w16_lazytable_data) + 64;
        break;
    case GF_MULT_BYTWO_p:
    case GF_MULT_BYTWO_b:
        return sizeof(gf_internal_t) + sizeof(struct gf_w16_bytwo_data);
        break;
    case GF_MULT_LOG_ZERO:
        return sizeof(gf_internal_t) + sizeof(struct gf_w16_zero_logtable_data) + 64;
        break;
    case GF_MULT_LOG_TABLE:
        return sizeof(gf_internal_t) + sizeof(struct gf_w16_logtable_data) + 64;
        break;
    case GF_MULT_DEFAULT:
    case GF_MULT_SPLIT_TABLE:
        if (arg1 == 8 && arg2 == 8) {
            return sizeof(gf_internal_t) + sizeof(struct gf_w16_split_8_8_data) + 64;
        } else if ((arg1 == 8 && arg2 == 16) || (arg2 == 8 && arg1 == 16)) {
            return sizeof(gf_internal_t) + sizeof(struct gf_w16_logtable_data) + 64;
        } else if (mult_type == GF_MULT_DEFAULT ||
                    (arg1 == 4 && arg2 == 16) || (arg2 == 4 && arg1 == 16)) {
            return sizeof(gf_internal_t) + sizeof(struct gf_w16_logtable_data) + 64;
        }
        return 0;
        break;
    case GF_MULT_GROUP:
        return sizeof(gf_internal_t) + sizeof(struct gf_w16_group_4_4_data) + 64;
        break;
    case GF_MULT_CARRY_FREE:
        return sizeof(gf_internal_t);
        break;
    case GF_MULT_SHIFT:
        return sizeof(gf_internal_t);
        break;
    case GF_MULT_COMPOSITE:
        return sizeof(gf_internal_t) + sizeof(struct gf_w16_composite_data) + 64;
        break;

    default:
        return 0;
}
return 0;
}

int gf_w16_init(gf_t *gf)
{
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    /* Allen: set default primitive polynomial / irreducible polynomial if needed */

    if (h->prim_poly == 0) {
        if (h->mult_type == GF_MULT_COMPOSITE) {
            h->prim_poly = gf_composite_get_default_poly(h->base_gf);
            if (h->prim_poly == 0) return 0;
        } else {

            /* Allen: use the following primitive polynomial to make
```

*src/gf\_w16.c lines 2401 to 2460*

carryless multiply work more efficiently for GF(2<sup>16</sup>).

h->prim\_poly = 0x1002d;

The following is the traditional primitive polynomial for GF(2<sup>16</sup>) \*/

h->prim\_poly = 0x1100b;

}

if (h->mult\_type != GF\_MULT\_COMPOSITE) h->prim\_poly |= (1 << 16);

gf->multiply.w32 = NULL;

gf->divide.w32 = NULL;

gf->inverse.w32 = NULL;

gf->multiply\_region.w32 = NULL;

switch(h->mult\_type) {

case GF\_MULT\_LOG\_ZERO: if (gf\_w16\_log\_zero\_init(gf) == 0) return 0; break;

case GF\_MULT\_LOG\_TABLE: if (gf\_w16\_log\_init(gf) == 0) return 0; break;

case GF\_MULT\_DEFAULT:

case GF\_MULT\_SPLIT\_TABLE: if (gf\_w16\_split\_init(gf) == 0) return 0; break;

case GF\_MULT\_TABLE: if (gf\_w16\_table\_init(gf) == 0) return 0; break;

case GF\_MULT\_CARRY\_FREE: if (gf\_w16\_cfm\_init(gf) == 0) return 0; break;

case GF\_MULT\_SHIFT: if (gf\_w16\_shift\_init(gf) == 0) return 0; break;

case GF\_MULT\_COMPOSITE: if (gf\_w16\_composite\_init(gf) == 0) return 0; break;

case GF\_MULT\_BYTWO\_p:

case GF\_MULT\_BYTWO\_b: if (gf\_w16\_bytwo\_init(gf) == 0) return 0; break;

case GF\_MULT\_GROUP: if (gf\_w16\_group\_init(gf) == 0) return 0; break;

default: return 0;

}

if (h->divide\_type == GF\_DIVIDE\_EUCLID) {

gf->divide.w32 = gf\_w16\_divide\_from\_inverse;

gf->inverse.w32 = gf\_w16\_euclid;

} else if (h->divide\_type == GF\_DIVIDE\_MATRIX) {

gf->divide.w32 = gf\_w16\_divide\_from\_inverse;

gf->inverse.w32 = gf\_w16\_matrix;

}

if (gf->divide.w32 == NULL) {

gf->divide.w32 = gf\_w16\_divide\_from\_inverse;

if (gf->inverse.w32 == NULL) gf->inverse.w32 = gf\_w16\_euclid;

}

if (gf->inverse.w32 == NULL) gf->inverse.w32 = gf\_w16\_inverse\_from\_divide;

if (h->region\_type & GF\_REGION\_ALTMAP) {

if (h->mult\_type == GF\_MULT\_COMPOSITE) {

gf->extract\_word.w32 = gf\_w16\_composite\_extract\_word;

} else {

gf->extract\_word.w32 = gf\_w16\_split\_extract\_word;

}

} else if (h->region\_type == GF\_REGION\_CAUCHY) {

gf->multiply\_region.w32 = gf\_wgen\_cauchy\_region;

gf->extract\_word.w32 = gf\_wgen\_extract\_word;

} else {

gf->extract\_word.w32 = gf\_w16\_extract\_word;

}

if (gf->multiply\_region.w32 == NULL) {

gf->multiply\_region.w32 = gf\_w16\_multiply\_region\_from\_single;



*src/gf\_w16.c lines 2461 to 2502*

```
    }
    return 1;
}

/* Inline setup functions */

uint16_t *gf_w16_get_log_table(gf_t *gf)
{
    struct gf_w16_logtable_data *ltd;

    if (gf->multiply.w32 == gf_w16_log_multiply) {
        ltd = (struct gf_w16_logtable_data *) ((gf_internal_t *) gf->scratch)->private;
        return (uint16_t *) ltd->log_tbl;
    }
    return NULL;
}

uint16_t *gf_w16_get_mult_aalog_table(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_w16_logtable_data *ltd;

    h = (gf_internal_t *) gf->scratch;
    if (gf->multiply.w32 == gf_w16_log_multiply) {
        ltd = (struct gf_w16_logtable_data *) h->private;
        return (uint16_t *) ltd->antilog_tbl;
    }
    return NULL;
}

uint16_t *gf_w16_get_div_aalog_table(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_w16_logtable_data *ltd;

    h = (gf_internal_t *) gf->scratch;
    if (gf->multiply.w32 == gf_w16_log_multiply) {
        ltd = (struct gf_w16_logtable_data *) h->private;
        return (uint16_t *) ltd->d_antilog;
    }
    return NULL;
}
```

*src/gf\_w32.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_w32.c
 *
 * Routines for 32-bit Galois fields
 */

#include "gf_int.h"
#include <stdio.h>
#include <stdlib.h>

#define GF_FIELD_WIDTH (32)
#define GF_FIRST_BIT (1 << 31)

#define GF_BASE_FIELD_WIDTH (16)
#define GF_BASE_FIELD_SIZE (1 << GF_BASE_FIELD_WIDTH)
#define GF_BASE_FIELD_GROUP_SIZE GF_BASE_FIELD_SIZE-1
#define GF_MULTBY_TWO(p) (((p) & GF_FIRST_BIT) ? ((p) << 1) ^ h->prim_poly) : (p) << 1)

struct gf_split_2_32_lazy_data {
    uint32_t tables[16][4];
    uint32_t last_value;
};

struct gf_w32_split_8_8_data {
    uint32_t tables[7][256][256];
    uint32_t region_tables[4][256];
    uint32_t last_value;
};

struct gf_w32_group_data {
    uint32_t *reduce;
    uint32_t *shift;
    int tshift;
    uint64_t rmask;
    uint32_t *memory;
};

struct gf_split_16_32_lazy_data {
    uint32_t tables[2][(1<<16)];
    uint32_t last_value;
};

struct gf_split_8_32_lazy_data {
    uint32_t tables[4][256];
    uint32_t last_value;
};

struct gf_split_4_32_lazy_data {
    uint32_t tables[8][16];
    uint32_t last_value;
};

struct gf_w32_bytwo_data {
    uint64_t prim_poly;
    uint64_t mask1;
```



*src/gf\_w32.c lines 61 to 120*

```
    uint64_t mask2;
};

struct gf_w32_composite_data {
    uint16_t *log;
    uint16_t *alog;
};

#define MM_PRINT32(s, r) { uint8_t blah[16], ii; printf("%-12s", s); \
    _mm_storeu_si128((__m128i *)blah, r); \
    for (ii = 0; ii < 16; ii += 4) \
        printf(" %02x%02x%02x%02x", blah[15-ii], blah[14-ii], \
            blah[13-ii], blah[12-ii]); printf("\n"); }

#define MM_PRINT8(s, r) { uint8_t blah[16], ii; printf("%-12s", s); \
    _mm_storeu_si128((__m128i *)blah, r); for (ii = 0; ii < 16; ii += 1) \
        printf("%s%02x", (ii%4==0) ? "    " : " ", blah[15-ii]); printf("\n"); }

#define AB2(ip, am1, am2, b, t1, t2) {\
    t1 = (b << 1) & am1; \
    t2 = b & am2; \
    t2 = ((t2 << 1) - (t2 >> (GF_FIELD_WIDTH-1))); \
    b = (t1 ^ (t2 & ip));}

#define SSE_AB2(pp, m1, m2, va, t1, t2) {\
    t1 = _mm_and_si128(_mm_slli_epi64(va, 1), m1); \
    t2 = _mm_and_si128(va, m2); \
    t2 = _mm_sub_epi64(_mm_slli_epi64(t2, 1), _mm_srli_epi64(t2, (GF_FIELD_WIDTH-1))); \
    va = _mm_xor_si128(t1, _mm_and_si128(t2, pp)); }

static
inline
uint32_t gf_w32_inverse_from_divide (gf_t *gf, uint32_t a)
{
    return gf->divide.w32(gf, 1, a);
}

static
inline
uint32_t gf_w32_divide_from_inverse (gf_t *gf, uint32_t a, uint32_t b)
{
    b = gf->inverse.w32(gf, b);
    return gf->multiply.w32(gf, a, b);
}

static
void
gf_w32_multiply_region_from_single(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int
xor)
{
    int i;
    uint32_t *s32;
    uint32_t *d32;

    s32 = (uint32_t *) src;
    d32 = (uint32_t *) dest;

    if (xor) {
        for (i = 0; i < bytes/sizeof(uint32_t); i++) {
            d32[i] ^= gf->multiply.w32(gf, val, s32[i]);
        }
    }
}
```

*src/gf\_w32.c lines 121 to 180*

```
    }
} else {
    for (i = 0; i < bytes/sizeof(uint32_t); i++) {
        d32[i] = gf->multiply.w32(gf, val, s32[i]);
    }
}
}

#if defined(INTEL_SSE4_PCLMUL)

static
void
gf_w32_clm_multiply_region_from_single_2(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    int i;
    uint32_t *s32;
    uint32_t *d32;

    __m128i a, b;
    __m128i result;
    __m128i prim_poly;
    __m128i w;
    gf_internal_t * h = gf->scratch;

    prim_poly = _mm_set_epi32(0, 0, 1, (uint32_t)(h->prim_poly & 0xffffffffULL));

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);
    s32 = (uint32_t *) src;
    d32 = (uint32_t *) dest;

    if (xor) {
        for (i = 0; i < bytes/sizeof(uint32_t); i++) {
            b = _mm_insert_epi32 (a, s32[i], 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            d32[i] ^= ((gf_val_32_t)_mm_extract_epi32(result, 0));
        }
    } else {
        for (i = 0; i < bytes/sizeof(uint32_t); i++) {
            b = _mm_insert_epi32 (a, s32[i], 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            d32[i] = ((gf_val_32_t)_mm_extract_epi32(result, 0));
        }
    }
}
}
#endif

#if defined(INTEL_SSE4_PCLMUL)
```



*src/gf\_w32.c lines 181 to 240*

```
static
void
gf_w32_clm_multiply_region_from_single_3(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    int i;
    uint32_t *s32;
    uint32_t *d32;

    __m128i a, b;
    __m128i result;
    __m128i prim_poly;
    __m128i w;
    gf_internal_t * h = gf->scratch;

    prim_poly = _mm_set_epi32(0, 0, 1, (uint32_t)(h->prim_poly & 0xffffffffULL));

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);

    s32 = (uint32_t *) src;
    d32 = (uint32_t *) dest;

    if (xor) {
        for (i = 0; i < bytes/sizeof(uint32_t); i++) {
            b = _mm_insert_epi32 (a, s32[i], 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            d32[i] ^= ((gf_val_32_t)_mm_extract_epi32(result, 0));
        }
    } else {
        for (i = 0; i < bytes/sizeof(uint32_t); i++) {
            b = _mm_insert_epi32 (a, s32[i], 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
            result = _mm_xor_si128 (result, w);
            d32[i] = ((gf_val_32_t)_mm_extract_epi32(result, 0));
        }
    }
}
#endif

#if defined(INTEL_SSE4_PCLMUL)
static
void
gf_w32_clm_multiply_region_from_single_4(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    int i;
    uint32_t *s32;
```

*src/gf\_w32.c lines 241 to 300*

```
uint32_t *d32;

__m128i      a, b;
__m128i      result;
__m128i      prim_poly;
__m128i      w;
gf_internal_t * h = gf->scratch;

prim_poly = _mm_set_epi32(0, 0, 1, (uint32_t)(h->prim_poly & 0xffffffffULL));

if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);

s32 = (uint32_t *) src;
d32 = (uint32_t *) dest;

if (xor) {
    for (i = 0; i < bytes/sizeof(uint32_t); i++) {
        b = _mm_insert_epi32 (a, s32[i], 0);
        result = _mm_clmulepi64_si128 (a, b, 0);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
        result = _mm_xor_si128 (result, w);
        d32[i] ^= ((gf_val_32_t)_mm_extract_epi32(result, 0));
    }
} else {
    for (i = 0; i < bytes/sizeof(uint32_t); i++) {
        b = _mm_insert_epi32 (a, s32[i], 0);
        result = _mm_clmulepi64_si128 (a, b, 0);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
        result = _mm_xor_si128 (result, w);
        d32[i] = ((gf_val_32_t)_mm_extract_epi32(result, 0));
    }
}
}
#endif

static
inline
uint32_t gf_w32_euclid (gf_t *gf, uint32_t b)
{
    uint32_t e_i, e_im1, e_ip1;
    uint32_t d_i, d_im1, d_ip1;
    uint32_t y_i, y_im1, y_ip1;
    uint32_t c_i;

    if (b == 0) return -1;
```



*src/gf\_w32.c lines 301 to 360*

```
e_im1 = ((gf_internal_t *) (gf->scratch))->prim_poly;
e_i = b;
d_im1 = 32;
for (d_i = d_im1-1; ((1 << d_i) & e_i) == 0; d_i--) ;
y_i = 1;
y_im1 = 0;

while (e_i != 1) {

    e_ip1 = e_im1;
    d_ip1 = d_im1;
    c_i = 0;

    while (d_ip1 >= d_i) {
        c_i ^= (1 << (d_ip1 - d_i));
        e_ip1 ^= (e_i << (d_ip1 - d_i));
        d_ip1--;
        if (e_ip1 == 0) return 0;
        while ((e_ip1 & (1 << d_ip1)) == 0) d_ip1--;
    }

    y_ip1 = y_im1 ^ gf->multiply.w32(gf, c_i, y_i);
    y_im1 = y_i;
    y_i = y_ip1;

    e_im1 = e_i;
    d_im1 = d_i;
    e_i = e_ip1;
    d_i = d_ip1;
}

return y_i;
}

static
gf_val_32_t gf_w32_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    uint32_t *r32, rv;

    r32 = (uint32_t *) start;
    rv = r32[index];
    return rv;
}

static
gf_val_32_t gf_w32_composite_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    int sub_size;
    gf_internal_t *h;
    uint8_t *r8, *top;
    uint32_t a, b, *r32;
    gf_region_data rd;

    h = (gf_internal_t *) gf->scratch;
    gf_set_region_data(&rd, gf, start, start, bytes, 0, 0, 32);
    r32 = (uint32_t *) start;
    if (r32 + index < (uint32_t *) rd.d_start) return r32[index];
    if (r32 + index >= (uint32_t *) rd.d_top) return r32[index];
    index -= ((uint32_t *) rd.d_start) - r32;
    r8 = (uint8_t *) rd.d_start;
```

*src/gf\_w32.c lines 361 to 420*

```
    top = (uint8_t *) rd.d_top;
    sub_size = (top-r8)/2;

    a = h->base_gf->extract_word.w32(h->base_gf, r8, sub_size, index);
    b = h->base_gf->extract_word.w32(h->base_gf, r8+sub_size, sub_size, index);
    return (a | (b << 16));
}

static
gf_val_32_t gf_w32_split_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    int i;
    uint32_t *r32, rv;
    uint8_t *r8;
    gf_region_data rd;

    gf_set_region_data(&rd, gf, start, start, bytes, 0, 0, 64);
    r32 = (uint32_t *) start;
    if (r32 + index < (uint32_t *) rd.d_start) return r32[index];
    if (r32 + index >= (uint32_t *) rd.d_top) return r32[index];
    index -= ((uint32_t *) rd.d_start) - r32;
    r8 = (uint8_t *) rd.d_start;
    r8 += ((index & 0xffffffff0)*4);
    r8 += (index & 0xf);
    r8 += 48;
    rv = 0;
    for (i = 0; i < 4; i++) {
        rv <= 8;
        rv |= *r8;
        r8 -= 16;
    }
    return rv;
}

static
inline
uint32_t gf_w32_matrix (gf_t *gf, uint32_t b)
{
    return gf_bitmatrix_inverse(b, 32, ((gf_internal_t *) (gf->scratch))->prim_poly);
}

/* JSP: GF_MULT_SHIFT: The world's dumbest multiplication algorithm. I only
   include it for completeness. It does have the feature that it requires no
   extra memory.
*/

static
inline
gf_val_32_t
gf_w32_cfm_gk_multiply (gf_t *gf, gf_val_32_t a32, gf_val_32_t b32)
{
    gf_val_32_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL
    __m128i a, b;
    __m128i result;
    __m128i w;
    __m128i g, q;
```



*src/gf\_w32.c lines 421 to 480*

```
gf_internal_t * h = gf->scratch;
uint64_t      g_star, q_plus;

q_plus = *(uint64_t *) h->private;
g_star = *((uint64_t *) h->private + 1);

a = _mm_insert_epi32 (_mm_setzero_si128(), a32, 0);
b = _mm_insert_epi32 (a, b32, 0);
g = _mm_insert_epi64 (a, g_star, 0);
q = _mm_insert_epi64 (a, q_plus, 0);

result = _mm_clmulepi64_si128 (a, b, 0);
w = _mm_clmulepi64_si128 (q, _mm_srli_si128 (result, 4), 0);
w = _mm_clmulepi64_si128 (g, _mm_srli_si128 (w, 4), 0);
result = _mm_xor_si128 (result, w);

/* Extracts 32 bit value from result. */
rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));
#endif
return rv;
}

#if defined(INTEL_SSE4_PCLMUL)

static
void
gf_w32_cfmkgk_multiply_region_from_single(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    int i;
    uint32_t *s32;
    uint32_t *d32;

    __m128i      a, b;
    __m128i      result;
    __m128i      w;
    __m128i      g, q;
    gf_internal_t * h = gf->scratch;
    uint64_t      g_star, q_plus;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    q_plus = *(uint64_t *) h->private;
    g_star = *((uint64_t *) h->private + 1);

    g = _mm_insert_epi64 (a, g_star, 0);
    q = _mm_insert_epi64 (a, q_plus, 0);
    a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);
    s32 = (uint32_t *) src;
    d32 = (uint32_t *) dest;

    if (xor) {
        for (i = 0; i < bytes/sizeof(uint32_t); i++) {
            b = _mm_insert_epi32 (a, s32[i], 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (q, _mm_srli_si128 (result, 4), 0);
            w = _mm_clmulepi64_si128 (g, _mm_srli_si128 (w, 4), 0);
            result = _mm_xor_si128 (result, w);
            d32[i] ^= ((gf_val_32_t)_mm_extract_epi32(result, 0));
        }
    }
}
```

*src/gf\_w32.c lines 481 to 540*

```
    }
} else {
    for (i = 0; i < bytes/sizeof(uint32_t); i++) {
        b = _mm_insert_epi32 (a, s32[i], 0);
        result = _mm_clmulepi64_si128 (a, b, 0);
        w = _mm_clmulepi64_si128 (q, _mm_srli_si128 (result, 4), 0);
        w = _mm_clmulepi64_si128 (g, _mm_srli_si128 (w, 4), 0);
        result = _mm_xor_si128 (result, w);
        d32[i] = ((gf_val_32_t)_mm_extract_epi32(result, 0));
    }
}
#endif

static
inline
gf_val_32_t
gf_w32_clm_multiply_2 (gf_t *gf, gf_val_32_t a32, gf_val_32_t b32)
{
    gf_val_32_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL

    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi32 (_mm_setzero_si128(), a32, 0);
    b = _mm_insert_epi32 (a, b32, 0);

    prim_poly = _mm_set_epi32(0, 0, 1, (uint32_t)(h->prim_poly & 0xffffffffULL));

    /* Do the initial multiply */

    result = _mm_clmulepi64_si128 (a, b, 0);

    /* Ben: Do prim_poly reduction twice. We are guaranteed that we will only
       have to do the reduction at most twice, because (w-2)/z == 2. Where
       z is equal to the number of zeros after the leading 1

       _mm_clmulepi64_si128 is the carryless multiply operation. Here
       _mm_srli_si128 shifts the result to the right by 4 bytes. This allows
       us to multiply the prim_poly by the leading bits of the result. We
       then xor the result of that operation back with the result.*/

    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
    result = _mm_xor_si128 (result, w);

    /* Extracts 32 bit value from result. */
    rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));
#endif
    return rv;
}
```



*src/gf\_w32.c lines 541 to 600*

```
static
inline
gf_val_32_t
gf_w32_clm_multiply_3 (gf_t *gf, gf_val_32_t a32, gf_val_32_t b32)
{
    gf_val_32_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL

    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi32 (_mm_setzero_si128(), a32, 0);
    b = _mm_insert_epi32 (a, b32, 0);

    prim_poly = _mm_set_epi32(0, 0, 1, (uint32_t)(h->prim_poly & 0xffffffffULL));

    /* Do the initial multiply */
    result = _mm_clmulepi64_si128 (a, b, 0);

    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
    result = _mm_xor_si128 (result, w);

    /* Extracts 32 bit value from result. */
    rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));
#endif
    return rv;
}
```

```
static
inline
gf_val_32_t
gf_w32_clm_multiply_4 (gf_t *gf, gf_val_32_t a32, gf_val_32_t b32)
{
    gf_val_32_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL

    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi32 (_mm_setzero_si128(), a32, 0);
    b = _mm_insert_epi32 (a, b32, 0);

    prim_poly = _mm_set_epi32(0, 0, 1, (uint32_t)(h->prim_poly & 0xffffffffULL));
```

*src/gf\_w32.c lines 601 to 660*

```
/* Do the initial multiply */

result = _mm_clmulepi64_si128 (a, b, 0);

w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
result = _mm_xor_si128 (result, w);
w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
result = _mm_xor_si128 (result, w);
w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
result = _mm_xor_si128 (result, w);
w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 4), 0);
result = _mm_xor_si128 (result, w);

/* Extracts 32 bit value from result. */

rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));
#endif
return rv;
}
```

```
static
inline
uint32_t
gf_w32_shift_multiply (gf_t *gf, uint32_t a32, uint32_t b32)
{
    uint64_t product, i, pp, a, b, one;
    gf_internal_t *h;

    a = a32;
    b = b32;
    h = (gf_internal_t *) gf->scratch;
    one = 1;
    pp = h->prim_poly | (one << 32);

    product = 0;

    for (i = 0; i < GF_FIELD_WIDTH; i++) {
        if (a & (one << i)) product ^= (b << i);
    }
    for (i = (GF_FIELD_WIDTH*2-2); i >= GF_FIELD_WIDTH; i--) {
        if (product & (one << i)) product ^= (pp << (i-GF_FIELD_WIDTH));
    }
    return product;
}
```

```
static
int gf_w32_cfmkgk_init(gf_t *gf)
{
    gf->inverse.w32 = gf_w32_euclid;
    gf->multiply_region.w32 = gf_w32_multiply_region_from_single;

#ifdef INTEL_SSE4_PCLMUL
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    gf->multiply.w32 = gf_w32_cfmkgk_multiply;
    gf->multiply_region.w32 = gf_w32_cfmkgk_multiply_region_from_single;

    uint64_t *q_plus = (uint64_t *) h->private;
```



*src/gf\_w32.c lines 661 to 720*

```
uint64_t *g_star = (uint64_t *) h->private + 1;

uint64_t tmp = h->prim_poly << 32;
*q_plus = 1ULL << 32;

int i;
for(i = 63; i >= 32; i--)
    if((1ULL << i) & tmp)
    {
        *q_plus |= 1ULL << (i-32);
        tmp ^= h->prim_poly << (i-32);
    }

*g_star = h->prim_poly & ((1ULL << 32) - 1);

    return 1;
#endif

    return 0;
}

static
int gf_w32_cfm_init(gf_t *gf)
{
    gf->inverse.w32 = gf_w32_euclid;
    gf->multiply_region.w32 = gf_w32_multiply_region_from_single;

    /*Ben: We also check to see if the prim poly will work for pclmul */
    /*Ben: Check to see how many reduction steps it will take*/

#ifdef INTEL_SSE4_PCLMUL
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    if ((0xfffe0000 & h->prim_poly) == 0){
        gf->multiply.w32 = gf_w32_clm_multiply_2;
        gf->multiply_region.w32 = gf_w32_clm_multiply_region_from_single_2;
    }else if ((0xffc00000 & h->prim_poly) == 0){
        gf->multiply.w32 = gf_w32_clm_multiply_3;
        gf->multiply_region.w32 = gf_w32_clm_multiply_region_from_single_3;
    }else if ((0xfe000000 & h->prim_poly) == 0){
        gf->multiply.w32 = gf_w32_clm_multiply_4;
        gf->multiply_region.w32 = gf_w32_clm_multiply_region_from_single_4;
    } else {
        return 0;
    }
    return 1;
#endif

    return 0;
}

static
int gf_w32_shift_init(gf_t *gf)
{
    gf->inverse.w32 = gf_w32_euclid;
    gf->multiply_region.w32 = gf_w32_multiply_region_from_single;
    gf->multiply.w32 = gf_w32_shift_multiply;
    return 1;
}
```

*src/gf\_w32.c lines 721 to 780*

```
}

static
void
gf_w32_group_set_shift_tables(uint32_t *shift, uint32_t val, gf_internal_t *h)
{
    int i;
    uint32_t j;

    shift[0] = 0;

    for (i = 1; i < (1 << h->arg1); i <= 1) {
        for (j = 0; j < i; j++) shift[i|j] = shift[j]^val;
        if (val & GF_FIRST_BIT) {
            val <<= 1;
            val ^= h->prim_poly;
        } else {
            val <<= 1;
        }
    }
}

static
void gf_w32_group_s_equals_r_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int leftover, rs;
    uint32_t p, l, ind, a32;
    int bits_left;
    int g_s;
    gf_region_data rd;
    uint32_t *s32, *d32, *top;
    struct gf_w32_group_data *gd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gd = (struct gf_w32_group_data *) h->private;
    g_s = h->arg1;
    gf_w32_group_set_shift_tables(gd->shift, val, h);

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);
    gf_do_initial_region_alignment(&rd);

    s32 = (uint32_t *) rd.s_start;
    d32 = (uint32_t *) rd.d_start;
    top = (uint32_t *) rd.d_top;

    leftover = 32 % g_s;
    if (leftover == 0) leftover = g_s;

    while (d32 < top) {
        rs = 32 - leftover;
        a32 = *s32;
        ind = a32 >> rs;
        a32 <<= leftover;
        p = gd->shift[ind];

        bits_left = rs;
        rs = 32 - g_s;
    }
}
```



*src/gf\_w32.c lines 781 to 840*

```
    while (bits_left > 0) {
        bits_left -= g_s;
        ind = a32 >> rs;
        a32 <<= g_s;
        l = p >> rs;
        p = (gd->shift[ind] ^ gd->reduce[l] ^ (p << g_s));
    }
    if (xor) p ^= *d32;
    *d32 = p;
    d32++;
    s32++;
}
gf_do_final_region_alignment(&rd);
}

static
void gf_w32_group_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint32_t *s32, *d32, *top;
    int i;
    int leftover;
    uint64_t p, l, r;
    uint32_t a32, ind;
    int g_s, g_r;
    struct gf_w32_group_data *gd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    g_s = h->arg1;
    g_r = h->arg2;
    gd = (struct gf_w32_group_data *) h->private;
    gf_w32_group_set_shift_tables(gd->shift, val, h);

    leftover = GF_FIELD_WIDTH % g_s;
    if (leftover == 0) leftover = g_s;

    gd = (struct gf_w32_group_data *) h->private;
    gf_w32_group_set_shift_tables(gd->shift, val, h);

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);
    gf_do_initial_region_alignment(&rd);

    s32 = (uint32_t *) rd.s_start;
    d32 = (uint32_t *) rd.d_start;
    top = (uint32_t *) rd.d_top;

    while (d32 < top) {
        a32 = *s32;
        ind = a32 >> (GF_FIELD_WIDTH - leftover);
        p = gd->shift[ind];
        p <<= g_s;
        a32 <<= leftover;

        i = (GF_FIELD_WIDTH - leftover);
        while (i > g_s) {
            ind = a32 >> (GF_FIELD_WIDTH - g_s);
```

*src/gf\_w32.c lines 841 to 900*

```
    p ^= gd->shift[ind];
    a32 <<= g_s;
    p <<= g_s;
    i -= g_s;
}

ind = a32 >> (GF_FIELD_WIDTH-g_s);
p ^= gd->shift[ind];

for (i = gd->tshift ; i >= 0; i -= g_r) {
    l = p & (gd->rmask << i);
    r = gd->reduce[l >> (i+32)];
    r <<= (i);
    p ^= r;
}

if (xor) p ^= *d32;
*d32 = p;
d32++;
s32++;
}
gf_do_final_region_alignment(&rd);
}

static
inline
gf_val_32_t
gf_w32_group_s_equals_r_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    int leftover, rs;
    uint32_t p, l, ind, a32;
    int bits_left;
    int g_s;

    struct gf_w32_group_data *gd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    g_s = h->arg1;

    gd = (struct gf_w32_group_data *) h->private;
    gf_w32_group_set_shift_tables(gd->shift, b, h);

    leftover = 32 % g_s;
    if (leftover == 0) leftover = g_s;

    rs = 32 - leftover;
    a32 = a;
    ind = a32 >> rs;
    a32 <<= leftover;
    p = gd->shift[ind];

    bits_left = rs;
    rs = 32 - g_s;

    while (bits_left > 0) {
        bits_left -= g_s;
        ind = a32 >> rs;
        a32 <<= g_s;
        l = p >> rs;
        p = (gd->shift[ind] ^ gd->reduce[l] ^ (p << g_s));
    }
}
```



[illegible]

```
static
inline
gf_val_32_t
gf_w32_group_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    int i;
    int leftover;
    uint64_t p, l, r;
    uint32_t a32, ind;
    int g_s, g_r;
```

*src/gf\_w32.c lines 961 to 1020*

```
    struct gf_w32_group_data *gd;

    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    g_s = h->arg1;
    g_r = h->arg2;
    gd = (struct gf_w32_group_data *) h->private;
    gf_w32_group_set_shift_tables(gd->shift, b, h);

    leftover = GF_FIELD_WIDTH % g_s;
    if (leftover == 0) leftover = g_s;

    a32 = a;
    ind = a32 >> (GF_FIELD_WIDTH - leftover);
    p = gd->shift[ind];
    p <<= g_s;
    a32 <<= leftover;

    i = (GF_FIELD_WIDTH - leftover);
    while (i > g_s) {
        ind = a32 >> (GF_FIELD_WIDTH - g_s);
        p ^= gd->shift[ind];
        a32 <<= g_s;
        p <<= g_s;
        i -= g_s;
    }

    ind = a32 >> (GF_FIELD_WIDTH - g_s);
    p ^= gd->shift[ind];

    for (i = gd->tshift ; i >= 0; i -= g_r) {
        l = p & (gd->rmask << i);
        r = gd->reduce[l >> (i+32)];
        r <<= (i);
        p ^= r;
    }
    return p;
}

static
inline
gf_val_32_t
gf_w32_bytwo_b_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint32_t prod, pp, bmask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;
    bmask = 0x80000000;

    while (1) {
        if (a & 1) prod ^= b;
        a >>= 1;
        if (a == 0) return prod;
        if (b & bmask) {
            b = ((b << 1) ^ pp);
        } else {
            b <<= 1;
        }
    }
}
```



*src/gf\_w32.c lines 1021 to 1080*

```
    }  
  }  
}  
  
static  
inline  
gf_val_32_t  
gf_w32_bytwo_p_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)  
{  
    uint32_t prod, pp, pmask, amask;  
    gf_internal_t *h;  
  
    h = (gf_internal_t *) gf->scratch;  
    pp = h->prim_poly;
```

```
  
    prod = 0;  
    pmask = 0x80000000;  
    amask = 0x80000000;  
  
    while (amask != 0) {  
        if (prod & pmask) {  
            prod = ((prod << 1) ^ pp);  
        } else {  
            prod <<= 1;  
        }  
        if (a & amask) prod ^= b;  
        amask >>= 1;  
    }  
    return prod;  
}
```

```
static  
void  
gf_w32_bytwo_p_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)  
{  
    uint64_t *s64, *d64, t1, t2, ta, prod, amask;  
    gf_region_data rd;  
    struct gf_w32_bytwo_data *btd;  
  
    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }  
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }  
  
    btd = (struct gf_w32_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;  
  
    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);  
    gf_do_initial_region_alignment(&rd);  
  
    s64 = (uint64_t *) rd.s_start;  
    d64 = (uint64_t *) rd.d_start;  
  
    if (xor) {  
        while (s64 < (uint64_t *) rd.s_top) {  
            prod = 0;  
            amask = 0x80000000;  
            ta = *s64;  
            while (amask != 0) {  
                AB2(btd->prim_poly, btd->mask1, btd->mask2, prod, t1, t2);  
                if (val & amask) prod ^= ta;  
                amask >>= 1;  
            }  
            *d64 = prod;  
            s64++;  
            d64++;  
        }  
    }  
}
```

*src/gf\_w32.c lines 1081 to 1140*

```
    }
    *d64 ^= prod;
    d64++;
    s64++;
}
} else {
while (s64 < (uint64_t *) rd.s_top) {
    prod = 0;
    amask = 0x80000000;
    ta = *s64;
    while (amask != 0) {
        AB2(btd->prim_poly, btd->mask1, btd->mask2, prod, t1, t2);
        if (val & amask) prod ^= ta;
        amask >>= 1;
    }
    *d64 = prod;
    d64++;
    s64++;
}
}
gf_do_final_region_alignment(&rd);
}

#define BYTWO_P_ONESTEP {\
    SSE_AB2(pp, m1, m2, prod, t1, t2); \
    t1 = _mm_and_si128(v, one); \
    t1 = _mm_sub_epi32(t1, one); \
    t1 = _mm_and_si128(t1, ta); \
    prod = _mm_xor_si128(prod, t1); \
    v = _mm_srli_epi64(v, 1); }

#ifdef INTEL_SSE2
static
void
gf_w32_bytwo_p_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int i;
    uint8_t *s8, *d8;
    uint32_t vrev;
    __m128i pp, m1, m2, ta, prod, t1, t2, tp, one, v;
    struct gf_w32_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    btd = (struct gf_w32_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    vrev = 0;
    for (i = 0; i < 32; i++) {
        vrev <<= 1;
        if (!(val & (1 << i))) vrev |= 1;
    }

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;
```



*src/gf\_w32.c lines 1141 to 1200*

```
pp = _mm_set1_epi32(btd->prim_poly&0xffffffff);
m1 = _mm_set1_epi32((btd->mask1)&0xffffffff);
m2 = _mm_set1_epi32((btd->mask2)&0xffffffff);
one = _mm_set1_epi32(1);

while (d8 < (uint8_t *) rd.d_top) {
    prod = _mm_setzero_si128();
    v = _mm_set1_epi32(vrev);
    ta = _mm_load_si128((__m128i *) s8);
    tp = (!xor) ? _mm_setzero_si128() : _mm_load_si128((__m128i *) d8);
    BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP;
    BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP;
    BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP;
    BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP;
    BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP;
    BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP;
    BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP;
    BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP; BYTWO_P_ONESTEP;
    _mm_store_si128((__m128i *) d8, _mm_xor_si128(prod, tp));
    d8 += 16;
    s8 += 16;
}
gf_do_final_region_alignment(&rd);
}
#endif

static
void
gf_w32_bytwo_b_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t *s64, *d64, t1, t2, ta, tb, prod;
    struct gf_w32_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 32);
    gf_do_initial_region_alignment(&rd);

    btd = (struct gf_w32_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;
    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;

    switch (val) {
    case 2:
        if (xor) {
            while (d64 < (uint64_t *) rd.d_top) {
                ta = *s64;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
                *d64 ^= ta;
                d64++;
                s64++;
            }
        } else {
            while (d64 < (uint64_t *) rd.d_top) {
                ta = *s64;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
                *d64 = ta;
                d64++;
            }
        }
    }
```

src/gf\_w32.c lines 1201 to 1260

```
        s64++;
    }
}
break;
case 3:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
            d64++;
            s64++;
        }
    }
}
break;
case 4:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= ta;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta;
            d64++;
            s64++;
        }
    }
}
break;
case 5:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
```



*src/gf\_w32.c lines 1261 to 1320*

```
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 = ta ^ prod;
        d64++;
        s64++;
    }
}
break;
default:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            prod = *d64;
            ta = *s64;
            tb = val;
            while (1) {
                if (tb & 1) prod ^= ta;
                tb >>= 1;
                if (tb == 0) break;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            }
            *d64 = prod;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            prod = 0;
            ta = *s64;
            tb = val;
            while (1) {
                if (tb & 1) prod ^= ta;
                tb >>= 1;
                if (tb == 0) break;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            }
            *d64 = prod;
            d64++;
            s64++;
        }
    }
}
break;
}
gf_do_final_region_alignment(&rd);
}

#ifdef INTEL_SSE2
static
void
gf_w32_bytwo_b_sse_region_2_noxor(gf_region_data *rd, struct gf_w32_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi32(btd->prim_poly&0xffffffff);
    m1 = _mm_set1_epi32((btd->mask1)&0xffffffff);
    m2 = _mm_set1_epi32((btd->mask2)&0xffffffff);
```

*src/gf\_w32.c lines 1321 to 1380*

```
while (d8 < (uint8_t *) rd->d_top) {
    va = _mm_load_si128 ((__m128i *) (s8));
    SSE_AB2(pp, m1, m2, va, t1, t2);
    _mm_store_si128((__m128i *)d8, va);
    d8 += 16;
    s8 += 16;
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w32_bytwo_b_sse_region_2_xor(gf_region_data *rd, struct gf_w32_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi32(btd->prim_poly&0xffffffff);
    m1 = _mm_set1_epi32((btd->mask1)&0xffffffff);
    m2 = _mm_set1_epi32((btd->mask2)&0xffffffff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        SSE_AB2(pp, m1, m2, va, t1, t2);
        vb = _mm_load_si128 ((__m128i *) (d8));
        vb = _mm_xor_si128(vb, va);
        _mm_store_si128((__m128i *)d8, vb);
        d8 += 16;
        s8 += 16;
    }
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w32_bytwo_b_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint32_t itb;
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va, vb;
    struct gf_w32_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    btd = (struct gf_w32_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

    if (val == 2) {
        if (xor) {
```



*src/gf\_w32.c lines 1381 to 1440*

```
    gf_w32_bytwo_b_sse_region_2_xor(&rd, btd);
} else {
    gf_w32_bytwo_b_sse_region_2_noxor(&rd, btd);
}
gf_do_final_region_alignment(&rd);
return;
}

s8 = (uint8_t *) rd.s_start;
d8 = (uint8_t *) rd.d_start;

pp = _mm_set1_epi32(btd->prim_poly&0xffffffff);
m1 = _mm_set1_epi32((btd->mask1)&0xffffffff);
m2 = _mm_set1_epi32((btd->mask2)&0xffffffff);

while (d8 < (uint8_t *) rd.d_top) {
    va = _mm_load_si128 ((__m128i *) (s8));
    vb = (!xor) ? _mm_setzero_si128() : _mm_load_si128 ((__m128i *) (d8));
    itb = va;
    while (1) {
        if (itb & 1) vb = _mm_xor_si128(vb, va);
        itb >>= 1;
        if (itb == 0) break;
        SSE_AB2(pp, m1, m2, va, t1, t2);
    }
    _mm_store_si128((__m128i *) d8, vb);
    d8 += 16;
    s8 += 16;
}

gf_do_final_region_alignment(&rd);
}
#endif

static
int gf_w32_bytwo_init(gf_t *gf)
{
    gf_internal_t *h;
    uint64_t ip, m1, m2;
    struct gf_w32_bytwo_data *btd;

    h = (gf_internal_t *) gf->scratch;
    btd = (struct gf_w32_bytwo_data *) (h->private);
    ip = h->prim_poly & 0xffffffff;
    m1 = 0xfffffffffe;
    m2 = 0x80000000;
    btd->prim_poly = 0;
    btd->mask1 = 0;
    btd->mask2 = 0;

    while (ip != 0) {
        btd->prim_poly |= ip;
        btd->mask1 |= m1;
        btd->mask2 |= m2;
        ip <<= GF_FIELD_WIDTH;
        m1 <<= GF_FIELD_WIDTH;
        m2 <<= GF_FIELD_WIDTH;
    }

    if (h->mult_type == GF_MULT_BYTWO_p) {
```

*src/gf\_w32.c lines 1441 to 1500*

```
gf->multiply.w32 = gf_w32_bytwo_p_multiply;
#ifdef INTEL_SSE2
    if (h->region_type & GF_REGION_NOSSE)
        gf->multiply_region.w32 = gf_w32_bytwo_p_nosse_multiply_region;
    else
        gf->multiply_region.w32 = gf_w32_bytwo_p_sse_multiply_region;
#else
    gf->multiply_region.w32 = gf_w32_bytwo_p_nosse_multiply_region;
    if(h->region_type & GF_REGION_SSE)
        return 0;
#endif
} else {
    gf->multiply.w32 = gf_w32_bytwo_b_multiply;
#ifdef INTEL_SSE2
    if (h->region_type & GF_REGION_NOSSE)
        gf->multiply_region.w32 = gf_w32_bytwo_b_nosse_multiply_region;
    else
        gf->multiply_region.w32 = gf_w32_bytwo_b_sse_multiply_region;
#else
    gf->multiply_region.w32 = gf_w32_bytwo_b_nosse_multiply_region;
    if(h->region_type & GF_REGION_SSE)
        return 0;
#endif
}

gf->inverse.w32 = gf_w32_euclid;
return 1;
}
```

```
static
inline
uint32_t
gf_w32_split_8_8_multiply (gf_t *gf, uint32_t a32, uint32_t b32)
{
    uint32_t product, i, j, mask, tb;
    gf_internal_t *h;
    struct gf_w32_split_8_8_data *d8;

    h = (gf_internal_t *) gf->scratch;
    d8 = (struct gf_w32_split_8_8_data *) h->private;
    product = 0;
    mask = 0xff;

    for (i = 0; i < 4; i++) {
        tb = b32;
        for (j = 0; j < 4; j++) {
            product ^= d8->tables[i+j][a32&mask][tb&mask];
            tb >>= 8;
        }
        a32 >>= 8;
    }
    return product;
}
```

```
static
inline
void
gf_w32_split_8_32_lazy_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    gf_internal_t *h;
```



*src/gf\_w32.c lines 1501 to 1560*

```
uint32_t *s32, *d32, *top, p, a, v;
struct gf_split_8_32_lazy_data *d8;
struct gf_w32_split_8_8_data *d88;
uint32_t *t[4];
int i, j, k, change;
uint32_t pp;
gf_region_data rd;

if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

h = (gf_internal_t *) gf->scratch;
if (h->arg1 == 32 || h->arg2 == 32 || h->mult_type == GF_MULT_DEFAULT) {
    d8 = (struct gf_split_8_32_lazy_data *) h->private;
    for (i = 0; i < 4; i++) t[i] = d8->tables[i];
    change = (val != d8->last_value);
    if (change) d8->last_value = val;
} else {
    d88 = (struct gf_w32_split_8_8_data *) h->private;
    for (i = 0; i < 4; i++) t[i] = d88->region_tables[i];
    change = (val != d88->last_value);
    if (change) d88->last_value = val;
}
pp = h->prim_poly;

gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);
gf_do_initial_region_alignment(&rd);

s32 = (uint32_t *) rd.s_start;
d32 = (uint32_t *) rd.d_start;
top = (uint32_t *) rd.d_top;

if (change) {
    v = val;
    for (i = 0; i < 4; i++) {
        t[i][0] = 0;
        for (j = 1; j < 256; j <= 1) {
            for (k = 0; k < j; k++) {
                t[i][k^j] = (v ^ t[i][k]);
            }
            v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
        }
    }
}

while (d32 < top) {
    p = (xor) ? *d32 : 0;
    a = *s32;
    i = 0;
    while (a != 0) {
        v = (a & 0xff);
        p ^= t[i][v];
        a >>= 8;
        i++;
    }
    *d32 = p;
    d32++;
    s32++;
}
gf_do_final_region_alignment(&rd);
```

*src/gf\_w32.c lines 1561 to 1620*

```
}

static
inline
void
gf_w32_split_16_32_lazy_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    gf_internal_t *h;
    uint32_t *s32, *d32, *top, p, a, v;
    struct gf_split_16_32_lazy_data *d16;
    uint32_t *t[2];
    int i, j, k, change;
    uint32_t pp;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;
    d16 = (struct gf_split_16_32_lazy_data *) h->private;
    for (i = 0; i < 2; i++) t[i] = d16->tables[i];
    change = (val != d16->last_value);
    if (change) d16->last_value = val;

    pp = h->prim_poly;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);
    gf_do_initial_region_alignment(&rd);

    s32 = (uint32_t *) rd.s_start;
    d32 = (uint32_t *) rd.d_start;
    top = (uint32_t *) rd.d_top;

    if (change) {
        v = val;
        for (i = 0; i < 2; i++) {
            t[i][0] = 0;
            for (j = 1; j < (1 << 16); j <= 1) {
                for (k = 0; k < j; k++) {
                    t[i][k^j] = (v ^ t[i][k]);
                }
                v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
            }
        }
    }

    while (d32 < top) {
        p = (xor) ? *d32 : 0;
        a = *s32;
        i = 0;
        while (a != 0 && i < 2) {
            v = (a & 0xffff);
            p ^= t[i][v];
            a >>= 16;
            i++;
        }
        *d32 = p;
        d32++;
        s32++;
    }
}
```



*src/gf\_w32.c lines 1621 to 1680*

```
gf_do_final_region_alignment(&rd);
}

static
void
gf_w32_split_2_32_lazy_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    gf_internal_t *h;
    struct gf_split_2_32_lazy_data *ld;
    int i;
    uint32_t pp, v, v2, s, *s32, *d32, *top;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);
    gf_do_initial_region_alignment(&rd);

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    ld = (struct gf_split_2_32_lazy_data *) h->private;

    if (ld->last_value != val) {
        v = val;
        for (i = 0; i < 16; i++) {
            v2 = (v << 1);
            if (v & GF_FIRST_BIT) v2 ^= pp;
            ld->tables[i][0] = 0;
            ld->tables[i][1] = v;
            ld->tables[i][2] = v2;
            ld->tables[i][3] = (v2 ^ v);
            v = (v2 << 1);
            if (v2 & GF_FIRST_BIT) v ^= pp;
        }
    }
    ld->last_value = val;

    s32 = (uint32_t *) rd.s_start;
    d32 = (uint32_t *) rd.d_start;
    top = (uint32_t *) rd.d_top;

    while (d32 != top) {
        v = (xor) ? *d32 : 0;
        s = *s32;
        i = 0;
        while (s != 0) {
            v ^= ld->tables[i][s&3];
            s >>= 2;
            i++;
        }
        *d32 = v;
        d32++;
        s32++;
    }
    gf_do_final_region_alignment(&rd);
}

#ifdef INTEL_SSSE3
```

*src/gf\_w32.c lines 1681 to 1740*

```
static
void
gf_w32_split_2_32_lazy_sse_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    gf_internal_t *h;
    int i, tindex;
    uint32_t pp, v, v2, *s32, *d32, *top;
    __m128i vi, si, pi, shuffler, tables[16], adder, xi, mask1, mask2;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 32);
    gf_do_initial_region_alignment(&rd);

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    s32 = (uint32_t *) rd.s_start;
    d32 = (uint32_t *) rd.d_start;
    top = (uint32_t *) rd.d_top;

    v = val;
    for (i = 0; i < 16; i++) {
        v2 = (v << 1);
        if (v & GF_FIRST_BIT) v2 ^= pp;
        tables[i] = _mm_set_epi32(v2 ^ v, v2, v, 0);
        v = (v2 << 1);
        if (v2 & GF_FIRST_BIT) v ^= pp;
    }

    shuffler = _mm_set_epi8(0xc, 0xc, 0xc, 0xc, 8, 8, 8, 8, 4, 4, 4, 4, 0, 0, 0, 0);
    adder = _mm_set_epi8(3, 2, 1, 0, 3, 2, 1, 0, 3, 2, 1, 0, 3, 2, 1, 0);
    mask1 = _mm_setl_epi8(0x3);
    mask2 = _mm_setl_epi8(0xc);

    while (d32 != top) {
        pi = (xor) ? _mm_load_si128((__m128i *) d32) : _mm_setzero_si128();
        vi = _mm_load_si128((__m128i *) s32);

        tindex = 0;
        for (i = 0; i < 4; i++) {
            si = _mm_shuffle_epi8(vi, shuffler);

            xi = _mm_and_si128(si, mask1);
            xi = _mm_slli_epi16(xi, 2);
            xi = _mm_xor_si128(xi, adder);
            pi = _mm_xor_si128(pi, _mm_shuffle_epi8(tables[tindex], xi));
            tindex++;

            xi = _mm_and_si128(si, mask2);
            xi = _mm_xor_si128(xi, adder);
            pi = _mm_xor_si128(pi, _mm_shuffle_epi8(tables[tindex], xi));
            si = _mm_srli_epi16(si, 2);
            tindex++;

            xi = _mm_and_si128(si, mask2);
            xi = _mm_xor_si128(xi, adder);
            pi = _mm_xor_si128(pi, _mm_shuffle_epi8(tables[tindex], xi));
```



*src/gf\_w32.c lines 1741 to 1800*

```
        si = _mm_srli_epi16(si, 2);
        tindex++;

        xi = _mm_and_si128(si, mask2);
        xi = _mm_xor_si128(xi, adder);
        pi = _mm_xor_si128(pi, _mm_shuffle_epi8(tables[tindex], xi));
        tindex++;

        vi = _mm_srli_epi32(vi, 8);
    }
    _mm_store_si128((__m128i *) d32, pi);
    d32 += 4;
    s32 += 4;
}

gf_do_final_region_alignment(&rd);

}
#endif

static
void
gf_w32_split_4_32_lazy_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    gf_internal_t *h;
    struct gf_split_4_32_lazy_data *ld;
    int i, j, k;
    uint32_t pp, v, s, *s32, *d32, *top;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    ld = (struct gf_split_4_32_lazy_data *) h->private;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);
    gf_do_initial_region_alignment(&rd);

    if (ld->last_value != val) {
        v = val;
        for (i = 0; i < 8; i++) {
            ld->tables[i][0] = 0;
            for (j = 1; j < 16; j <= 1) {
                for (k = 0; k < j; k++) {
                    ld->tables[i][k^j] = (v ^ ld->tables[i][k]);
                }
                v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
            }
        }
    }
    ld->last_value = val;

    s32 = (uint32_t *) rd.s_start;
    d32 = (uint32_t *) rd.d_start;
    top = (uint32_t *) rd.d_top;

    while (d32 != top) {
```

*src/gf\_w32.c lines 1801 to 1860*

```
    v = (xor) ? *d32 : 0;
    s = *s32;
    i = 0;
    while (s != 0) {
        v ^= ld->tables[i][s&0xf];
        s >>= 4;
        i++;
    }
    *d32 = v;
    d32++;
    s32++;
}
gf_do_final_region_alignment(&rd);
}

static
void
gf_w32_split_4_32_lazy_sse_altmap_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
#ifdef INTEL_SSSE3
    gf_internal_t *h;
    int i, j, k;
    uint32_t pp, v, *s32, *d32, *top;
    __m128i si, tables[8][4], p0, p1, p2, p3, mask1, v0, v1, v2, v3;
    struct gf_split_4_32_lazy_data *ld;
    uint8_t btable[16];
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 64);
    gf_do_initial_region_alignment(&rd);

    s32 = (uint32_t *) rd.s_start;
    d32 = (uint32_t *) rd.d_start;
    top = (uint32_t *) rd.d_top;

    ld = (struct gf_split_4_32_lazy_data *) h->private;

    v = val;
    for (i = 0; i < 8; i++) {
        ld->tables[i][0] = 0;
        for (j = 1; j < 16; j <= 1) {
            for (k = 0; k < j; k++) {
                ld->tables[i][k^j] = (v ^ ld->tables[i][k]);
            }
            v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
        }
        for (j = 0; j < 4; j++) {
            for (k = 0; k < 16; k++) {
                btable[k] = (uint8_t) ld->tables[i][k];
                ld->tables[i][k] >>= 8;
            }
            tables[i][j] = _mm_loadu_si128((__m128i *) btable);
        }
    }
}
```



*src/gf\_w32.c lines 1861 to 1920*

```
mask1 = _mm_set1_epi8(0xf);

if (xor) {
    while (d32 != top) {
        p0 = _mm_load_si128((__m128i *) d32);
        p1 = _mm_load_si128((__m128i *) (d32+4));
        p2 = _mm_load_si128((__m128i *) (d32+8));
        p3 = _mm_load_si128((__m128i *) (d32+12));

        v0 = _mm_load_si128((__m128i *) s32); s32 += 4;
        v1 = _mm_load_si128((__m128i *) s32); s32 += 4;
        v2 = _mm_load_si128((__m128i *) s32); s32 += 4;
        v3 = _mm_load_si128((__m128i *) s32); s32 += 4;

        si = _mm_and_si128(v0, mask1);
        p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[0][0], si));
        p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[0][1], si));
        p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[0][2], si));
        p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[0][3], si));

        v0 = _mm_srli_epi32(v0, 4);
        si = _mm_and_si128(v0, mask1);
        p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[1][0], si));
        p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[1][1], si));
        p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[1][2], si));
        p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[1][3], si));

        si = _mm_and_si128(v1, mask1);
        p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[2][0], si));
        p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[2][1], si));
        p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[2][2], si));
        p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[2][3], si));

        v1 = _mm_srli_epi32(v1, 4);
        si = _mm_and_si128(v1, mask1);
        p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[3][0], si));
        p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[3][1], si));
        p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[3][2], si));
        p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[3][3], si));

        si = _mm_and_si128(v2, mask1);
        p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[4][0], si));
        p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[4][1], si));
        p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[4][2], si));
        p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[4][3], si));

        v2 = _mm_srli_epi32(v2, 4);
        si = _mm_and_si128(v2, mask1);
        p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[5][0], si));
        p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[5][1], si));
        p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[5][2], si));
        p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[5][3], si));

        si = _mm_and_si128(v3, mask1);
        p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[6][0], si));
        p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[6][1], si));
        p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[6][2], si));
        p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[6][3], si));
    }
}
```



*src/gf\_w32.c lines 1921 to 1980*

```
v3 = _mm_srli_epi32(v3, 4);
si = _mm_and_si128(v3, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[7][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[7][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[7][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[7][3], si));

_mm_store_si128((__m128i *) d32, p0);
_mm_store_si128((__m128i *) (d32+4), p1);
_mm_store_si128((__m128i *) (d32+8), p2);
_mm_store_si128((__m128i *) (d32+12), p3);
d32 += 16;
}
} else {
while (d32 != top) {

v0 = _mm_load_si128((__m128i *) s32); s32 += 4;
v1 = _mm_load_si128((__m128i *) s32); s32 += 4;
v2 = _mm_load_si128((__m128i *) s32); s32 += 4;
v3 = _mm_load_si128((__m128i *) s32); s32 += 4;

si = _mm_and_si128(v0, mask1);
p0 = _mm_shuffle_epi8(tables[0][0], si);
p1 = _mm_shuffle_epi8(tables[0][1], si);
p2 = _mm_shuffle_epi8(tables[0][2], si);
p3 = _mm_shuffle_epi8(tables[0][3], si);

v0 = _mm_srli_epi32(v0, 4);
si = _mm_and_si128(v0, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[1][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[1][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[1][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[1][3], si));

si = _mm_and_si128(v1, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[2][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[2][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[2][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[2][3], si));

v1 = _mm_srli_epi32(v1, 4);
si = _mm_and_si128(v1, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[3][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[3][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[3][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[3][3], si));

si = _mm_and_si128(v2, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[4][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[4][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[4][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[4][3], si));

v2 = _mm_srli_epi32(v2, 4);
si = _mm_and_si128(v2, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[5][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[5][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[5][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[5][3], si));
```



*src/gf\_w32.c lines 1981 to 2040*

```
    si = _mm_and_si128(v3, mask1);
    p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[6][0], si));
    p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[6][1], si));
    p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[6][2], si));
    p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[6][3], si));

    v3 = _mm_srli_epi32(v3, 4);
    si = _mm_and_si128(v3, mask1);
    p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[7][0], si));
    p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[7][1], si));
    p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[7][2], si));
    p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[7][3], si));

    _mm_store_si128((__m128i *) d32, p0);
    _mm_store_si128((__m128i *) (d32+4), p1);
    _mm_store_si128((__m128i *) (d32+8), p2);
    _mm_store_si128((__m128i *) (d32+12), p3);
    d32 += 16;
}
}

gf_do_final_region_alignment(&rd);

#endif
}

static
void
gf_w32_split_4_32_lazy_sse_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
#ifdef INTEL_SSSE3
    gf_internal_t *h;
    int i, j, k;
    uint32_t pp, v, *s32, *d32, *top, tmp_table[16];
    __m128i si, tables[8][4], p0, p1, p2, p3, mask1, v0, v1, v2, v3, mask8;
    __m128i tv1, tv2, tv3, tv0;
    uint8_t btable[16];
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 64);
    gf_do_initial_region_alignment(&rd);

    s32 = (uint32_t *) rd.s_start;
    d32 = (uint32_t *) rd.d_start;
    top = (uint32_t *) rd.d_top;

    v = val;
    for (i = 0; i < 8; i++) {
        tmp_table[0] = 0;
        for (j = 1; j < 16; j <= 1) {
            for (k = 0; k < j; k++) {
                tmp_table[k^j] = (v ^ tmp_table[k]);
            }
        }
    }
#endif
}
```

*src/gf\_w32.c lines 2041 to 2100*

```
    v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
}
for (j = 0; j < 4; j++) {
    for (k = 0; k < 16; k++) {
        btable[k] = (uint8_t) tmp_table[k];
        tmp_table[k] >>= 8;
    }
    tables[i][j] = _mm_loadu_si128((__m128i *) btable);
}
}

mask1 = _mm_set1_epi8(0xf);
mask8 = _mm_set1_epi16(0xff);

if (xor) {
    while (d32 != top) {
        v0 = _mm_load_si128((__m128i *) s32); s32 += 4;
        v1 = _mm_load_si128((__m128i *) s32); s32 += 4;
        v2 = _mm_load_si128((__m128i *) s32); s32 += 4;
        v3 = _mm_load_si128((__m128i *) s32); s32 += 4;

        p0 = _mm_srli_epi16(v0, 8);
        p1 = _mm_srli_epi16(v1, 8);
        p2 = _mm_srli_epi16(v2, 8);
        p3 = _mm_srli_epi16(v3, 8);

        tv0 = _mm_and_si128(v0, mask8);
        tv1 = _mm_and_si128(v1, mask8);
        tv2 = _mm_and_si128(v2, mask8);
        tv3 = _mm_and_si128(v3, mask8);

        v0 = _mm_packus_epi16(p1, p0);
        v1 = _mm_packus_epi16(tv1, tv0);
        v2 = _mm_packus_epi16(p3, p2);
        v3 = _mm_packus_epi16(tv3, tv2);

        p0 = _mm_srli_epi16(v0, 8);
        p1 = _mm_srli_epi16(v1, 8);
        p2 = _mm_srli_epi16(v2, 8);
        p3 = _mm_srli_epi16(v3, 8);

        tv0 = _mm_and_si128(v0, mask8);
        tv1 = _mm_and_si128(v1, mask8);
        tv2 = _mm_and_si128(v2, mask8);
        tv3 = _mm_and_si128(v3, mask8);

        v0 = _mm_packus_epi16(p2, p0);
        v1 = _mm_packus_epi16(p3, p1);
        v2 = _mm_packus_epi16(tv2, tv0);
        v3 = _mm_packus_epi16(tv3, tv1);

        si = _mm_and_si128(v0, mask1);
        p0 = _mm_shuffle_epi8(tables[6][0], si);
        p1 = _mm_shuffle_epi8(tables[6][1], si);
        p2 = _mm_shuffle_epi8(tables[6][2], si);
        p3 = _mm_shuffle_epi8(tables[6][3], si);

        v0 = _mm_srli_epi32(v0, 4);
        si = _mm_and_si128(v0, mask1);
        p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[7][0], si));
    }
}
```



*src/gf\_w32.c lines 2101 to 2160*

```
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[7][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[7][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[7][3], si));

si = _mm_and_si128(v1, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[4][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[4][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[4][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[4][3], si));

v1 = _mm_srli_epi32(v1, 4);
si = _mm_and_si128(v1, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[5][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[5][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[5][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[5][3], si));

si = _mm_and_si128(v2, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[2][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[2][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[2][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[2][3], si));

v2 = _mm_srli_epi32(v2, 4);
si = _mm_and_si128(v2, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[3][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[3][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[3][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[3][3], si));

si = _mm_and_si128(v3, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[0][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[0][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[0][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[0][3], si));

v3 = _mm_srli_epi32(v3, 4);
si = _mm_and_si128(v3, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[1][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[1][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[1][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[1][3], si));

tv0 = _mm_unpackhi_epi8(p1, p3);
tv1 = _mm_unpackhi_epi8(p0, p2);
tv2 = _mm_unpacklo_epi8(p1, p3);
tv3 = _mm_unpacklo_epi8(p0, p2);

p0 = _mm_unpackhi_epi8(tv1, tv0);
p1 = _mm_unpacklo_epi8(tv1, tv0);
p2 = _mm_unpackhi_epi8(tv3, tv2);
p3 = _mm_unpacklo_epi8(tv3, tv2);

v0 = _mm_load_si128((__m128i *) d32);
v1 = _mm_load_si128((__m128i *) (d32+4));
v2 = _mm_load_si128((__m128i *) (d32+8));
v3 = _mm_load_si128((__m128i *) (d32+12));

p0 = _mm_xor_si128(p0, v0);
p1 = _mm_xor_si128(p1, v1);
```



*src/gf\_w32.c lines 2161 to 2220*

```
    p2 = _mm_xor_si128(p2, v2);
    p3 = _mm_xor_si128(p3, v3);

    _mm_store_si128((__m128i *) d32, p0);
    _mm_store_si128((__m128i *) (d32+4), p1);
    _mm_store_si128((__m128i *) (d32+8), p2);
    _mm_store_si128((__m128i *) (d32+12), p3);
    d32 += 16;
}
} else {
    while (d32 != top) {
        v0 = _mm_load_si128((__m128i *) s32); s32 += 4;
        v1 = _mm_load_si128((__m128i *) s32); s32 += 4;
        v2 = _mm_load_si128((__m128i *) s32); s32 += 4;
        v3 = _mm_load_si128((__m128i *) s32); s32 += 4;

        p0 = _mm_srli_epi16(v0, 8);
        p1 = _mm_srli_epi16(v1, 8);
        p2 = _mm_srli_epi16(v2, 8);
        p3 = _mm_srli_epi16(v3, 8);

        tv0 = _mm_and_si128(v0, mask8);
        tv1 = _mm_and_si128(v1, mask8);
        tv2 = _mm_and_si128(v2, mask8);
        tv3 = _mm_and_si128(v3, mask8);

        v0 = _mm_packus_epi16(p1, p0);
        v1 = _mm_packus_epi16(tv1, tv0);
        v2 = _mm_packus_epi16(p3, p2);
        v3 = _mm_packus_epi16(tv3, tv2);

        p0 = _mm_srli_epi16(v0, 8);
        p1 = _mm_srli_epi16(v1, 8);
        p2 = _mm_srli_epi16(v2, 8);
        p3 = _mm_srli_epi16(v3, 8);

        tv0 = _mm_and_si128(v0, mask8);
        tv1 = _mm_and_si128(v1, mask8);
        tv2 = _mm_and_si128(v2, mask8);
        tv3 = _mm_and_si128(v3, mask8);

        v0 = _mm_packus_epi16(p2, p0);
        v1 = _mm_packus_epi16(p3, p1);
        v2 = _mm_packus_epi16(tv2, tv0);
        v3 = _mm_packus_epi16(tv3, tv1);

        si = _mm_and_si128(v0, mask1);
        p0 = _mm_shuffle_epi8(tables[6][0], si);
        p1 = _mm_shuffle_epi8(tables[6][1], si);
        p2 = _mm_shuffle_epi8(tables[6][2], si);
        p3 = _mm_shuffle_epi8(tables[6][3], si);

        v0 = _mm_srli_epi32(v0, 4);
        si = _mm_and_si128(v0, mask1);
        p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[7][0], si));
        p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[7][1], si));
        p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[7][2], si));
        p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[7][3], si));

        si = _mm_and_si128(v1, mask1);
```



*src/gf\_w32.c lines 2221 to 2280*

```
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[4][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[4][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[4][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[4][3], si));
```

```
v1 = _mm_srli_epi32(v1, 4);
si = _mm_and_si128(v1, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[5][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[5][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[5][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[5][3], si));
```

```
si = _mm_and_si128(v2, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[2][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[2][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[2][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[2][3], si));
```

```
v2 = _mm_srli_epi32(v2, 4);
si = _mm_and_si128(v2, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[3][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[3][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[3][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[3][3], si));
```

```
si = _mm_and_si128(v3, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[0][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[0][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[0][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[0][3], si));
```

```
v3 = _mm_srli_epi32(v3, 4);
si = _mm_and_si128(v3, mask1);
p0 = _mm_xor_si128(p0, _mm_shuffle_epi8(tables[1][0], si));
p1 = _mm_xor_si128(p1, _mm_shuffle_epi8(tables[1][1], si));
p2 = _mm_xor_si128(p2, _mm_shuffle_epi8(tables[1][2], si));
p3 = _mm_xor_si128(p3, _mm_shuffle_epi8(tables[1][3], si));
```

```
tv0 = _mm_unpackhi_epi8(p1, p3);
tv1 = _mm_unpackhi_epi8(p0, p2);
tv2 = _mm_unpacklo_epi8(p1, p3);
tv3 = _mm_unpacklo_epi8(p0, p2);
```

```
p0 = _mm_unpackhi_epi8(tv1, tv0);
p1 = _mm_unpacklo_epi8(tv1, tv0);
p2 = _mm_unpackhi_epi8(tv3, tv2);
p3 = _mm_unpacklo_epi8(tv3, tv2);
```

```
_mm_store_si128((__m128i *) d32, p0);
_mm_store_si128((__m128i *) (d32+4), p1);
_mm_store_si128((__m128i *) (d32+8), p2);
_mm_store_si128((__m128i *) (d32+12), p3);
d32 += 16;
```

```
    }
}
gf_do_final_region_alignment(&rd);
```

```
#endif
}
```

*src/gf\_w32.c lines 2281 to 2340*

```
static
int gf_w32_split_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_split_2_32_lazy_data *ld2;
    struct gf_split_4_32_lazy_data *ld4;
    struct gf_w32_split_8_8_data *d8;
    struct gf_split_8_32_lazy_data *d32;
    struct gf_split_16_32_lazy_data *d16;
    uint32_t p, basep;
    int i, j, exp, ispclmul, issse3;

#ifdef INTEL_SSE4_PCLMUL
    ispclmul = 1;
#else
    ispclmul = 0;
#endif

#ifdef INTEL_SSSE3
    issse3 = 1;
#else
    issse3 = 0;
#endif

    h = (gf_internal_t *) gf->scratch;

    /* Defaults */

    gf->inverse.w32 = gf_w32_euclid;

    /* JSP: First handle single multiplication:
       If args == 8, then we're doing split 8 8.
       Otherwise, if PCLMUL, we use that.
       Otherwise, we use bytwo_p.
    */

    if (h->arg1 == 8 && h->arg2 == 8) {
        gf->multiply.w32 = gf_w32_split_8_8_multiply;
    } else if (ispclmul) {
        if ((0xfffe0000 & h->prim_poly) == 0) {
            gf->multiply.w32 = gf_w32_clm_multiply_2;
        } else if ((0xffc00000 & h->prim_poly) == 0) {
            gf->multiply.w32 = gf_w32_clm_multiply_3;
        } else if ((0xfe000000 & h->prim_poly) == 0) {
            gf->multiply.w32 = gf_w32_clm_multiply_4;
        }
    } else {
        gf->multiply.w32 = gf_w32_bytwo_p_multiply;
    }

    /* Easy cases: 16/32 and 2/32 */

    if ((h->arg1 == 16 && h->arg2 == 32) || (h->arg1 == 32 && h->arg2 == 16)) {
        d16 = (struct gf_split_16_32_lazy_data *) h->private;
        d16->last_value = 0;
        gf->multiply_region.w32 = gf_w32_split_16_32_lazy_multiply_region;
        return 1;
    }

    if ((h->arg1 == 2 && h->arg2 == 32) || (h->arg1 == 32 && h->arg2 == 2)) {
```



*src/gf\_w32.c lines 2341 to 2400*

```
ld2 = (struct gf_split_2_32_lazy_data *) h->private;
ld2->last_value = 0;
#ifdef INTEL_SSSE3
    if (!(h->region_type & GF_REGION_NOSSE))
        gf->multiply_region.w32 = gf_w32_split_2_32_lazy_sse_multiply_region;
    else
        gf->multiply_region.w32 = gf_w32_split_2_32_lazy_multiply_region;
#else
    gf->multiply_region.w32 = gf_w32_split_2_32_lazy_multiply_region;
    if(h->region_type & GF_REGION_SSE) return 0;
#endif
return 1;
}

/* 4/32 or Default + SSE - There is no ALTMAP/NOSSE. */

if ((h->arg1 == 4 && h->arg2 == 32) || (h->arg1 == 32 && h->arg2 == 4) ||
    (issse3 && h->mult_type == GF_REGION_DEFAULT)) {
    ld4 = (struct gf_split_4_32_lazy_data *) h->private;
    ld4->last_value = 0;
    if ((h->region_type & GF_REGION_NOSSE) || !issse3) {
        gf->multiply_region.w32 = gf_w32_split_4_32_lazy_multiply_region;
    } else if (h->region_type & GF_REGION_ALTMAP) {
        gf->multiply_region.w32 = gf_w32_split_4_32_lazy_sse_altmap_multiply_region;
    } else {
        gf->multiply_region.w32 = gf_w32_split_4_32_lazy_sse_multiply_region;
    }
    return 1;
}

/* 8/32 or Default + no SSE */

if ((h->arg1 == 8 && h->arg2 == 32) || (h->arg1 == 32 && h->arg2 == 8) ||
    h->mult_type == GF_MULT_DEFAULT) {
    d32 = (struct gf_split_8_32_lazy_data *) h->private;
    d32->last_value = 0;
    gf->multiply_region.w32 = gf_w32_split_8_32_lazy_multiply_region;
    return 1;
}

/* Finally, if args == 8, then we have to set up the tables here. */

if (h->arg1 == 8 && h->arg2 == 8) {
    d8 = (struct gf_w32_split_8_8_data *) h->private;
    d8->last_value = 0;
    gf->multiply.w32 = gf_w32_split_8_8_multiply;
    gf->multiply_region.w32 = gf_w32_split_8_32_lazy_multiply_region;
    basep = 1;
    for (exp = 0; exp < 7; exp++) {
        for (j = 0; j < 256; j++) d8->tables[exp][0][j] = 0;
        for (i = 0; i < 256; i++) d8->tables[exp][i][0] = 0;
        d8->tables[exp][1][1] = basep;
        for (i = 2; i < 256; i++) {
            if (i&1) {
                p = d8->tables[exp][i^1][1];
                d8->tables[exp][i][1] = p ^ basep;
            } else {
                p = d8->tables[exp][i>>1][1];
                d8->tables[exp][i][1] = GF_MULTBY_TWO(p);
            }
        }
    }
}
```

*src/gf\_w32.c lines 2401 to 2460*

```
    }
    for (i = 1; i < 256; i++) {
        p = d8->tables[exp][i][1];
        for (j = 1; j < 256; j++) {
            if (j&1) {
                d8->tables[exp][i][j] = d8->tables[exp][i][j^1] ^ p;
            } else {
                d8->tables[exp][i][j] = GF_MULTBY_TWO(d8->tables[exp][i][j>>1]);
            }
        }
    }
    for (i = 0; i < 8; i++) basep = GF_MULTBY_TWO(basep);
}
return 1;
}

/* If we get here, then the arguments were bad. */

return 0;
}
```

```
static
int gf_w32_group_init(gf_t *gf)
{
    uint32_t i, j, p, index;
    struct gf_w32_group_data *gd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    int g_r, g_s;

    g_s = h->arg1;
    g_r = h->arg2;

    gd = (struct gf_w32_group_data *) h->private;
    gd->shift = (uint32_t *) (&(gd->memory));
    gd->reduce = gd->shift + (1 << g_s);

    gd->rmask = (1 << g_r) - 1;
    gd->rmask <=& 32;

    gd->tshift = 32 % g_s;
    if (gd->tshift == 0) gd->tshift = g_s;
    gd->tshift = (32 - gd->tshift);
    gd->tshift = ((gd->tshift-1)/g_r) * g_r;

    gd->reduce[0] = 0;
    for (i = 0; i < (1 << g_r); i++) {
        p = 0;
        index = 0;
        for (j = 0; j < g_r; j++) {
            if (i & (1 << j)) {
                p ^= (h->prim_poly << j);
                index ^= (1 << j);
                index ^= (h->prim_poly >> (32-j));
            }
        }
        gd->reduce[index] = p;
    }

    if (g_s == g_r) {
        gf->multiply.w32 = gf_w32_group_s_equals_r_multiply;
    }
}
```



*src/gf\_w32.c lines 2461 to 2520*

```
    gf->multiply_region.w32 = gf_w32_group_s_equals_r_multiply_region;
} else {
    gf->multiply.w32 = gf_w32_group_multiply;
    gf->multiply_region.w32 = gf_w32_group_multiply_region;
}
gf->divide.w32 = NULL;
gf->inverse.w32 = gf_w32_euclid;

return 1;
}
```

```
static
uint32_t
gf_w32_composite_multiply_recursive(gf_t *gf, uint32_t a, uint32_t b)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint32_t b0 = b & 0x0000ffff;
    uint32_t b1 = (b & 0xffff0000) >> 16;
    uint32_t a0 = a & 0x0000ffff;
    uint32_t a1 = (a & 0xffff0000) >> 16;
    uint32_t alb1;
    uint32_t rv;
    alb1 = base_gf->multiply.w32(base_gf, a1, b1);

    rv = ((base_gf->multiply.w32(base_gf, a1, b0) ^
          base_gf->multiply.w32(base_gf, a0, b1) ^
          base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 16) |
          (base_gf->multiply.w32(base_gf, a0, b0) ^ alb1);
    return rv;
}
```

/\* JSP: This could be made faster. Someday, when I'm bored. \*/

```
static
uint32_t
gf_w32_composite_multiply_inline(gf_t *gf, uint32_t a, uint32_t b)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    uint32_t b0 = b & 0x0000ffff;
    uint32_t b1 = b >> 16;
    uint32_t a0 = a & 0x0000ffff;
    uint32_t a1 = a >> 16;
    uint32_t alb1, prod;
    uint16_t *log, *alog;
    struct gf_w32_composite_data *cd;

    cd = (struct gf_w32_composite_data *) h->private;
    log = cd->log;
    alog = cd->alog;

    alb1 = GF_W16_INLINE_MULT(log, alog, a1, b1);
    prod = GF_W16_INLINE_MULT(log, alog, a1, b0);
    prod ^= GF_W16_INLINE_MULT(log, alog, a0, b1);
    prod ^= GF_W16_INLINE_MULT(log, alog, alb1, h->prim_poly);
    prod <<= 16;
    prod ^= GF_W16_INLINE_MULT(log, alog, a0, b0);
    prod ^= alb1;
    return prod;
}
```

*src/gf\_w32.c lines 2521 to 2580*

```
}

/*
 * Composite field division trick (explained in 2007 tech report)
 * Compute  $a / b = a \cdot b^{-1}$ , where  $p(x) = x^2 + sx + 1$ 
 * let  $c = b^{-1}$ 
 *  $c \cdot b = (s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1)x + (b_1 c_1 + b_0 c_0)$ 
 * want  $(s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1) = 0$  and  $(b_1 c_1 + b_0 c_0) = 1$ 
 * let  $d = b_1 c_1$  and  $d+1 = b_0 c_0$ 
 * solve  $s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1 = 0$ 
 * solution:  $d = (b_1 b_0^{-1}) (b_1 b_0^{-1} + b_0 b_1^{-1} + s)^{-1}$ 
 *  $c_0 = (d+1) b_0^{-1}$ 
 *  $c_1 = d \cdot b_1^{-1}$ 
 *  $a / b = a * c$ 
 */
```

```
static
uint32_t
gf_w32_composite_inverse(gf_t *gf, uint32_t a)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint16_t a0 = a & 0x0000ffff;
    uint16_t a1 = (a & 0xffff0000) >> 16;
    uint16_t c0, c1, d, tmp;
    uint32_t c;
    uint16_t a0inv, a1inv;

    if (a0 == 0) {
        a1inv = base_gf->inverse.w32(base_gf, a1);
        c0 = base_gf->multiply.w32(base_gf, a1inv, h->prim_poly);
        c1 = a1inv;
    } else if (a1 == 0) {
        c0 = base_gf->inverse.w32(base_gf, a0);
        c1 = 0;
    } else {
        a1inv = base_gf->inverse.w32(base_gf, a1);
        a0inv = base_gf->inverse.w32(base_gf, a0);

        d = base_gf->multiply.w32(base_gf, a1, a0inv);

        tmp = (base_gf->multiply.w32(base_gf, a1, a0inv) ^ base_gf->multiply.w32(base_gf, a0, a1inv) ^ h->prim_poly);
        tmp = base_gf->inverse.w32(base_gf, tmp);

        d = base_gf->multiply.w32(base_gf, d, tmp);

        c0 = base_gf->multiply.w32(base_gf, (d^1), a0inv);
        c1 = base_gf->multiply.w32(base_gf, d, a1inv);
    }

    c = c0 | (c1 << 16);
}
```



*src/gf\_w32.c lines 2581 to 2640*

```
    return c;
}

static
void
gf_w32_composite_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint32_t b0 = val & 0x0000ffff;
    uint32_t b1 = (val & 0xffff0000) >> 16;
    uint32_t *s32, *d32, *top;
    uint16_t a0, a1, alb1, *log, *alog;
    uint32_t prod;
    gf_region_data rd;
    struct gf_w32_composite_data *cd;

    cd = (struct gf_w32_composite_data *) h->private;
    log = cd->log;
    alog = cd->alog;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);

    s32 = rd.s_start;
    d32 = rd.d_start;
    top = rd.d_top;

    if (log == NULL) {
        if (xor) {
            while (d32 < top) {
                a0 = *s32 & 0x0000ffff;
                a1 = (*s32 & 0xffff0000) >> 16;
                alb1 = base_gf->multiply.w32(base_gf, a1, b1);

                *d32 ^= ((base_gf->multiply.w32(base_gf, a0, b0) ^ alb1) |
                        ((base_gf->multiply.w32(base_gf, a1, b0) ^
                          base_gf->multiply.w32(base_gf, a0, b1) ^
                          base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 16));

                s32++;
                d32++;
            }
        } else {
            while (d32 < top) {
                a0 = *s32 & 0x0000ffff;
                a1 = (*s32 & 0xffff0000) >> 16;
                alb1 = base_gf->multiply.w32(base_gf, a1, b1);

                *d32 = ((base_gf->multiply.w32(base_gf, a0, b0) ^ alb1) |
                        ((base_gf->multiply.w32(base_gf, a1, b0) ^
                          base_gf->multiply.w32(base_gf, a0, b1) ^
                          base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 16));

                s32++;
                d32++;
            }
        }
    } else {
        if (xor) {
            while (d32 < top) {
```

*src/gf\_w32.c lines 2641 to 2700*

```
        a0 = *s32 & 0x0000ffff;
        a1 = (*s32 & 0xffff0000) >> 16;
        alb1 = GF_W16_INLINE_MULT(log, alog, a1, b1);

        prod = GF_W16_INLINE_MULT(log, alog, a1, b0);
        prod ^= GF_W16_INLINE_MULT(log, alog, a0, b1);
        prod ^= GF_W16_INLINE_MULT(log, alog, alb1, h->prim_poly);
        prod <=< 16;
        prod ^= GF_W16_INLINE_MULT(log, alog, a0, b0);
        prod ^= alb1;
        *d32 ^= prod;
        s32++;
        d32++;
    }
} else {
    while (d32 < top) {
        a0 = *s32 & 0x0000ffff;
        a1 = (*s32 & 0xffff0000) >> 16;
        alb1 = GF_W16_INLINE_MULT(log, alog, a1, b1);

        prod = GF_W16_INLINE_MULT(log, alog, a1, b0);
        prod ^= GF_W16_INLINE_MULT(log, alog, a0, b1);
        prod ^= GF_W16_INLINE_MULT(log, alog, alb1, h->prim_poly);
        prod <=< 16;
        prod ^= GF_W16_INLINE_MULT(log, alog, a0, b0);
        prod ^= alb1;

        *d32 = prod;
        s32++;
        d32++;
    }
}
}
```

```
static
void
gf_w32_composite_multiply_region_alt(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
```

```
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint16_t val0 = val & 0x0000ffff;
    uint16_t val1 = (val & 0xffff0000) >> 16;
    gf_region_data rd;
    int sub_reg_size;
    uint8_t *slow, *shigh;
    uint8_t *dlow, *dhigh, *top;
```

```
    /* JSP: I want the two pointers aligned wrt each other on 16 byte
       boundaries. So I'm going to make sure that the area on
       which the two operate is a multiple of 32. Of course, that
       junks up the mapping, but so be it -- that's why we have extract_word.... */
```

```
    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 32);
    gf_do_initial_region_alignment(&rd);
```

```
    slow = (uint8_t *) rd.s_start;
    dlow = (uint8_t *) rd.d_start;
    top = (uint8_t *) rd.d_top;
    sub_reg_size = (top - dlow)/2;
```



*src/gf\_w32.c lines 2701 to 2760*

```
    shigh = slow + sub_reg_size;
    dhigh = dlow + sub_reg_size;

    base_gf->multiply_region.w32(base_gf, slow, dlow, val0, sub_reg_size, xor);
    base_gf->multiply_region.w32(base_gf, shigh, dlow, val1, sub_reg_size, 1);
    base_gf->multiply_region.w32(base_gf, slow, dhigh, val1, sub_reg_size, xor);
    base_gf->multiply_region.w32(base_gf, shigh, dhigh, val0, sub_reg_size, 1);
    base_gf->multiply_region.w32(base_gf, shigh, dhigh,
                                base_gf->multiply.w32(base_gf, h->prim_poly, val1), sub_reg_size, 1);

    gf_do_final_region_alignment(&rd);
}

static
int gf_w32_composite_init(gf_t *gf)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    struct gf_w32_composite_data *cd;

    if (h->base_gf == NULL) return 0;

    cd = (struct gf_w32_composite_data *) h->private;
    cd->log = gf_w16_get_log_table(h->base_gf);
    cd->a_log = gf_w16_get_mult_a_log_table(h->base_gf);

    if (h->region_type & GF_REGION_ALTMAP) {
        gf->multiply_region.w32 = gf_w32_composite_multiply_region_alt;
    } else {
        gf->multiply_region.w32 = gf_w32_composite_multiply_region;
    }

    if (cd->log == NULL) {
        gf->multiply.w32 = gf_w32_composite_multiply_recursive;
    } else {
        gf->multiply.w32 = gf_w32_composite_multiply_inline;
    }
    gf->divide.w32 = NULL;
    gf->inverse.w32 = gf_w32_composite_inverse;

    return 1;
}
```

```
int gf_w32_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2)
{
    int issse3 = 0;

#ifdef INTEL_SSSE3
    issse3 = 1;
#endif

    switch(mult_type)
    {
        case GF_MULT_BYTWO_p:
        case GF_MULT_BYTWO_b:
            return sizeof(gf_internal_t) + sizeof(struct gf_w32_bytwo_data) + 64;
            break;
        case GF_MULT_GROUP:
            return sizeof(gf_internal_t) + sizeof(struct gf_w32_group_data) +
```

*src/gf\_w32.c lines 2761 to 2820*

```
        sizeof(uint32_t) * (1 << arg1) +
        sizeof(uint32_t) * (1 << arg2) + 64;
    break;
case GF_MULT_DEFAULT:
case GF_MULT_SPLIT_TABLE:
    if (arg1 == 8 && arg2 == 8) {
        return sizeof(gf_internal_t) + sizeof(struct gf_w32_split_8_8_data) + 64;
    }
    if ((arg1 == 16 && arg2 == 32) || (arg2 == 16 && arg1 == 32)) {
        return sizeof(gf_internal_t) + sizeof(struct gf_split_16_32_lazy_data) + 64;
    }
    if ((arg1 == 2 && arg2 == 32) || (arg2 == 2 && arg1 == 32)) {
        return sizeof(gf_internal_t) + sizeof(struct gf_split_2_32_lazy_data) + 64;
    }
    if ((arg1 == 8 && arg2 == 32) || (arg2 == 8 && arg1 == 32) ||
        (mult_type == GF_MULT_DEFAULT && !issse3)) {
        return sizeof(gf_internal_t) + sizeof(struct gf_split_8_32_lazy_data) + 64;
    }
    if ((arg1 == 4 && arg2 == 32) ||
        (arg2 == 4 && arg1 == 32) ||
        mult_type == GF_MULT_DEFAULT) {
        return sizeof(gf_internal_t) + sizeof(struct gf_split_4_32_lazy_data) + 64;
    }
    return 0;
case GF_MULT_CARRY_FREE:
    return sizeof(gf_internal_t);
    break;
case GF_MULT_CARRY_FREE_GK:
    return sizeof(gf_internal_t) + sizeof(uint64_t)*2;
    break;
case GF_MULT_SHIFT:
    return sizeof(gf_internal_t);
    break;
case GF_MULT_COMPOSITE:
    return sizeof(gf_internal_t) + sizeof(struct gf_w32_composite_data) + 64;
    break;

    default:
        return 0;
}
return 0;
}

int gf_w32_init(gf_t *gf)
{
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    /* Allen: set default primitive polynomial / irreducible polynomial if needed */

    if (h->prim_poly == 0) {
        if (h->mult_type == GF_MULT_COMPOSITE) {
            h->prim_poly = gf_composite_get_default_poly(h->base_gf);
            if (h->prim_poly == 0) return 0; /* This shouldn't happen */
        } else {

            /* Allen: use the following primitive polynomial to make carryless multiply work more efficiently for GF(2^32).*/
```



*src/gf\_w32.c lines 2821 to 2877*

```
    /* h->prim_poly = 0xc5; */

    /* Allen: The following is the traditional primitive polynomial for GF(2^32) */
    h->prim_poly = 0x400007;
}

/* No leading one */

if(h->mult_type != GF_MULT_COMPOSITE) h->prim_poly &= 0xffffffff;

gf->multiply.w32 = NULL;
gf->divide.w32 = NULL;
gf->inverse.w32 = NULL;
gf->multiply_region.w32 = NULL;

switch(h->mult_type) {
case GF_MULT_CARRY_FREE:      if (gf_w32_cfm_init(gf) == 0) return 0; break;
case GF_MULT_CARRY_FREE_GK:  if (gf_w32_cfm_gk_init(gf) == 0) return 0; break;
case GF_MULT_SHIFT:          if (gf_w32_shift_init(gf) == 0) return 0; break;
case GF_MULT_COMPOSITE:      if (gf_w32_composite_init(gf) == 0) return 0; break;
case GF_MULT_DEFAULT:
case GF_MULT_SPLIT_TABLE:    if (gf_w32_split_init(gf) == 0) return 0; break;
case GF_MULT_GROUP:          if (gf_w32_group_init(gf) == 0) return 0; break;
case GF_MULT_BYTWO_p:
case GF_MULT_BYTWO_b:        if (gf_w32_bytwo_init(gf) == 0) return 0; break;
default: return 0;
}

if (h->divide_type == GF_DIVIDE_EUCLID) {
    gf->divide.w32 = gf_w32_divide_from_inverse;
    gf->inverse.w32 = gf_w32_euclid;
} else if (h->divide_type == GF_DIVIDE_MATRIX) {
    gf->divide.w32 = gf_w32_divide_from_inverse;
    gf->inverse.w32 = gf_w32_matrix;
}

if (gf->inverse.w32 != NULL && gf->divide.w32 == NULL) {
    gf->divide.w32 = gf_w32_divide_from_inverse;
}

if (gf->inverse.w32 == NULL && gf->divide.w32 != NULL) {
    gf->inverse.w32 = gf_w32_inverse_from_divide;
}

if (h->region_type == GF_REGION_CAUCHY) {
    gf->extract_word.w32 = gf_wgen_extract_word;
    gf->multiply_region.w32 = gf_wgen_cauchy_region;
} else if (h->region_type & GF_REGION_ALTMAT) {
    if (h->mult_type == GF_MULT_COMPOSITE) {
        gf->extract_word.w32 = gf_w32_composite_extract_word;
    } else {
        gf->extract_word.w32 = gf_w32_split_extract_word;
    }
} else {
    gf->extract_word.w32 = gf_w32_extract_word;
}

return 1;
}
```

*src/gf\_w4.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_w4.c
 *
 * Routines for 4-bit Galois fields
 */

#include "gf_int.h"
#include <stdio.h>
#include <stdlib.h>

#define GF_FIELD_WIDTH      4
#define GF_DOUBLE_WIDTH    (GF_FIELD_WIDTH*2)
#define GF_FIELD_SIZE      (1 << GF_FIELD_WIDTH)
#define GF_MULT_GROUP_SIZE (GF_FIELD_SIZE-1)

/* -----
   JSP: Each implementation has its own data, which is allocated
   at one time as part of the handle. For that reason, it
   shouldn't be hierarchical -- i.e. one should be able to
   allocate it with one call to malloc. */

struct gf_logtable_data {
    uint8_t    log_tbl[GF_FIELD_SIZE];
    uint8_t    antilog_tbl[GF_FIELD_SIZE * 2];
    uint8_t    *antilog_tbl_div;
};

struct gf_single_table_data {
    uint8_t    mult[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint8_t    div[GF_FIELD_SIZE][GF_FIELD_SIZE];
};

struct gf_double_table_data {
    uint8_t    div[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint8_t    mult[GF_FIELD_SIZE][GF_FIELD_SIZE*GF_FIELD_SIZE];
};

struct gf_quad_table_data {
    uint8_t    div[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint16_t   mult[GF_FIELD_SIZE][(1<<16)];
};

struct gf_quad_table_lazy_data {
    uint8_t    div[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint8_t    smult[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint16_t   mult[(1 << 16)];
};

struct gf_bytwo_data {
    uint64_t   prim_poly;
    uint64_t   mask1;
    uint64_t   mask2;
};

#define AB2(ip, am1 ,am2, b, t1, t2) {\
    t1 = (b << 1) & am1;\
    t2 = b & am2; \
}
```



*src/gf\_w4.c lines 61 to 120*

```
    t2 = ((t2 << 1) - (t2 >> (GF_FIELD_WIDTH-1))); \
    b = (t1 ^ (t2 & ip));}

// ToDo(KMG/JSP): Why is 0x88 hard-coded?
#define SSE_AB2(pp, m1, va, t1, t2) {\
    t1 = _mm_and_si128(_mm_slli_epi64(va, 1), m1); \
    t2 = _mm_and_si128(va, _mm_set1_epi8(0x88)); \
    t2 = _mm_sub_epi64(_mm_slli_epi64(t2, 1), _mm_srli_epi64(t2, (GF_FIELD_WIDTH-1))); \
    va = _mm_xor_si128(t1, _mm_and_si128(t2, pp));}

/* -----
   JSP: These are basic and work from multiple implementations.
*/

static
inline
gf_val_32_t gf_w4_inverse_from_divide (gf_t *gf, gf_val_32_t a)
{
    return gf->divide.w32(gf, 1, a);
}

static
inline
gf_val_32_t gf_w4_divide_from_inverse (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    b = gf->inverse.w32(gf, b);
    return gf->multiply.w32(gf, a, b);
}

static
inline
gf_val_32_t gf_w4_euclid (gf_t *gf, gf_val_32_t b)
{
    gf_val_32_t e_i, e_im1, e_ip1;
    gf_val_32_t d_i, d_im1, d_ip1;
    gf_val_32_t y_i, y_im1, y_ip1;
    gf_val_32_t c_i;

    if (b == 0) return -1;
    e_im1 = ((gf_internal_t *) (gf->scratch))->prim_poly;
    e_i = b;
    d_im1 = 4;
    for (d_i = d_im1; ((1 << d_i) & e_i) == 0; d_i--) ;
    y_i = 1;
    y_im1 = 0;

    while (e_i != 1) {
        e_ip1 = e_im1;
        d_ip1 = d_im1;
        c_i = 0;

        while (d_ip1 >= d_i) {
            c_i ^= (1 << (d_ip1 - d_i));
            e_ip1 ^= (e_i << (d_ip1 - d_i));
            if (e_ip1 == 0) return 0;
            while ((e_ip1 & (1 << d_ip1)) == 0) d_ip1--;
        }

        y_ip1 = y_im1 ^ gf->multiply.w32(gf, c_i, y_i);
        y_im1 = y_i;
    }
}
```

*src/gf\_w4.c lines 121 to 180*

```
    y_i = y_ip1;

    e_im1 = e_i;
    d_im1 = d_i;
    e_i = e_ip1;
    d_i = d_ip1;
}

return y_i;
}
```

```
static
gf_val_32_t gf_w4_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    uint8_t *r8, v;

    r8 = (uint8_t *) start;
    v = r8[index/2];
    if (index%2) {
        return v >> 4;
    } else {
        return v&0xf;
    }
}
```

```
static
inline
gf_val_32_t gf_w4_matrix (gf_t *gf, gf_val_32_t b)
{
    return gf_bitmatrix_inverse(b, 4, ((gf_internal_t *) (gf->scratch))->prim_poly);
}
```

```
static
inline
gf_val_32_t
gf_w4_shift_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint8_t product, i, pp;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    product = 0;

    for (i = 0; i < GF_FIELD_WIDTH; i++) {
        if (a & (1 << i)) product ^= (b << i);
    }
    for (i = (GF_FIELD_WIDTH*2-2); i >= GF_FIELD_WIDTH; i--) {
        if (product & (1 << i)) product ^= (pp << (i-GF_FIELD_WIDTH));
    }
    return product;
}
```

/\* Ben: This function works, but it is 33% slower than the normal shift mult \*/

```
static
inline
```



*src/gf\_w4.c lines 181 to 240*

```
gf_val_32_t
gf_w4_clm_multiply (gf_t *gf, gf_val_32_t a4, gf_val_32_t b4)
{
    gf_val_32_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL

    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi32 (_mm_setzero_si128(), a4, 0);
    b = _mm_insert_epi32 (a, b4, 0);

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1fULL));

    /* Do the initial multiply */

    result = _mm_clmulepi64_si128 (a, b, 0);

    /* Ben/JSP: Do prim_poly reduction once. We are guaranteed that we will only
       have to do the reduction only once, because (w-2)/z == 1. Where
       z is equal to the number of zeros after the leading 1.

       _mm_clmulepi64_si128 is the carryless multiply operation. Here
       _mm_srli_epi64 shifts the result to the right by 4 bits. This allows
       us to multiply the prim_poly by the leading bits of the result. We
       then xor the result of that operation back with the result. */

    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_epi64 (result, 4), 0);
    result = _mm_xor_si128 (result, w);

    /* Extracts 32 bit value from result. */

    rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));
#endif
    return rv;
}

static
void
gf_w4_multiply_region_from_single(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int
xor)
{
    gf_region_data rd;
    uint8_t *s8;
    uint8_t *d8;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 1);
    gf_do_initial_region_alignment(&rd);

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    if (xor) {
```

*src/gf\_w4.c lines 241 to 300*

```
    while (d8 < ((uint8_t *) rd.d_top)) {
        *d8 ^= (gf->multiply.w32(gf, val, (*s8 & 0xf)) |
                ((gf->multiply.w32(gf, val, (*s8 >> 4))) << 4));
        d8++;
        s8++;
    }
} else {
    while (d8 < ((uint8_t *) rd.d_top)) {
        *d8 = (gf->multiply.w32(gf, val, (*s8 & 0xf)) |
                ((gf->multiply.w32(gf, val, (*s8 >> 4))) << 4));
        d8++;
        s8++;
    }
}
gf_do_final_region_alignment(&rd);
}
```

```
/* -----
IMPLEMENTATION: LOG_TABLE:

JSP: This is a basic log-antilog implementation.
      I'm not going to spend any time optimizing it because the
      other techniques are faster for both single and region
      operations.
*/
```

```
static
inline
gf_val_32_t
gf_w4_log_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_logtable_data *ltd;

    ltd = (struct gf_logtable_data *) ((gf_internal_t *) (gf->scratch))->private;
    return (a == 0 || b == 0) ? 0 : ltd->antilog_tbl[(unsigned) (ltd->log_tbl[a] + ltd->log_tbl[b])];
}
```

```
static
inline
gf_val_32_t
gf_w4_log_divide (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    int log_sum = 0;
    struct gf_logtable_data *ltd;

    if (a == 0 || b == 0) return 0;
    ltd = (struct gf_logtable_data *) ((gf_internal_t *) (gf->scratch))->private;

    log_sum = ltd->log_tbl[a] - ltd->log_tbl[b];
    return (ltd->antilog_tbl_div[log_sum]);
}
```

```
static
void
gf_w4_log_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int i;
    uint8_t lv, b, c;
    uint8_t *s8, *d8;
```



*src/gf\_w4.c lines 301 to 360*

```
struct gf_logtable_data *ltd;

if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

ltd = (struct gf_logtable_data *) ((gf_internal_t *) (gf->scratch))->private;
s8 = (uint8_t *) src;
d8 = (uint8_t *) dest;

lv = ltd->log_tbl[val];

for (i = 0; i < bytes; i++) {
    c = (xor) ? d8[i] : 0;
    b = (s8[i] >> GF_FIELD_WIDTH);
    c ^= (b == 0) ? 0 : (ltd->antilog_tbl[lv + ltd->log_tbl[b]] << GF_FIELD_WIDTH);
    b = (s8[i] & 0xf);
    c ^= (b == 0) ? 0 : ltd->antilog_tbl[lv + ltd->log_tbl[b]];
    d8[i] = c;
}

static
int gf_w4_log_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_logtable_data *ltd;
    int i, b;

    h = (gf_internal_t *) gf->scratch;
    ltd = h->private;

    for (i = 0; i < GF_FIELD_SIZE; i++)
        ltd->log_tbl[i] = 0;

    ltd->antilog_tbl_div = ltd->antilog_tbl + (GF_FIELD_SIZE-1);
    b = 1;
    i = 0;
    do {
        if (ltd->log_tbl[b] != 0 && i != 0) {
            fprintf(stderr, "Cannot construct log table: Polynomial is not primitive.\n\n");
            return 0;
        }
        ltd->log_tbl[b] = i;
        ltd->antilog_tbl[i] = b;
        ltd->antilog_tbl[i+GF_FIELD_SIZE-1] = b;
        b <<= 1;
        i++;
        if (b & GF_FIELD_SIZE) b = b ^ h->prim_poly;
    } while (b != 1);

    if (i != GF_FIELD_SIZE - 1) {
        _gf_errno = GF_E_LOGPOLY;
        return 0;
    }

    gf->inverse.w32 = gf_w4_inverse_from_divide;
    gf->divide.w32 = gf_w4_log_divide;
    gf->multiply.w32 = gf_w4_log_multiply;
    gf->multiply_region.w32 = gf_w4_log_multiply_region;
    return 1;
}
```

*src/gf\_w4.c lines 361 to 420*

```
}

/* -----
IMPLEMENTATION: SINGLE TABLE: JSP.
*/

static
inline
gf_val_32_t
gf_w4_single_table_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_single_table_data *std;

    std = (struct gf_single_table_data *) ((gf_internal_t *) (gf->scratch))->private;
    return std->mult[a][b];
}

static
inline
gf_val_32_t
gf_w4_single_table_divide (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_single_table_data *std;

    std = (struct gf_single_table_data *) ((gf_internal_t *) (gf->scratch))->private;
    return std->div[a][b];
}

static
void
gf_w4_single_table_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int i;
    uint8_t b, c;
    uint8_t *s8, *d8;

    struct gf_single_table_data *std;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    std = (struct gf_single_table_data *) ((gf_internal_t *) (gf->scratch))->private;
    s8 = (uint8_t *) src;
    d8 = (uint8_t *) dest;

    for (i = 0; i < bytes; i++) {
        c = (xor) ? d8[i] : 0;
        b = (s8[i] >> GF_FIELD_WIDTH);
        c ^= (std->mult[val][b] << GF_FIELD_WIDTH);
        b = (s8[i] & 0xf);
        c ^= (std->mult[val][b]);
        d8[i] = c;
    }
}

#define MM_PRINT(s, r) { uint8_t blah[16]; printf("%-12s", s); \
    _mm_storeu_si128((__m128i *)blah, r); \
    for (i = 0; i < 16; i++) \
        printf(" %02x", blah[i]); printf("\n"); }
```



*src/gf\_w4.c lines 421 to 480*

```
#ifndef INTEL_SSSE3
static
void
gf_w4_single_table_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    gf_region_data rd;
    uint8_t *base, *sptr, *dptr, *top;
    __m128i tl, loset, r, va, th;

    struct gf_single_table_data *std;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);

    std = (struct gf_single_table_data *) ((gf_internal_t *) (gf->scratch))->private;
    base = (uint8_t *) std->mult;
    base += (val << GF_FIELD_WIDTH);

    gf_do_initial_region_alignment(&rd);

    tl = _mm_loadu_si128((__m128i *)base);
    th = _mm_slli_epi64(tl, 4);
    loset = _mm_set1_epi8 (0x0f);

    sptr = rd.s_start;
    dptr = rd.d_start;
    top = rd.s_top;

    while (sptr < (uint8_t *) top) {
        va = _mm_load_si128 ((__m128i *) (sptr));
        r = _mm_and_si128 (loset, va);
        r = _mm_shuffle_epi8 (tl, r);
        va = _mm_srli_epi64 (va, 4);
        va = _mm_and_si128 (loset, va);
        va = _mm_shuffle_epi8 (th, va);
        r = _mm_xor_si128 (r, va);
        va = (xor) ? _mm_load_si128 ((__m128i *) (dptr)) : _mm_setzero_si128();
        r = _mm_xor_si128 (r, va);
        _mm_store_si128 ((__m128i *) (dptr), r);
        dptr += 16;
        sptr += 16;
    }
    gf_do_final_region_alignment(&rd);
}
#endif

static
int gf_w4_single_table_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_single_table_data *std;
    int a, b, prod;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_single_table_data *)h->private;
```

*src/gf\_w4.c lines 481 to 540*

```
bzero(std->mult, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
bzero(std->div, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);

for (a = 1; a < GF_FIELD_SIZE; a++) {
    for (b = 1; b < GF_FIELD_SIZE; b++) {
        prod = gf_w4_shift_multiply(gf, a, b);
        std->mult[a][b] = prod;
        std->div[prod][b] = a;
    }
}

gf->inverse.w32 = NULL;
gf->divide.w32 = gf_w4_single_table_divide;
gf->multiply.w32 = gf_w4_single_table_multiply;
#ifdef INTEL_SSSE3
    if(h->region_type & (GF_REGION_NOSSE | GF_REGION_CAUCHY))
        gf->multiply_region.w32 = gf_w4_single_table_multiply_region;
    else
        gf->multiply_region.w32 = gf_w4_single_table_sse_multiply_region;
#else
    gf->multiply_region.w32 = gf_w4_single_table_multiply_region;
    if (h->region_type & GF_REGION_SSE) return 0;
#endif

return 1;
}

/* -----
IMPLEMENTATION: DOUBLE TABLE: JSP.
*/

static
inline
gf_val_32_t
gf_w4_double_table_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_double_table_data *std;

    std = (struct gf_double_table_data *) ((gf_internal_t *) (gf->scratch))->private;
    return std->mult[a][b];
}

static
inline
gf_val_32_t
gf_w4_double_table_divide (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_double_table_data *std;

    std = (struct gf_double_table_data *) ((gf_internal_t *) (gf->scratch))->private;
    return std->div[a][b];
}

static
void
gf_w4_double_table_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int i;
    uint8_t *s8, *d8, *base;
    gf_region_data rd;
```



*src/gf\_w4.c lines 541 to 600*

```
struct gf_double_table_data *std;

if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);

std = (struct gf_double_table_data *) ((gf_internal_t *) (gf->scratch))->private;
s8 = (uint8_t *) src;
d8 = (uint8_t *) dest;
base = (uint8_t *) std->mult;
base += (val << GF_DOUBLE_WIDTH);

if (xor) {
    for (i = 0; i < bytes; i++) d8[i] ^= base[s8[i]];
} else {
    for (i = 0; i < bytes; i++) d8[i] = base[s8[i]];
}
}

static
int gf_w4_double_table_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_double_table_data *std;
    int a, b, c, prod, ab;
    uint8_t mult[GF_FIELD_SIZE][GF_FIELD_SIZE];

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_double_table_data *)h->private;

    bzero(mult, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
    bzero(std->div, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);

    for (a = 1; a < GF_FIELD_SIZE; a++) {
        for (b = 1; b < GF_FIELD_SIZE; b++) {
            prod = gf_w4_shift_multiply(gf, a, b);
            mult[a][b] = prod;
            std->div[prod][b] = a;
        }
    }
    bzero(std->mult, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE * GF_FIELD_SIZE);
    for (a = 0; a < GF_FIELD_SIZE; a++) {
        for (b = 0; b < GF_FIELD_SIZE; b++) {
            ab = mult[a][b];
            for (c = 0; c < GF_FIELD_SIZE; c++) {
                std->mult[a][(b << 4) | c] = ((ab << 4) | mult[a][c]);
            }
        }
    }

    gf->inverse.w32 = NULL;
    gf->divide.w32 = gf_w4_double_table_divide;
    gf->multiply.w32 = gf_w4_double_table_multiply;
    gf->multiply_region.w32 = gf_w4_double_table_multiply_region;
    return 1;
}
```

static

*src/gf\_w4.c lines 601 to 660*

```
inline
gf_val_32_t
gf_w4_quad_table_lazy_divide (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_quad_table_lazy_data *std;

    std = (struct gf_quad_table_lazy_data *) ((gf_internal_t *) (gf->scratch))->private;
    return std->div[a][b];
}

static
inline
gf_val_32_t
gf_w4_quad_table_lazy_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_quad_table_lazy_data *std;

    std = (struct gf_quad_table_lazy_data *) ((gf_internal_t *) (gf->scratch))->private;
    return std->smult[a][b];
}

static
inline
gf_val_32_t
gf_w4_quad_table_divide (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_quad_table_data *std;

    std = (struct gf_quad_table_data *) ((gf_internal_t *) (gf->scratch))->private;
    return std->div[a][b];
}

static
inline
gf_val_32_t
gf_w4_quad_table_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_quad_table_data *std;
    uint16_t v;

    std = (struct gf_quad_table_data *) ((gf_internal_t *) (gf->scratch))->private;
    v = std->mult[a][b];
    return v;
}

static
void
gf_w4_quad_table_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint16_t *base;
    gf_region_data rd;
    struct gf_quad_table_data *std;
    struct gf_quad_table_lazy_data *ltd;
    gf_internal_t *h;
    int a, b, c, d, va, vb, vc, vd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) (gf->scratch);
```



*src/gf\_w4.c lines 661 to 720*

```
if (h->region_type & GF_REGION_LAZY) {
    ltd = (struct gf_quad_table_lazy_data *) ((gf_internal_t *) (gf->scratch))->private;
    base = ltd->mult;
    for (a = 0; a < 16; a++) {
        va = (ltd->smult[val][a] << 12);
        for (b = 0; b < 16; b++) {
            vb = (ltd->smult[val][b] << 8);
            for (c = 0; c < 16; c++) {
                vc = (ltd->smult[val][c] << 4);
                for (d = 0; d < 16; d++) {
                    vd = ltd->smult[val][d];
                    base[(a << 12) | (b << 8) | (c << 4) | d] = (va | vb | vc | vd);
                }
            }
        }
    }
} else {
    std = (struct gf_quad_table_data *) ((gf_internal_t *) (gf->scratch))->private;
    base = &(std->mult[val][0]);
}

gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
gf_do_initial_region_alignment(&rd);
gf_two_byte_region_table_multiply(&rd, base);
gf_do_final_region_alignment(&rd);
}
```

```
static
int gf_w4_quad_table_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_quad_table_data *std;
    int prod, val, a, b, c, d, va, vb, vc, vd;
    uint8_t mult[GF_FIELD_SIZE][GF_FIELD_SIZE];

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_quad_table_data *) h->private;

    bzero(mult, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
    bzero(std->div, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);

    for (a = 1; a < GF_FIELD_SIZE; a++) {
        for (b = 1; b < GF_FIELD_SIZE; b++) {
            prod = gf_w4_shift_multiply(gf, a, b);
            mult[a][b] = prod;
            std->div[prod][b] = a;
        }
    }

    for (val = 0; val < 16; val++) {
        for (a = 0; a < 16; a++) {
            va = (mult[val][a] << 12);
            for (b = 0; b < 16; b++) {
                vb = (mult[val][b] << 8);
                for (c = 0; c < 16; c++) {
                    vc = (mult[val][c] << 4);
                    for (d = 0; d < 16; d++) {
                        vd = mult[val][d];
                        std->mult[val][(a << 12) | (b << 8) | (c << 4) | d] = (va | vb | vc | vd);
                    }
                }
            }
        }
    }
}
```

*src/gf\_w4.c lines 721 to 780*

```
    }
    }
}

gf->inverse.w32 = NULL;
gf->divide.w32 = gf_w4_quad_table_divide;
gf->multiply.w32 = gf_w4_quad_table_multiply;
gf->multiply_region.w32 = gf_w4_quad_table_multiply_region;
return 1;
}

static
int gf_w4_quad_table_lazy_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_quad_table_lazy_data *std;
    int a, b, prod, loga, logb;
    uint8_t log_tbl[GF_FIELD_SIZE];
    uint8_t antilog_tbl[GF_FIELD_SIZE*2];

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_quad_table_lazy_data *)h->private;

    b = 1;
    for (a = 0; a < GF_MULT_GROUP_SIZE; a++) {
        log_tbl[b] = a;
        antilog_tbl[a] = b;
        antilog_tbl[a+GF_MULT_GROUP_SIZE] = b;
        b <<= 1;
        if (b & GF_FIELD_SIZE) {
            b = b ^ h->prim_poly;
        }
    }

    bzero(std->smult, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
    bzero(std->div, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);

    for (a = 1; a < GF_FIELD_SIZE; a++) {
        loga = log_tbl[a];
        for (b = 1; b < GF_FIELD_SIZE; b++) {
            logb = log_tbl[b];
            prod = antilog_tbl[loga+logb];
            std->smult[a][b] = prod;
            std->div[prod][b] = a;
        }
    }

    gf->inverse.w32 = NULL;
    gf->divide.w32 = gf_w4_quad_table_lazy_divide;
    gf->multiply.w32 = gf_w4_quad_table_lazy_multiply;
    gf->multiply_region.w32 = gf_w4_quad_table_multiply_region;
    return 1;
}

static
int gf_w4_table_init(gf_t *gf)
{
    int rt;
    gf_internal_t *h;
    int issse3 = 0;
```



*src/gf\_w4.c lines 781 to 840*

```
#ifndef INTEL_SSSE3
    issse3 = 1;
#endif

    h = (gf_internal_t *) gf->scratch;
    rt = (h->region_type);

    if (h->mult_type == GF_MULT_DEFAULT && !issse3) rt |= GF_REGION_DOUBLE_TABLE;

    if (rt & GF_REGION_DOUBLE_TABLE) {
        return gf_w4_double_table_init(gf);
    } else if (rt & GF_REGION_QUAD_TABLE) {
        if (rt & GF_REGION_LAZY) {
            return gf_w4_quad_table_lazy_init(gf);
        } else {
            return gf_w4_quad_table_init(gf);
        }
    } else {
        return gf_w4_single_table_init(gf);
    }
}

/* -----
   JSP: GF_MULT_BYTWO_p and _b: See the paper.
*/

static
inline
gf_val_32_t
gf_w4_bytwo_p_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint32_t prod, pp, pmask, amask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;
    pmask = 0x8;
    amask = 0x8;

    while (amask != 0) {
        if (prod & pmask) {
            prod = ((prod << 1) ^ pp);
        } else {
            prod <<= 1;
        }
        if (a & amask) prod ^= b;
        amask >>= 1;
    }
    return prod;
}

static
inline
gf_val_32_t
gf_w4_bytwo_b_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
```

*src/gf\_w4.c lines 841 to 900*

```
{
    uint32_t prod, pp, bmask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;
    bmask = 0x8;

    while (1) {
        if (a & 1) prod ^= b;
        a >>= 1;
        if (a == 0) return prod;
        if (b & bmask) {
            b = ((b << 1) ^ pp);
        } else {
            b <<= 1;
        }
    }
}

static
void
gf_w4_bytwo_p_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t *s64, *d64, t1, t2, ta, prod, amask;
    gf_region_data rd;
    struct gf_bytwo_data *btd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    btd = (struct gf_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
    gf_do_initial_region_alignment(&rd);

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;

    if (xor) {
        while (s64 < (uint64_t *) rd.s_top) {
            prod = 0;
            amask = 0x8;
            ta = *s64;
            while (amask != 0) {
                AB2(btd->prim_poly, btd->mask1, btd->mask2, prod, t1, t2);
                if (val & amask) prod ^= ta;
                amask >>= 1;
            }
            *d64 ^= prod;
            d64++;
            s64++;
        }
    } else {
        while (s64 < (uint64_t *) rd.s_top) {
            prod = 0;
            amask = 0x8;
            ta = *s64;
```



*src/gf\_w4.c lines 901 to 960*

```
        while (amask != 0) {
            AB2(btd->prim_poly, btd->mask1, btd->mask2, prod, t1, t2);
            if (val & amask) prod ^= ta;
            amask >>= 1;
        }
        *d64 = prod;
        d64++;
        s64++;
    }
}
gf_do_final_region_alignment(&rd);
}

#define BYTWO_P_ONESTEP {\
    SSE_AB2(pp, m1, prod, t1, t2); \
    t1 = _mm_and_si128(v, one); \
    t1 = _mm_sub_epi8(t1, one); \
    t1 = _mm_and_si128(t1, ta); \
    prod = _mm_xor_si128(prod, t1); \
    v = _mm_srli_epi64(v, 1); }

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_p_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int i;
    uint8_t *s8, *d8;
    uint8_t vrev;
    __m128i pp, m1, ta, prod, t1, t2, tp, one, v;
    struct gf_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    btd = (struct gf_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    vrev = 0;
    for (i = 0; i < 4; i++) {
        vrev <=< 1;
        if (!(val & (1 << i))) vrev |= 1;
    }

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);
    one = _mm_set1_epi8(1);

    while (d8 < (uint8_t *) rd.d_top) {
        prod = _mm_setzero_si128();
        v = _mm_set1_epi8(vrev);
        ta = _mm_load_si128((__m128i *) s8);
        tp = (!xor) ? _mm_setzero_si128() : _mm_load_si128((__m128i *) d8);
        BYTWO_P_ONESTEP;
    }
}
```

*src/gf\_w4.c lines 961 to 1020*

```
    BYTWO_P_ONESTEP;
    BYTWO_P_ONESTEP;
    BYTWO_P_ONESTEP;
    _mm_store_si128((__m128i *) d8, _mm_xor_si128(prod, tp));
    d8 += 16;
    s8 += 16;
}
gf_do_final_region_alignment(&rd);
}
#endif

/*
static
void
gf_w4_bytwo_b_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
#ifdef INTEL_SSE2
    uint8_t *d8, *s8, tb;
    __m128i pp, m1, m2, t1, t2, va, vb;
    struct gf_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    btd = (struct gf_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);
    m2 = _mm_set1_epi8((btd->mask2)&0xff);

    if (xor) {
        while (d8 < (uint8_t *) rd.d_top) {
            va = _mm_load_si128((__m128i *) (s8));
            vb = _mm_load_si128((__m128i *) (d8));
            tb = val;
            while (1) {
                if (tb & 1) vb = _mm_xor_si128(vb, va);
                tb >>= 1;
                if (tb == 0) break;
                SSE_AB2(pp, m1, m2, va, t1, t2);
            }
            _mm_store_si128((__m128i *) d8, vb);
            d8 += 16;
            s8 += 16;
        }
    } else {
        while (d8 < (uint8_t *) rd.d_top) {
            va = _mm_load_si128((__m128i *) (s8));
            vb = _mm_setzero_si128();
            tb = val;
            while (1) {
                if (tb & 1) vb = _mm_xor_si128(vb, va);
                tb >>= 1;
            }
        }
    }
}
```



*src/gf\_w4.c lines 1021 to 1080*

```
        if (tb == 0) break;
        t1 = _mm_and_si128(_mm_slli_epi64(va, 1), m1);
        t2 = _mm_and_si128(va, m2);
        t2 = _mm_sub_epi64 (
            _mm_slli_epi64(t2, 1), _mm_srli_epi64(t2, (GF_FIELD_WIDTH-1)));
        va = _mm_xor_si128(t1, _mm_and_si128(t2, pp));
    }
    _mm_store_si128((__m128i *)d8, vb);
    d8 += 16;
    s8 += 16;
}
}
gf_do_final_region_alignment(&rd);
#endif
}
*/

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_2_noxor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        SSE_AB2(pp, m1, va, t1, t2);
        _mm_store_si128((__m128i *)d8, va);
        d8 += 16;
        s8 += 16;
    }
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_2_xor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        SSE_AB2(pp, m1, va, t1, t2);
        vb = _mm_load_si128 ((__m128i *) (d8));
        vb = _mm_xor_si128(vb, va);
    }
}
```

*src/gf\_w4.c lines 1081 to 1140*

```
    __mm_store_si128((__m128i *)d8, vb);
    d8 += 16;
    s8 += 16;
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_4_noxor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128((__m128i *) (s8));
        SSE_AB2(pp, m1, va, t1, t2);
        SSE_AB2(pp, m1, va, t1, t2);
        __mm_store_si128((__m128i *)d8, va);
        d8 += 16;
        s8 += 16;
    }
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_4_xor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128((__m128i *) (s8));
        SSE_AB2(pp, m1, va, t1, t2);
        SSE_AB2(pp, m1, va, t1, t2);
        vb = _mm_load_si128((__m128i *) (d8));
        vb = _mm_xor_si128(vb, va);
        __mm_store_si128((__m128i *)d8, vb);
        d8 += 16;
        s8 += 16;
    }
}
}
#endif
```



*src/gf\_w4.c lines 1141 to 1200*

```
#ifndef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_3_noxor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        vb = va;
        SSE_AB2(pp, m1, va, t1, t2);
        va = _mm_xor_si128(va, vb);
        _mm_store_si128((__m128i *) d8, va);
        d8 += 16;
        s8 += 16;
    }
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_3_xor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        vb = _mm_xor_si128(_mm_load_si128 ((__m128i *) (d8)), va);
        SSE_AB2(pp, m1, va, t1, t2);
        vb = _mm_xor_si128(vb, va);
        _mm_store_si128((__m128i *) d8, vb);
        d8 += 16;
        s8 += 16;
    }
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_5_noxor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;
```

*src/gf\_w4.c lines 1201 to 1260*

```
s8 = (uint8_t *) rd->s_start;
d8 = (uint8_t *) rd->d_start;

pp = _mm_set1_epi8(btd->prim_poly&0xff);
m1 = _mm_set1_epi8((btd->mask1)&0xff);

while (d8 < (uint8_t *) rd->d_top) {
    va = _mm_load_si128 ((__m128i *) (s8));
    vb = va;
    SSE_AB2(pp, m1, va, t1, t2);
    SSE_AB2(pp, m1, va, t1, t2);
    va = _mm_xor_si128(va, vb);
    _mm_store_si128((__m128i *) d8, va);
    d8 += 16;
    s8 += 16;
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_5_xor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        vb = _mm_xor_si128(_mm_load_si128 ((__m128i *) (d8)), va);
        SSE_AB2(pp, m1, va, t1, t2);
        SSE_AB2(pp, m1, va, t1, t2);
        vb = _mm_xor_si128(vb, va);
        _mm_store_si128((__m128i *) d8, vb);
        d8 += 16;
        s8 += 16;
    }
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_7_noxor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);
```



*src/gf\_w4.c lines 1261 to 1320*

```
while (d8 < (uint8_t *) rd->d_top) {
    va = _mm_load_si128 ((__m128i *) (s8));
    vb = va;
    SSE_AB2(pp, m1, va, t1, t2);
    vb = _mm_xor_si128(va, vb);
    SSE_AB2(pp, m1, va, t1, t2);
    va = _mm_xor_si128(va, vb);
    _mm_store_si128((__m128i *)d8, va);
    d8 += 16;
    s8 += 16;
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_7_xor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        vb = _mm_xor_si128(_mm_load_si128 ((__m128i *) (d8)), va);
        SSE_AB2(pp, m1, va, t1, t2);
        vb = _mm_xor_si128(vb, va);
        SSE_AB2(pp, m1, va, t1, t2);
        vb = _mm_xor_si128(vb, va);
        _mm_store_si128((__m128i *)d8, vb);
        d8 += 16;
        s8 += 16;
    }
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_6_noxor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        SSE_AB2(pp, m1, va, t1, t2);
        vb = va;
    }
}
```

*src/gf\_w4.c lines 1321 to 1380*

```
    SSE_AB2(pp, m1, va, t1, t2);
    va = _mm_xor_si128(va, vb);
    _mm_store_si128((__m128i *)d8, va);
    d8 += 16;
    s8 += 16;
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_region_6_xor(gf_region_data *rd, struct gf_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128((__m128i *) (s8));
        SSE_AB2(pp, m1, va, t1, t2);
        vb = _mm_xor_si128(_mm_load_si128((__m128i *) (d8)), va);
        SSE_AB2(pp, m1, va, t1, t2);
        vb = _mm_xor_si128(vb, va);
        _mm_store_si128((__m128i *)d8, vb);
        d8 += 16;
        s8 += 16;
    }
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w4_bytwo_b_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint8_t *d8, *s8, tb;
    __m128i pp, m1, m2, t1, t2, va, vb;
    struct gf_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    btd = (struct gf_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

    switch (val) {
        case 2:
            if (!xor) {
```



*src/gf\_w4.c lines 1381 to 1440*

```
    gf_w4_bytwo_b_sse_region_2_noxor(&rd, btd);
} else {
    gf_w4_bytwo_b_sse_region_2_xor(&rd, btd);
}
gf_do_final_region_alignment(&rd);
return;
case 3:
    if (!xor) {
        gf_w4_bytwo_b_sse_region_3_noxor(&rd, btd);
    } else {
        gf_w4_bytwo_b_sse_region_3_xor(&rd, btd);
    }
    gf_do_final_region_alignment(&rd);
    return;
case 4:
    if (!xor) {
        gf_w4_bytwo_b_sse_region_4_noxor(&rd, btd);
    } else {
        gf_w4_bytwo_b_sse_region_4_xor(&rd, btd);
    }
    gf_do_final_region_alignment(&rd);
    return;
case 5:
    if (!xor) {
        gf_w4_bytwo_b_sse_region_5_noxor(&rd, btd);
    } else {
        gf_w4_bytwo_b_sse_region_5_xor(&rd, btd);
    }
    gf_do_final_region_alignment(&rd);
    return;
case 6:
    if (!xor) {
        gf_w4_bytwo_b_sse_region_6_noxor(&rd, btd);
    } else {
        gf_w4_bytwo_b_sse_region_6_xor(&rd, btd);
    }
    gf_do_final_region_alignment(&rd);
    return;
case 7:
    if (!xor) {
        gf_w4_bytwo_b_sse_region_7_noxor(&rd, btd);
    } else {
        gf_w4_bytwo_b_sse_region_7_xor(&rd, btd);
    }
    gf_do_final_region_alignment(&rd);
    return;
}
```

```
pp = _mm_set1_epi8(btd->prim_poly&0xff);
m1 = _mm_set1_epi8((btd->mask1)&0xff);
m2 = _mm_set1_epi8((btd->mask2)&0xff);
```

```
if (xor) {
    while (d8 < (uint8_t *) rd.d_top) {
        va = _mm_load_si128((__m128i *) (s8));
        vb = _mm_load_si128((__m128i *) (d8));
        tb = val;
        while (1) {
            if (tb & 1) vb = _mm_xor_si128(vb, va);
            tb >>= 1;
        }
    }
}
```

*src/gf\_w4.c lines 1441 to 1500*

```
        if (tb == 0) break;
        SSE_AB2(pp, m1, va, t1, t2);
    }
    _mm_store_si128((__m128i *)d8, vb);
    d8 += 16;
    s8 += 16;
}
} else {
    while (d8 < (uint8_t *) rd.d_top) {
        va = _mm_load_si128((__m128i *) (s8));
        vb = _mm_setzero_si128();
        tb = val;
        while (1) {
            if (tb & 1) vb = _mm_xor_si128(vb, va);
            tb >>= 1;
            if (tb == 0) break;
            t1 = _mm_and_si128(_mm_slli_epi64(va, 1), m1);
            t2 = _mm_and_si128(va, m2);
            t2 = _mm_sub_epi64(
                _mm_slli_epi64(t2, 1), _mm_srli_epi64(t2, (GF_FIELD_WIDTH-1)));
            va = _mm_xor_si128(t1, _mm_and_si128(t2, pp));
        }
        _mm_store_si128((__m128i *)d8, vb);
        d8 += 16;
        s8 += 16;
    }
}
gf_do_final_region_alignment(&rd);
}
#endif

static
void
gf_w4_bytwo_b_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t *s64, *d64, t1, t2, ta, tb, prod;
    struct gf_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    btd = (struct gf_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;
    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;

    switch (val) {
    case 1:
        if (xor) {
            while (d64 < (uint64_t *) rd.d_top) {
                *d64 ^= *s64;
                d64++;
                s64++;
            }
        } else {
            while (d64 < (uint64_t *) rd.d_top) {
                *d64 = *s64;
            }
        }
    }
```



*src/gf\_w4.c lines 1501 to 1560*

```
        d64++;
        s64++;
    }
}
break;
case 2:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= ta;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta;
            d64++;
            s64++;
        }
    }
}
break;
case 3:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
            d64++;
            s64++;
        }
    }
}
break;
case 4:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= ta;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
```

*src/gf\_w4.c lines 1561 to 1620*

```
        *d64 = ta;
        d64++;
        s64++;
    }
}
break;
case 5:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta ^ prod;
            d64++;
            s64++;
        }
    }
}
break;
case 6:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta ^ prod;
            d64++;
            s64++;
        }
    }
}
break;
case 7:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
```



*src/gf\_w4.c lines 1621 to 1680*

```
        *d64 ^= (ta ^ prod);
        d64++;
        s64++;
    }
} else {
    while (d64 < (uint64_t *) rd.d_top) {
        ta = *s64;
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod ^= ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 = ta ^ prod;
        d64++;
        s64++;
    }
}
break;
case 8:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= ta;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta;
            d64++;
            s64++;
        }
    }
}
break;
case 9:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
        }
    }
}
```

*src/gf\_w4.c lines 1681 to 1740*

```
        d64++;
        s64++;
    }
}
break;
case 10:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
            d64++;
            s64++;
        }
    }
}
break;
case 11:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
            d64++;
            s64++;
        }
    }
}
break;
case 12:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
```



*src/gf\_w4.c lines 1741 to 1800*

```
        ta = *s64;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 ^= (ta ^ prod);
        d64++;
        s64++;
    }
} else {
    while (d64 < (uint64_t *) rd.d_top) {
        ta = *s64;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 = (ta ^ prod);
        d64++;
        s64++;
    }
}
break;
case 13:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
            d64++;
            s64++;
        }
    }
}
break;
case 14:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
        }
    }
```

*src/gf\_w4.c lines 1801 to 1860*

```
        s64++;
    }
} else {
    while (d64 < (uint64_t *) rd.d_top) {
        ta = *s64;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod ^= ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 = (ta ^ prod);
        d64++;
        s64++;
    }
}
break;
case 15:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
            d64++;
            s64++;
        }
    }
}
break;
default:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            prod = *d64 ;
            ta = *s64;
            tb = val;
            while (1) {
                if (tb & 1) prod ^= ta;
                tb >>= 1;
                if (tb == 0) break;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            }
            *d64 = prod;
            d64++;
            s64++;
        }
    }
```



*src/gf\_w4.c lines 1861 to 1920*

```
    }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            prod = 0;
            ta = *s64;
            tb = val;
            while (1) {
                if (tb & 1) prod ^= ta;
                tb >>= 1;
                if (tb == 0) break;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            }
            *d64 = prod;
            d64++;
            s64++;
        }
    }
    break;
}
gf_do_final_region_alignment(&rd);
}

static
int gf_w4_bytwo_init(gf_t *gf)
{
    gf_internal_t *h;
    uint64_t ip, m1, m2;
    struct gf_bytwo_data *btd;

    h = (gf_internal_t *) gf->scratch;
    btd = (struct gf_bytwo_data *) (h->private);
    ip = h->prim_poly & 0xf;
    m1 = 0xe;
    m2 = 0x8;
    btd->prim_poly = 0;
    btd->mask1 = 0;
    btd->mask2 = 0;

    while (ip != 0) {
        btd->prim_poly |= ip;
        btd->mask1 |= m1;
        btd->mask2 |= m2;
        ip <<= GF_FIELD_WIDTH;
        m1 <<= GF_FIELD_WIDTH;
        m2 <<= GF_FIELD_WIDTH;
    }

    if (h->mult_type == GF_MULT_BYTWO_p) {
        gf->multiply.w32 = gf_w4_bytwo_p_multiply;
#ifdef INTEL_SSE2
        if (h->region_type & GF_REGION_NOSSE)
            gf->multiply_region.w32 = gf_w4_bytwo_p_nosse_multiply_region;
        else
            gf->multiply_region.w32 = gf_w4_bytwo_p_sse_multiply_region;
#else
        gf->multiply_region.w32 = gf_w4_bytwo_p_nosse_multiply_region;
        if (h->region_type & GF_REGION_SSE)
            return 0;
#endif
    } else {
```

*src/gf\_w4.c lines 1921 to 1980*

```
gf->multiply.w32 = gf_w4_bytwo_b_multiply;
#ifdef INTEL_SSE2
    if (h->region_type & GF_REGION_NOSSE)
        gf->multiply_region.w32 = gf_w4_bytwo_b_nosse_multiply_region;
    else
        gf->multiply_region.w32 = gf_w4_bytwo_b_sse_multiply_region;
#else
    gf->multiply_region.w32 = gf_w4_bytwo_b_nosse_multiply_region;
    if (h->region_type & GF_REGION_SSE)
        return 0;
#endif
}
return 1;
}
```

```
static
int gf_w4_cfm_init(gf_t *gf)
{
    #if defined(INTEL_SSE4_PCLMUL)
        gf->multiply.w32 = gf_w4_clm_multiply;
        return 1;
    #endif
    return 0;
}
```

```
static
int gf_w4_shift_init(gf_t *gf)
{
    gf->multiply.w32 = gf_w4_shift_multiply;
    return 1;
}
```

/\* JSP: I'm putting all error-checking into gf\_error\_check(), so you don't  
have to do error checking in scratch\_size or in init \*/

```
int gf_w4_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2)
{
    int issse3 = 0;
```

```
#ifdef INTEL_SSSE3
    issse3 = 1;
#endif
```

```
    switch(mult_type)
    {
        case GF_MULT_BYTWO_p:
        case GF_MULT_BYTWO_b:
            return sizeof(gf_internal_t) + sizeof(struct gf_bytwo_data);
            break;
        case GF_MULT_DEFAULT:
        case GF_MULT_TABLE:
            if (region_type == GF_REGION_CAUCHY) {
                return sizeof(gf_internal_t) + sizeof(struct gf_single_table_data) + 64;
            }

            if (mult_type == GF_MULT_DEFAULT && !issse3) region_type = GF_REGION_DOUBLE_TABLE;

            if (region_type & GF_REGION_DOUBLE_TABLE) {
                return sizeof(gf_internal_t) + sizeof(struct gf_double_table_data) + 64;
            }
    }
```



*src/gf\_w4.c lines 1981 to 2040*

```
    } else if (region_type & GF_REGION_QUAD_TABLE) {
        if ((region_type & GF_REGION_LAZY) == 0) {
            return sizeof(gf_internal_t) + sizeof(struct gf_quad_table_data) + 64;
        } else {
            return sizeof(gf_internal_t) + sizeof(struct gf_quad_table_lazy_data) + 64;
        }
    } else {
        return sizeof(gf_internal_t) + sizeof(struct gf_single_table_data) + 64;
    }
}
break;

case GF_MULT_LOG_TABLE:
    return sizeof(gf_internal_t) + sizeof(struct gf_logtable_data) + 64;
break;
case GF_MULT_CARRY_FREE:
    return sizeof(gf_internal_t);
break;
case GF_MULT_SHIFT:
    return sizeof(gf_internal_t);
break;
default:
    return 0;
}
return 0;
}

int
gf_w4_init (gf_t *gf)
{
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    if (h->prim_poly == 0) h->prim_poly = 0x13;
    h->prim_poly |= 0x10;
    gf->multiply.w32 = NULL;
    gf->divide.w32 = NULL;
    gf->inverse.w32 = NULL;
    gf->multiply_region.w32 = NULL;
    gf->extract_word.w32 = gf_w4_extract_word;

    switch(h->mult_type) {
        case GF_MULT_CARRY_FREE: if (gf_w4_cfm_init(gf) == 0) return 0; break;
        case GF_MULT_SHIFT:      if (gf_w4_shift_init(gf) == 0) return 0; break;
        case GF_MULT_BYTWO_p:
        case GF_MULT_BYTWO_b:     if (gf_w4_bytwo_init(gf) == 0) return 0; break;
        case GF_MULT_LOG_TABLE:   if (gf_w4_log_init(gf) == 0) return 0; break;
        case GF_MULT_DEFAULT:
        case GF_MULT_TABLE:       if (gf_w4_table_init(gf) == 0) return 0; break;
        default: return 0;
    }

    if (h->divide_type == GF_DIVIDE_EUCLID) {
        gf->divide.w32 = gf_w4_divide_from_inverse;
        gf->inverse.w32 = gf_w4_euclid;
    } else if (h->divide_type == GF_DIVIDE_MATRIX) {
        gf->divide.w32 = gf_w4_divide_from_inverse;
        gf->inverse.w32 = gf_w4_matrix;
    }

    if (gf->divide.w32 == NULL) {
```

*src/gf\_w4.c lines 2041 to 2086*

```
    gf->divide.w32 = gf_w4_divide_from_inverse;
    if (gf->inverse.w32 == NULL) gf->inverse.w32 = gf_w4_euclid;
}

if (gf->inverse.w32 == NULL) gf->inverse.w32 = gf_w4_inverse_from_divide;

if (h->region_type == GF_REGION_CAUCHY) {
    gf->multiply_region.w32 = gf_wgen_cauchy_region;
    gf->extract_word.w32 = gf_wgen_extract_word;
}

if (gf->multiply_region.w32 == NULL) {
    gf->multiply_region.w32 = gf_w4_multiply_region_from_single;
}

return 1;
}

/* Inline setup functions */

uint8_t *gf_w4_get_mult_table(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_single_table_data *std;

    h = (gf_internal_t *) gf->scratch;
    if (gf->multiply.w32 == gf_w4_single_table_multiply) {
        std = (struct gf_single_table_data *) h->private;
        return (uint8_t *) std->mult;
    }
    return NULL;
}

uint8_t *gf_w4_get_div_table(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_single_table_data *std;

    h = (gf_internal_t *) gf->scratch;
    if (gf->multiply.w32 == gf_w4_single_table_multiply) {
        std = (struct gf_single_table_data *) h->private;
        return (uint8_t *) std->div;
    }
    return NULL;
}
```



*src/gf\_w64.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_w64.c
 *
 * Routines for 64-bit Galois fields
 */

#include "gf_int.h"
#include <stdio.h>
#include <stdlib.h>

#define GF_FIELD_WIDTH (64)
#define GF_FIRST_BIT (1ULL << 63)

#define GF_BASE_FIELD_WIDTH (32)
#define GF_BASE_FIELD_SIZE (1ULL << GF_BASE_FIELD_WIDTH)
#define GF_BASE_FIELD_GROUP_SIZE GF_BASE_FIELD_SIZE-1

struct gf_w64_group_data {
    uint64_t *reduce;
    uint64_t *shift;
    uint64_t *memory;
};

struct gf_split_4_64_lazy_data {
    uint64_t tables[16][16];
    uint64_t last_value;
};

struct gf_split_8_64_lazy_data {
    uint64_t tables[8][(1<<8)];
    uint64_t last_value;
};

struct gf_split_16_64_lazy_data {
    uint64_t tables[4][(1<<16)];
    uint64_t last_value;
};

struct gf_split_8_8_data {
    uint64_t tables[15][256][256];
};

static
inline
gf_val_64_t gf_w64_inverse_from_divide (gf_t *gf, gf_val_64_t a)
{
    return gf->divide.w64(gf, 1, a);
}

#define MM_PRINT8(s, r) { uint8_t blah[16], ii; printf("%-12s", s); \
    _mm_storeu_si128((__m128i *)blah, r); \
    for (ii = 0; ii < 16; ii += 1) \
        printf("%s%02x", (ii%4==0) ? "    " : " ", blah[15-ii]); printf("\n"); }

static
inline
```

*src/gf\_w64.c lines 61 to 120*

```
gf_val_64_t gf_w64_divide_from_inverse (gf_t *gf, gf_val_64_t a, gf_val_64_t b)
{
    b = gf->inverse.w64(gf, b);
    return gf->multiply.w64(gf, a, b);
}
```

```
static
void
gf_w64_multiply_region_from_single(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int
xor)
{
    int i;
    gf_val_64_t *s64;
    gf_val_64_t *d64;

    s64 = (gf_val_64_t *) src;
    d64 = (gf_val_64_t *) dest;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    if (xor) {
        for (i = 0; i < bytes/sizeof(gf_val_64_t); i++) {
            d64[i] ^= gf->multiply.w64(gf, val, s64[i]);
        }
    } else {
        for (i = 0; i < bytes/sizeof(gf_val_64_t); i++) {
            d64[i] = gf->multiply.w64(gf, val, s64[i]);
        }
    }
}
```

```
#if defined(INTEL_SSE4_PCLMUL)
static
void
gf_w64_clm_multiply_region_from_single_2(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int
xor)
{
    gf_val_64_t *s64, *d64, *top;
    gf_region_data rd;

    __m128i a, b;
    __m128i result, r1;
    __m128i prim_poly;
    __m128i w;
    __m128i m1, m3, m4;
    gf_internal_t * h = gf->scratch;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0xffffffffULL));
    b = _mm_insert_epi64(_mm_setzero_si128(), val, 0);
    m1 = _mm_set_epi32(0, 0, 0, (uint32_t)0xffffffff);
    m3 = _mm_slli_si128(m1, 8);
    m4 = _mm_slli_si128(m3, 4);
}
```



*src/gf\_w64.c lines 121 to 180*

```
s64 = (gf_val_64_t *) rd.s_start;
d64 = (gf_val_64_t *) rd.d_start;
top = (gf_val_64_t *) rd.d_top;

if (xor) {
    while (d64 != top) {
        a = _mm_load_si128((__m128i *) s64);
        result = _mm_clmulepi64_si128 (a, b, 1);

        w = _mm_clmulepi64_si128 (_mm_and_si128(result, m4), prim_poly, 1);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (_mm_and_si128(result, m3), prim_poly, 1);
        r1 = _mm_xor_si128 (result, w);

        result = _mm_clmulepi64_si128 (a, b, 0);

        w = _mm_clmulepi64_si128 (_mm_and_si128(result, m4), prim_poly, 1);
        result = _mm_xor_si128 (result, w);

        w = _mm_clmulepi64_si128 (_mm_and_si128(result, m3), prim_poly, 1);
        result = _mm_xor_si128 (result, w);

        result = _mm_unpacklo_epi64(result, r1);

        r1 = _mm_load_si128((__m128i *) d64);
        result = _mm_xor_si128(r1, result);
        _mm_store_si128((__m128i *) d64, result);
        d64 += 2;
        s64 += 2;
    }
} else {
    while (d64 != top) {

        a = _mm_load_si128((__m128i *) s64);
        result = _mm_clmulepi64_si128 (a, b, 1);

        w = _mm_clmulepi64_si128 (_mm_and_si128(result, m4), prim_poly, 1);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (_mm_and_si128(result, m3), prim_poly, 1);
        r1 = _mm_xor_si128 (result, w);

        result = _mm_clmulepi64_si128 (a, b, 0);

        w = _mm_clmulepi64_si128 (_mm_and_si128(result, m4), prim_poly, 1);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (_mm_and_si128(result, m3), prim_poly, 1);
        result = _mm_xor_si128 (result, w);

        result = _mm_unpacklo_epi64(result, r1);

        _mm_store_si128((__m128i *) d64, result);
        d64 += 2;
        s64 += 2;
    }
}
gf_do_final_region_alignment(&rd);
}
#endif

#if defined(INTEL_SSE4_PCLMUL)
```

*src/gf\_w64.c lines 181 to 240*

```
static
void
gf_w64_clm_multiply_region_from_single_4(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int
xor)
{
    gf_val_64_t *s64, *d64, *top;
    gf_region_data rd;

    __m128i      a, b;
    __m128i      result, r1;
    __m128i      prim_poly;
    __m128i      w;
    __m128i      m1, m3, m4;
    gf_internal_t * h = gf->scratch;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0xffffffffULL));
    b = _mm_insert_epi64(_mm_setzero_si128(), val, 0);
    m1 = _mm_set_epi32(0, 0, 0, (uint32_t)0xffffffff);
    m3 = _mm_slli_si128(m1, 8);
    m4 = _mm_slli_si128(m3, 4);

    s64 = (gf_val_64_t *) rd.s_start;
    d64 = (gf_val_64_t *) rd.d_start;
    top = (gf_val_64_t *) rd.d_top;

    if (xor) {
        while (d64 != top) {
            a = _mm_load_si128((__m128i *) s64);
            result = _mm_clmulepi64_si128(a, b, 1);

            w = _mm_clmulepi64_si128(_mm_and_si128(result, m4), prim_poly, 1);
            result = _mm_xor_si128(result, w);
            w = _mm_clmulepi64_si128(_mm_and_si128(result, m3), prim_poly, 1);
            r1 = _mm_xor_si128(result, w);

            result = _mm_clmulepi64_si128(a, b, 0);

            w = _mm_clmulepi64_si128(_mm_and_si128(result, m4), prim_poly, 1);
            result = _mm_xor_si128(result, w);

            w = _mm_clmulepi64_si128(_mm_and_si128(result, m3), prim_poly, 1);
            result = _mm_xor_si128(result, w);

            result = _mm_unpacklo_epi64(result, r1);

            r1 = _mm_load_si128((__m128i *) d64);
            result = _mm_xor_si128(r1, result);
            _mm_store_si128((__m128i *) d64, result);
            d64 += 2;
            s64 += 2;
        }
    } else {
        while (d64 != top) {
            a = _mm_load_si128((__m128i *) s64);
```



*src/gf\_w64.c lines 241 to 300*

```
    result = _mm_clmulepi64_si128 (a, b, 1);

    w = _mm_clmulepi64_si128 (_mm_and_si128(result, m4), prim_poly, 1);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (_mm_and_si128(result, m3), prim_poly, 1);
    r1 = _mm_xor_si128 (result, w);

    result = _mm_clmulepi64_si128 (a, b, 0);

    w = _mm_clmulepi64_si128 (_mm_and_si128(result, m4), prim_poly, 1);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (_mm_and_si128(result, m3), prim_poly, 1);
    result = _mm_xor_si128 (result, w);

    result = _mm_unpacklo_epi64(result, r1);

    _mm_store_si128((__m128i *) d64, result);
    d64 += 2;
    s64 += 2;
}
}
gf_do_final_region_alignment(&rd);
}
#endif
```

```
static
inline
gf_val_64_t gf_w64_euclid (gf_t *gf, gf_val_64_t b)
{
    gf_val_64_t e_i, e_im1, e_ip1;
    gf_val_64_t d_i, d_im1, d_ip1;
    gf_val_64_t y_i, y_im1, y_ip1;
    gf_val_64_t c_i;
    gf_val_64_t one = 1;

    if (b == 0) return -1;
    e_im1 = ((gf_internal_t *) (gf->scratch))->prim_poly;
    e_i = b;
    d_im1 = 64;
    for (d_i = d_im1-1; ((one << d_i) & e_i) == 0; d_i--) ;
    y_i = 1;
    y_im1 = 0;

    while (e_i != 1) {

        e_ip1 = e_im1;
        d_ip1 = d_im1;
        c_i = 0;

        while (d_ip1 >= d_i) {
            c_i ^= (one << (d_ip1 - d_i));
            e_ip1 ^= (e_i << (d_ip1 - d_i));
            d_ip1--;
            if (e_ip1 == 0) return 0;
            while ((e_ip1 & (one << d_ip1)) == 0) d_ip1--;
        }

        y_ip1 = y_im1 ^ gf->multiply.w64(gf, c_i, y_i);
        y_im1 = y_i;
        y_i = y_ip1;
    }
}
```

*src/gf\_w64.c lines 301 to 360*

```
    e_im1 = e_i;
    d_im1 = d_i;
    e_i = e_ip1;
    d_i = d_ip1;
}

return y_i;
}

/* JSP: GF_MULT_SHIFT: The world's dumbest multiplication algorithm. I only
   include it for completeness. It does have the feature that it requires no
   extra memory.
*/

static
inline
gf_val_64_t
gf_w64_shift_multiply (gf_t *gf, gf_val_64_t a64, gf_val_64_t b64)
{
    uint64_t pl, pr, ppl, ppr, i, a, bl, br, one, lbit;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    /* Allen: set leading one of primitive polynomial */

    a = a64;
    bl = 0;
    br = b64;
    one = 1;
    lbit = (one << 63);

    pl = 0; /* Allen: left side of product */
    pr = 0; /* Allen: right side of product */

    /* Allen: unlike the corresponding functions for smaller word sizes,
     * this loop carries out the initial carryless multiply by
     * shifting b itself rather than simply looking at successively
     * higher shifts of b */

    for (i = 0; i < GF_FIELD_WIDTH; i++) {
        if (a & (one << i)) {
            pl ^= bl;
            pr ^= br;
        }

        bl <<= 1;
        if (br & lbit) bl ^= 1;
        br <<= 1;
    }

    /* Allen: the name of the variable "one" is no longer descriptive at this point */

    one = lbit >> 1;
    ppl = (h->prim_poly >> 2) | one;
    ppr = (h->prim_poly << (GF_FIELD_WIDTH-2));
    while (one != 0) {
        if (pl & one) {
            pl ^= ppl;

```



src/gf\_w64.c lines 361 to 420

```
    pr ^= ppr;
}
one >>= 1;
ppr >>= 1;
if (ppl & 1) ppr ^= lbit;
ppl >>= 1;
}
return pr;
}

/*
 * ELM: Use the Intel carryless multiply instruction to do very fast 64x64 multiply.
 */

static
inline
gf_val_64_t
gf_w64_clm_multiply_2 (gf_t *gf, gf_val_64_t a64, gf_val_64_t b64)
{
    gf_val_64_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL
    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      v, w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi64 (_mm_setzero_si128(), a64, 0);
    b = _mm_insert_epi64 (a, b64, 0);
    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0xffffffffULL));
    /* Do the initial multiply */

    result = _mm_clmulepi64_si128 (a, b, 0);

    /* Mask off the high order 32 bits using subtraction of the polynomial.
     * NOTE: this part requires that the polynomial have at least 32 leading 0 bits.
     */

    /* Adam: We cant include the leading one in the 64 bit pclmul,
     so we need to split up the high 8 bytes of the result into two
     parts before we multiply them with the prim_poly.*/

    v = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 0);
    w = _mm_clmulepi64_si128 (prim_poly, v, 0);
    result = _mm_xor_si128 (result, w);
    v = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 1);
    w = _mm_clmulepi64_si128 (prim_poly, v, 0);
    result = _mm_xor_si128 (result, w);

    rv = ((gf_val_64_t)_mm_extract_epi64(result, 0));
#endif
    return rv;
}

static
inline
gf_val_64_t
gf_w64_clm_multiply_4 (gf_t *gf, gf_val_64_t a64, gf_val_64_t b64)
```

*src/gf\_w64.c lines 421 to 480*

```
{
    gf_val_64_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL

    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      v, w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi64 (_mm_setzero_si128(), a64, 0);
    b = _mm_insert_epi64 (a, b64, 0);
    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0xffffffffULL));

    /* Do the initial multiply */

    result = _mm_clmulepi64_si128 (a, b, 0);

    v = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 0);
    w = _mm_clmulepi64_si128 (prim_poly, v, 0);
    result = _mm_xor_si128 (result, w);
    v = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 1);
    w = _mm_clmulepi64_si128 (prim_poly, v, 0);
    result = _mm_xor_si128 (result, w);

    v = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 0);
    w = _mm_clmulepi64_si128 (prim_poly, v, 0);
    result = _mm_xor_si128 (result, w);
    v = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 1);
    w = _mm_clmulepi64_si128 (prim_poly, v, 0);
    result = _mm_xor_si128 (result, w);

    rv = ((gf_val_64_t)_mm_extract_epi64(result, 0));
#endif
    return rv;
}

void
gf_w64_clm_multiply_region(gf_t *gf, void *src, void *dest, uint64_t val, int bytes, int xor)
{
#ifdef INTEL_SSE4_PCLMUL
    gf_internal_t *h;
    uint8_t *s8, *d8, *dtop;
    gf_region_data rd;
    __m128i v, b, m, prim_poly, c, fr, w, result;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;
    dtop = (uint8_t *) rd.d_top;
#endif
}
```



*src/gf\_w64.c lines 481 to 540*

```
v = _mm_insert_epi64(_mm_setzero_si128(), val, 0);
m = _mm_set_epi32(0, 0, 0xffffffff, 0xffffffff);
prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0xffffffffULL));

if (xor) {
    while (d8 != dtop) {
        b = _mm_load_si128((__m128i *) s8);
        result = _mm_clmulepi64_si128 (b, v, 0);
        c = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 0);
        w = _mm_clmulepi64_si128 (prim_poly, c, 0);
        result = _mm_xor_si128 (result, w);
        c = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 1);
        w = _mm_clmulepi64_si128 (prim_poly, c, 0);
        fr = _mm_xor_si128 (result, w);
        fr = _mm_and_si128 (fr, m);

        result = _mm_clmulepi64_si128 (b, v, 1);
        c = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 0);
        w = _mm_clmulepi64_si128 (prim_poly, c, 0);
        result = _mm_xor_si128 (result, w);
        c = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 1);
        w = _mm_clmulepi64_si128 (prim_poly, c, 0);
        result = _mm_xor_si128 (result, w);
        result = _mm_slli_si128 (result, 8);
        fr = _mm_xor_si128 (result, fr);
        result = _mm_load_si128((__m128i *) d8);
        fr = _mm_xor_si128 (result, fr);

        _mm_store_si128((__m128i *) d8, fr);
        d8 += 16;
        s8 += 16;
    }
} else {
    while (d8 < dtop) {
        b = _mm_load_si128((__m128i *) s8);
        result = _mm_clmulepi64_si128 (b, v, 0);
        c = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 0);
        w = _mm_clmulepi64_si128 (prim_poly, c, 0);
        result = _mm_xor_si128 (result, w);
        c = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 1);
        w = _mm_clmulepi64_si128 (prim_poly, c, 0);
        fr = _mm_xor_si128 (result, w);
        fr = _mm_and_si128 (fr, m);

        result = _mm_clmulepi64_si128 (b, v, 1);
        c = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 0);
        w = _mm_clmulepi64_si128 (prim_poly, c, 0);
        result = _mm_xor_si128 (result, w);
        c = _mm_insert_epi32 (_mm_srli_si128 (result, 8), 0, 1);
        w = _mm_clmulepi64_si128 (prim_poly, c, 0);
        result = _mm_xor_si128 (result, w);
        result = _mm_slli_si128 (result, 8);
        fr = _mm_xor_si128 (result, fr);

        _mm_store_si128((__m128i *) d8, fr);
        d8 += 16;
        s8 += 16;
    }
}
gf_do_final_region_alignment(&rd);
```

*src/gf\_w64.c lines 541 to 600*

```
#endif
}

void
gf_w64_split_4_64_lazy_multiply_region(gf_t *gf, void *src, void *dest, uint64_t val, int bytes, int xor)
{
    gf_internal_t *h;
    struct gf_split_4_64_lazy_data *ld;
    int i, j, k;
    uint64_t pp, v, s, *s64, *d64, *top;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    ld = (struct gf_split_4_64_lazy_data *) h->private;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
    gf_do_initial_region_alignment(&rd);

    if (ld->last_value != val) {
        v = val;
        for (i = 0; i < 16; i++) {
            ld->tables[i][0] = 0;
            for (j = 1; j < 16; j <= 1) {
                for (k = 0; k < j; k++) {
                    ld->tables[i][k^j] = (v ^ ld->tables[i][k]);
                }
                v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
            }
        }
    }
    ld->last_value = val;

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    top = (uint64_t *) rd.d_top;

    while (d64 != top) {
        v = (xor) ? *d64 : 0;
        s = *s64;
        i = 0;
        while (s != 0) {
            v ^= ld->tables[i][s&0xf];
            s >>= 4;
            i++;
        }
        *d64 = v;
        d64++;
        s64++;
    }
    gf_do_final_region_alignment(&rd);
}

static
inline
uint64_t
```



*src/gf\_w64.c lines 601 to 660*

```
gf_w64_split_8_8_multiply (gf_t *gf, uint64_t a64, uint64_t b64)
{
    uint64_t product, i, j, mask, tb;
    gf_internal_t *h;
    struct gf_split_8_8_data *d8;

    h = (gf_internal_t *) gf->scratch;
    d8 = (struct gf_split_8_8_data *) h->private;
    product = 0;
    mask = 0xff;

    for (i = 0; a64 != 0; i++) {
        tb = b64;
        for (j = 0; tb != 0; j++) {
            product ^= d8->tables[i+j][a64&mask][tb&mask];
            tb >>= 8;
        }
        a64 >>= 8;
    }
    return product;
}
```

```
void
gf_w64_split_8_64_lazy_multiply_region(gf_t *gf, void *src, void *dest, uint64_t val, int bytes, int xor)
{
    gf_internal_t *h;
    struct gf_split_8_64_lazy_data *ld;
    int i, j, k;
    uint64_t pp, v, s, *s64, *d64, *top;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    ld = (struct gf_split_8_64_lazy_data *) h->private;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);
    gf_do_initial_region_alignment(&rd);

    if (ld->last_value != val) {
        v = val;
        for (i = 0; i < 8; i++) {
            ld->tables[i][0] = 0;
            for (j = 1; j < 256; j <= 1) {
                for (k = 0; k < j; k++) {
                    ld->tables[i][k^j] = (v ^ ld->tables[i][k]);
                }
                v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
            }
        }
    }
    ld->last_value = val;

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    top = (uint64_t *) rd.d_top;
}
```

*src/gf\_w64.c lines 661 to 720*

```
while (d64 != top) {
    v = (xor) ? *d64 : 0;
    s = *s64;
    i = 0;
    while (s != 0) {
        v ^= ld->tables[i][s&0xff];
        s >>= 8;
        i++;
    }
    *d64 = v;
    d64++;
    s64++;
}
gf_do_final_region_alignment(&rd);
}
```

```
void
gf_w64_split_16_64_lazy_multiply_region(gf_t *gf, void *src, void *dest, uint64_t val, int bytes, int xor)
{
```

```
    gf_internal_t *h;
    struct gf_split_16_64_lazy_data *ld;
    int i, j, k;
    uint64_t pp, v, s, *s64, *d64, *top;
    gf_region_data rd;
```

```
    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }
```

```
    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;
```

```
    ld = (struct gf_split_16_64_lazy_data *) h->private;
```

```
    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);
    gf_do_initial_region_alignment(&rd);
```

```
    if (ld->last_value != val) {
        v = val;
        for (i = 0; i < 4; i++) {
            ld->tables[i][0] = 0;
            for (j = 1; j < (1<<16); j <= 1) {
                for (k = 0; k < j; k++) {
                    ld->tables[i][k^j] = (v ^ ld->tables[i][k]);
                }
                v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
            }
        }
    }
```

```
    ld->last_value = val;
```

```
    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    top = (uint64_t *) rd.d_top;
```

```
while (d64 != top) {
    v = (xor) ? *d64 : 0;
    s = *s64;
    i = 0;
    while (s != 0) {
        v ^= ld->tables[i][s&0xffff];
```



*src/gf\_w64.c lines 721 to 780*

```
        s >>= 16;
        i++;
    }
    *d64 = v;
    d64++;
    s64++;
}
gf_do_final_region_alignment(&rd);
}
```

```
static
int gf_w64_shift_init(gf_t *gf)
{
    gf->multiply.w64 = gf_w64_shift_multiply;
    gf->inverse.w64 = gf_w64_euclid;
    gf->multiply_region.w64 = gf_w64_multiply_region_from_single;
    return 1;
}
```

```
static
int gf_w64_cfm_init(gf_t *gf)
{
    gf->inverse.w64 = gf_w64_euclid;
    gf->multiply_region.w64 = gf_w64_multiply_region_from_single;
}
```

```
#if defined(INTEL_SSE4_PCLMUL)
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    if ((0xfffffffffe00000000ULL & h->prim_poly) == 0){
        gf->multiply.w64 = gf_w64_clm_multiply_2;
        gf->multiply_region.w64 = gf_w64_clm_multiply_region_from_single_2;
    }else if((0xffffe00000000000ULL & h->prim_poly) == 0){
        gf->multiply.w64 = gf_w64_clm_multiply_4;
        gf->multiply_region.w64 = gf_w64_clm_multiply_region_from_single_4;
    } else {
        return 0;
    }
    return 1;
#endif

    return 0;
}
```

```
static
void
gf_w64_group_set_shift_tables(uint64_t *shift, uint64_t val, gf_internal_t *h)
{
    int i;
    uint64_t j;
    uint64_t one = 1;
    int g_s;

    g_s = h->arg1;
    shift[0] = 0;

    for (i = 1; i < (1 << g_s); i <= 1) {
        for (j = 0; j < i; j++) shift[i|j] = shift[j]^val;
        if (val & (one << 63)) {
```

*src/gf\_w64.c lines 781 to 840*

```
        val <=< 1;
        val ^= h->prim_poly;
    } else {
        val <=< 1;
    }
}

static
inline
gf_val_64_t
gf_w64_group_multiply(gf_t *gf, gf_val_64_t a, gf_val_64_t b)
{
    uint64_t top, bot, mask, tp;
    int g_s, g_r, lshift, rshift;
    struct gf_w64_group_data *gd;

    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    g_s = h->arg1;
    g_r = h->arg2;
    gd = (struct gf_w64_group_data *) h->private;
    gf_w64_group_set_shift_tables(gd->shift, b, h);

    mask = ((1 << g_s) - 1);
    top = 0;
    bot = gd->shift[a&mask];
    a >>= g_s;

    if (a == 0) return bot;
    lshift = 0;
    rshift = 64;

    do {
        /* Shifting out is straightfoward */
        lshift += g_s;
        rshift -= g_s;
        tp = gd->shift[a&mask];
        top ^= (tp >> rshift);
        bot ^= (tp << lshift);
        a >>= g_s;
    } while (a != 0);

    /* Reducing is a bit gross, because I don't zero out the index bits of top.
       The reason is that we throw top away. Even better, that last (tp >> rshift)
       is going to be ignored, so it doesn't matter how (tp >> 64) is implemented. */

    lshift = ((lshift-1) / g_r) * g_r;
    rshift = 64 - lshift;
    mask = (1 << g_r) - 1;
    while (lshift >= 0) {
        tp = gd->reduce[(top >> lshift) & mask];
        top ^= (tp >> rshift);
        bot ^= (tp << lshift);
        lshift -= g_r;
        rshift += g_r;
    }

    return bot;
}

static
```



*src/gf\_w64.c lines 841 to 900*

```
void gf_w64_group_multiply_region(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int xor)
{
    int i, fzb;
    uint64_t a64, smask, rmask, top, bot, tp;
    int lshift, rshift, g_s, g_r;
    gf_region_data rd;
    uint64_t *s64, *d64, *dtop;
    struct gf_w64_group_data *gd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gd = (struct gf_w64_group_data *) h->private;
    g_s = h->arg1;
    g_r = h->arg2;
    gf_w64_group_set_shift_tables(gd->shift, val, h);

    for (i = 63; !(val & (1ULL << i)); i--) ;
    i += g_s;

    /* i is the bit position of the first zero bit in any element of
       gd->shift[] */

    if (i > 64) i = 64;

    fzb = i;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);

    gf_do_initial_region_alignment(&rd);

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    dtop = (uint64_t *) rd.d_top;

    smask = (1 << g_s) - 1;
    rmask = (1 << g_r) - 1;

    while (d64 < dtop) {
        a64 = *s64;

        top = 0;
        bot = gd->shift[a64&smask];
        a64 >>= g_s;
        i = fzb;

        if (a64 != 0) {
            lshift = 0;
            rshift = 64;

            do {
                lshift += g_s;
                rshift -= g_s;
                tp = gd->shift[a64&smask];
                top ^= (tp >> rshift);
                bot ^= (tp << lshift);
                a64 >>= g_s;
            } while (a64 != 0);
            i += lshift;
        }
    }
}
```

*src/gf\_w64.c lines 901 to 960*

```
        lshift = ((i-64-1) / g_r) * g_r;
        rshift = 64 - lshift;
        while (lshift >= 0) {
            tp = gd->reduce[(top >> lshift) & rmask];
            top ^= (tp >> rshift);
            bot ^= (tp << lshift);
            lshift -= g_r;
            rshift += g_r;
        }
    }

    if (xor) bot ^= *d64;
    *d64 = bot;
    d64++;
    s64++;
}
gf_do_final_region_alignment(&rd);
}
```

```
static
inline
gf_val_64_t
gf_w64_group_s_equals_r_multiply(gf_t *gf, gf_val_64_t a, gf_val_64_t b)
{
    int leftover, rs;
    uint64_t p, l, ind, a64;
    int bits_left;
    int g_s;

    struct gf_w64_group_data *gd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    g_s = h->arg1;

    gd = (struct gf_w64_group_data *) h->private;
    gf_w64_group_set_shift_tables(gd->shift, b, h);

    leftover = 64 % g_s;
    if (leftover == 0) leftover = g_s;

    rs = 64 - leftover;
    a64 = a;
    ind = a64 >> rs;
    a64 <<= leftover;
    p = gd->shift[ind];

    bits_left = rs;
    rs = 64 - g_s;

    while (bits_left > 0) {
        bits_left -= g_s;
        ind = a64 >> rs;
        a64 <<= g_s;
        l = p >> rs;
        p = (gd->shift[ind] ^ gd->reduce[l] ^ (p << g_s));
    }
    return p;
}
```

static



*src/gf\_w64.c lines 961 to 1020*

```
void gf_w64_group_s_equals_r_multiply_region(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int xor)
{
    int leftover, rs;
    uint64_t p, l, ind, a64;
    int bits_left;
    int g_s;
    gf_region_data rd;
    uint64_t *s64, *d64, *top;
    struct gf_w64_group_data *gd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gd = (struct gf_w64_group_data *) h->private;
    g_s = h->arg1;
    gf_w64_group_set_shift_tables(gd->shift, val, h);

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 4);
    gf_do_initial_region_alignment(&rd);

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    top = (uint64_t *) rd.d_top;

    leftover = 64 % g_s;
    if (leftover == 0) leftover = g_s;

    while (d64 < top) {
        rs = 64 - leftover;
        a64 = *s64;
        ind = a64 >> rs;
        a64 <=< leftover;
        p = gd->shift[ind];

        bits_left = rs;
        rs = 64 - g_s;

        while (bits_left > 0) {
            bits_left -= g_s;
            ind = a64 >> rs;
            a64 <=< g_s;
            l = p >> rs;
            p = (gd->shift[ind] ^ gd->reduce[l] ^ (p << g_s));
        }
        if (xor) p ^= *d64;
        *d64 = p;
        d64++;
        s64++;
    }
    gf_do_final_region_alignment(&rd);
}
```

```
static
int gf_w64_group_init(gf_t *gf)
{
    uint64_t i, j, p, index;
    struct gf_w64_group_data *gd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
```

*src/gf\_w64.c lines 1021 to 1080*

```
int g_r, g_s;

g_s = h->arg1;
g_r = h->arg2;

gd = (struct gf_w64_group_data *) h->private;
gd->shift = (uint64_t *) (&(gd->memory));
gd->reduce = gd->shift + (1 << g_s);

gd->reduce[0] = 0;
for (i = 0; i < (1 << g_r); i++) {
    p = 0;
    index = 0;
    for (j = 0; j < g_r; j++) {
        if (i & (1 << j)) {
            p ^= (h->prim_poly << j);
            index ^= (1 << j);
            if (j > 0) index ^= (h->prim_poly >> (64-j));
        }
    }
    gd->reduce[index] = p;
}

if (g_s == g_r) {
    gf->multiply.w64 = gf_w64_group_s_equals_r_multiply;
    gf->multiply_region.w64 = gf_w64_group_s_equals_r_multiply_region;
} else {
    gf->multiply.w64 = gf_w64_group_multiply;
    gf->multiply_region.w64 = gf_w64_group_multiply_region;
}
gf->divide.w64 = NULL;
gf->inverse.w64 = gf_w64_euclid;

return 1;
}

static
gf_val_64_t gf_w64_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    uint64_t *r64, rv;

    r64 = (uint64_t *) start;
    rv = r64[index];
    return rv;
}

static
gf_val_64_t gf_w64_composite_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    int sub_size;
    gf_internal_t *h;
    uint8_t *r8, *top;
    uint64_t a, b, *r64;
    gf_region_data rd;

    h = (gf_internal_t *) gf->scratch;
    gf_set_region_data(&rd, gf, start, start, bytes, 0, 0, 32);
    r64 = (uint64_t *) start;
    if (r64 + index < (uint64_t *) rd.d_start) return r64[index];
    if (r64 + index >= (uint64_t *) rd.d_top) return r64[index];
}
```



*src/gf\_w64.c lines 1081 to 1140*

```
    index -= (((uint64_t *) rd.d_start) - r64);
    r8 = (uint8_t *) rd.d_start;
    top = (uint8_t *) rd.d_top;
    sub_size = (top-r8)/2;

    a = h->base_gf->extract_word.w32(h->base_gf, r8, sub_size, index);
    b = h->base_gf->extract_word.w32(h->base_gf, r8+sub_size, sub_size, index);
    return (a | ((uint64_t)b << 32));
}
```

```
static
gf_val_64_t gf_w64_split_extract_word(gf_t *gf, void *start, int bytes, int index)
```

```
{
    int i;
    uint64_t *r64, rv;
    uint8_t *r8;
    gf_region_data rd;

    gf_set_region_data(&rd, gf, start, start, bytes, 0, 0, 128);
    r64 = (uint64_t *) start;
    if (r64 + index < (uint64_t *) rd.d_start) return r64[index];
    if (r64 + index >= (uint64_t *) rd.d_top) return r64[index];
    index -= (((uint64_t *) rd.d_start) - r64);
    r8 = (uint8_t *) rd.d_start;
    r8 += ((index & 0xffffffff0)*8);
    r8 += (index & 0xf);
    r8 += 112;
    rv = 0;
    for (i = 0; i < 8; i++) {
        rv <<= 8;
        rv |= *r8;
        r8 -= 16;
    }
    return rv;
}
```

```
static
inline
gf_val_64_t
gf_w64_bytwo_b_multiply (gf_t *gf, gf_val_64_t a, gf_val_64_t b)
```

```
{
    uint64_t prod, pp, bmask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;
```

```
    prod = 0;
    bmask = 0x8000000000000000ULL;
```

```
    while (1) {
        if (a & 1) prod ^= b;
        a >>= 1;
        if (a == 0) return prod;
        if (b & bmask) {
            b = ((b << 1) ^ pp);
        } else {
            b <<= 1;
        }
    }
}
```

*src/gf\_w64.c lines 1141 to 1200*

```
}

static
inline
gf_val_64_t
gf_w64_bytwo_p_multiply (gf_t *gf, gf_val_64_t a, gf_val_64_t b)
{
    uint64_t prod, pp, pmask, amask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;

    /* changed from declare then shift to just declare.*/

    pmask = 0x8000000000000000ULL;
    amask = 0x8000000000000000ULL;

    while (amask != 0) {
        if (prod & pmask) {
            prod = ((prod << 1) ^ pp);
        } else {
            prod <<= 1;
        }
        if (a & amask) prod ^= b;
        amask >>= 1;
    }
    return prod;
}
```

```
static
void
gf_w64_bytwo_p_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int xor)
{
    uint64_t *s64, *d64, ta, prod, amask, pmask, pp;
    gf_region_data rd;
    gf_internal_t *h;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
    gf_do_initial_region_alignment(&rd);

    h = (gf_internal_t *) gf->scratch;

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    pmask = 0x80000000;
    pmask <<= 32;
    pp = h->prim_poly;

    if (xor) {
        while (s64 < (uint64_t *) rd.s_top) {
            prod = 0;
            amask = pmask;
            ta = *s64;
            while (amask != 0) {
```



*src/gf\_w64.c lines 1201 to 1260*

```
        prod = (prod & pmask) ? ((prod << 1) ^ pp) : (prod << 1);
        if (val & amask) prod ^= ta;
        amask >>= 1;
    }
    *d64 ^= prod;
    d64++;
    s64++;
}
} else {
    while (s64 < (uint64_t *) rd.s_top) {
        prod = 0;
        amask = pmask;
        ta = *s64;
        while (amask != 0) {
            prod = (prod & pmask) ? ((prod << 1) ^ pp) : (prod << 1);
            if (val & amask) prod ^= ta;
            amask >>= 1;
        }
        *d64 = prod;
        d64++;
        s64++;
    }
}
gf_do_final_region_alignment(&rd);
}
```

```
static
void
gf_w64_bytwo_b_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int xor)
{
    uint64_t *s64, *d64, ta, tb, prod, bmask, pp;
    gf_region_data rd;
    gf_internal_t *h;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
    gf_do_initial_region_alignment(&rd);

    h = (gf_internal_t *) gf->scratch;

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    bmask = 0x80000000;
    bmask <=> 32;
    pp = h->prim_poly;

    if (xor) {
        while (s64 < (uint64_t *) rd.s_top) {
            prod = 0;
            tb = val;
            ta = *s64;
            while (1) {
                if (tb & 1) prod ^= ta;
                tb >>= 1;
                if (tb == 0) break;
                ta = (ta & bmask) ? ((ta << 1) ^ pp) : (ta << 1);
            }
            *d64 ^= prod;
        }
    }
}
```

*src/gf\_w64.c lines 1261 to 1320*

```
        d64++;
        s64++;
    }
} else {
    while (s64 < (uint64_t *) rd.s_top) {
        prod = 0;
        tb = val;
        ta = *s64;
        while (1) {
            if (tb & 1) prod ^= ta;
            tb >>= 1;
            if (tb == 0) break;
            ta = (ta & bmask) ? ((ta << 1) ^ pp) : (ta << 1);
        }
        *d64 = prod;
        d64++;
        s64++;
    }
}
gf_do_final_region_alignment(&rd);
}

#define SSE_AB2(pp, m1, m2, va, t1, t2) {\
    t1 = _mm_and_si128(_mm_slli_epi64(va, 1), m1); \
    t2 = _mm_and_si128(va, m2); \
    t2 = _mm_sub_epi64(_mm_slli_epi64(t2, 1), _mm_srli_epi64(t2, (GF_FIELD_WIDTH-1))); \
    va = _mm_xor_si128(t1, _mm_and_si128(t2, pp)); }

#define BYTWO_P_ONESTEP {\
    SSE_AB2(pp, m1, m2, prod, t1, t2); \
    t1 = _mm_and_si128(v, one); \
    t1 = _mm_sub_epi64(t1, one); \
    t1 = _mm_and_si128(t1, ta); \
    prod = _mm_xor_si128(prod, t1); \
    v = _mm_srli_epi64(v, 1); }
```

```
void gf_w64_bytwo_p_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int xor)
```

```
{
#ifdef INTEL_SSE2
    int i;
    uint8_t *s8, *d8;
    uint64_t vrev, one64;
    uint64_t amask;
    __m128i pp, m1, m2, ta, prod, t1, t2, tp, one, v;
    gf_region_data rd;
    gf_internal_t *h;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    h = (gf_internal_t *) gf->scratch;
    one64 = 1;
    vrev = 0;
    for (i = 0; i < 64; i++) {
        vrev <=> 1;
        if (!(val & (one64 << i))) vrev |= 1;
    }
#endif
}
```



*src/gf w64.c lines 1321 to 1380*

}

```
s8 = (uint8_t *) rd.s_start;
d8 = (uint8_t *) rd.d_start;
```

```
amask = -1;
amask ^= 1;
pp = _mm_set1_epi64x(h->prim_poly);
m1 = _mm_set1_epi64x(amask);
m2 = _mm_set1_epi64x(one64 << 63);
one = _mm_set1_epi64x(1);
```

[illegible]

```
gf_do_final_region_alignment(&rd);
```

```
#endif
```

$$\left. \begin{array}{l} \text{---} \\ \text{---} \end{array} \right\}$$

```
#ifndef INTEL SSE2
```

```
static
```

```
void
```

```
gf_w64_bytwo_b_sse_region_2_xor(gf_region_data *rd)
```

$$\{$$

```
uint64_t one64, amask;
uint8_t *d8, *s8;
__m128i pp, m1, m2, t1, t2, va, vb;
qf_internal t *h;
```

```
s8 = (uint8_t *) rd->s_start;
d8 = (uint8_t *) rd->d_start;
```

```
h = (gf_internal_t *) rd->gf->scratch;
one64 = 1;
amask = -1;
amask ^= 1;
pp = _mm_set1_epi64x(h->prim_poly);
ml = _mm_set1_epi64x(amask);
```

*src/gf\_w64.c lines 1381 to 1440*

```
m2 = _mm_set1_epi64x(one64 << 63);

while (d8 < (uint8_t *) rd->d_top) {
    va = _mm_load_si128 ((__m128i *) (s8));
    SSE_AB2(pp, m1, m2, va, t1, t2);
    vb = _mm_load_si128 ((__m128i *) (d8));
    vb = _mm_xor_si128(vb, va);
    _mm_store_si128((__m128i *)d8, vb);
    d8 += 16;
    s8 += 16;
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w64_bytwo_b_sse_region_2_noxor(gf_region_data *rd)
{
    uint64_t one64, amask;
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va;
    gf_internal_t *h;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    h = (gf_internal_t *) rd->gf->scratch;
    one64 = 1;
    amask = -1;
    amask ^= 1;
    pp = _mm_set1_epi64x(h->prim_poly);
    m1 = _mm_set1_epi64x(amask);
    m2 = _mm_set1_epi64x(one64 << 63);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        SSE_AB2(pp, m1, m2, va, t1, t2);
        _mm_store_si128((__m128i *)d8, va);
        d8 += 16;
        s8 += 16;
    }
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w64_bytwo_b_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int xor)
{
    uint64_t itb, amask, one64;
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va, vb;
    gf_region_data rd;
    gf_internal_t *h;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
```



*src/gf\_w64.c lines 1441 to 1500*

```
gf_do_initial_region_alignment(&rd);

if (val == 2) {
    if (xor) {
        gf_w64_bytwo_b_sse_region_2_xor(&rd);
    } else {
        gf_w64_bytwo_b_sse_region_2_noxor(&rd);
    }
    gf_do_final_region_alignment(&rd);
    return;
}

s8 = (uint8_t *) rd.s_start;
d8 = (uint8_t *) rd.d_start;
h = (gf_internal_t *) gf->scratch;

one64 = 1;
amask = -1;
amask ^= 1;
pp = _mm_set1_epi64x(h->prim_poly);
m1 = _mm_set1_epi64x(amask);
m2 = _mm_set1_epi64x(one64 << 63);

while (d8 < (uint8_t *) rd.d_top) {
    va = _mm_load_si128 ((__m128i *) (s8));
    vb = (!xor) ? _mm_setzero_si128() : _mm_load_si128 ((__m128i *) (d8));
    itb = val;
    while (1) {
        if (itb & 1) vb = _mm_xor_si128(vb, va);
        itb >>= 1;
        if (itb == 0) break;
        SSE_AB2(pp, m1, m2, va, t1, t2);
    }
    _mm_store_si128((__m128i *) d8, vb);
    d8 += 16;
    s8 += 16;
}

gf_do_final_region_alignment(&rd);
}
#endif
```

```
static
int gf_w64_bytwo_init(gf_t *gf)
{
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    if (h->mult_type == GF_MULT_BYTWO_p) {
        gf->multiply.w64 = gf_w64_bytwo_p_multiply;
#ifdef INTEL_SSE2
        if (h->region_type & GF_REGION_NOSSE)
            gf->multiply_region.w64 = gf_w64_bytwo_p_nosse_multiply_region;
        else
            gf->multiply_region.w64 = gf_w64_bytwo_p_sse_multiply_region;
#else
        gf->multiply_region.w64 = gf_w64_bytwo_p_nosse_multiply_region;
        if (h->region_type & GF_REGION_SSE)

```

*src/gf\_w64.c lines 1501 to 1560*

```
        return 0;
    #endif
} else {
    gf->multiply.w64 = gf_w64_bytwo_b_multiply;
    #ifdef INTEL_SSE2
        if (h->region_type & GF_REGION_NOSSE)
            gf->multiply_region.w64 = gf_w64_bytwo_b_nosse_multiply_region;
        else
            gf->multiply_region.w64 = gf_w64_bytwo_b_sse_multiply_region;
    #else
        gf->multiply_region.w64 = gf_w64_bytwo_b_nosse_multiply_region;
        if(h->region_type & GF_REGION_SSE)
            return 0;
    #endif
}
gf->inverse.w64 = gf_w64_euclid;
return 1;
}
```

```
static
gf_val_64_t
gf_w64_composite_multiply(gf_t *gf, gf_val_64_t a, gf_val_64_t b)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint32_t b0 = b & 0x00000000ffffffff;
    uint32_t b1 = (b & 0xffffffff00000000) >> 32;
    uint32_t a0 = a & 0x00000000ffffffff;
    uint32_t a1 = (a & 0xffffffff00000000) >> 32;
    uint32_t a1b1;

    a1b1 = base_gf->multiply.w32(base_gf, a1, b1);

    return ((uint64_t) (base_gf->multiply.w32(base_gf, a0, b0) ^ a1b1) |
            ((uint64_t) (base_gf->multiply.w32(base_gf, a1, b0) ^
                base_gf->multiply.w32(base_gf, a0, b1) ^
                base_gf->multiply.w32(base_gf, a1b1, h->prim_poly)) << 32));
}
```

```
/*
 * Composite field division trick (explained in 2007 tech report)
 *
 * Compute  $a / b = a \cdot b^{-1}$ , where  $p(x) = x^2 + sx + 1$ 
 *
 * let  $c = b^{-1}$ 
 *
 *  $c \cdot b = (s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1)x + (b_1 c_1 + b_0 c_0)$ 
 *
 * want  $(s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1) = 0$  and  $(b_1 c_1 + b_0 c_0) = 1$ 
 *
 * let  $d = b_1 c_1$  and  $d+1 = b_0 c_0$ 
 *
 * solve  $s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1 = 0$ 
 *
 * solution:  $d = (b_1 b_0^{-1}) (b_1 b_0^{-1} + b_0 b_1^{-1} + s)^{-1}$ 
 *
 *  $c_0 = (d+1) b_0^{-1}$ 
 *  $c_1 = d \cdot b_1^{-1}$ 
 *
```



*src/gf\_w64.c lines 1561 to 1620*

```
* a / b = a * c
*/

static
gf_val_64_t
gf_w64_composite_inverse(gf_t *gf, gf_val_64_t a)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint32_t a0 = a & 0x00000000ffffffff;
    uint32_t a1 = (a & 0xffffffff00000000) >> 32;
    uint32_t c0, c1, d, tmp;
    uint64_t c;
    uint32_t a0inv, a1inv;

    if (a0 == 0) {
        a1inv = base_gf->inverse.w32(base_gf, a1);
        c0 = base_gf->multiply.w32(base_gf, a1inv, h->prim_poly);
        c1 = a1inv;
    } else if (a1 == 0) {
        c0 = base_gf->inverse.w32(base_gf, a0);
        c1 = 0;
    } else {
        a1inv = base_gf->inverse.w32(base_gf, a1);
        a0inv = base_gf->inverse.w32(base_gf, a0);

        d = base_gf->multiply.w32(base_gf, a1, a0inv);

        tmp = (base_gf->multiply.w32(base_gf, a1, a0inv) ^ base_gf->multiply.w32(base_gf, a0, a1inv) ^ h->prim_poly);
        tmp = base_gf->inverse.w32(base_gf, tmp);

        d = base_gf->multiply.w32(base_gf, d, tmp);

        c0 = base_gf->multiply.w32(base_gf, (d^1), a0inv);
        c1 = base_gf->multiply.w32(base_gf, d, a1inv);
    }

    c = c0 | ((uint64_t)c1 << 32);

    return c;
}

static
void
gf_w64_composite_multiply_region(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int xor)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint32_t b0 = val & 0x00000000ffffffff;
    uint32_t b1 = (val & 0xffffffff00000000) >> 32;
    uint64_t *s64, *d64;
    uint64_t *top;
    uint64_t a0, a1, a1b1;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);

    s64 = rd.s_start;
    d64 = rd.d_start;
```

*src/gf\_w64.c lines 1621 to 1680*

```
top = rd.d_top;

if (xor) {
    while (d64 < top) {
        a0 = *s64 & 0x00000000ffffffff;
        a1 = (*s64 & 0xffffffff00000000) >> 32;
        alb1 = base_gf->multiply.w32(base_gf, a1, b1);

        *d64 ^= ((uint64_t)(base_gf->multiply.w32(base_gf, a0, b0) ^ alb1) |
                ((uint64_t)(base_gf->multiply.w32(base_gf, a1, b0) ^
                base_gf->multiply.w32(base_gf, a0, b1) ^
                base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 32));

        s64++;
        d64++;
    }
} else {
    while (d64 < top) {
        a0 = *s64 & 0x00000000ffffffff;
        a1 = (*s64 & 0xffffffff00000000) >> 32;
        alb1 = base_gf->multiply.w32(base_gf, a1, b1);

        *d64 = ((base_gf->multiply.w32(base_gf, a0, b0) ^ alb1) |
                ((uint64_t)(base_gf->multiply.w32(base_gf, a1, b0) ^
                base_gf->multiply.w32(base_gf, a0, b1) ^
                base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 32));

        s64++;
        d64++;
    }
}
}

static
void
gf_w64_composite_multiply_region_alt(gf_t *gf, void *src, void *dest, gf_val_64_t val, int bytes, int xor)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    gf_val_32_t val0 = val & 0x00000000ffffffff;
    gf_val_32_t val1 = (val & 0xffffffff00000000) >> 32;
    uint8_t *slow, *shigh;
    uint8_t *dlow, *dhigh, *top;
    int sub_reg_size;
    gf_region_data rd;

    if (!xor) {
        memset(dest, 0, bytes);
    }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 32);
    gf_do_initial_region_alignment(&rd);

    slow = (uint8_t *) rd.s_start;
    dlow = (uint8_t *) rd.d_start;
    top = (uint8_t *) rd.d_top;
    sub_reg_size = (top - dlow) / 2;
    shigh = slow + sub_reg_size;
    dhigh = dlow + sub_reg_size;

    base_gf->multiply_region.w32(base_gf, slow, dlow, val0, sub_reg_size, xor);
    base_gf->multiply_region.w32(base_gf, shigh, dlow, val1, sub_reg_size, 1);
}
```



*src/gf\_w64.c lines 1681 to 1740*

```
base_gf->multiply_region.w32(base_gf, slow, dhigh, vall, sub_reg_size, xor);
base_gf->multiply_region.w32(base_gf, shigh, dhigh, val0, sub_reg_size, 1);
base_gf->multiply_region.w32(base_gf, shigh, dhigh,
    base_gf->multiply.w32(base_gf, h->prim_poly, vall), sub_reg_size, 1);

gf_do_final_region_alignment(&rd);
}
```

```
static
int gf_w64_composite_init(gf_t *gf)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;

    if (h->region_type & GF_REGION_ALTMAP) {
        gf->multiply_region.w64 = gf_w64_composite_multiply_region_alt;
    } else {
        gf->multiply_region.w64 = gf_w64_composite_multiply_region;
    }

    gf->multiply.w64 = gf_w64_composite_multiply;
    gf->divide.w64 = NULL;
    gf->inverse.w64 = gf_w64_composite_inverse;

    return 1;
}
```

```
#ifdef INTEL_SSSE3
static
void
gf_w64_split_4_64_lazy_sse_altmap_multiply_region(gf_t *gf, void *src, void *dest, uint64_t val, int bytes, int xor)
{
    gf_internal_t *h;
    int i, j, k;
    uint64_t pp, v, *s64, *d64, *top;
    __m128i si, tables[16][8], p[8], v0, mask1;
    struct gf_split_4_64_lazy_data *ld;
    uint8_t btable[16];
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 128);
    gf_do_initial_region_alignment(&rd);

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;
    top = (uint64_t *) rd.d_top;

    ld = (struct gf_split_4_64_lazy_data *) h->private;

    v = val;
    for (i = 0; i < 16; i++) {
        ld->tables[i][0] = 0;
        for (j = 1; j < 16; j <= 1) {
```

*src/gf\_w64.c lines 1741 to 1800*

```
    for (k = 0; k < j; k++) {
        ld->tables[i][k^j] = (v ^ ld->tables[i][k]);
    }
    v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
}
for (j = 0; j < 8; j++) {
    for (k = 0; k < 16; k++) {
        btable[k] = (uint8_t) ld->tables[i][k];
        ld->tables[i][k] >>= 8;
    }
    tables[i][j] = _mm_loadu_si128((__m128i *) btable);
}
}

mask1 = _mm_set1_epi8(0xf);

while (d64 != top) {

    if (xor) {
        for (i = 0; i < 8; i++) p[i] = _mm_load_si128 ((__m128i *) (d64+i*2));
    } else {
        for (i = 0; i < 8; i++) p[i] = _mm_setzero_si128();
    }
    i = 0;
    for (k = 0; k < 8; k++) {
        v0 = _mm_load_si128((__m128i *) s64);
        /* MM_PRINT8("v", v0); */
        s64 += 2;

        si = _mm_and_si128(v0, mask1);

        for (j = 0; j < 8; j++) {
            p[j] = _mm_xor_si128(p[j], _mm_shuffle_epi8(tables[i][j], si));
        }
        i++;
        v0 = _mm_srli_epi32(v0, 4);
        si = _mm_and_si128(v0, mask1);
        for (j = 0; j < 8; j++) {
            p[j] = _mm_xor_si128(p[j], _mm_shuffle_epi8(tables[i][j], si));
        }
        i++;
    }
    for (i = 0; i < 8; i++) {
        /* MM_PRINT8("v", p[i]); */
        _mm_store_si128((__m128i *) d64, p[i]);
        d64 += 2;
    }
}
gf_do_final_region_alignment(&rd);
}
#endif

#ifdef INTEL_SSE4
static
void
gf_w64_split_4_64_lazy_sse_multiply_region(gf_t *gf, void *src, void *dest, uint64_t val, int bytes, int xor)
{
    gf_internal_t *h;
    int i, j, k;
    uint64_t pp, v, *s64, *d64, *top;
```



*src/gf\_w64.c lines 1801 to 1860*

```
__m128i si, tables[16][8], p[8], st[8], mask1, mask8, mask16, t1;
struct gf_split_4_64_lazy_data *ld;
uint8_t btable[16];
gf_region_data rd;

if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

h = (gf_internal_t *) gf->scratch;
pp = h->prim_poly;

gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 128);
gf_do_initial_region_alignment(&rd);

s64 = (uint64_t *) rd.s_start;
d64 = (uint64_t *) rd.d_start;
top = (uint64_t *) rd.d_top;

ld = (struct gf_split_4_64_lazy_data *) h->private;

v = val;
for (i = 0; i < 16; i++) {
    ld->tables[i][0] = 0;
    for (j = 1; j < 16; j <= 1) {
        for (k = 0; k < j; k++) {
            ld->tables[i][k^j] = (v ^ ld->tables[i][k]);
        }
        v = (v & GF_FIRST_BIT) ? ((v << 1) ^ pp) : (v << 1);
    }
    for (j = 0; j < 8; j++) {
        for (k = 0; k < 16; k++) {
            btable[k] = (uint8_t) ld->tables[i][k];
            ld->tables[i][k] >>= 8;
        }
        tables[i][j] = _mm_loadu_si128((__m128i *) btable);
    }
}

mask1 = _mm_set1_epi8(0xf);
mask8 = _mm_set1_epi16(0xff);
mask16 = _mm_set1_epi32(0xffff);

while (d64 != top) {

    for (i = 0; i < 8; i++) p[i] = _mm_setzero_si128();

    for (k = 0; k < 8; k++) {
        st[k] = _mm_load_si128((__m128i *) s64);
        s64 += 2;
    }

    for (k = 0; k < 4; k++) {
        st[k] = _mm_shuffle_epi32(st[k], _MM_SHUFFLE(3,1,2,0));
        st[k+4] = _mm_shuffle_epi32(st[k+4], _MM_SHUFFLE(2,0,3,1));
        t1 = _mm_blend_epi16(st[k], st[k+4], 0xf0);
        st[k] = _mm_srli_si128(st[k], 8);
        st[k+4] = _mm_slli_si128(st[k+4], 8);
        st[k+4] = _mm_blend_epi16(st[k], st[k+4], 0xf0);
        st[k] = t1;
    }
}
```

*src/gf\_w64.c lines 1861 to 1920*

```
/*
    printf("After pack pass 1\n");
    for (k = 0; k < 8; k++) {
        MM_PRINT8("v", st[k]);
    }
    printf("\n");
*/

t1 = _mm_packus_epi32(_mm_and_si128(st[0], mask16), _mm_and_si128(st[2], mask16));
st[2] = _mm_packus_epi32(_mm_srli_epi32(st[0], 16), _mm_srli_epi32(st[2], 16));
st[0] = t1;
t1 = _mm_packus_epi32(_mm_and_si128(st[1], mask16), _mm_and_si128(st[3], mask16));
st[3] = _mm_packus_epi32(_mm_srli_epi32(st[1], 16), _mm_srli_epi32(st[3], 16));
st[1] = t1;
t1 = _mm_packus_epi32(_mm_and_si128(st[4], mask16), _mm_and_si128(st[6], mask16));
st[6] = _mm_packus_epi32(_mm_srli_epi32(st[4], 16), _mm_srli_epi32(st[6], 16));
st[4] = t1;
t1 = _mm_packus_epi32(_mm_and_si128(st[5], mask16), _mm_and_si128(st[7], mask16));
st[7] = _mm_packus_epi32(_mm_srli_epi32(st[5], 16), _mm_srli_epi32(st[7], 16));
st[5] = t1;

/*
    printf("After pack pass 2\n");
    for (k = 0; k < 8; k++) {
        MM_PRINT8("v", st[k]);
    }
    printf("\n");
*/

t1 = _mm_packus_epil6(_mm_and_si128(st[0], mask8), _mm_and_si128(st[1], mask8));
st[1] = _mm_packus_epil6(_mm_srli_epil6(st[0], 8), _mm_srli_epil6(st[1], 8));
st[0] = t1;
t1 = _mm_packus_epil6(_mm_and_si128(st[2], mask8), _mm_and_si128(st[3], mask8));
st[3] = _mm_packus_epil6(_mm_srli_epil6(st[2], 8), _mm_srli_epil6(st[3], 8));
st[2] = t1;
t1 = _mm_packus_epil6(_mm_and_si128(st[4], mask8), _mm_and_si128(st[5], mask8));
st[5] = _mm_packus_epil6(_mm_srli_epil6(st[4], 8), _mm_srli_epil6(st[5], 8));
st[4] = t1;
t1 = _mm_packus_epil6(_mm_and_si128(st[6], mask8), _mm_and_si128(st[7], mask8));
st[7] = _mm_packus_epil6(_mm_srli_epil6(st[6], 8), _mm_srli_epil6(st[7], 8));
st[6] = t1;

/*
    printf("After final pack pass 2\n");
    for (k = 0; k < 8; k++) {
        MM_PRINT8("v", st[k]);
    }
*/

i = 0;
for (k = 0; k < 8; k++) {
    si = _mm_and_si128(st[k], mask1);

    for (j = 0; j < 8; j++) {
        p[j] = _mm_xor_si128(p[j], _mm_shuffle_epi8(tables[i][j], si));
    }
    i++;
    st[k] = _mm_srli_epi32(st[k], 4);
    si = _mm_and_si128(st[k], mask1);
    for (j = 0; j < 8; j++) {
        p[j] = _mm_xor_si128(p[j], _mm_shuffle_epi8(tables[i][j], si));
    }
}
```



*src/gf\_w64.c lines 1921 to 1980*

```
    }
    i++;
}

t1 = _mm_unpacklo_epi8(p[0], p[1]);
p[1] = _mm_unpackhi_epi8(p[0], p[1]);
p[0] = t1;
t1 = _mm_unpacklo_epi8(p[2], p[3]);
p[3] = _mm_unpackhi_epi8(p[2], p[3]);
p[2] = t1;
t1 = _mm_unpacklo_epi8(p[4], p[5]);
p[5] = _mm_unpackhi_epi8(p[4], p[5]);
p[4] = t1;
t1 = _mm_unpacklo_epi8(p[6], p[7]);
p[7] = _mm_unpackhi_epi8(p[6], p[7]);
p[6] = t1;

/*
printf("After unpack pass 1:\n");
for (i = 0; i < 8; i++) {
    MM_PRINT8("v", p[i]);
}
*/

t1 = _mm_unpacklo_epi16(p[0], p[2]);
p[2] = _mm_unpackhi_epi16(p[0], p[2]);
p[0] = t1;
t1 = _mm_unpacklo_epi16(p[1], p[3]);
p[3] = _mm_unpackhi_epi16(p[1], p[3]);
p[1] = t1;
t1 = _mm_unpacklo_epi16(p[4], p[6]);
p[6] = _mm_unpackhi_epi16(p[4], p[6]);
p[4] = t1;
t1 = _mm_unpacklo_epi16(p[5], p[7]);
p[7] = _mm_unpackhi_epi16(p[5], p[7]);
p[5] = t1;

/*
printf("After unpack pass 2:\n");
for (i = 0; i < 8; i++) {
    MM_PRINT8("v", p[i]);
}
*/

t1 = _mm_unpacklo_epi32(p[0], p[4]);
p[4] = _mm_unpackhi_epi32(p[0], p[4]);
p[0] = t1;
t1 = _mm_unpacklo_epi32(p[1], p[5]);
p[5] = _mm_unpackhi_epi32(p[1], p[5]);
p[1] = t1;
t1 = _mm_unpacklo_epi32(p[2], p[6]);
p[6] = _mm_unpackhi_epi32(p[2], p[6]);
p[2] = t1;
t1 = _mm_unpacklo_epi32(p[3], p[7]);
p[7] = _mm_unpackhi_epi32(p[3], p[7]);
p[3] = t1;

if (xor) {
    for (i = 0; i < 8; i++) {
        t1 = _mm_load_si128((__m128i *) d64);
```

*src/gf\_w64.c lines 1981 to 2040*

```
        _mm_store_si128((__m128i *) d64, _mm_xor_si128(p[i], t1));
        d64 += 2;
    }
} else {
    for (i = 0; i < 8; i++) {
        _mm_store_si128((__m128i *) d64, p[i]);
        d64 += 2;
    }
}

}

gf_do_final_region_alignment(&rd);
}
#endif

#define GF_MULTBY_TWO(p) (((p) & GF_FIRST_BIT) ? ((p) << 1) ^ h->prim_poly) : (p) << 1);

static
int gf_w64_split_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_split_4_64_lazy_data *d4;
    struct gf_split_8_64_lazy_data *d8;
    struct gf_split_8_8_data *d88;
    struct gf_split_16_64_lazy_data *d16;
    uint64_t p, basep;
    int exp, i, j;

    h = (gf_internal_t *) gf->scratch;

    /* Defaults */

    gf->multiply_region.w64 = gf_w64_multiply_region_from_single;

    gf->multiply.w64 = gf_w64_bytwo_p_multiply;

#ifdef INTEL_SSE4_PCLMUL
    if (!(h->region_type & GF_REGION_NOSSE) &&
        (h->arg1 == 64 || h->arg2 == 64)) ||
        h->mult_type == GF_MULT_DEFAULT){

        if ((0xfffffffffe00000000ULL & h->prim_poly) == 0){
            gf->multiply.w64 = gf_w64_clm_multiply_2;
            gf->multiply_region.w64 = gf_w64_clm_multiply_region_from_single_2;
        } else if ((0xfffe000000000000ULL & h->prim_poly) == 0){
            gf->multiply.w64 = gf_w64_clm_multiply_4;
            gf->multiply_region.w64 = gf_w64_clm_multiply_region_from_single_4;
        } else{
            return 0;
        }
    }
}
#endif

gf->inverse.w64 = gf_w64_euclid;

/* Allen: set region pointers for default mult type. Single pointers are
 * taken care of above (explicitly for sse, implicitly for no sse). */

#ifdef INTEL_SSE4
```



*src/gf\_w64.c lines 2041 to 2100*

```
if (h->mult_type == GF_MULT_DEFAULT) {
    d4 = (struct gf_split_4_64_lazy_data *) h->private;
    d4->last_value = 0;
    gf->multiply_region.w64 = gf_w64_split_4_64_lazy_sse_multiply_region;
}
#else
if (h->mult_type == GF_MULT_DEFAULT) {
    d8 = (struct gf_split_8_64_lazy_data *) h->private;
    d8->last_value = 0;
    gf->multiply_region.w64 = gf_w64_split_8_64_lazy_multiply_region;
}
#endif

if ((h->arg1 == 4 && h->arg2 == 64) || (h->arg1 == 64 && h->arg2 == 4)) {
    d4 = (struct gf_split_4_64_lazy_data *) h->private;
    d4->last_value = 0;

    if((h->region_type & GF_REGION_ALTMAP) && (h->region_type & GF_REGION_NOSSE)) return 0;
    if(h->region_type & GF_REGION_ALTMAP)
    {
        #ifdef INTEL_SSSE3
            gf->multiply_region.w64 = gf_w64_split_4_64_lazy_sse_altmap_multiply_region;
        #else
            return 0;
        #endif
    }
    else //no altmap
    {
        #ifdef INTEL_SSE4
            if(h->region_type & GF_REGION_NOSSE)
                gf->multiply_region.w64 = gf_w64_split_4_64_lazy_multiply_region;
            else
                gf->multiply_region.w64 = gf_w64_split_4_64_lazy_sse_multiply_region;
        #else
            gf->multiply_region.w64 = gf_w64_split_4_64_lazy_multiply_region;
            if(h->region_type & GF_REGION_SSE)
                return 0;
        #endif
    }
}

if ((h->arg1 == 8 && h->arg2 == 64) || (h->arg1 == 64 && h->arg2 == 8)) {
    d8 = (struct gf_split_8_64_lazy_data *) h->private;
    d8->last_value = 0;
    gf->multiply_region.w64 = gf_w64_split_8_64_lazy_multiply_region;
}

if ((h->arg1 == 16 && h->arg2 == 64) || (h->arg1 == 64 && h->arg2 == 16)) {
    d16 = (struct gf_split_16_64_lazy_data *) h->private;
    d16->last_value = 0;
    gf->multiply_region.w64 = gf_w64_split_16_64_lazy_multiply_region;
}

if ((h->arg1 == 8 && h->arg2 == 8)) {
    d88 = (struct gf_split_8_8_data *) h->private;
    gf->multiply.w64 = gf_w64_split_8_8_multiply;

    /* The performance of this guy sucks, so don't bother with a region op */

    basep = 1;
    for (exp = 0; exp < 15; exp++) {
        for (j = 0; j < 256; j++) d88->tables[exp][0][j] = 0;
        for (i = 0; i < 256; i++) d88->tables[exp][i][0] = 0;
    }
}
```

*src/gf\_w64.c lines 2101 to 2160*

```
d88->tables[exp][1][1] = basep;
for (i = 2; i < 256; i++) {
    if (i&1) {
        p = d88->tables[exp][i^1][1];
        d88->tables[exp][i][1] = p ^ basep;
    } else {
        p = d88->tables[exp][i>>1][1];
        d88->tables[exp][i][1] = GF_MULTBY_TWO(p);
    }
}
for (i = 1; i < 256; i++) {
    p = d88->tables[exp][i][1];
    for (j = 1; j < 256; j++) {
        if (j&1) {
            d88->tables[exp][i][j] = d88->tables[exp][i][j^1] ^ p;
        } else {
            d88->tables[exp][i][j] = GF_MULTBY_TWO(d88->tables[exp][i][j>>1]);
        }
    }
}
for (i = 0; i < 8; i++) basep = GF_MULTBY_TWO(basep);
}
}
return 1;
}

int gf_w64_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2)
{
    switch(mult_type)
    {
        case GF_MULT_SHIFT:
            return sizeof(gf_internal_t);
            break;
        case GF_MULT_CARRY_FREE:
            return sizeof(gf_internal_t);
            break;
        case GF_MULT_BYTWO_p:
        case GF_MULT_BYTWO_b:
            return sizeof(gf_internal_t);
            break;

        case GF_MULT_DEFAULT:

            /* Allen: set the *local* arg1 and arg2, just for scratch size purposes,
             * then fall through to split table scratch size code. */

#ifdef INTEL_SSE4
            arg1 = 64;
            arg2 = 4;
#else
            arg1 = 64;
            arg2 = 8;
#endif

        case GF_MULT_SPLIT_TABLE:
            if (arg1 == 8 && arg2 == 8) {
                return sizeof(gf_internal_t) + sizeof(struct gf_split_8_8_data) + 64;
            }
            if ((arg1 == 16 && arg2 == 64) || (arg2 == 16 && arg1 == 64)) {
                return sizeof(gf_internal_t) + sizeof(struct gf_split_16_64_lazy_data) + 64;
            }
    }
}
```



*src/gf\_w64.c lines 2161 to 2220*

```
    }
    if ((arg1 == 8 && arg2 == 64) || (arg2 == 8 && arg1 == 64)) {
        return sizeof(gf_internal_t) + sizeof(struct gf_split_8_64_lazy_data) + 64;
    }

    if ((arg1 == 64 && arg2 == 4) || (arg1 == 4 && arg2 == 64)) {
        return sizeof(gf_internal_t) + sizeof(struct gf_split_4_64_lazy_data) + 64;
    }
    return 0;
case GF_MULT_GROUP:
    return sizeof(gf_internal_t) + sizeof(struct gf_w64_group_data) +
        sizeof(uint64_t) * (1 << arg1) +
        sizeof(uint64_t) * (1 << arg2) + 64;
    break;
case GF_MULT_COMPOSITE:
    if (arg1 == 2) return sizeof(gf_internal_t) + 64;
    return 0;
    break;
default:
    return 0;
}
}

int gf_w64_init(gf_t *gf)
{
    gf_internal_t *h;
    int no_default_flag = 0;

    h = (gf_internal_t *) gf->scratch;

    /* Allen: set default primitive polynomial / irreducible polynomial if needed */

    /* Omitting the leftmost 1 as in w=32 */

    if (h->prim_poly == 0) {
        if (h->mult_type == GF_MULT_COMPOSITE) {
            h->prim_poly = gf_composite_get_default_poly(h->base_gf);
            if (h->prim_poly == 0) return 0; /* This shouldn't happen */
        } else {
            h->prim_poly = 0x1b;
        }
    }
    if (no_default_flag == 1) {
        fprintf(stderr, "Code contains no default irreducible polynomial for given base field\n");
        return 0;
    }
}

gf->multiply.w64 = NULL;
gf->divide.w64 = NULL;
gf->inverse.w64 = NULL;
gf->multiply_region.w64 = NULL;

switch(h->mult_type) {
case GF_MULT_CARRY_FREE:    if (gf_w64_cfm_init(gf) == 0) return 0; break;
case GF_MULT_SHIFT:        if (gf_w64_shift_init(gf) == 0) return 0; break;
case GF_MULT_COMPOSITE:    if (gf_w64_composite_init(gf) == 0) return 0; break;
case GF_MULT_DEFAULT:
case GF_MULT_SPLIT_TABLE:  if (gf_w64_split_init(gf) == 0) return 0; break;
case GF_MULT_GROUP:        if (gf_w64_group_init(gf) == 0) return 0; break;
case GF_MULT_BYTWO_p:
```

*src/gf\_w64.c lines 2221 to 2249*

```
    case GF_MULT_BYTWO_b:      if (gf_w64_bytwo_init(gf) == 0) return 0; break;
    default: return 0;
}
if (h->divide_type == GF_DIVIDE_EUCLID) {
    gf->divide.w64 = gf_w64_divide_from_inverse;
    gf->inverse.w64 = gf_w64_euclid;
}

if (gf->inverse.w64 != NULL && gf->divide.w64 == NULL) {
    gf->divide.w64 = gf_w64_divide_from_inverse;
}
if (gf->inverse.w64 == NULL && gf->divide.w64 != NULL) {
    gf->inverse.w64 = gf_w64_inverse_from_divide;
}

if (h->region_type == GF_REGION_CAUCHY) return 0;

if (h->region_type & GF_REGION_ALTMAP) {
    if (h->mult_type == GF_MULT_COMPOSITE) {
        gf->extract_word.w64 = gf_w64_composite_extract_word;
    } else if (h->mult_type == GF_MULT_SPLIT_TABLE) {
        gf->extract_word.w64 = gf_w64_split_extract_word;
    }
} else {
    gf->extract_word.w64 = gf_w64_extract_word;
}

return 1;
}
```



*src/gf\_w8.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_w8.c
 *
 * Routines for 8-bit Galois fields
 */

#include "gf_int.h"
#include <stdio.h>
#include <stdlib.h>

#define GF_FIELD_WIDTH (8)
#define GF_FIELD_SIZE (1 << GF_FIELD_WIDTH)
#define GF_HALF_SIZE (1 << (GF_FIELD_WIDTH/2))
#define GF_MULT_GROUP_SIZE GF_FIELD_SIZE-1

#define GF_BASE_FIELD_WIDTH (4)
#define GF_BASE_FIELD_SIZE (1 << GF_BASE_FIELD_WIDTH)

struct gf_w8_logtable_data {
    uint8_t log_tbl[GF_FIELD_SIZE];
    uint8_t antilog_tbl[GF_FIELD_SIZE * 2];
    uint8_t inv_tbl[GF_FIELD_SIZE];
};

struct gf_w8_logzero_table_data {
    short log_tbl[GF_FIELD_SIZE]; /* Make this signed, so that we can divide easily */
    uint8_t antilog_tbl[512+512+1];
    uint8_t *div_tbl;
    uint8_t *inv_tbl;
};

struct gf_w8_logzero_small_table_data {
    short log_tbl[GF_FIELD_SIZE]; /* Make this signed, so that we can divide easily */
    uint8_t antilog_tbl[255*3];
    uint8_t inv_tbl[GF_FIELD_SIZE];
    uint8_t *div_tbl;
};

struct gf_w8_composite_data {
    uint8_t *mult_table;
};

/* Don't change the order of these relative to gf_w8_half_table_data */

struct gf_w8_default_data {
    uint8_t high[GF_FIELD_SIZE][GF_HALF_SIZE];
    uint8_t low[GF_FIELD_SIZE][GF_HALF_SIZE];
    uint8_t divtable[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint8_t multtable[GF_FIELD_SIZE][GF_FIELD_SIZE];
};

struct gf_w8_half_table_data {
    uint8_t high[GF_FIELD_SIZE][GF_HALF_SIZE];
    uint8_t low[GF_FIELD_SIZE][GF_HALF_SIZE];
};
```

*src/gf\_w8.c lines 61 to 120*

```
struct gf_w8_single_table_data {
    uint8_t      divtable[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint8_t      multtable[GF_FIELD_SIZE][GF_FIELD_SIZE];
};

struct gf_w8_double_table_data {
    uint8_t      div[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint16_t     mult[GF_FIELD_SIZE][GF_FIELD_SIZE*GF_FIELD_SIZE];
};

struct gf_w8_double_table_lazy_data {
    uint8_t      div[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint8_t      smult[GF_FIELD_SIZE][GF_FIELD_SIZE];
    uint16_t     mult[GF_FIELD_SIZE*GF_FIELD_SIZE];
};

struct gf_w4_logtable_data {
    uint8_t      log_tbl[GF_BASE_FIELD_SIZE];
    uint8_t      antilog_tbl[GF_BASE_FIELD_SIZE * 2];
    uint8_t      *antilog_tbl_div;
};

struct gf_w4_single_table_data {
    uint8_t      div[GF_BASE_FIELD_SIZE][GF_BASE_FIELD_SIZE];
    uint8_t      mult[GF_BASE_FIELD_SIZE][GF_BASE_FIELD_SIZE];
};

struct gf_w8_bytwo_data {
    uint64_t prim_poly;
    uint64_t mask1;
    uint64_t mask2;
};

#define AB2(ip, am1, am2, b, t1, t2) {\
    t1 = (b << 1) & am1;\
    t2 = b & am2;\
    t2 = ((t2 << 1) - (t2 >> (GF_FIELD_WIDTH-1))); \
    b = (t1 ^ (t2 & ip));}

#define SSE_AB2(pp, m1, m2, va, t1, t2) {\
    t1 = _mm_and_si128(_mm_slli_epi64(va, 1), m1);\
    t2 = _mm_and_si128(va, m2);\
    t2 = _mm_sub_epi64(_mm_slli_epi64(t2, 1), _mm_srli_epi64(t2, (GF_FIELD_WIDTH-1))); \
    va = _mm_xor_si128(t1, _mm_and_si128(t2, pp)); }

#define MM_PRINT(s, r) { uint8_t blah[16], ii; printf("%-12s", s); \
    _mm_storeu_si128((__m128i *)blah, r);\
    for (ii = 0; ii < 16; ii += 2) \
    printf("  %02x %02x", blah[15-ii], blah[14-ii]); printf("\n"); }

static
inline
uint32_t gf_w8_inverse_from_divide (gf_t *gf, uint32_t a)
{
    return gf->divide.w32(gf, 1, a);
}

static
inline
uint32_t gf_w8_divide_from_inverse (gf_t *gf, uint32_t a, uint32_t b)
```



*src/gf\_w8.c lines 121 to 180*

```
{
    b = gf->inverse.w32(gf, b);
    return gf->multiply.w32(gf, a, b);
}

static
inline
uint32_t gf_w8_euclid (gf_t *gf, uint32_t b)
{
    uint32_t e_i, e_im1, e_ip1;
    uint32_t d_i, d_im1, d_ip1;
    uint32_t y_i, y_im1, y_ip1;
    uint32_t c_i;

    if (b == 0) return -1;
    e_im1 = ((gf_internal_t *) (gf->scratch))->prim_poly;
    e_i = b;
    d_im1 = 8;
    for (d_i = d_im1; ((1 << d_i) & e_i) == 0; d_i--) ;
    y_i = 1;
    y_im1 = 0;

    while (e_i != 1) {

        e_ip1 = e_im1;
        d_ip1 = d_im1;
        c_i = 0;

        while (d_ip1 >= d_i) {
            c_i ^= (1 << (d_ip1 - d_i));
            e_ip1 ^= (e_i << (d_ip1 - d_i));
            if (e_ip1 == 0) return 0;
            while ((e_ip1 & (1 << d_ip1)) == 0) d_ip1--;
        }

        y_ip1 = y_im1 ^ gf->multiply.w32(gf, c_i, y_i);
        y_im1 = y_i;
        y_i = y_ip1;

        e_im1 = e_i;
        d_im1 = d_i;
        e_i = e_ip1;
        d_i = d_ip1;
    }

    return y_i;
}

static
gf_val_32_t gf_w8_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    uint8_t *r8;

    r8 = (uint8_t *) start;
    return r8[index];
}

static
gf_val_32_t gf_w8_composite_extract_word(gf_t *gf, void *start, int bytes, int index)
{

```

*src/gf\_w8.c lines 181 to 240*

```
int sub_size;
gf_internal_t *h;
uint8_t *r8, *top;
uint8_t a, b;
gf_region_data rd;

h = (gf_internal_t *) gf->scratch;
gf_set_region_data(&rd, gf, start, start, bytes, 0, 0, 32);
r8 = (uint8_t *) start;
if (r8 + index < (uint8_t *) rd.d_start) return r8[index];
if (r8 + index >= (uint8_t *) rd.d_top) return r8[index];
index -= (((uint8_t *) rd.d_start) - r8);
r8 = (uint8_t *) rd.d_start;
top = (uint8_t *) rd.d_top;
sub_size = (top-r8)/2;

a = h->base_gf->extract_word.w32(h->base_gf, r8, sub_size, index);
b = h->base_gf->extract_word.w32(h->base_gf, r8+sub_size, sub_size, index);
return (a | (b << 4));
}

static
inline
uint32_t gf_w8_matrix (gf_t *gf, uint32_t b)
{
    return gf_bitmatrix_inverse(b, 8, ((gf_internal_t *) (gf->scratch))->prim_poly);
}

static
inline
gf_val_32_t
gf_w8_clm_multiply_2 (gf_t *gf, gf_val_32_t a8, gf_val_32_t b8)
{
    gf_val_32_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL

    __m128i a, b;
    __m128i result;
    __m128i prim_poly;
    __m128i w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi32 (_mm_setzero_si128(), a8, 0);
    b = _mm_insert_epi32 (a, b8, 0);

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1fffULL));

    /* Do the initial multiply */

    result = _mm_clmulepi64_si128 (a, b, 0);

    /* Ben: Do prim_poly reduction twice. We are guaranteed that we will only
       have to do the reduction at most twice, because (w-2)/z == 2. Where
       z is equal to the number of zeros after the leading 1

       _mm_clmulepi64_si128 is the carryless multiply operation. Here
       _mm_srli_si128 shifts the result to the right by 1 byte. This allows
       us to multiply the prim_poly by the leading bits of the result. We
```



*src/gf\_w8.c lines 241 to 300*

```
        then xor the result of that operation back with the result.*/

w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
result = _mm_xor_si128 (result, w);
w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
result = _mm_xor_si128 (result, w);

/* Extracts 32 bit value from result. */

rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));

#endif
return rv;
}

static
inline
gf_val_32_t
gf_w8_clm_multiply_3 (gf_t *gf, gf_val_32_t a8, gf_val_32_t b8)
{
    gf_val_32_t rv = 0;

#ifdef INTEL_SSE4_PCLMUL

    __m128i          a, b;
    __m128i          result;
    __m128i          prim_poly;
    __m128i          w;
    gf_internal_t * h = gf->scratch;

    a = _mm_insert_epi32 (_mm_setzero_si128(), a8, 0);
    b = _mm_insert_epi32 (a, b8, 0);

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1fffULL));

    /* Do the initial multiply */

    result = _mm_clmulepi64_si128 (a, b, 0);

    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
    result = _mm_xor_si128 (result, w);

    /* Extracts 32 bit value from result. */

    rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));

#endif
return rv;
}

static
inline
gf_val_32_t
gf_w8_clm_multiply_4 (gf_t *gf, gf_val_32_t a8, gf_val_32_t b8)
{
    gf_val_32_t rv = 0;
```

*src/gf\_w8.c lines 301 to 360*

```
#if defined(INTEL_SSE4_PCLMUL)
```

```
    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      w;
    gf_internal_t * h = gf->scratch;
```

```
    a = _mm_insert_epi32 (_mm_setzero_si128(), a8, 0);
    b = _mm_insert_epi32 (a, b8, 0);
```

```
    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1fffULL));
```

```
    /* Do the initial multiply */
```

```
    result = _mm_clmulepi64_si128 (a, b, 0);
```

```
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
    result = _mm_xor_si128 (result, w);
    w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
    result = _mm_xor_si128 (result, w);
```

```
    /* Extracts 32 bit value from result. */
    rv = ((gf_val_32_t)_mm_extract_epi32(result, 0));
```

```
#endif
    return rv;
}
```

```
static
void
gf_w8_multiply_region_from_single(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int
    xor)
```

```
{
    gf_region_data rd;
    uint8_t *s8;
    uint8_t *d8;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }
```

```
    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 1);
    gf_do_initial_region_alignment(&rd);
```

```
    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;
```

```
    if (xor) {
        while (d8 < ((uint8_t *) rd.d_top)) {
            *d8 ^= gf->multiply.w32(gf, val, *s8);
            d8++;
            s8++;
        }
    } else {
```



*src/gf\_w8.c lines 361 to 420*

```
    while (d8 < ((uint8_t *) rd.d_top)) {
        *d8 = gf->multiply.w32(gf, val, *s8);
        d8++;
        s8++;
    }
}
gf_do_final_region_alignment(&rd);
}

#if defined(INTEL_SSE4_PCLMUL)
static
void
gf_w8_clm_multiply_region_from_single_2(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int
xor)
{
    gf_region_data rd;
    uint8_t *s8;
    uint8_t *d8;

    __m128i a, b;
    __m128i result;
    __m128i prim_poly;
    __m128i w;
    gf_internal_t * h = gf->scratch;

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1fffULL));

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 1);
    gf_do_initial_region_alignment(&rd);

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    if (xor) {
        while (d8 < ((uint8_t *) rd.d_top)) {
            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s8), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            *d8 ^= ((gf_val_32_t)_mm_extract_epi32(result, 0));
            d8++;
            s8++;
        }
    } else {
        while (d8 < ((uint8_t *) rd.d_top)) {
            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s8), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            *d8 = ((gf_val_32_t)_mm_extract_epi32(result, 0));
            d8++;
        }
    }
}
```

*src/gf\_w8.c lines 421 to 480*

```
        s8++;
    }
}
gf_do_final_region_alignment(&rd);
}
#endif

#if defined(INTEL_SSE4_PCLMUL)
static
void
gf_w8_clm_multiply_region_from_single_3(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int
xor)
{
    gf_region_data rd;
    uint8_t *s8;
    uint8_t *d8;

    __m128i      a, b;
    __m128i      result;
    __m128i      prim_poly;
    __m128i      w;
    gf_internal_t * h = gf->scratch;

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1fffULL));

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 1);
    gf_do_initial_region_alignment(&rd);

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    if (xor) {
        while (d8 < ((uint8_t *) rd.d_top)) {
            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s8), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            *d8 ^= ((gf_val_32_t)_mm_extract_epi32(result, 0));
            d8++;
            s8++;
        }
    } else {
        while (d8 < ((uint8_t *) rd.d_top)) {
            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s8), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
        }
    }
}
```



*src/gf\_w8.c lines 481 to 540*

```
        *d8 = ((gf_val_32_t)_mm_extract_epi32(result, 0));
        d8++;
        s8++;
    }
}
gf_do_final_region_alignment(&rd);
}
#endif

#if defined(INTEL_SSE4_PCLMUL)
static
void
gf_w8_clm_multiply_region_from_single_4(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int
xor)
{
    gf_region_data rd;
    uint8_t *s8;
    uint8_t *d8;

    __m128i          a, b;
    __m128i          result;
    __m128i          prim_poly;
    __m128i          w;
    gf_internal_t * h = gf->scratch;

    prim_poly = _mm_set_epi32(0, 0, 0, (uint32_t)(h->prim_poly & 0x1fffULL));

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    a = _mm_insert_epi32 (_mm_setzero_si128(), val, 0);

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 1);
    gf_do_initial_region_alignment(&rd);

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    if (xor) {
        while (d8 < ((uint8_t *) rd.d_top)) {
            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s8), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
            *d8 ^= ((gf_val_32_t)_mm_extract_epi32(result, 0));
            d8++;
            s8++;
        }
    } else {
        while (d8 < ((uint8_t *) rd.d_top)) {
            b = _mm_insert_epi32 (a, (gf_val_32_t)(*s8), 0);
            result = _mm_clmulepi64_si128 (a, b, 0);
            w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
            result = _mm_xor_si128 (result, w);
```

*src/gf\_w8.c lines 541 to 600*

```
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
        result = _mm_xor_si128 (result, w);
        w = _mm_clmulepi64_si128 (prim_poly, _mm_srli_si128 (result, 1), 0);
        result = _mm_xor_si128 (result, w);
        *d8 = ((gf_val_32_t) _mm_extract_epi32(result, 0));
        d8++;
        s8++;
    }
}
gf_do_final_region_alignment(&rd);
}
#endif
```

/\* -----

IMPLEMENTATION: SHIFT:

JSP: The world's dumbest multiplication algorithm. I only include it for completeness. It does have the feature that it requires no extra memory.  
\*/

```
static
inline
uint32_t
gf_w8_shift_multiply (gf_t *gf, uint32_t a8, uint32_t b8)
{
    uint16_t product, i, pp, a, b;
    gf_internal_t *h;

    a = a8;
    b = b8;
    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    product = 0;

    for (i = 0; i < GF_FIELD_WIDTH; i++) {
        if (a & (1 << i)) product ^= (b << i);
    }
    for (i = (GF_FIELD_WIDTH*2-2); i >= GF_FIELD_WIDTH; i--) {
        if (product & (1 << i)) product ^= (pp << (i-GF_FIELD_WIDTH));
    }
    return product;
}
```

```
static
int gf_w8_cfm_init(gf_t *gf)
{
#ifdef INTEL_SSE4_PCLMUL
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    if ((0xe0 & h->prim_poly) == 0){
        gf->multiply.w32 = gf_w8_clm_multiply_2;
        gf->multiply_region.w32 = gf_w8_clm_multiply_region_from_single_2;
    }else if ((0xc0 & h->prim_poly) == 0){
        gf->multiply.w32 = gf_w8_clm_multiply_3;
```



*src/gf\_w8.c lines 601 to 660*

```
        gf->multiply_region.w32 = gf_w8_clm_multiply_region_from_single_3;
    }else if ((0x80 & h->prim_poly) == 0){
        gf->multiply.w32 = gf_w8_clm_multiply_4;
        gf->multiply_region.w32 = gf_w8_clm_multiply_region_from_single_4;
    }else{
        return 0;
    }
    return 1;
#endif

    return 0;
}

static
int gf_w8_shift_init(gf_t *gf)
{
    gf->multiply.w32 = gf_w8_shift_multiply; /* The others will be set automatically */
    return 1;
}

/* -----
IMPLEMENTATION: LOG_TABLE:

JSP: Kevin wrote this, and I'm converting it to my structure.
*/

static
inline
uint32_t
gf_w8_logzero_multiply (gf_t *gf, uint32_t a, uint32_t b)
{
    struct gf_w8_logzero_table_data *ltd;

    ltd = (struct gf_w8_logzero_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return ltd->antilog_tbl[ltd->log_tbl[a] + ltd->log_tbl[b]];
}

static
inline
uint32_t
gf_w8_logzero_divide (gf_t *gf, uint32_t a, uint32_t b)
{
    struct gf_w8_logzero_table_data *ltd;

    ltd = (struct gf_w8_logzero_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return ltd->div_tbl[ltd->log_tbl[a] - ltd->log_tbl[b]];
}

static
inline
uint32_t
gf_w8_logzero_small_multiply (gf_t *gf, uint32_t a, uint32_t b)
{
    struct gf_w8_logzero_small_table_data *std;

    std = (struct gf_w8_logzero_small_table_data *) ((gf_internal_t *) gf->scratch)->private;
    if (b == 0) return 0;
    return std->antilog_tbl[std->log_tbl[a] + std->log_tbl[b]];
}
```

*src/gf\_w8.c lines 661 to 720*

```
static
inline
uint32_t
gf_w8_logzero_small_divide (gf_t *gf, uint32_t a, uint32_t b)
{
    struct gf_w8_logzero_small_table_data *std;

    std = (struct gf_w8_logzero_small_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return std->div_tbl[std->log_tbl[a] - std->log_tbl[b]];
}

static
inline
uint32_t
gf_w8_log_multiply (gf_t *gf, uint32_t a, uint32_t b)
{
    struct gf_w8_logtable_data *ltd;

    ltd = (struct gf_w8_logtable_data *) ((gf_internal_t *) gf->scratch)->private;
    return (a == 0 || b == 0) ? 0 : ltd->antilog_tbl[(unsigned) (ltd->log_tbl[a] + ltd->log_tbl[b])];
}

static
inline
uint32_t
gf_w8_log_divide (gf_t *gf, uint32_t a, uint32_t b)
{
    int log_sum = 0;
    struct gf_w8_logtable_data *ltd;

    if (a == 0 || b == 0) return 0;
    ltd = (struct gf_w8_logtable_data *) ((gf_internal_t *) gf->scratch)->private;

    log_sum = ltd->log_tbl[a] - ltd->log_tbl[b] + (GF_MULT_GROUP_SIZE);
    return (ltd->antilog_tbl[log_sum]);
}

static
uint32_t
gf_w8_log_inverse (gf_t *gf, uint32_t a)
{
    struct gf_w8_logtable_data *ltd;

    ltd = (struct gf_w8_logtable_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ltd->inv_tbl[a]);
}

static
uint32_t
gf_w8_logzero_inverse (gf_t *gf, uint32_t a)
{
    struct gf_w8_logzero_table_data *ltd;

    ltd = (struct gf_w8_logzero_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ltd->inv_tbl[a]);
}

static
uint32_t
```



*src/gf\_w8.c lines 721 to 780*

```
gf_w8_logzero_small_inverse (gf_t *gf, uint32_t a)
{
    struct gf_w8_logzero_small_table_data *std;

    std = (struct gf_w8_logzero_small_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return (std->inv_tbl[a]);
}

static
void
gf_w8_log_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    int i;
    uint8_t lv;
    uint8_t *s8, *d8;
    struct gf_w8_logtable_data *ltd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    ltd = (struct gf_w8_logtable_data *) ((gf_internal_t *) gf->scratch)->private;
    s8 = (uint8_t *) src;
    d8 = (uint8_t *) dest;

    lv = ltd->log_tbl[val];

    if (xor) {
        for (i = 0; i < bytes; i++) {
            d8[i] ^= (s8[i] == 0 ? 0 : ltd->antilog_tbl[lv + ltd->log_tbl[s8[i]]]);
        }
    } else {
        for (i = 0; i < bytes; i++) {
            d8[i] = (s8[i] == 0 ? 0 : ltd->antilog_tbl[lv + ltd->log_tbl[s8[i]]]);
        }
    }
}

static
void
gf_w8_logzero_multiply_region(gf_t *gf, void *src, void *dest, uint32_t val, int bytes, int xor)
{
    int i;
    uint8_t lv;
    uint8_t *s8, *d8;
    struct gf_w8_logzero_table_data *ltd;
    struct gf_w8_logzero_small_table_data *std;
    short *log;
    uint8_t *alt;
    gf_internal_t *h;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;

    if (h->arg1 == 1) {
        std = (struct gf_w8_logzero_small_table_data *) h->private;
        log = std->log_tbl;
        alt = std->antilog_tbl;
    } else {
```

*src/gf\_w8.c lines 781 to 840*

```
    ltd = (struct gf_w8_logzero_table_data *) h->private;
    log = ltd->log_tbl;
    alt = ltd->antilog_tbl;
}
s8 = (uint8_t *) src;
d8 = (uint8_t *) dest;

lv = log[val];

if (xor) {
    for (i = 0; i < bytes; i++) {
        d8[i] ^= (alt[lv + log[s8[i]]]);
    }
} else {
    for (i = 0; i < bytes; i++) {
        d8[i] = (alt[lv + log[s8[i]]]);
    }
}
}

static
int gf_w8_log_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_w8_logtable_data *ltd = NULL;
    struct gf_w8_logzero_table_data *ztd = NULL;
    struct gf_w8_logzero_small_table_data *std = NULL;
    uint8_t *alt;
    uint8_t *inv;
    int i, b;
    int check = 0;

    h = (gf_internal_t *) gf->scratch;
    if (h->mult_type == GF_MULT_LOG_TABLE) {
        ltd = h->private;
        alt = ltd->antilog_tbl;
        inv = ltd->inv_tbl;
    } else if (h->mult_type == GF_MULT_LOG_ZERO) {
        std = h->private;
        alt = std->antilog_tbl;
        std->div_tbl = (alt + 255);
        inv = std->inv_tbl;
    } else {
        ztd = h->private;
        alt = ztd->antilog_tbl;
        ztd->inv_tbl = (alt + 512 + 256);
        ztd->div_tbl = (alt + 255);
        inv = ztd->inv_tbl;
    }

    for (i = 0; i < GF_MULT_GROUP_SIZE+1; i++) {
        if (h->mult_type == GF_MULT_LOG_TABLE)
            ltd->log_tbl[i] = 0;
        else if (h->mult_type == GF_MULT_LOG_ZERO)
            std->log_tbl[i] = 0;
        else
            ztd->log_tbl[i] = 0;
    }

    if (h->mult_type == GF_MULT_LOG_TABLE) {
```



*src/gf\_w8.c lines 841 to 900*

```
    ltd->log_tbl[0] = 0;
} else if (h->mult_type == GF_MULT_LOG_ZERO) {
    std->log_tbl[0] = 510;
} else {
    ztd->log_tbl[0] = 512;
}

b = 1;
for (i = 0; i < GF_MULT_GROUP_SIZE; i++) {
    if (h->mult_type == GF_MULT_LOG_TABLE) {
        if (ltd->log_tbl[b] != 0) check = 1;
        ltd->log_tbl[b] = i;
    } else if (h->mult_type == GF_MULT_LOG_ZERO) {
        if (std->log_tbl[b] != 0) check = 1;
        std->log_tbl[b] = i;
    } else {
        if (ztd->log_tbl[b] != 0) check = 1;
        ztd->log_tbl[b] = i;
    }
    alt[i] = b;
    alt[i+GF_MULT_GROUP_SIZE] = b;
    b <<= 1;
    if (b & GF_FIELD_SIZE) {
        b = b ^ h->prim_poly;
    }
}
if (check) {
    _gf_errno = GF_E_LOGPOLY;
    return 0;
}

if (h->mult_type == GF_MULT_LOG_ZERO) bzero(alt+510, 255);

if (h->mult_type == GF_MULT_LOG_ZERO_EXT) {
    bzero(alt+512, 255);
    alt[512+512] = 0;
}

inv[0] = 0; /* Not really, but we need to fill it with something */
i = 1;
b = GF_MULT_GROUP_SIZE;
do {
    inv[i] = alt[b];
    i <<= 1;
    if (i & (1 << 8)) i ^= h->prim_poly;
    b--;
} while (i != 1);

if (h->mult_type == GF_MULT_LOG_TABLE) {
    gf->inverse.w32 = gf_w8_log_inverse;
    gf->divide.w32 = gf_w8_log_divide;
    gf->multiply.w32 = gf_w8_log_multiply;
    gf->multiply_region.w32 = gf_w8_log_multiply_region;
} else if (h->mult_type == GF_MULT_LOG_ZERO) {
    gf->inverse.w32 = gf_w8_logzero_small_inverse;
    gf->divide.w32 = gf_w8_logzero_small_divide;
    gf->multiply.w32 = gf_w8_logzero_small_multiply;
    gf->multiply_region.w32 = gf_w8_logzero_multiply_region;
} else {
    gf->inverse.w32 = gf_w8_logzero_inverse;
```

*src/gf\_w8.c lines 901 to 960*

```
    gf->divide.w32 = gf_w8_logzero_divide;
    gf->multiply.w32 = gf_w8_logzero_multiply;
    gf->multiply_region.w32 = gf_w8_logzero_multiply_region;
}
return 1;
}

/* -----
IMPLEMENTATION: FULL_TABLE:

JSP: Kevin wrote this, and I'm converting it to my structure.
*/

static
gf_val_32_t
gf_w8_table_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w8_single_table_data *ftd;

    ftd = (struct gf_w8_single_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ftd->multtable[a][b]);
}

static
gf_val_32_t
gf_w8_table_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w8_single_table_data *ftd;

    ftd = (struct gf_w8_single_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ftd->divtable[a][b]);
}

static
gf_val_32_t
gf_w8_default_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w8_default_data *ftd;

    ftd = (struct gf_w8_default_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ftd->multtable[a][b]);
}

#ifdef INTEL_SSSE3
static
gf_val_32_t
gf_w8_default_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w8_default_data *ftd;

    ftd = (struct gf_w8_default_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ftd->divtable[a][b]);
}
#endif

static
gf_val_32_t
gf_w8_double_table_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w8_double_table_data *ftd;
```



*src/gf\_w8.c lines 961 to 1020*

```
    ftd = (struct gf_w8_double_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ftd->mult[a][b]);
}
```

```
static
gf_val_32_t
gf_w8_double_table_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w8_double_table_data *ftd;

    ftd = (struct gf_w8_double_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ftd->div[a][b]);
}
```

```
static
void
gf_w8_double_table_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint16_t *base;
    uint32_t b, c, vc, vb;
    gf_internal_t *h;
    struct gf_w8_double_table_data *dtd;
    struct gf_w8_double_table_lazy_data *ltd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) (gf->scratch);
    if (h->region_type & GF_REGION_LAZY) {
        ltd = (struct gf_w8_double_table_lazy_data *) h->private;
        base = ltd->mult;
        for (b = 0; b < GF_FIELD_SIZE; b++) {
            vb = (ltd->smult[val][b] << 8);
            for (c = 0; c < GF_FIELD_SIZE; c++) {
                vc = ltd->smult[val][c];
                base[(b << 8) | c] = (vb | vc);
            }
        }
    }
    else {
        dtd = (struct gf_w8_double_table_data *) h->private;
        base = &(dtd->mult[val][0]);
    }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
    gf_do_initial_region_alignment(&rd);
    gf_two_byte_region_table_multiply(&rd, base);
    gf_do_final_region_alignment(&rd);
}
```

```
static
gf_val_32_t
gf_w8_double_table_lazy_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w8_double_table_lazy_data *ftd;

    ftd = (struct gf_w8_double_table_lazy_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ftd->smult[a][b]);
}
```

*src/gf\_w8.c lines 1021 to 1080*

```
}

static
gf_val_32_t
gf_w8_double_table_lazy_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w8_double_table_lazy_data *ftd;

    ftd = (struct gf_w8_double_table_lazy_data *) ((gf_internal_t *) gf->scratch)->private;
    return (ftd->div[a][b]);
}

static
void
gf_w8_table_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int i;
    uint8_t *s8, *d8;
    struct gf_w8_single_table_data *ftd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    ftd = (struct gf_w8_single_table_data *) ((gf_internal_t *) gf->scratch)->private;
    s8 = (uint8_t *) src;
    d8 = (uint8_t *) dest;

    if (xor) {
        for (i = 0; i < bytes; i++) {
            d8[i] ^= ftd->multtable[s8[i]][val];
        }
    } else {
        for (i = 0; i < bytes; i++) {
            d8[i] = ftd->multtable[s8[i]][val];
        }
    }
}

#ifdef INTEL_SSSE3
static
void
gf_w8_split_multiply_region_sse(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint8_t *bh, *bl, *sptr, *dptr;
    __m128i loset, t1, r, va, mth, mtl;
    struct gf_w8_half_table_data *htd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    htd = (struct gf_w8_half_table_data *) ((gf_internal_t *) (gf->scratch))->private;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    bh = (uint8_t *) htd->high;
    bh += (val << 4);
    bl = (uint8_t *) htd->low;
    bl += (val << 4);
}
```



*src/gf\_w8.c lines 1081 to 1140*

```
sptr = rd.s_start;
dptr = rd.d_start;

mth = _mm_loadu_si128 ((__m128i *) (bh));
mtl = _mm_loadu_si128 ((__m128i *) (bl));
loset = _mm_set1_epi8 (0x0f);

if (xor) {
    while (sptr < (uint8_t *) rd.s_top) {
        va = _mm_load_si128 ((__m128i *) (sptr));
        t1 = _mm_and_si128 (loset, va);
        r = _mm_shuffle_epi8 (mtl, t1);
        va = _mm_srli_epi64 (va, 4);
        t1 = _mm_and_si128 (loset, va);
        r = _mm_xor_si128 (r, _mm_shuffle_epi8 (mth, t1));
        va = _mm_load_si128 ((__m128i *) (dptr));
        r = _mm_xor_si128 (r, va);
        _mm_store_si128 ((__m128i *) (dptr), r);
        dptr += 16;
        sptr += 16;
    }
} else {
    while (sptr < (uint8_t *) rd.s_top) {
        va = _mm_load_si128 ((__m128i *) (sptr));
        t1 = _mm_and_si128 (loset, va);
        r = _mm_shuffle_epi8 (mtl, t1);
        va = _mm_srli_epi64 (va, 4);
        t1 = _mm_and_si128 (loset, va);
        r = _mm_xor_si128 (r, _mm_shuffle_epi8 (mth, t1));
        _mm_store_si128 ((__m128i *) (dptr), r);
        dptr += 16;
        sptr += 16;
    }
}

gf_do_final_region_alignment(&rd);
}
#endif

/* -----
IMPLEMENTATION: FULL_TABLE:
*/

static
gf_val_32_t
gf_w8_split_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    struct gf_w8_half_table_data *htd;
    htd = (struct gf_w8_half_table_data *) ((gf_internal_t *) gf->scratch)->private;

    return htd->high[b][a>>4] ^ htd->low[b][a&0xf];
}

static
void
gf_w8_split_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int i;
```

*src/gf\_w8.c lines 1141 to 1200*

```
uint8_t *s8, *d8;
struct gf_w8_half_table_data *htd;

if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

htd = (struct gf_w8_half_table_data *) ((gf_internal_t *) gf->scratch)->private;
s8 = (uint8_t *) src;
d8 = (uint8_t *) dest;

if (xor) {
    for (i = 0; i < bytes; i++) {
        d8[i] ^= (htd->high[val][s8[i]>>4] ^ htd->low[val][s8[i]&0xf]);
    }
} else {
    for (i = 0; i < bytes; i++) {
        d8[i] = (htd->high[val][s8[i]>>4] ^ htd->low[val][s8[i]&0xf]);
    }
}
}

static
int gf_w8_split_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_w8_half_table_data *htd;
    int a, b;

    h = (gf_internal_t *) gf->scratch;
    htd = (struct gf_w8_half_table_data *)h->private;

    bzero(htd->high, sizeof(uint8_t)*GF_FIELD_SIZE*GF_HALF_SIZE);
    bzero(htd->low, sizeof(uint8_t)*GF_FIELD_SIZE*GF_HALF_SIZE);

    for (a = 1; a < GF_FIELD_SIZE; a++) {
        for (b = 1; b < GF_HALF_SIZE; b++) {
            htd->low[a][b] = gf_w8_shift_multiply(gf, a, b);
            htd->high[a][b] = gf_w8_shift_multiply(gf, a, b<<4);
        }
    }

    gf->multiply.w32 = gf_w8_split_multiply;

#ifdef INTEL_SSSE3
    if (h->region_type & GF_REGION_NOSSE)
        gf->multiply_region.w32 = gf_w8_split_multiply_region;
    else
        gf->multiply_region.w32 = gf_w8_split_multiply_region_sse;
#else
    gf->multiply_region.w32 = gf_w8_split_multiply_region;
    if (h->region_type & GF_REGION_SSE)
        return 0;
#endif

    return 1;
}

/* JSP: This is disgusting, but it is what it is. If there is no SSE,
   then the default is equivalent to single table. If there is SSE, then
```



*src/gf\_w8.c lines 1201 to 1260*

```
we use the "gf_w8_default_data" which is a hybrid of SPLIT & TABLE. */

static
int gf_w8_table_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_w8_single_table_data *ftd = NULL;
    struct gf_w8_double_table_data *dtd = NULL;
    struct gf_w8_double_table_lazy_data *ltd = NULL;
    struct gf_w8_default_data *dd = NULL;
    int a, b, c, prod, scase, issse;

    h = (gf_internal_t *) gf->scratch;

#ifdef INTEL_SSSE3
    issse = 1;
#else
    issse = 0;
#endif

    if (h->mult_type == GF_MULT_DEFAULT && issse) {
        dd = (struct gf_w8_default_data *)h->private;
        scase = 3;
        bzero(dd->high, sizeof(uint8_t) * GF_FIELD_SIZE * GF_HALF_SIZE);
        bzero(dd->low, sizeof(uint8_t) * GF_FIELD_SIZE * GF_HALF_SIZE);
        bzero(dd->divtable, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
        bzero(dd->multtable, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
    } else if (h->mult_type == GF_MULT_DEFAULT ||
               h->region_type == 0 || (h->region_type & GF_REGION_CAUCHY)) {
        ftd = (struct gf_w8_single_table_data *)h->private;
        bzero(ftd->divtable, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
        bzero(ftd->multtable, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
        scase = 0;
    } else if (h->region_type == GF_REGION_DOUBLE_TABLE) {
        dtd = (struct gf_w8_double_table_data *)h->private;
        bzero(dtd->div, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
        bzero(dtd->mult, sizeof(uint16_t) * GF_FIELD_SIZE * GF_FIELD_SIZE * GF_FIELD_SIZE);
        scase = 1;
    } else if (h->region_type == (GF_REGION_DOUBLE_TABLE | GF_REGION_LAZY)) {
        ltd = (struct gf_w8_double_table_lazy_data *)h->private;
        bzero(ltd->div, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
        bzero(ltd->smult, sizeof(uint8_t) * GF_FIELD_SIZE * GF_FIELD_SIZE);
        scase = 2;
    } else {
        fprintf(stderr, "Internal error in gf_w8_table_init\n");
        exit(0);
    }

    for (a = 1; a < GF_FIELD_SIZE; a++) {
        for (b = 1; b < GF_FIELD_SIZE; b++) {
            prod = gf_w8_shift_multiply(gf, a, b);
            switch (scase) {
                case 0:
                    ftd->multtable[a][b] = prod;
                    ftd->divtable[prod][b] = a;
                    break;
                case 1:
                    dtd->div[prod][b] = a;
                    for (c = 0; c < GF_FIELD_SIZE; c++) {
                        dtd->mult[a][(c<<8)|b] |= prod;
                    }
                case 2:
                    ltd->div[prod][b] = a;
                    ltd->smult[a][b] = prod;
            }
        }
    }
}
```



*src/gf\_w8.c lines 1261 to 1320*

```
        dtd->mult[a][(b<<8)|c] |= (prod<<8);
    }
    break;
case 2:
    ltd->div[prod][b] = a;
    ltd->smult[a][b] = prod;
    break;
case 3:
    dd->multtable[a][b] = prod;
    dd->divtable[prod][b] = a;
    if ((b & 0xf) == b) { dd->low[a][b] = prod; }
    if ((b & 0xf0) == b) { dd->high[a][b>>4] = prod; }
    break;
}
}
}

gf->inverse.w32 = NULL; /* Will set from divide */
switch (scase) {
    case 0:
        gf->divide.w32 = gf_w8_table_divide;
        gf->multiply.w32 = gf_w8_table_multiply;
        gf->multiply_region.w32 = gf_w8_table_multiply_region;
        break;
    case 1:
        gf->divide.w32 = gf_w8_double_table_divide;
        gf->multiply.w32 = gf_w8_double_table_multiply;
        gf->multiply_region.w32 = gf_w8_double_table_multiply_region;
        break;
    case 2:
        gf->divide.w32 = gf_w8_double_table_lazy_divide;
        gf->multiply.w32 = gf_w8_double_table_lazy_multiply;
        gf->multiply_region.w32 = gf_w8_double_table_multiply_region;
        break;
    case 3:
#ifdef INTEL_SSSE3
        gf->divide.w32 = gf_w8_default_divide;
        gf->multiply.w32 = gf_w8_default_multiply;
        gf->multiply_region.w32 = gf_w8_split_multiply_region_sse;
#endif
    break;
}
return 1;
}

static
void
gf_w8_composite_multiply_region_alt(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint8_t val0 = val & 0x0f;
    uint8_t val1 = (val & 0xf0) >> 4;
    gf_region_data rd;
    int sub_reg_size;

    if (val == 0) {
        if (xor) return;
        bzero(dest, bytes);
        return;
    }
}
```



*src/gf\_w8.c lines 1321 to 1380*

```
    }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 32);
    gf_do_initial_region_alignment(&rd);

    sub_reg_size = ((uint8_t *)rd.d_top - (uint8_t *)rd.d_start) / 2;

    base_gf->multiply_region.w32(base_gf, rd.s_start, rd.d_start, val0, sub_reg_size, xor);
    base_gf->multiply_region.w32(base_gf, (uint8_t *)rd.s_start+sub_reg_size,
        rd.d_start, val1, sub_reg_size, 1);
    base_gf->multiply_region.w32(base_gf, rd.s_start,
        (uint8_t *)rd.d_start+sub_reg_size, val1, sub_reg_size, xor);
    base_gf->multiply_region.w32(base_gf, (uint8_t *)rd.s_start+sub_reg_size,
        (uint8_t *)rd.d_start+sub_reg_size, val0, sub_reg_size, 1);
    base_gf->multiply_region.w32(base_gf, (uint8_t *)rd.s_start+sub_reg_size,
        (uint8_t *)rd.d_start+sub_reg_size,
        base_gf->multiply.w32(base_gf, h->prim_poly, val1), sub_reg_size, 1);

    gf_do_final_region_alignment(&rd);
}
```

```
static
gf_val_32_t
gf_w8_composite_multiply_recursive(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint8_t b0 = b & 0x0f;
    uint8_t b1 = (b & 0xf0) >> 4;
    uint8_t a0 = a & 0x0f;
    uint8_t a1 = (a & 0xf0) >> 4;
    uint8_t alb1;

    alb1 = base_gf->multiply.w32(base_gf, a1, b1);

    return ((base_gf->multiply.w32(base_gf, a0, b0) ^ alb1) |
        ((base_gf->multiply.w32(base_gf, a1, b0) ^
            base_gf->multiply.w32(base_gf, a0, b1) ^
            base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 4));
}
```

```
static
gf_val_32_t
gf_w8_composite_multiply_inline(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    uint8_t b0 = b & 0x0f;
    uint8_t b1 = (b & 0xf0) >> 4;
    uint8_t a0 = a & 0x0f;
    uint8_t a1 = (a & 0xf0) >> 4;
    uint8_t alb1, *mt;
    struct gf_w8_composite_data *cd;

    cd = (struct gf_w8_composite_data *) h->private;
    mt = cd->mult_table;

    alb1 = GF_W4_INLINE_MULTDIV(mt, a1, b1);

    return ((GF_W4_INLINE_MULTDIV(mt, a0, b0) ^ alb1) |
        ((GF_W4_INLINE_MULTDIV(mt, a1, b0) ^
```

*src/gf\_w8.c lines 1381 to 1440*

```
        GF_W4_INLINE_MULTDIV(mt, a0, b1) ^
        GF_W4_INLINE_MULTDIV(mt, alb1, h->prim_poly)) << 4));
}

/*
 * Composite field division trick (explained in 2007 tech report)
 *
 * Compute  $a / b = a \cdot b^{-1}$ , where  $p(x) = x^2 + sx + 1$ 
 *
 * let  $c = b^{-1}$ 
 *
 *  $c \cdot b = (s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1)x + (b_1 c_1 + b_0 c_0)$ 
 *
 * want  $(s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1) = 0$  and  $(b_1 c_1 + b_0 c_0) = 1$ 
 *
 * let  $d = b_1 c_1$  and  $d+1 = b_0 c_0$ 
 *
 * solve  $s \cdot b_1 c_1 + b_1 c_0 + b_0 c_1 = 0$ 
 *
 * solution:  $d = (b_1 b_0^{-1}) (b_1 b_0^{-1} + b_0 b_1^{-1} + s)^{-1}$ 
 *
 *  $c_0 = (d+1) b_0^{-1}$ 
 *  $c_1 = d \cdot b_1^{-1}$ 
 *
 *  $a / b = a * c$ 
 */
```

```
static
gf_val_32_t
gf_w8_composite_inverse(gf_t *gf, gf_val_32_t a)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint8_t a0 = a & 0xf;
    uint8_t a1 = (a & 0xf0) >> 4;
    uint8_t c0, c1, c, d, tmp;
    uint8_t a0inv, a1inv;

    if (a0 == 0) {
        a1inv = base_gf->inverse.w32(base_gf, a1) & 0xf;
        c0 = base_gf->multiply.w32(base_gf, a1inv, h->prim_poly);
        c1 = a1inv;
    } else if (a1 == 0) {
        c0 = base_gf->inverse.w32(base_gf, a0);
        c1 = 0;
    } else {
        a1inv = base_gf->inverse.w32(base_gf, a1) & 0xf;
        a0inv = base_gf->inverse.w32(base_gf, a0) & 0xf;

        d = base_gf->multiply.w32(base_gf, a1, a0inv) & 0xf;

        tmp = (base_gf->multiply.w32(base_gf, a1, a0inv) ^ base_gf->multiply.w32(base_gf, a0, a1inv) ^ h->prim_poly) & 0xf;
        tmp = base_gf->inverse.w32(base_gf, tmp) & 0xf;

        d = base_gf->multiply.w32(base_gf, d, tmp) & 0xf;

        c0 = base_gf->multiply.w32(base_gf, (d^1), a0inv) & 0xf;
        c1 = base_gf->multiply.w32(base_gf, d, a1inv) & 0xf;
    }
}
```



*src/gf\_w8.c lines 1441 to 1500*

```
    c = c0 | (c1 << 4);

    return c;
}

static
void
gf_w8_composite_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    gf_region_data rd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    gf_t *base_gf = h->base_gf;
    uint8_t b0 = val & 0x0f;
    uint8_t b1 = (val & 0xf0) >> 4;
    uint8_t *s8;
    uint8_t *d8;
    uint8_t *mt;
    uint8_t a0, a1, alb1;
    struct gf_w8_composite_data *cd;

    cd = (struct gf_w8_composite_data *) h->private;

    if (val == 0) {
        if (xor) return;
        bzero(dest, bytes);
        return;
    }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 1);
    gf_do_initial_region_alignment(&rd);

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    mt = cd->mult_table;
    if (mt == NULL) {
        if (xor) {
            while (d8 < (uint8_t *) rd.d_top) {
                a0 = *s8 & 0x0f;
                a1 = (*s8 & 0xf0) >> 4;
                alb1 = base_gf->multiply.w32(base_gf, a1, b1);

                *d8 ^= ((base_gf->multiply.w32(base_gf, a0, b0) ^ alb1) |
                    ((base_gf->multiply.w32(base_gf, a1, b0) ^
                    base_gf->multiply.w32(base_gf, a0, b1) ^
                    base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 4));

                s8++;
                d8++;
            }
        } else {
            while (d8 < (uint8_t *) rd.d_top) {
                a0 = *s8 & 0x0f;
                a1 = (*s8 & 0xf0) >> 4;
                alb1 = base_gf->multiply.w32(base_gf, a1, b1);

                *d8 = ((base_gf->multiply.w32(base_gf, a0, b0) ^ alb1) |
                    ((base_gf->multiply.w32(base_gf, a1, b0) ^
                    base_gf->multiply.w32(base_gf, a0, b1) ^
                    base_gf->multiply.w32(base_gf, alb1, h->prim_poly)) << 4));
            }
        }
    }
}
```

*src/gf\_w8.c lines 1501 to 1560*

```
        s8++;
        d8++;
    }
} else {
    if (xor) {
        while (d8 < (uint8_t *) rd.d_top) {
            a0 = *s8 & 0x0f;
            a1 = (*s8 & 0xf0) >> 4;
            alb1 = GF_W4_INLINE_MULTDIV(mt, a1, b1);

            *d8 ^= ((GF_W4_INLINE_MULTDIV(mt, a0, b0) ^ alb1) |
                ((GF_W4_INLINE_MULTDIV(mt, a1, b0) ^
                    GF_W4_INLINE_MULTDIV(mt, a0, b1) ^
                    GF_W4_INLINE_MULTDIV(mt, alb1, h->prim_poly)) << 4));

            s8++;
            d8++;
        }
    } else {
        while (d8 < (uint8_t *) rd.d_top) {
            a0 = *s8 & 0x0f;
            a1 = (*s8 & 0xf0) >> 4;
            alb1 = GF_W4_INLINE_MULTDIV(mt, a1, b1);

            *d8 = ((GF_W4_INLINE_MULTDIV(mt, a0, b0) ^ alb1) |
                ((GF_W4_INLINE_MULTDIV(mt, a1, b0) ^
                    GF_W4_INLINE_MULTDIV(mt, a0, b1) ^
                    GF_W4_INLINE_MULTDIV(mt, alb1, h->prim_poly)) << 4));

            s8++;
            d8++;
        }
    }
}
}
gf_do_final_region_alignment(&rd);
return;
}
```

```
static
int gf_w8_composite_init(gf_t *gf)
{
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    struct gf_w8_composite_data *cd;

    if (h->base_gf == NULL) return 0;

    cd = (struct gf_w8_composite_data *) h->private;
    cd->mult_table = gf_w4_get_mult_table(h->base_gf);

    if (h->region_type & GF_REGION_ALTMAP) {
        gf->multiply_region.w32 = gf_w8_composite_multiply_region_alt;
    } else {
        gf->multiply_region.w32 = gf_w8_composite_multiply_region;
    }

    if (cd->mult_table == NULL) {
        gf->multiply.w32 = gf_w8_composite_multiply_recursive;
    } else {
        gf->multiply.w32 = gf_w8_composite_multiply_inline;
    }
    gf->divide.w32 = NULL;
}
```



*src/gf\_w8.c lines 1561 to 1620*

```
gf->inverse.w32 = gf_w8_composite_inverse;

return 1;
}

static
inline
gf_val_32_t
gf_w8_bytwo_p_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint32_t prod, pp, pmask, amask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;
    pmask = 0x80;
    amask = 0x80;

    while (amask != 0) {
        if (prod & pmask) {
            prod = ((prod << 1) ^ pp);
        } else {
            prod <<= 1;
        }
        if (a & amask) prod ^= b;
        amask >>= 1;
    }
    return prod;
}

static
inline
gf_val_32_t
gf_w8_bytwo_b_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint32_t prod, pp, bmask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;
    bmask = 0x80;

    while (1) {
        if (a & 1) prod ^= b;
        a >>= 1;
        if (a == 0) return prod;
        if (b & bmask) {
            b = ((b << 1) ^ pp);
        } else {
            b <<= 1;
        }
    }
}

static
```

*src/gf\_w8.c lines 1621 to 1680*

```
void
gf_w8_bytwo_p_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t *s64, *d64, t1, t2, ta, prod, amask;
    gf_region_data rd;
    struct gf_w8_bytwo_data *btd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    btd = (struct gf_w8_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 8);
    gf_do_initial_region_alignment(&rd);

    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;

    if (xor) {
        while (s64 < (uint64_t *) rd.s_top) {
            prod = 0;
            amask = 0x80;
            ta = *s64;
            while (amask != 0) {
                AB2(btd->prim_poly, btd->mask1, btd->mask2, prod, t1, t2);
                if (val & amask) prod ^= ta;
                amask >>= 1;
            }
            *d64 ^= prod;
            d64++;
            s64++;
        }
    } else {
        while (s64 < (uint64_t *) rd.s_top) {
            prod = 0;
            amask = 0x80;
            ta = *s64;
            while (amask != 0) {
                AB2(btd->prim_poly, btd->mask1, btd->mask2, prod, t1, t2);
                if (val & amask) prod ^= ta;
                amask >>= 1;
            }
            *d64 = prod;
            d64++;
            s64++;
        }
    }
    gf_do_final_region_alignment(&rd);
}

#define BYTWO_P_ONESTEP {\
    SSE_AB2(pp, m1, m2, prod, t1, t2); \
    t1 = _mm_and_si128(v, one); \
    t1 = _mm_sub_epi8(t1, one); \
    t1 = _mm_and_si128(t1, ta); \
    prod = _mm_xor_si128(prod, t1); \
    v = _mm_srli_epi64(v, 1); }

#ifdef INTEL_SSE2
static
```



*src/gf\_w8.c lines 1681 to 1740*

```
void
gf_w8_bytwo_p_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int i;
    uint8_t *s8, *d8;
    uint8_t vrev;
    __m128i pp, m1, m2, ta, prod, t1, t2, tp, one, v;
    struct gf_w8_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    btd = (struct gf_w8_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    vrev = 0;
    for (i = 0; i < 8; i++) {
        vrev <=< 1;
        if (!(val & (1 << i))) vrev |= 1;
    }

    s8 = (uint8_t *) rd.s_start;
    d8 = (uint8_t *) rd.d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);
    m2 = _mm_set1_epi8((btd->mask2)&0xff);
    one = _mm_set1_epi8(1);

    while (d8 < (uint8_t *) rd.d_top) {
        prod = _mm_setzero_si128();
        v = _mm_set1_epi8(vrev);
        ta = _mm_load_si128((__m128i *) s8);
        tp = (!xor) ? _mm_setzero_si128() : _mm_load_si128((__m128i *) d8);
        BYTWO_P_ONESTEP;
        BYTWO_P_ONESTEP;
        BYTWO_P_ONESTEP;
        BYTWO_P_ONESTEP;
        BYTWO_P_ONESTEP;
        BYTWO_P_ONESTEP;
        BYTWO_P_ONESTEP;
        BYTWO_P_ONESTEP;
        BYTWO_P_ONESTEP;
        BYTWO_P_ONESTEP;
        _mm_store_si128((__m128i *) d8, _mm_xor_si128(prod, tp));
        d8 += 16;
        s8 += 16;
    }
    gf_do_final_region_alignment(&rd);
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w8_bytwo_b_sse_region_2_noxor(gf_region_data *rd, struct gf_w8_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va;
```

*src/gf\_w8.c lines 1741 to 1800*

```
s8 = (uint8_t *) rd->s_start;
d8 = (uint8_t *) rd->d_start;

pp = _mm_set1_epi8(btd->prim_poly&0xff);
m1 = _mm_set1_epi8((btd->mask1)&0xff);
m2 = _mm_set1_epi8((btd->mask2)&0xff);

while (d8 < (uint8_t *) rd->d_top) {
    va = _mm_load_si128 ((__m128i *) (s8));
    SSE_AB2(pp, m1, m2, va, t1, t2);
    _mm_store_si128((__m128i *) d8, va);
    d8 += 16;
    s8 += 16;
}
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w8_bytwo_b_sse_region_2_xor(gf_region_data *rd, struct gf_w8_bytwo_data *btd)
{
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va, vb;

    s8 = (uint8_t *) rd->s_start;
    d8 = (uint8_t *) rd->d_start;

    pp = _mm_set1_epi8(btd->prim_poly&0xff);
    m1 = _mm_set1_epi8((btd->mask1)&0xff);
    m2 = _mm_set1_epi8((btd->mask2)&0xff);

    while (d8 < (uint8_t *) rd->d_top) {
        va = _mm_load_si128 ((__m128i *) (s8));
        SSE_AB2(pp, m1, m2, va, t1, t2);
        vb = _mm_load_si128 ((__m128i *) (d8));
        vb = _mm_xor_si128(vb, va);
        _mm_store_si128((__m128i *) d8, vb);
        d8 += 16;
        s8 += 16;
    }
}
#endif

#ifdef INTEL_SSE2
static
void
gf_w8_bytwo_b_sse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    int itb;
    uint8_t *d8, *s8;
    __m128i pp, m1, m2, t1, t2, va, vb;
    struct gf_w8_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }
```



*src/gf\_w8.c lines 1801 to 1860*

```
gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
gf_do_initial_region_alignment(&rd);

btd = (struct gf_w8_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;

if (val == 2) {
    if (xor) {
        gf_w8_bytwo_b_sse_region_2_xor(&rd, btd);
    } else {
        gf_w8_bytwo_b_sse_region_2_noxor(&rd, btd);
    }
    gf_do_final_region_alignment(&rd);
    return;
}

s8 = (uint8_t *) rd.s_start;
d8 = (uint8_t *) rd.d_start;

pp = _mm_set1_epi8(btd->prim_poly&0xff);
m1 = _mm_set1_epi8((btd->mask1)&0xff);
m2 = _mm_set1_epi8((btd->mask2)&0xff);

while (d8 < (uint8_t *) rd.d_top) {
    va = _mm_load_si128((__m128i *) (s8));
    vb = (!xor) ? _mm_setzero_si128() : _mm_load_si128((__m128i *) (d8));
    itb = val;
    while (1) {
        if (itb & 1) vb = _mm_xor_si128(vb, va);
        itb >>= 1;
        if (itb == 0) break;
        SSE_AB2(pp, m1, m2, va, t1, t2);
    }
    _mm_store_si128((__m128i *) d8, vb);
    d8 += 16;
    s8 += 16;
}

gf_do_final_region_alignment(&rd);
}
#endif

static
void
gf_w8_bytwo_b_nosse_multiply_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
{
    uint64_t *s64, *d64, t1, t2, ta, tb, prod;
    struct gf_w8_bytwo_data *btd;
    gf_region_data rd;

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, 16);
    gf_do_initial_region_alignment(&rd);

    btd = (struct gf_w8_bytwo_data *) ((gf_internal_t *) (gf->scratch))->private;
    s64 = (uint64_t *) rd.s_start;
    d64 = (uint64_t *) rd.d_start;

    switch (val) {
```

*src/gf\_w8.c lines 1861 to 1920*

```
case 2:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= ta;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta;
            d64++;
            s64++;
        }
    }
    break;
case 3:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
            d64++;
            s64++;
        }
    }
    break;
case 4:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= ta;
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta;
            d64++;
            s64++;
        }
    }
}
```



*src/gf\_w8.c lines 1921 to 1980*

```
break;
case 5:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta ^ prod;
            d64++;
            s64++;
        }
    }
    break;
case 6:
    if (xor) {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 ^= (ta ^ prod);
            d64++;
            s64++;
        }
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = ta ^ prod;
            d64++;
            s64++;
        }
    }
    break;
/*
    case 7:
        if (xor) {
            while (d64 < (uint64_t *) rd.d_top) {
                ta = *s64;
                prod = ta;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
                prod ^= ta;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
                *d64 ^= (ta ^ prod);
                d64++;
                s64++;
            }
        }
    */
```





src/gf\_w8.c lines 2041 to 2100

```
}
}
break;
case 10:
if (xor) {
while (d64 < (uint64_t *) rd.d_top) {
ta = *s64;
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
prod = ta;
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
*d64 ^= (ta ^ prod);
d64++;
s64++;
}
} else {
while (d64 < (uint64_t *) rd.d_top) {
ta = *s64;
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
prod = ta;
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
*d64 = (ta ^ prod);
d64++;
s64++;
}
}
break;
case 11:
if (xor) {
while (d64 < (uint64_t *) rd.d_top) {
ta = *s64;
prod = ta;
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
prod ^= ta;
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
*d64 ^= (ta ^ prod);
d64++;
s64++;
}
} else {
while (d64 < (uint64_t *) rd.d_top) {
ta = *s64;
prod = ta;
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
prod ^= ta;
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
*d64 = (ta ^ prod);
d64++;
s64++;
}
}
}
break;
case 12:
if (xor) {
while (d64 < (uint64_t *) rd.d_top) {
ta = *s64;
AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
```

*src/gf\_w8.c lines 2101 to 2160*

```
    AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
    prod = ta;
    AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
    *d64 ^= (ta ^ prod);
    d64++;
    s64++;
}
} else {
    while (d64 < (uint64_t *) rd.d_top) {
        ta = *s64;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 = (ta ^ prod);
        d64++;
        s64++;
    }
}
break;
case 13:
if (xor) {
    while (d64 < (uint64_t *) rd.d_top) {
        ta = *s64;
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod ^= ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 ^= (ta ^ prod);
        d64++;
        s64++;
    }
} else {
    while (d64 < (uint64_t *) rd.d_top) {
        ta = *s64;
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod ^= ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 = (ta ^ prod);
        d64++;
        s64++;
    }
}
}
break;
case 14:
if (xor) {
    while (d64 < (uint64_t *) rd.d_top) {
        ta = *s64;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod ^= ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 ^= (ta ^ prod);
        d64++;
        s64++;
    }
}
```



*src/gf\_w8.c lines 2161 to 2220*

```
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            ta = *s64;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod = ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            prod ^= ta;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            *d64 = (ta ^ prod);
            d64++;
            s64++;
        }
    }
}
break;
case 15:
if (xor) {
    while (d64 < (uint64_t *) rd.d_top) {
        ta = *s64;
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod ^= ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod ^= ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 ^= (ta ^ prod);
        d64++;
        s64++;
    }
} else {
    while (d64 < (uint64_t *) rd.d_top) {
        ta = *s64;
        prod = ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod ^= ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        prod ^= ta;
        AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        *d64 = (ta ^ prod);
        d64++;
        s64++;
    }
}
}
break;
*/
default:
if (xor) {
    while (d64 < (uint64_t *) rd.d_top) {
        prod = *d64 ;
        ta = *s64;
        tb = val;
        while (1) {
            if (tb & 1) prod ^= ta;
            tb >>= 1;
            if (tb == 0) break;
            AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
        }
        *d64 = prod;
        d64++;
        s64++;
    }
}
```

*src/gf\_w8.c lines 2221 to 2280*

```
    } else {
        while (d64 < (uint64_t *) rd.d_top) {
            prod = 0;
            ta = *s64;
            tb = val;
            while (1) {
                if (tb & 1) prod ^= ta;
                tb >>= 1;
                if (tb == 0) break;
                AB2(btd->prim_poly, btd->mask1, btd->mask2, ta, t1, t2);
            }
            *d64 = prod;
            d64++;
            s64++;
        }
    }
    break;
}
gf_do_final_region_alignment(&rd);
}

static
int gf_w8_bytwo_init(gf_t *gf)
{
    gf_internal_t *h;
    uint64_t ip, m1, m2;
    struct gf_w8_bytwo_data *btd;

    h = (gf_internal_t *) gf->scratch;
    btd = (struct gf_w8_bytwo_data *) (h->private);
    ip = h->prim_poly & 0xff;
    m1 = 0xfe;
    m2 = 0x80;
    btd->prim_poly = 0;
    btd->mask1 = 0;
    btd->mask2 = 0;

    while (ip != 0) {
        btd->prim_poly |= ip;
        btd->mask1 |= m1;
        btd->mask2 |= m2;
        ip <<= GF_FIELD_WIDTH;
        m1 <<= GF_FIELD_WIDTH;
        m2 <<= GF_FIELD_WIDTH;
    }

    if (h->mult_type == GF_MULT_BYTWO_p) {
        gf->multiply.w32 = gf_w8_bytwo_p_multiply;
#ifdef INTEL_SSE2
        if (h->region_type & GF_REGION_NOSSE)
            gf->multiply_region.w32 = gf_w8_bytwo_p_nosse_multiply_region;
        else
            gf->multiply_region.w32 = gf_w8_bytwo_p_sse_multiply_region;
#else
        gf->multiply_region.w32 = gf_w8_bytwo_p_nosse_multiply_region;
        if (h->region_type & GF_REGION_SSE)
            return 0;
#endif
    } else {
        gf->multiply.w32 = gf_w8_bytwo_b_multiply;
    }
}
```



*src/gf\_w8.c lines 2281 to 2340*

```
#ifndef INTEL_SSE2
    if (h->region_type & GF_REGION_NOSSE)
        gf->multiply_region.w32 = gf_w8_bytwo_b_nosse_multiply_region;
    else
        gf->multiply_region.w32 = gf_w8_bytwo_b_sse_multiply_region;
#else
    gf->multiply_region.w32 = gf_w8_bytwo_b_nosse_multiply_region;
    if(h->region_type & GF_REGION_SSE)
        return 0;
#endif
}
return 1;
}

/* -----
   General procedures.
   You don't need to error check here on in init, because it's done
   for you in gf_error_check().
*/

int gf_w8_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2)
{
    switch(mult_type)
    {
        case GF_MULT_DEFAULT:
#ifdef INTEL_SSSE3
            return sizeof(gf_internal_t) + sizeof(struct gf_w8_default_data) + 64;
#endif
            return sizeof(gf_internal_t) + sizeof(struct gf_w8_single_table_data) + 64;
        case GF_MULT_TABLE:
            if (region_type == GF_REGION_CAUCHY) {
                return sizeof(gf_internal_t) + sizeof(struct gf_w8_single_table_data) + 64;
            }

            if (region_type == GF_REGION_DEFAULT) {
                return sizeof(gf_internal_t) + sizeof(struct gf_w8_single_table_data) + 64;
            }
            if (region_type & GF_REGION_DOUBLE_TABLE) {
                if (region_type == GF_REGION_DOUBLE_TABLE) {
                    return sizeof(gf_internal_t) + sizeof(struct gf_w8_double_table_data) + 64;
                } else if (region_type == (GF_REGION_DOUBLE_TABLE | GF_REGION_LAZY)) {
                    return sizeof(gf_internal_t) + sizeof(struct gf_w8_double_table_lazy_data) + 64;
                } else {
                    return 0;
                }
            }
            return 0;
        break;
        case GF_MULT_BYTWO_p:
        case GF_MULT_BYTWO_b:
            return sizeof(gf_internal_t) + sizeof(struct gf_w8_bytwo_data);
        break;
        case GF_MULT_SPLIT_TABLE:
            if ((arg1 == 4 && arg2 == 8) || (arg1 == 8 && arg2 == 4)) {
                return sizeof(gf_internal_t) + sizeof(struct gf_w8_half_table_data) + 64;
            }
        break;
        case GF_MULT_LOG_TABLE:
            return sizeof(gf_internal_t) + sizeof(struct gf_w8_logtable_data) + 64;
    }
}
```

*src/gf\_w8.c lines 2341 to 2400*

```
        break;
    case GF_MULT_LOG_ZERO:
        return sizeof(gf_internal_t) + sizeof(struct gf_w8_logzero_small_table_data) + 64;
        break;
    case GF_MULT_LOG_ZERO_EXT:
        return sizeof(gf_internal_t) + sizeof(struct gf_w8_logzero_table_data) + 64;
        break;
    case GF_MULT_CARRY_FREE:
        return sizeof(gf_internal_t);
        break;
    case GF_MULT_SHIFT:
        return sizeof(gf_internal_t);
        break;
    case GF_MULT_COMPOSITE:
        return sizeof(gf_internal_t) + sizeof(struct gf_w8_composite_data) + 64;
    default:
        return 0;
}
return 0;
}

int gf_w8_init(gf_t *gf)
{
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;

    /* Allen: set default primitive polynomial / irreducible polynomial if needed */

    if (h->prim_poly == 0) {
        if (h->mult_type == GF_MULT_COMPOSITE) {
            h->prim_poly = gf_composite_get_default_poly(h->base_gf);
            if (h->prim_poly == 0) return 0; /* JSP: This shouldn't happen, but just in case. */
        } else {
            h->prim_poly = 0x11d;
        }
    }
    if (h->mult_type != GF_MULT_COMPOSITE) {
        h->prim_poly |= 0x100;
    }

    gf->multiply.w32 = NULL;
    gf->divide.w32 = NULL;
    gf->inverse.w32 = NULL;
    gf->multiply_region.w32 = NULL;
    gf->extract_word.w32 = gf_w8_extract_word;

    switch(h->mult_type) {
        case GF_MULT_DEFAULT:
        case GF_MULT_TABLE:
            if (gf_w8_table_init(gf) == 0) return 0; break;
        case GF_MULT_BYTWO_p:
        case GF_MULT_BYTWO_b:
            if (gf_w8_bytwo_init(gf) == 0) return 0; break;
        case GF_MULT_LOG_ZERO:
        case GF_MULT_LOG_ZERO_EXT:
        case GF_MULT_LOG_TABLE:
            if (gf_w8_log_init(gf) == 0) return 0; break;
        case GF_MULT_CARRY_FREE:
            if (gf_w8_cfm_init(gf) == 0) return 0; break;
        case GF_MULT_SHIFT:
            if (gf_w8_shift_init(gf) == 0) return 0; break;
        case GF_MULT_SPLIT_TABLE:
            if (gf_w8_split_init(gf) == 0) return 0; break;
        case GF_MULT_COMPOSITE:
            if (gf_w8_composite_init(gf) == 0) return 0; break;
        default: return 0;
    }
}
```



*src/gf\_w8.c lines 2401 to 2460*

```
    }

    if (h->divide_type == GF_DIVIDE_EUCLID) {
        gf->divide.w32 = gf_w8_divide_from_inverse;
        gf->inverse.w32 = gf_w8_euclid;
    } else if (h->divide_type == GF_DIVIDE_MATRIX) {
        gf->divide.w32 = gf_w8_divide_from_inverse;
        gf->inverse.w32 = gf_w8_matrix;
    }

    if (gf->divide.w32 == NULL) {
        gf->divide.w32 = gf_w8_divide_from_inverse;
        if (gf->inverse.w32 == NULL) gf->inverse.w32 = gf_w8_euclid;
    }

    if (gf->inverse.w32 == NULL) gf->inverse.w32 = gf_w8_inverse_from_divide;

    if (h->mult_type == GF_MULT_COMPOSITE && (h->region_type & GF_REGION_ALTMAP)) {
        gf->extract_word.w32 = gf_w8_composite_extract_word;
    }

    if (h->region_type == GF_REGION_CAUCHY) {
        gf->multiply_region.w32 = gf_wgen_cauchy_region;
        gf->extract_word.w32 = gf_wgen_extract_word;
    }

    if (gf->multiply_region.w32 == NULL) {
        gf->multiply_region.w32 = gf_w8_multiply_region_from_single;
    }

    return 1;
}
```

*/\* Inline setup functions \*/*

```
uint8_t *gf_w8_get_mult_table(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_w8_default_data *ftd;
    struct gf_w8_single_table_data *std;

    h = (gf_internal_t *) gf->scratch;
    if (gf->multiply.w32 == gf_w8_default_multiply) {
        ftd = (struct gf_w8_default_data *) h->private;
        return (uint8_t *) ftd->multtable;
    } else if (gf->multiply.w32 == gf_w8_table_multiply) {
        std = (struct gf_w8_single_table_data *) h->private;
        return (uint8_t *) std->multtable;
    }
    return NULL;
}
```

```
uint8_t *gf_w8_get_div_table(gf_t *gf)
{
    struct gf_w8_default_data *ftd;
    struct gf_w8_single_table_data *std;

    if (gf->multiply.w32 == gf_w8_default_multiply) {
        ftd = (struct gf_w8_default_data *) ((gf_internal_t *) gf->scratch)->private;
```

*src/gf\_w8.c lines 2461 to 2467*

```
    return (uint8_t *) ftd->divtable;
} else if (gf->multiply.w32 == gf_w8_table_multiply) {
    std = (struct gf_w8_single_table_data *) ((gf_internal_t *) gf->scratch)->private;
    return (uint8_t *) std->divtable;
}
return NULL;
}
```



*src/gf\_wgen.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_wgen.c
 *
 * Routines for Galois fields for general w < 32. For specific w,
 * like 4, 8, 16, 32, 64 and 128, see the other files.
 */

#include "gf_int.h"
#include <stdio.h>
#include <stdlib.h>

struct gf_wgen_table_w8_data {
    uint8_t *mult;
    uint8_t *div;
    uint8_t base;
};

struct gf_wgen_table_w16_data {
    uint16_t *mult;
    uint16_t *div;
    uint16_t base;
};

struct gf_wgen_log_w8_data {
    uint8_t *log;
    uint8_t *anti;
    uint8_t *danti;
    uint8_t base;
};

struct gf_wgen_log_w16_data {
    uint16_t *log;
    uint16_t *anti;
    uint16_t *danti;
    uint16_t base;
};

struct gf_wgen_log_w32_data {
    uint32_t *log;
    uint32_t *anti;
    uint32_t *danti;
    uint32_t base;
};

struct gf_wgen_group_data {
    uint32_t *reduce;
    uint32_t *shift;
    uint32_t mask;
    uint64_t rmask;
    int tshift;
    uint32_t memory;
};

static
inline
gf_val_32_t gf_wgen_inverse_from_divide (gf_t *gf, gf_val_32_t a)
```

*src/gf\_wgen.c lines 61 to 120*

```
{
    return gf->divide.w32(gf, 1, a);
}

static
inline
gf_val_32_t gf_wgen_divide_from_inverse (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    b = gf->inverse.w32(gf, b);
    return gf->multiply.w32(gf, a, b);
}

static
inline
gf_val_32_t gf_wgen_euclid (gf_t *gf, gf_val_32_t b)
{
    gf_val_32_t e_i, e_im1, e_ip1;
    gf_val_32_t d_i, d_im1, d_ip1;
    gf_val_32_t y_i, y_im1, y_ip1;
    gf_val_32_t c_i;

    if (b == 0) return -1;
    e_im1 = ((gf_internal_t *) (gf->scratch))->prim_poly;
    e_i = b;
    d_im1 = ((gf_internal_t *) (gf->scratch))->w;
    for (d_i = d_im1; ((1 << d_i) & e_i) == 0; d_i--) ;
    y_i = 1;
    y_im1 = 0;

    while (e_i != 1) {
        e_ip1 = e_im1;
        d_ip1 = d_im1;
        c_i = 0;

        while (d_ip1 >= d_i) {
            c_i ^= (1 << (d_ip1 - d_i));
            e_ip1 ^= (e_i << (d_ip1 - d_i));
            if (e_ip1 == 0) return 0;
            while ((e_ip1 & (1 << d_ip1)) == 0) d_ip1--;
        }

        y_ip1 = y_im1 ^ gf->multiply.w32(gf, c_i, y_i);
        y_im1 = y_i;
        y_i = y_ip1;

        e_im1 = e_i;
        d_im1 = d_i;
        e_i = e_ip1;
        d_i = d_ip1;
    }

    return y_i;
}

gf_val_32_t gf_wgen_extract_word(gf_t *gf, void *start, int bytes, int index)
{
    uint8_t *ptr;
    uint32_t rv;
```



*src/gf\_wgen.c lines 121 to 180*

```
int rs;
int byte, bit, i;
gf_internal_t *h;

h = (gf_internal_t *) gf->scratch;
rs = bytes / h->w;
byte = index/8;
bit = index%8;

ptr = (uint8_t *) start;
ptr += bytes;
ptr -= rs;
ptr += byte;

rv = 0;
for (i = 0; i < h->w; i++) {
    rv <<= 1;
    if ((*ptr) & (1 << bit)) rv |= 1;
    ptr -= rs;
}

return rv;
}

static
inline
gf_val_32_t gf_wgen_matrix (gf_t *gf, gf_val_32_t b)
{
    return gf_bitmatrix_inverse(b, ((gf_internal_t *) (gf->scratch))->w,
                                ((gf_internal_t *) (gf->scratch))->prim_poly);
}

static
inline
uint32_t
gf_wgen_shift_multiply (gf_t *gf, uint32_t a32, uint32_t b32)
{
    uint64_t product, i, pp, a, b, one;
    gf_internal_t *h;

    a = a32;
    b = b32;
    h = (gf_internal_t *) gf->scratch;
    one = 1;
    pp = h->prim_poly | (one << h->w);

    product = 0;

    for (i = 0; i < h->w; i++) {
        if (a & (one << i)) product ^= (b << i);
    }
    for (i = h->w*2-1; i >= h->w; i--) {
        if (product & (one << i)) product ^= (pp << (i-h->w));
    }
    return product;
}

static
int gf_wgen_shift_init(gf_t *gf)
{

```

*src/gf\_wgen.c lines 181 to 240*

```
gf->multiply.w32 = gf_wgen_shift_multiply;
gf->inverse.w32 = gf_wgen_euclid;
return 1;
}
```

```
static
gf_val_32_t
gf_wgen_bytwo_b_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint32_t prod, pp, bmask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;
    bmask = (1 << (h->w-1));

    while (1) {
        if (a & 1) prod ^= b;
        a >>= 1;
        if (a == 0) return prod;
        if (b & bmask) {
            b = ((b << 1) ^ pp);
        } else {
            b <<= 1;
        }
    }
}
```

```
static
int gf_wgen_bytwo_b_init(gf_t *gf)
{
    gf->multiply.w32 = gf_wgen_bytwo_b_multiply;
    gf->inverse.w32 = gf_wgen_euclid;
    return 1;
}
```

```
static
inline
gf_val_32_t
gf_wgen_bytwo_p_multiply (gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    uint32_t prod, pp, pmask, amask;
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    pp = h->prim_poly;

    prod = 0;
    pmask = (1 << ((h->w)-1)); /*Ben: Had an operator precedence warning here*/
    amask = pmask;

    while (amask != 0) {
        if (prod & pmask) {
            prod = ((prod << 1) ^ pp);
        } else {
            prod <<= 1;
        }
        if (a & amask) prod ^= b;
    }
}
```



*src/gf\_wgen.c lines 241 to 300*

```
    amask >>= 1;
}
return prod;
}
```

```
static
int gf_wgen_bytwo_p_init(gf_t *gf)
{
    gf->multiply.w32 = gf_wgen_bytwo_p_multiply;
    gf->inverse.w32 = gf_wgen_euclid;
    return 1;
}
```

```
static
void
gf_wgen_group_set_shift_tables(uint32_t *shift, uint32_t val, gf_internal_t *h)
{
    int i;
    uint32_t j;
    int g_s;

    if (h->mult_type == GF_MULT_DEFAULT) {
        g_s = 2;
    } else {
        g_s = h->arg1;
    }

    shift[0] = 0;

    for (i = 1; i < (1 << g_s); i <= 1) {
        for (j = 0; j < i; j++) shift[i|j] = shift[j]^val;
        if (val & (1 << (h->w-1))) {
            val <<= 1;
            val ^= h->prim_poly;
        } else {
            val <<= 1;
        }
    }
}
```

```
static
inline
gf_val_32_t
gf_wgen_group_s_equals_r_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    int leftover, rs;
    uint32_t p, l, ind, a32;
    int bits_left;
    int g_s;
    int w;

    struct gf_wgen_group_data *gd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    g_s = h->arg1;
    w = h->w;

    gd = (struct gf_wgen_group_data *) h->private;
    gf_wgen_group_set_shift_tables(gd->shift, b, h);
```

*src/gf\_wgen.c lines 301 to 360*

```
    leftover = w % g_s;
    if (leftover == 0) leftover = g_s;

    rs = w - leftover;
    a32 = a;
    ind = a32 >> rs;
    a32 <<= leftover;
    a32 &= gd->mask;
    p = gd->shift[ind];

    bits_left = rs;
    rs = w - g_s;

    while (bits_left > 0) {
        bits_left -= g_s;
        ind = a32 >> rs;
        a32 <<= g_s;
        a32 &= gd->mask;
        l = p >> rs;
        p = (gd->shift[ind] ^ gd->reduce[l] ^ (p << g_s)) & gd->mask;
    }
    return p;
}
```

```
char *bits(uint32_t v)
{
    char *rv;
    int i, j;

    rv = malloc(30);
    j = 0;
    for (i = 27; i >= 0; i--) {
        rv[j] = '0' + ((v & (1 << i)) ? 1 : 0);
        j++;
    }
    rv[j] = '\\0';
    return rv;
}
```

```
char *bits_56(uint64_t v)
{
    char *rv;
    int i, j;
    uint64_t one;

    one = 1;

    rv = malloc(60);
    j = 0;
    for (i = 55; i >= 0; i--) {
        rv[j] = '0' + ((v & (one << i)) ? 1 : 0);
        j++;
    }
    rv[j] = '\\0';
    return rv;
}
```

```
static
inline
gf_val_32_t
gf_wgen_group_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
```



*src/gf\_wgen.c lines 361 to 420*

```
{
    int i;
    int leftover;
    uint64_t p, l, r;
    uint32_t a32, ind;
    int g_s, g_r;
    struct gf_wgen_group_data *gd;
    int w;

    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    if (h->mult_type == GF_MULT_DEFAULT) {
        g_s = 2;
        g_r = 8;
    } else {
        g_s = h->arg1;
        g_r = h->arg2;
    }
    w = h->w;
    gd = (struct gf_wgen_group_data *) h->private;
    gf_wgen_group_set_shift_tables(gd->shift, b, h);

    leftover = w % g_s;
    if (leftover == 0) leftover = g_s;

    a32 = a;
    ind = a32 >> (w - leftover);
    p = gd->shift[ind];
    p <<= g_s;
    a32 <<= leftover;
    a32 &= gd->mask;

    i = (w - leftover);
    while (i > g_s) {
        ind = a32 >> (w - g_s);
        p ^= gd->shift[ind];
        a32 <<= g_s;
        a32 &= gd->mask;
        p <<= g_s;
        i -= g_s;
    }

    ind = a32 >> (h->w - g_s);
    p ^= gd->shift[ind];

    for (i = gd->tshift ; i >= 0; i -= g_r) {
        l = p & (gd->rmask << i);
        r = gd->reduce[l >> (i + w)];
        r <<= (i);
        p ^= r;
    }
    return p & gd->mask;
}
```

```
static
int gf_wgen_group_init(gf_t *gf)
{
    uint32_t i, j, p, index;
    struct gf_wgen_group_data *gd;
    gf_internal_t *h = (gf_internal_t *) gf->scratch;
    int g_s, g_r;
```

*src/gf\_wgen.c lines 421 to 480*

```
if (h->mult_type == GF_MULT_DEFAULT) {
    g_s = 2;
    g_r = 8;
} else {
    g_s = h->arg1;
    g_r = h->arg2;
}
gd = (struct gf_wgen_group_data *) h->private;
gd->shift = &(gd->memory);
gd->reduce = gd->shift + (1 << g_s);
gd->mask = (h->w != 31) ? ((1 << h->w)-1) : 0x7fffffff;

gd->rmask = (1 << g_r) - 1;
gd->rmask <=< h->w;

gd->tshift = h->w % g_s;
if (gd->tshift == 0) gd->tshift = g_s;
gd->tshift = (h->w - gd->tshift);
gd->tshift = ((gd->tshift-1)/g_r) * g_r;

gd->reduce[0] = 0;
for (i = 0; i < (1 << g_r); i++) {
    p = 0;
    index = 0;
    for (j = 0; j < g_r; j++) {
        if (i & (1 << j)) {
            p ^= (h->prim_poly << j);
            index ^= (h->prim_poly >> (h->w-j));
        }
    }
    gd->reduce[index] = (p & gd->mask);
}

if (g_s == g_r) {
    gf->multiply.w32 = gf_wgen_group_s_equals_r_multiply;
} else {
    gf->multiply.w32 = gf_wgen_group_multiply;
}
gf->divide.w32 = NULL;
gf->divide.w32 = NULL;
return 1;
}
```

```
static
gf_val_32_t
gf_wgen_table_8_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_table_w8_data *std;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_table_w8_data *) h->private;

    return (std->mult[(a<<h->w)+b]);
}
```

```
static
gf_val_32_t
```



*src/gf\_wgen.c lines 481 to 540*

```
gf_wgen_table_8_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_table_w8_data *std;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_table_w8_data *) h->private;

    return (std->div[(a<<h->w)+b]);
}
```

```
static
int gf_wgen_table_8_init(gf_t *gf)
{
    gf_internal_t *h;
    int w;
    struct gf_wgen_table_w8_data *std;
    uint32_t a, b, p;

    h = (gf_internal_t *) gf->scratch;
    w = h->w;
    std = (struct gf_wgen_table_w8_data *) h->private;

    std->mult = &(std->base);
    std->div = std->mult + ((1<<h->w) * (1<<h->w));

    for (a = 0; a < (1 << w); a++) {
        std->mult[a] = 0;
        std->mult[a<<w] = 0;
        std->div[a] = 0;
        std->div[a<<w] = 0;
    }

    for (a = 1; a < (1 << w); a++) {
        for (b = 1; b < (1 << w); b++) {
            p = gf_wgen_shift_multiply(gf, a, b);
            std->mult[(a<<w)|b] = p;
            std->div[(p<<w)|a] = b;
        }
    }

    gf->multiply.w32 = gf_wgen_table_8_multiply;
    gf->divide.w32 = gf_wgen_table_8_divide;
    return 1;
}
```

```
static
gf_val_32_t
gf_wgen_table_16_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_table_w16_data *std;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_table_w16_data *) h->private;

    return (std->mult[(a<<h->w)+b]);
}
```

static

*src/gf\_wgen.c lines 541 to 600*

```
gf_val_32_t
gf_wgen_table_16_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_table_w16_data *std;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_table_w16_data *) h->private;

    return (std->div[(a<<h->w)+b]);
}
```

```
static
int gf_wgen_table_16_init(gf_t *gf)
{
    gf_internal_t *h;
    int w;
    struct gf_wgen_table_w16_data *std;
    uint32_t a, b, p;

    h = (gf_internal_t *) gf->scratch;
    w = h->w;
    std = (struct gf_wgen_table_w16_data *) h->private;

    std->mult = &(std->base);
    std->div = std->mult + ((1<<h->w)*(1<<h->w));

    for (a = 0; a < (1 << w); a++) {
        std->mult[a] = 0;
        std->mult[a<<w] = 0;
        std->div[a] = 0;
        std->div[a<<w] = 0;
    }

    for (a = 1; a < (1 << w); a++) {
        for (b = 1; b < (1 << w); b++) {
            p = gf_wgen_shift_multiply(gf, a, b);
            std->mult[(a<<w)|b] = p;
            std->div[(p<<w)|a] = b;
        }
    }

    gf->multiply.w32 = gf_wgen_table_16_multiply;
    gf->divide.w32 = gf_wgen_table_16_divide;
    return 1;
}
```

```
static
int gf_wgen_table_init(gf_t *gf)
{
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    if (h->w <= 8) return gf_wgen_table_8_init(gf);
    if (h->w <= 14) return gf_wgen_table_16_init(gf);

    /* Returning zero to make the compiler happy, but this won't get
       executed, because it is tested in _scratch_space. */

    return 0;
}
```



*src/gf\_wgen.c lines 601 to 660*

```
}

static
gf_val_32_t
gf_wgen_log_8_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_log_w8_data *std;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_log_w8_data *) h->private;

    if (a == 0 || b == 0) return 0;
    return (std->anti[std->log[a]+std->log[b]]);
}
```

```
static
gf_val_32_t
gf_wgen_log_8_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_log_w8_data *std;
    int index;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_log_w8_data *) h->private;

    if (a == 0 || b == 0) return 0;
    index = std->log[a];
    index -= std->log[b];

    return (std->danti[index]);
}
```

```
static
int gf_wgen_log_8_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_wgen_log_w8_data *std;
    int w;
    uint32_t a, i;
    int check = 0;

    h = (gf_internal_t *) gf->scratch;
    w = h->w;
    std = (struct gf_wgen_log_w8_data *) h->private;

    std->log = &(std->base);
    std->anti = std->log + (1<<h->w);
    std->danti = std->anti + (1<<h->w)-1;

    for (i = 0; i < (1 << w); i++)
        std->log[i] = 0;

    a = 1;
    for(i=0; i < (1<<w)-1; i++)
    {
        if (std->log[a] != 0) check = 1;
        std->log[a] = i;
        std->anti[i] = a;
    }
```

*src/gf\_wgen.c lines 661 to 720*

```
    std->danti[i] = a;
    a <<= 1;
    if(a & (1<<w))
        a ^= h->prim_poly;
    //a &= ((1<<w)-1);
}

if (check != 0) {
    _gf_errno = GF_E_LOGPOLY;
    return 0;
}

gf->multiply.w32 = gf_wgen_log_8_multiply;
gf->divide.w32 = gf_wgen_log_8_divide;
return 1;
}

static
gf_val_32_t
gf_wgen_log_16_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_log_w16_data *std;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_log_w16_data *) h->private;

    if (a == 0 || b == 0) return 0;
    return (std->anti[std->log[a]+std->log[b]]);
}

static
gf_val_32_t
gf_wgen_log_16_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_log_w16_data *std;
    int index;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_log_w16_data *) h->private;

    if (a == 0 || b == 0) return 0;
    index = std->log[a];
    index -= std->log[b];

    return (std->danti[index]);
}

static
int gf_wgen_log_16_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_wgen_log_w16_data *std;
    int w;
    uint32_t a, i;
    int check = 0;

    h = (gf_internal_t *) gf->scratch;
    w = h->w;
```



*src/gf\_wgen.c lines 721 to 780*

```
std = (struct gf_wgen_log_w16_data *) h->private;

std->log = &(std->base);
std->anti = std->log + (1<<h->w);
std->danti = std->anti + (1<<h->w)-1;

for (i = 0; i < (1 << w); i++)
    std->log[i] = 0;

a = 1;
for(i=0; i < (1<<w)-1; i++)
{
    if (std->log[a] != 0) check = 1;
    std->log[a] = i;
    std->anti[i] = a;
    std->danti[i] = a;
    a <<= 1;
    if(a & (1<<w))
        a ^= h->prim_poly;
    //a &= ((1 << w)-1);
}

if (check) {
    if (h->mult_type != GF_MULT_LOG_TABLE) return gf_wgen_shift_init(gf);
    _gf_errno = GF_E_LOGPOLY;
    return 0;
}

gf->multiply.w32 = gf_wgen_log_16_multiply;
gf->divide.w32 = gf_wgen_log_16_divide;
return 1;
}

static
gf_val_32_t
gf_wgen_log_32_multiply(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_log_w32_data *std;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_log_w32_data *) h->private;

    if (a == 0 || b == 0) return 0;
    return (std->anti[std->log[a]+std->log[b]]);
}

static
gf_val_32_t
gf_wgen_log_32_divide(gf_t *gf, gf_val_32_t a, gf_val_32_t b)
{
    gf_internal_t *h;
    struct gf_wgen_log_w32_data *std;
    int index;

    h = (gf_internal_t *) gf->scratch;
    std = (struct gf_wgen_log_w32_data *) h->private;

    if (a == 0 || b == 0) return 0;
    index = std->log[a];
```

*src/gf\_wgen.c lines 781 to 840*

```
    index -= std->log[b];

    return (std->danti[index]);
}

static
int gf_wgen_log_32_init(gf_t *gf)
{
    gf_internal_t *h;
    struct gf_wgen_log_w32_data *std;
    int w;
    uint32_t a, i;
    int check = 0;

    h = (gf_internal_t *) gf->scratch;
    w = h->w;
    std = (struct gf_wgen_log_w32_data *) h->private;

    std->log = &(std->base);
    std->anti = std->log + (1<<h->w);
    std->danti = std->anti + (1<<h->w)-1;

    for (i = 0; i < (1 << w); i++)
        std->log[i] = 0;

    a = 1;
    for(i=0; i < (1<<w)-1; i++)
    {
        if (std->log[a] != 0) check = 1;
        std->log[a] = i;
        std->anti[i] = a;
        std->danti[i] = a;
        a <<= 1;
        if(a & (1<<w))
            a ^= h->prim_poly;
        //a &= ((1 << w)-1);
    }

    if (check != 0) {
        _gf_errno = GF_E_LOGPOLY;
        return 0;
    }

    gf->multiply.w32 = gf_wgen_log_32_multiply;
    gf->divide.w32 = gf_wgen_log_32_divide;
    return 1;
}
```

```
static
int gf_wgen_log_init(gf_t *gf)
{
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    if (h->w <= 8) return gf_wgen_log_8_init(gf);
    if (h->w <= 16) return gf_wgen_log_16_init(gf);
    if (h->w <= 32) return gf_wgen_log_32_init(gf);

    /* Returning zero to make the compiler happy, but this won't get
       executed, because it is tested in _scratch_space. */
}
```



*src/gf\_wgen.c lines 841 to 900*

```
    return 0;
}

int gf_wgen_scratch_size(int w, int mult_type, int region_type, int divide_type, int arg1, int arg2)
{
    switch(mult_type)
    {
        case GF_MULT_DEFAULT:
            if (w <= 8) {
                return sizeof(gf_internal_t) + sizeof(struct gf_wgen_table_w8_data) +
                    sizeof(uint8_t)*(1 << w)*(1<<w)*2 + 64;
            } else if (w <= 16) {
                return sizeof(gf_internal_t) + sizeof(struct gf_wgen_log_w16_data) +
                    sizeof(uint16_t)*(1 << w)*3;
            } else {
                return sizeof(gf_internal_t) + sizeof(struct gf_wgen_group_data) +
                    sizeof(uint32_t) * (1 << 2) +
                    sizeof(uint32_t) * (1 << 8) + 64;
            }
        case GF_MULT_SHIFT:
        case GF_MULT_BYTWO_b:
        case GF_MULT_BYTWO_p:
            return sizeof(gf_internal_t);
            break;
        case GF_MULT_GROUP:
            return sizeof(gf_internal_t) + sizeof(struct gf_wgen_group_data) +
                sizeof(uint32_t) * (1 << arg1) +
                sizeof(uint32_t) * (1 << arg2) + 64;

            break;

        case GF_MULT_TABLE:
            if (w <= 8) {
                return sizeof(gf_internal_t) + sizeof(struct gf_wgen_table_w8_data) +
                    sizeof(uint8_t)*(1 << w)*(1<<w)*2 + 64;
            } else if (w < 15) {
                return sizeof(gf_internal_t) + sizeof(struct gf_wgen_table_w16_data) +
                    sizeof(uint16_t)*(1 << w)*(1<<w)*2 + 64;
            }
            return 0;
        case GF_MULT_LOG_TABLE:
            if (w <= 8) {
                return sizeof(gf_internal_t) + sizeof(struct gf_wgen_log_w8_data) +
                    sizeof(uint8_t)*(1 << w)*3;
            } else if (w <= 16) {
                return sizeof(gf_internal_t) + sizeof(struct gf_wgen_log_w16_data) +
                    sizeof(uint16_t)*(1 << w)*3;
            } else if (w <= 27) {
                return sizeof(gf_internal_t) + sizeof(struct gf_wgen_log_w32_data) +
                    sizeof(uint32_t)*(1 << w)*3;
            } else
                return 0;
        default:
            return 0;
    }
}

void
gf_wgen_cauchy_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor)
```

*src/gf\_wgen.c lines 901 to 960*

```
{
    gf_internal_t *h;
    gf_region_data rd;
    int written;
    int rs, i, j;

    gf_set_region_data(&rd, gf, src, dest, bytes, val, xor, -1);

    if (val == 0) { gf_multby_zero(dest, bytes, xor); return; }
    if (val == 1) { gf_multby_one(src, dest, bytes, xor); return; }

    h = (gf_internal_t *) gf->scratch;
    rs = bytes / (h->w);

    written = (xor) ? 0xffffffff : 0;
    for (i = 0; i < h->w; i++) {
        for (j = 0; j < h->w; j++) {
            if (val & (1 << j)) {
                gf_multby_one(src, ((uint8_t *)dest) + j*rs, rs, (written & (1 << j)));
                written |= (1 << j);
            }
        }
        src = (uint8_t *)src + rs;
        val = gf->multiply.w32(gf, val, 2);
    }
}
```

```
int gf_wgen_init(gf_t *gf)
{
    gf_internal_t *h;

    h = (gf_internal_t *) gf->scratch;
    if (h->prim_poly == 0) {
        switch (h->w) {
            case 1: h->prim_poly = 1; break;
            case 2: h->prim_poly = 7; break;
            case 3: h->prim_poly = 013; break;
            case 4: h->prim_poly = 023; break;
            case 5: h->prim_poly = 045; break;
            case 6: h->prim_poly = 0103; break;
            case 7: h->prim_poly = 0211; break;
            case 8: h->prim_poly = 0435; break;
            case 9: h->prim_poly = 01021; break;
            case 10: h->prim_poly = 02011; break;
            case 11: h->prim_poly = 04005; break;
            case 12: h->prim_poly = 010123; break;
            case 13: h->prim_poly = 020033; break;
            case 14: h->prim_poly = 042103; break;
            case 15: h->prim_poly = 0100003; break;
            case 16: h->prim_poly = 0210013; break;
            case 17: h->prim_poly = 0400011; break;
            case 18: h->prim_poly = 01000201; break;
            case 19: h->prim_poly = 02000047; break;
            case 20: h->prim_poly = 04000011; break;
            case 21: h->prim_poly = 010000005; break;
            case 22: h->prim_poly = 020000003; break;
            case 23: h->prim_poly = 040000041; break;
            case 24: h->prim_poly = 0100000207; break;
            case 25: h->prim_poly = 0200000011; break;
            case 26: h->prim_poly = 0400000107; break;
        }
    }
}
```



*src/gf\_wgen.c lines 961 to 1019*

```
    case 27: h->prim_poly = 010000000047; break;
    case 28: h->prim_poly = 020000000011; break;
    case 29: h->prim_poly = 040000000005; break;
    case 30: h->prim_poly = 010040000007; break;
    case 31: h->prim_poly = 020000000011; break;
    case 32: h->prim_poly = 000200000007; break;
    default: fprintf(stderr, "gf_wgen_init: w not defined yet\n"); exit(1);
}
} else {
    if (h->w == 32) {
        h->prim_poly &= 0xffffffff;
    } else {
        h->prim_poly |= (1 << h->w);
        if (h->prim_poly & ~((1ULL<<(h->w+1))-1)) return 0;
    }
}

gf->multiply.w32 = NULL;
gf->divide.w32 = NULL;
gf->inverse.w32 = NULL;
gf->multiply_region.w32 = gf_wgen_cauchy_region;
gf->extract_word.w32 = gf_wgen_extract_word;

switch(h->mult_type) {
    case GF_MULT_DEFAULT:
        if (h->w <= 8) {
            if (gf_wgen_table_init(gf) == 0) return 0;
        } else if (h->w <= 16) {
            if (gf_wgen_log_init(gf) == 0) return 0;
        } else {
            if (gf_wgen_bytwo_p_init(gf) == 0) return 0;
        }
        break;
    case GF_MULT_SHIFT:      if (gf_wgen_shift_init(gf) == 0) return 0; break;
    case GF_MULT_BYTWO_b:    if (gf_wgen_bytwo_b_init(gf) == 0) return 0; break;
    case GF_MULT_BYTWO_p:    if (gf_wgen_bytwo_p_init(gf) == 0) return 0; break;
    case GF_MULT_GROUP:      if (gf_wgen_group_init(gf) == 0) return 0; break;
    case GF_MULT_TABLE:      if (gf_wgen_table_init(gf) == 0) return 0; break;
    case GF_MULT_LOG_TABLE:  if (gf_wgen_log_init(gf) == 0) return 0; break;
    default: return 0;
}
if (h->divide_type == GF_DIVIDE_EUCLID) {
    gf->divide.w32 = gf_wgen_divide_from_inverse;
    gf->inverse.w32 = gf_wgen_euclid;
} else if (h->divide_type == GF_DIVIDE_MATRIX) {
    gf->divide.w32 = gf_wgen_divide_from_inverse;
    gf->inverse.w32 = gf_wgen_matrix;
}

if (gf->inverse.w32== NULL && gf->divide.w32 == NULL) gf->inverse.w32 = gf_wgen_euclid;

if (gf->inverse.w32 != NULL && gf->divide.w32 == NULL) {
    gf->divide.w32 = gf_wgen_divide_from_inverse;
}
if (gf->inverse.w32 == NULL && gf->divide.w32 != NULL) {
    gf->inverse.w32 = gf_wgen_inverse_from_divide;
}
return 1;
}
```

*include/config.h lines 1 to 60*

```
/* include/config.h.  Generated from config.h.in by configure.  */
/* include/config.h.in.  Generated from configure.ac by autoheader.  */

/* Support Altivec instructions */
/* #undef HAVE_ALTIVEC */

/* Support AVX (Advanced Vector Extensions) instructions */
/* #undef HAVE_AVX */

/* Define to 1 if you have the <dlfcn.h> header file.  */
#define HAVE_DLFCN_H 1

/* Define to 1 if you have the <inttypes.h> header file.  */
#define HAVE_INTTYPES_H 1

/* Define to 1 if you have the <memory.h> header file.  */
#define HAVE_MEMORY_H 1

/* Support mmx instructions */
#define HAVE_MMX /**/

/* Support (PCLMULDQ) Carry-Free Multiplication */
/* #undef HAVE_PCLMULDQ */

/* Support SSE (Streaming SIMD Extensions) instructions */
#define HAVE_SSE /**/

/* Support SSE2 (Streaming SIMD Extensions 2) instructions */
#define HAVE_SSE2 /**/

/* Support SSE3 (Streaming SIMD Extensions 3) instructions */
#define HAVE_SSE3 /**/

/* Support SSSE4.1 (Streaming SIMD Extensions 4.1) instructions */
#define HAVE_SSSE4_1 /**/

/* Support SSSE4.2 (Streaming SIMD Extensions 4.2) instructions */
#define HAVE_SSSE4_2 /**/

/* Support SSSE3 (Supplemental Streaming SIMD Extensions 3) instructions */
#define HAVE_SSSE3 /**/

/* Define to 1 if you have the <stdint.h> header file.  */
#define HAVE_STDINT_H 1

/* Define to 1 if you have the <stdlib.h> header file.  */
#define HAVE_STDLIB_H 1

/* Define to 1 if you have the <strings.h> header file.  */
#define HAVE_STRINGS_H 1

/* Define to 1 if you have the <string.h> header file.  */
#define HAVE_STRING_H 1

/* Define to 1 if you have the <sys/stat.h> header file.  */
#define HAVE_SYS_STAT_H 1

/* Define to 1 if you have the <sys/types.h> header file.  */
#define HAVE_SYS_TYPES_H 1
```



# *include/config.h lines 61 to 93*

```
/* Define to 1 if you have the <unistd.h> header file. */
#define HAVE_UNISTD_H 1

/* Define to the sub-directory in which libtool stores uninstalled libraries.
   */
#define LT_OBJDIR ".libs/"

/* Name of package */
#define PACKAGE "gf-complete"

/* Define to the address where bug reports for this package should be sent. */
#define PACKAGE_BUGREPORT ""

/* Define to the full name of this package. */
#define PACKAGE_NAME "gf-complete"

/* Define to the full name and version of this package. */
#define PACKAGE_STRING "gf-complete 1.0"

/* Define to the one symbol short name of this package. */
#define PACKAGE_TARNAME "gf-complete"

/* Define to the home page for this package. */
#define PACKAGE_URL ""

/* Define to the version of this package. */
#define PACKAGE_VERSION "1.0"

/* Define to 1 if you have the ANSI C header files. */
#define STDC_HEADERS 1

/* Version number of package */
#define VERSION "1.0"
```

*include/gf\_complete.h lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_complete.h
 *
 * The main include file for gf_complete.
 */
```

```
#ifndef _GF_COMPLETE_H_
#define _GF_COMPLETE_H_
#include <stdint.h>
```

```
#ifdef INTEL_SSE4
    #include <nmmintrin.h>
#endif
```

```
#ifdef INTEL_SSSE3
    #include <tmmintrin.h>
#endif
```

```
#ifdef INTEL_SSE2
    #include <emmintrin.h>
#endif
```

```
#ifdef INTEL_SSE4_PCLMUL
    #include <wmmmintrin.h>
#endif
```

```
/* These are the different ways to perform multiplication.
   Not all are implemented for all values of w.
   See the paper for an explanation of how they work. */
```

```
typedef enum {GF_MULT_DEFAULT,
              GF_MULT_SHIFT,
              GF_MULT_CARRY_FREE,
              GF_MULT_CARRY_FREE_GK,
              GF_MULT_GROUP,
              GF_MULT_BYTWO_p,
              GF_MULT_BYTWO_b,
              GF_MULT_TABLE,
              GF_MULT_LOG_TABLE,
              GF_MULT_LOG_ZERO,
              GF_MULT_LOG_ZERO_EXT,
              GF_MULT_SPLIT_TABLE,
              GF_MULT_COMPOSITE } gf_mult_type_t;
```

```
/* These are the different ways to optimize region
   operations. They are bits because you can compose them.
   Certain optimizations only apply to certain gf_mult_type_t's.
   Again, please see documentation for how to use these */
```

```
#define GF_REGION_DEFAULT      (0x0)
#define GF_REGION_DOUBLE_TABLE (0x1)
#define GF_REGION_QUAD_TABLE  (0x2)
#define GF_REGION_LAZY         (0x4)
#define GF_REGION_SSE          (0x8)
#define GF_REGION_NOSSE        (0x10)
```



*include/gf\_complete.h lines 61 to 120*

```
#define GF_REGION_ALTMAP          (0x20)
#define GF_REGION_CAUCHY         (0x40)

typedef uint32_t gf_region_type_t;

/* These are different ways to implement division.
   Once again, it's best to use "DEFAULT". However,
   there are times when you may want to experiment
   with the others. */

typedef enum { GF_DIVIDE_DEFAULT,
               GF_DIVIDE_MATRIX,
               GF_DIVIDE_EUCLID } gf_division_type_t;

/* We support w=4,8,16,32,64 and 128 with their own data types and
   operations for multiplication, division, etc. We also support
   a "gen" type so that you can do general gf arithmetic for any
   value of w from 1 to 32. You can perform a "region" operation
   on these if you use "CAUCHY" as the mapping.
   */

typedef uint32_t    gf_val_32_t;
typedef uint64_t    gf_val_64_t;
typedef uint64_t    *gf_val_128_t;

extern int _gf_errno;
extern void gf_error();

typedef struct gf *GFP;

typedef union gf_func_a_b {
    gf_val_32_t    (*w32) (GFP gf, gf_val_32_t a, gf_val_32_t b);
    gf_val_64_t    (*w64) (GFP gf, gf_val_64_t a, gf_val_64_t b);
    void           (*w128) (GFP gf, gf_val_128_t a, gf_val_128_t b, gf_val_128_t c);
} gf_func_a_b;

typedef union {
    gf_val_32_t    (*w32) (GFP gf, gf_val_32_t a);
    gf_val_64_t    (*w64) (GFP gf, gf_val_64_t a);
    void           (*w128) (GFP gf, gf_val_128_t a, gf_val_128_t b);
} gf_func_a;

typedef union {
    void    (*w32) (GFP gf, void *src, void *dest, gf_val_32_t val, int bytes, int add);
    void    (*w64) (GFP gf, void *src, void *dest, gf_val_64_t val, int bytes, int add);
    void    (*w128) (GFP gf, void *src, void *dest, gf_val_128_t val, int bytes, int add);
} gf_region;

typedef union {
    gf_val_32_t    (*w32) (GFP gf, void *start, int bytes, int index);
    gf_val_64_t    (*w64) (GFP gf, void *start, int bytes, int index);
    void           (*w128) (GFP gf, void *start, int bytes, int index, gf_val_128_t rv);
} gf_extract;

typedef struct gf {
    gf_func_a_b    multiply;
    gf_func_a_b    divide;
    gf_func_a      inverse;
    gf_region      multiply_region;
    gf_extract      extract_word;
}
```

*include/gf\_complete.h lines 121 to 180*

```
void          *scratch;
} gf_t;

/* Initializes the GF to defaults.  Pass it a pointer to a gf_t.
   Returns 0 on failure, 1 on success. */

extern int gf_init_easy(GFP gf, int w);

/* Initializes the GF changing the defaults.
   Returns 0 on failure, 1 on success.
   Pass it a pointer to a gf_t.
   For mult_type and divide_type, use one of gf_mult_type_t gf_divide_type_t .
   For region_type, OR together the GF_REGION_xxx's defined above.
   Use 0 as prim_poly for defaults.  Otherwise, the leading 1 is optional.
   Use NULL for scratch_memory to have init_hard allocate memory.  Otherwise,
   use gf_scratch_size() to determine how big scratch_memory has to be.
*/

extern int gf_init_hard(GFP gf,
                        int w,
                        int mult_type,
                        int region_type,
                        int divide_type,
                        uint64_t prim_poly,
                        int arg1,
                        int arg2,
                        GFP base_gf,
                        void *scratch_memory);

/* Determines the size for scratch_memory.
   Returns 0 on failure and non-zero on success. */

extern int gf_scratch_size(int w,
                           int mult_type,
                           int region_type,
                           int divide_type,
                           int arg1,
                           int arg2);

/* This reports the gf_scratch_size of a gf_t that has already been created */

extern int gf_size(GFP gf);

/* Frees scratch memory if gf_init_easy/gf_init_hard called malloc.
   If recursive = 1, then it calls itself recursively on base_gf. */

extern int gf_free(GFP gf, int recursive);

/* This is support for inline single multiplications and divisions.
   I know it's yucky, but if you've got to be fast, you've got to be fast.
   We support inlining for w=4, w=8 and w=16.

   To use inline multiplication and division with w=4 or 8, you should use the
   default gf_t, or one with a single table.  Otherwise, gf_w4/8_get_mult_table()
   will return NULL.  Similarly, with w=16, the gf_t must be LOG */

uint8_t *gf_w4_get_mult_table(GFP gf);
uint8_t *gf_w4_get_div_table(GFP gf);

#define GF_W4_INLINE_MULTDIV(table, a, b) (table[((a)<<4)|(b)])
```



*include/gf\_complete.h lines 181 to 193*

```
uint8_t *gf_w8_get_mult_table(GFP gf);  
uint8_t *gf_w8_get_div_table(GFP gf);
```

```
#define GF_W8_INLINE_MULTDIV(table, a, b) (table[(((uint32_t) (a))<<8)|(b)])
```

```
uint16_t *gf_w16_get_log_table(GFP gf);  
uint16_t *gf_w16_get_mult_alog_table(GFP gf);  
uint16_t *gf_w16_get_div_alog_table(GFP gf);
```

```
#define GF_W16_INLINE_MULT(log, alog, a, b) ((a) == 0 || (b) == 0) ? 0 : (alog[(uint32_t)log[a]+(uint32_t)log[b]])  
#define GF_W16_INLINE_DIV(log, alog, a, b) ((a) == 0 || (b) == 0) ? 0 : (alog[(int)log[a]-(int)log[b]])  
#endif
```

*include/gf\_general.h lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_general.h
 *
 * This file has helper routines for doing basic GF operations with any
 * legal value of w. The problem is that w <= 32, w=64 and w=128 all have
 * different data types, which is a pain. The procedures in this file try
 * to alleviate that pain. They are used in gf_unit and gf_time.
 */

#pragma once

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#include "gf_complete.h"

typedef union {
    uint32_t w32;
    uint64_t w64;
    uint64_t w128[2];
} gf_general_t;

void gf_general_set_zero(gf_general_t *v, int w);
void gf_general_set_one(gf_general_t *v, int w);
void gf_general_set_two(gf_general_t *v, int w);

int gf_general_is_zero(gf_general_t *v, int w);
int gf_general_is_one(gf_general_t *v, int w);
int gf_general_are_equal(gf_general_t *v1, gf_general_t *v2, int w);

void gf_general_val_to_s(gf_general_t *v, int w, char *s, int hex);
int gf_general_s_to_val(gf_general_t *v, int w, char *s, int hex);

void gf_general_set_random(gf_general_t *v, int w, int zero_ok);

void gf_general_add(gf_t *gf, gf_general_t *a, gf_general_t *b, gf_general_t *c);
void gf_general_multiply(gf_t *gf, gf_general_t *a, gf_general_t *b, gf_general_t *c);
void gf_general_divide(gf_t *gf, gf_general_t *a, gf_general_t *b, gf_general_t *c);
void gf_general_inverse(gf_t *gf, gf_general_t *a, gf_general_t *b);

void gf_general_do_region_multiply(gf_t *gf, gf_general_t *a,
                                   void *ra, void *rb,
                                   int bytes, int xor);

void gf_general_do_region_check(gf_t *gf, gf_general_t *a,
                                void *orig_a, void *orig_target, void *final_target,
                                int bytes, int xor);

/* Which is M, D or I for multiply, divide or inverse. */

void gf_general_set_up_single_timing_test(int w, void *ra, void *rb, int size);
```



*include/gf\_general.h lines 61 to 61*

```
int  gf_general_do_single_timing_test(gf_t *gf, void *ra, void *rb, int size, char which);
```

*include/gf\_int.h lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_int.h
 *
 * Internal code for Galois field routines. This is not meant for
 * users to include, but for the internal GF files to use.
 */

#pragma once

#include "gf_complete.h"

#include <string.h>

extern void      timer_start (double *t);
extern double    timer_split (const double *t);
extern void      galois_fill_random (void *buf, int len, unsigned int seed);

typedef struct {
    int mult_type;
    int region_type;
    int divide_type;
    int w;
    uint64_t prim_poly;
    int free_me;
    int arg1;
    int arg2;
    gf_t *base_gf;
    void *private;
} gf_internal_t;

extern int gf_w4_init (gf_t *gf);
extern int gf_w4_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2);

extern int gf_w8_init (gf_t *gf);
extern int gf_w8_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2);

extern int gf_w16_init (gf_t *gf);
extern int gf_w16_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2);

extern int gf_w32_init (gf_t *gf);
extern int gf_w32_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2);

extern int gf_w64_init (gf_t *gf);
extern int gf_w64_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2);

extern int gf_w128_init (gf_t *gf);
extern int gf_w128_scratch_size(int mult_type, int region_type, int divide_type, int arg1, int arg2);

extern int gf_wgen_init (gf_t *gf);
extern int gf_wgen_scratch_size(int w, int mult_type, int region_type, int divide_type, int arg1, int arg2);

void gf_wgen_cauchy_region(gf_t *gf, void *src, void *dest, gf_val_32_t val, int bytes, int xor);
gf_val_32_t gf_wgen_extract_word(gf_t *gf, void *start, int bytes, int index);

extern void gf_alignment_error(char *s, int a);
```



*include/gf\_int.h lines 61 to 120*

```
extern uint32_t gf_bitmatrix_inverse(uint32_t y, int w, uint32_t pp);
```

```
/* This returns the correct default for prim_poly when base is used as the base
   field for COMPOSITE. It returns 0 if we don't have a default prim_poly. */
```

```
extern uint64_t gf_composite_get_default_poly(gf_t *base);
```

```
/* This structure lets you define a region multiply. It helps because you can handle
   unaligned portions of the data with the procedures below, which really cleans
   up the code. */
```

```
typedef struct {
    gf_t *gf;
    void *src;
    void *dest;
    int bytes;
    uint64_t val;
    int xor;
    int align;           /* The number of bytes to which to align. */
    void *s_start;       /* The start and the top of the aligned region. */
    void *d_start;
    void *s_top;
    void *d_top;
} gf_region_data;
```

```
/* This lets you set up one of these in one call. It also sets the start/top pointers. */
```

```
void gf_set_region_data(gf_region_data *rd,
                        gf_t *gf,
                        void *src,
                        void *dest,
                        int bytes,
                        uint64_t val,
                        int xor,
                        int align);
```

```
/* This performs gf->multiply.32() on all of the unaligned bytes in the beginning of the region */
```

```
extern void gf_do_initial_region_alignment(gf_region_data *rd);
```

```
/* This performs gf->multiply.32() on all of the unaligned bytes in the end of the region */
```

```
extern void gf_do_final_region_alignment(gf_region_data *rd);
```

```
extern void gf_two_byte_region_table_multiply(gf_region_data *rd, uint16_t *base);
```

```
extern void gf_multby_zero(void *dest, int bytes, int xor);
```

```
extern void gf_multby_one(void *src, void *dest, int bytes, int xor);
```

```
typedef enum {GF_E_MDEFDIV, /* Dev != Default && Mult == Default */
              GF_E_MDEFREG, /* Reg != Default && Mult == Default */
              GF_E_MDEFARG, /* Args != Default && Mult == Default */
              GF_E_DIVCOMP, /* Mult == Composite && Div != Default */
              GF_E_CAUCOMP, /* Mult == Composite && Reg == CAUCHY */
              GF_E_DOUQUAD, /* Reg == DOUBLE && Reg == QUAD */
              GF_E_SSE_NO, /* Reg == SSE && Reg == NOSSE */
              GF_E_CAUCHYB, /* Reg == CAUCHY && Other Reg */
              GF_E_CAUGT32, /* Reg == CAUCHY && w > 32 */
              GF_E_ARG1SET, /* Arg1 != 0 && Mult \notin COMPOSITE/SPLIT/GROUP */
              GF_E_ARG2SET, /* Arg2 != 0 && Mult \notin SPLIT/GROUP */
```



*include/gf\_int.h lines 121 to 180*

```
GF_E_MATRIXW, /* Div == MATRIX && w > 32 */
GF_E_BAD_W, /* Illegal w */
GF_E_DOUBLET, /* Reg == DOUBLE && Mult != TABLE */
GF_E_DOUBLEW, /* Reg == DOUBLE && w \notin {4,8} */
GF_E_DOUBLEJ, /* Reg == DOUBLE && other Reg */
GF_E_DOUBLEL, /* Reg == DOUBLE & LAZY but w = 4 */
GF_E_QUAD_T, /* Reg == QUAD && Mult != TABLE */
GF_E_QUAD_W, /* Reg == QUAD && w != 4 */
GF_E_QUAD_J, /* Reg == QUAD && other Reg */
GF_E_LAZY_X, /* Reg == LAZY && not DOUBLE or QUAD */
GF_E_ALTSHIF, /* Mult == Shift && Reg == ALTMAP */
GF_E_SSESHIF, /* Mult == Shift && Reg == SSE|NOSSE */
GF_E_ALT_CFM, /* Mult == CARRY_FREE && Reg == ALTMAP */
GF_E_SSE_CFM, /* Mult == CARRY_FREE && Reg == SSE|NOSSE */
GF_E_PCLMULX, /* Mult == Carry_Free && No PCLMUL */
GF_E_ALT_BY2, /* Mult == Bytwo_x && Reg == ALTMAP */
GF_E_BY2_SSE, /* Mult == Bytwo_x && Reg == SSE && No SSE2 */
GF_E_LOGBADW, /* Mult == LOGx, w too big */
GF_E_LOG_J, /* Mult == LOGx, && Reg == SSE|ALTMAP|NOSSE */
GF_E_ZERBADW, /* Mult == LOG_ZERO, w \notin {8,16} */
GF_E_ZEXBADW, /* Mult == LOG_ZERO_EXT, w != 8 */
GF_E_LOGPOLY, /* Mult == LOG & poly not primitive */
GF_E_GR_ARGX, /* Mult == GROUP, Bad arg1/2 */
GF_E_GR_W_48, /* Mult == GROUP, w \in { 4, 8 } */
GF_E_GR_W_16, /* Mult == GROUP, w == 16, arg1 != 4 || arg2 != 4 */
GF_E_GR_128A, /* Mult == GROUP, w == 128, bad args */
GF_E_GR_A_27, /* Mult == GROUP, either arg > 27 */
GF_E_GR_AR_W, /* Mult == GROUP, either arg > w */
GF_E_GR_J, /* Mult == GROUP, Reg == SSE|ALTMAP|NOSSE */
GF_E_TABLE_W, /* Mult == TABLE, w too big */
GF_E_TAB_SSE, /* Mult == TABLE, SSE|NOSSE only apply to w == 4 */
GF_E_TABSSE3, /* Mult == TABLE, Need SSSE3 for SSE */
GF_E_TAB_ALT, /* Mult == TABLE, Reg == ALTMAP */
GF_E_SP128AR, /* Mult == SPLIT, w=128, Bad arg1/arg2 */
GF_E_SP128AL, /* Mult == SPLIT, w=128, SSE requires ALTMAP */
GF_E_SP128AS, /* Mult == SPLIT, w=128, ALTMAP requires SSE */
GF_E_SP128_A, /* Mult == SPLIT, w=128, ALTMAP only with 4/128 */
GF_E_SP128_S, /* Mult == SPLIT, w=128, SSE only with 4/128 */
GF_E_SPLIT_W, /* Mult == SPLIT, Bad w (8, 16, 32, 64, 128) */
GF_E_SP_16AR, /* Mult == SPLIT, w=16, Bad arg1/arg2 */
GF_E_SP_16_A, /* Mult == SPLIT, w=16, ALTMAP only with 4/16 */
GF_E_SP_16_S, /* Mult == SPLIT, w=16, SSE only with 4/16 */
GF_E_SP_32AR, /* Mult == SPLIT, w=32, Bad arg1/arg2 */
GF_E_SP_32AS, /* Mult == SPLIT, w=32, ALTMAP requires SSE */
GF_E_SP_32_A, /* Mult == SPLIT, w=32, ALTMAP only with 4/32 */
GF_E_SP_32_S, /* Mult == SPLIT, w=32, SSE only with 4/32 */
GF_E_SP_64AR, /* Mult == SPLIT, w=64, Bad arg1/arg2 */
GF_E_SP_64AS, /* Mult == SPLIT, w=64, ALTMAP requires SSE */
GF_E_SP_64_A, /* Mult == SPLIT, w=64, ALTMAP only with 4/64 */
GF_E_SP_64_S, /* Mult == SPLIT, w=64, SSE only with 4/64 */
GF_E_SP_8_AR, /* Mult == SPLIT, w=8, Bad arg1/arg2 */
GF_E_SP_8_A, /* Mult == SPLIT, w=8, no ALTMAP */
GF_E_SP_SSE3, /* Mult == SPLIT, Need SSSE3 for SSE */
GF_E_COMP_A2, /* Mult == COMP, arg1 must be = 2 */
GF_E_COMP_SS, /* Mult == COMP, SSE|NOSSE */
GF_E_COMP_W, /* Mult == COMP, Bad w. */
GF_E_UNKFLAG, /* Unknown flag in create_from.... */
GF_E_UNKNOWN, /* Unknown mult_type. */
GF_E_UNK_REG, /* Unknown region_type. */
GF_E_UNK_DIV, /* Unknown divide_type. */
```



*include/gf\_int.h lines 181 to 200*

```
GF_E_CFM___W, /* Mult == CFM, Bad w. */
GF_E_CFM4POL, /* Mult == CFM & Prim Poly has high bits set. */
GF_E_CFM8POL, /* Mult == CFM & Prim Poly has high bits set. */
GF_E_CF16POL, /* Mult == CFM & Prim Poly has high bits set. */
GF_E_CF32POL, /* Mult == CFM & Prim Poly has high bits set. */
GF_E_CF64POL, /* Mult == CFM & Prim Poly has high bits set. */
GF_E_FEWARGS, /* Too few args in argc/argv. */
GF_E_BADPOLY, /* Bad primitive polynomial -- too many bits set. */
GF_E_COMP_PP, /* Bad primitive polynomial -- bigger than sub-field. */
GF_E_COMPPPP, /* Can't derive a default pp for composite field. */
GF_E_BASE___W, /* Composite -- Base field is the wrong size. */
GF_E_TWOMULT, /* In create_from... two -m's. */
GF_E_TWO_DIV, /* In create_from... two -d's. */
GF_E_POLYSPC, /* Bad numbera after -p. */
GF_E_SPLITAR, /* Ran out of arguments in SPLIT */
GF_E_SPLITNU, /* Arguments not integers in SPLIT. */
GF_E_GROUPAR, /* Ran out of arguments in GROUP */
GF_E_GROUPNU, /* Arguments not integers in GROUP. */
GF_E_DEFAULT } gf_error_type_t;
```

*include/gf\_method.h lines 1 to 20*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_method.h
 *
 * Parses argv to figure out the flags and arguments.  Creates the gf.
 */
```

```
#pragma once
```

```
#include "gf_complete.h"
```

```
/* Parses argv starting at "starting".
```

```
    Returns 0 on failure.
```

```
    On success, it returns one past the last argument it read in argv. */
```

```
extern int create_gf_from_argv(gf_t *gf, int w, int argc, char **argv, int starting);
```



*include/gf\_rand.h lines 1 to 22*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_rand.h
 *
 * Random number generation, using the "Mother of All" random number generator.  */

#pragma once
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* These are all pretty self-explanatory */
uint32_t MOA_Random_32();
uint64_t MOA_Random_64();
void      MOA_Random_128(uint64_t *x);
uint32_t MOA_Random_W(int w, int zero_ok);
void MOA_Fill_Random_Region(void *reg, int size);    /* reg should be aligned to 4 bytes, but
                                                    size can be anything. */
void      MOA_Seed(uint32_t seed);
```

*examples/gf\_example\_1.c lines 1 to 57*

```
#include <strings.h>
#include "gf_complete.h"
#include "gf_method.h"
#include "gf_rand.h"

int main(int argc, char **argv)
{
    uint16_t region[16*4];
    uint16_t prod[16*4];
    uint8_t *r8, *p8;
    gf_t gf;
    int i, j;

    r8 = (uint8_t *) region;
    p8 = (uint8_t *) prod;

    bzero(region, 16*4*2);
    region[0] = 0x640f; region[1] = 0x07e5; region[2] = 0x2fba; region[3] = 0xcdef;
    region[4] = 0xf1f9; region[5] = 0x3ab8; region[6] = 0xcd18; region[7] = 0x1d97;
    region[8] = 0x45a7; region[9] = 0x0160; region[10] = 0x0160; region[11] = 0x0160;
    region[12] = 0x0160; region[13] = 0x0160; region[14] = 0x0160; region[15] = 0x0160;

    if (create_gf_from_argv(&gf, 16, argc, argv, 1) == 0) {
        printf("Bad args\n");
        exit(1);
    }

    printf("Printing the first 16 16-bit words:\n");
    gf.multiply_region.w32(&gf, (void *) region, (void *) prod, 0x1234, 16*4*2, 0);
    printf("Reg: "); for (i = 0; i < 16; i++) printf(" %04x", region[i]); printf("\n");
    printf("Prod:"); for (i = 0; i < 16; i++) printf(" %04x", prod[i]); printf("\n");

    printf("Using extract_word\n");

    printf("Reg: ");
    for (i = 0; i < 16; i++) {
        printf(" %04x", gf.extract_word.w32(&gf, (void *) region, 16*4*2, i));
    }
    printf("\n");

    printf("Prod:");
    for (i = 0; i < 16; i++) {
        printf(" %04x", gf.extract_word.w32(&gf, (void *) prod, 16*4*2, i));
    }
    printf("\n");

    printf("Printing all 128 bytes of region (left) and prod (right):\n");
    for (i = 0; i < 128; i += 8) {
        printf("%03d - %03d:", i, i+7);
        for (j = 0; j < 8; j++) printf(" %02x", r8[i+j]);
        printf(" ");
        for (j = 0; j < 8; j++) printf(" %02x", p8[i+j]);
        printf("\n");
    }

    exit(0);
}
```



*examples/gf\_example\_2.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_example_2.c
 *
 * Demonstrates using the procedures for examples in GF(2^w) for w <= 32.
 */
```

```
#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
```

```
#include "gf_complete.h"
#include "gf_rand.h"
```

```
void usage(char *s)
{
    fprintf(stderr, "usage: gf_example_2 w - w must be between 1 and 32\n");
    exit(1);
}
```

```
int main(int argc, char **argv)
{
```

```
    uint32_t a, b, c;
    uint8_t *r1, *r2;
    uint16_t *r16 = NULL;
    uint32_t *r32 = NULL;
    int w, i;
    gf_t gf;
```

```
    if (argc != 2) usage(NULL);
    w = atoi(argv[1]);
    if (w <= 0 || w > 32) usage("Bad w");
```

```
    /* Get two random numbers in a and b */
```

```
    MOA_Seed(time(0));
    a = MOA_Random_W(w, 0);
    b = MOA_Random_W(w, 0);
```

```
    /* Create the proper instance of the gf_t object using defaults: */
```

```
    gf_init_easy(&gf, w);
```

```
    /* And multiply a and b using the galois field: */
```

```
    c = gf.multiply.w32(&gf, a, b);
    printf("%u * %u = %u\n", a, b, c);
```

```
    /* Divide the product by a and b */
```

```
    printf("%u / %u = %u\n", c, a, gf.divide.w32(&gf, c, a));
    printf("%u / %u = %u\n", c, b, gf.divide.w32(&gf, c, b));
```

```
    /* If w is 4, 8, 16 or 32, do a very small region operation */
```

*examples/gf\_example\_2.c lines 61 to 107*

```
if (w == 4 || w == 8 || w == 16 || w == 32) {
    r1 = (uint8_t *) malloc(16);
    r2 = (uint8_t *) malloc(16);

    if (w == 4 || w == 8) {
        r1[0] = b;
        for (i = 1; i < 16; i++) r1[i] = MOA_Random_W(8, 1);
    } else if (w == 16) {
        r16 = (uint16_t *) r1;
        r16[0] = b;
        for (i = 1; i < 8; i++) r16[i] = MOA_Random_W(16, 1);
    } else {
        r32 = (uint32_t *) r1;
        r32[0] = b;
        for (i = 1; i < 4; i++) r32[i] = MOA_Random_W(32, 1);
    }

    gf.multiply_region.w32(&gf, r1, r2, a, 16, 0);

    printf("\nmultiply_region by 0x%x (%u)\n\n", a, a);
    printf("R1 (the source): ");
    if (w == 4) {
        for (i = 0; i < 16; i++) printf(" %x %x", r1[i] >> 4, r1[i] & 0xf);
    } else if (w == 8) {
        for (i = 0; i < 16; i++) printf(" %02x", r1[i]);
    } else if (w == 16) {
        for (i = 0; i < 8; i++) printf(" %04x", r16[i]);
    } else if (w == 32) {
        for (i = 0; i < 4; i++) printf(" %08x", r32[i]);
    }
    printf("\nR2 (the product): ");
    if (w == 4) {
        for (i = 0; i < 16; i++) printf(" %x %x", r2[i] >> 4, r2[i] & 0xf);
    } else if (w == 8) {
        for (i = 0; i < 16; i++) printf(" %02x", r2[i]);
    } else if (w == 16) {
        r16 = (uint16_t *) r2;
        for (i = 0; i < 8; i++) printf(" %04x", r16[i]);
    } else if (w == 32) {
        r32 = (uint32_t *) r2;
        for (i = 0; i < 4; i++) printf(" %08x", r32[i]);
    }
    printf("\n");
}
exit(0);
}
```



*examples/gf\_example\_3.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_example_3.c
 *
 * Identical to example_2 except it works in GF(2^64)
 */

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#include "gf_complete.h"
#include "gf_rand.h"

void usage(char *s)
{
    fprintf(stderr, "usage: gf_example_3\n");
    exit(1);
}

int main(int argc, char **argv)
{
    uint64_t a, b, c;
    uint64_t *r1, *r2;
    int i;
    gf_t gf;

    if (argc != 1) usage(NULL);

    /* Get two random numbers in a and b */

    MOA_Seed(time(0));
    a = MOA_Random_64();
    b = MOA_Random_64();

    /* Create the proper instance of the gf_t object using defaults: */

    gf_init_easy(&gf, 64);

    /* And multiply a and b using the galois field: */

    c = gf.multiply.w64(&gf, a, b);
    printf("%llx * %llx = %llx\n", (long long unsigned int) a, (long long unsigned int) b, (long long unsigned int) c);

    /* Divide the product by a and b */

    printf("%llx / %llx = %llx\n", (long long unsigned int) c,
           (long long unsigned int) a, (long long unsigned int) gf.divide.w64(&gf, c, a));
    printf("%llx / %llx = %llx\n", (long long unsigned int) c,
           (long long unsigned int) b, (long long unsigned int) gf.divide.w64(&gf, c, b));

    r1 = (uint64_t *) malloc(32);
    r2 = (uint64_t *) malloc(32);
```

*examples/gf\_example\_3.c lines 61 to 76*

```
    r1[0] = b;

    for (i = 1; i < 4; i++) r1[i] = MOA_Random_64();

    gf.multiply_region.w64(&gf, r1, r2, a, 32, 0);

    printf("\nmultiply_region by %llx\n\n", (long long unsigned int) a);
    printf("R1 (the source): ");
    for (i = 0; i < 4; i++) printf(" %016llx", (long long unsigned int) r1[i]);

    printf("\nR2 (the product): ");
    for (i = 0; i < 4; i++) printf(" %016llx", (long long unsigned int) r2[i]);
    printf("\n");

    exit(0);
}
```



*examples/gf\_example\_4.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_example_4.c
 *
 * Identical to example_3 except it works in GF(2^128)
 */
```

```
#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
```

```
#include "gf_complete.h"
#include "gf_rand.h"
```

```
#define LLUI (long long unsigned int)
```

```
void usage(char *s)
{
    fprintf(stderr, "usage: gf_example_3\n");
    exit(1);
}
```

```
int main(int argc, char **argv)
{
    uint64_t a[2], b[2], c[2];
    uint64_t *r1, *r2;
    int i;
    gf_t gf;
```

```
    if (argc != 1) usage(NULL);
```

```
    /* Get two random numbers in a and b */
```

```
    MOA_Seed(time(0));
    MOA_Random_128(a);
    MOA_Random_128(b);
```

```
    /* Create the proper instance of the gf_t object using defaults: */
```

```
    gf_init_easy(&gf, 128);
```

```
    /* And multiply a and b using the galois field: */
```

```
    gf.multiply.w128(&gf, a, b, c);
    printf("%016llx%016llx * %016llx%016llx =\n%016llx%016llx\n",
          LLUI a[0], LLUI a[1], LLUI b[0], LLUI b[1], LLUI c[0], LLUI c[1]);
```

```
    r1 = (uint64_t *) malloc(32);
    r2 = (uint64_t *) malloc(32);
```

```
    for (i = 0; i < 4; i++) r1[i] = MOA_Random_64();
```

```
    gf.multiply_region.w128(&gf, r1, r2, a, 32, 0);
```

*examples/gf\_example\_4.c lines 61 to 69*

```
printf("\nmultiply_region by %016llx%016llx\n\n", LLUI a[0], LLUI a[1]);
printf("R1 (the source): ");
for (i = 0; i < 4; i += 2) printf(" %016llx%016llx", LLUI r1[i], LLUI r1[i+1]);

printf("\nR2 (the product): ");
for (i = 0; i < 4; i += 2) printf(" %016llx%016llx", LLUI r2[i], LLUI r2[i+1]);
printf("\n");
exit(0);
}
```



*examples/gf\_example\_5.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_example_5.c
 *
 * Demonstrating altmap and extract_word
 */

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#include "gf_complete.h"
#include "gf_rand.h"

void usage(char *s)
{
    fprintf(stderr, "usage: gf_example_5\n");
    exit(1);
}

int main(int argc, char **argv)
{
    uint16_t *a, *b;
    int i, j;
    gf_t gf;

    if (gf_init_hard(&gf, 16, GF_MULT_SPLIT_TABLE, GF_REGION_ALTMAP, GF_DIVIDE_DEFAULT,
                    0, 16, 4, NULL, NULL) == 0) {
        fprintf(stderr, "gf_init_hard failed\n");
        exit(1);
    }

    a = (uint16_t *) malloc(200);
    b = (uint16_t *) malloc(200);

    a += 6;
    b += 6;

    MOA_Seed(0);

    for (i = 0; i < 30; i++) a[i] = MOA_Random_W(16, 1);

    gf.multiply_region.w32(&gf, a, b, 0x1234, 30*2, 0);

    printf("a: 0x%lx    b: 0x%lx\n", (unsigned long) a, (unsigned long) b);

    for (i = 0; i < 30; i += 10) {
        printf("\n");
        printf("    ");
        for (j = 0; j < 10; j++) printf(" %4d", i+j);
        printf("\n");

        printf("a:");
        for (j = 0; j < 10; j++) printf(" %04x", a[i+j]);
    }
}
```

*examples/gf\_example\_5.c lines 61 to 78*

```
    printf("\n");

    printf("b:");
    for (j = 0; j < 10; j++) printf(" %04x", b[i+j]);
    printf("\n");
    printf("\n");
}

for (i = 0; i < 15; i++) {
    printf("Word %2d: 0x%04x * 0x1234 = 0x%04x      ", i,
          gf.extract_word.w32(&gf, a, 30*2, i),
          gf.extract_word.w32(&gf, b, 30*2, i));
    printf("Word %2d: 0x%04x * 0x1234 = 0x%04x\n", i+15,
          gf.extract_word.w32(&gf, a, 30*2, i+15),
          gf.extract_word.w32(&gf, b, 30*2, i+15));
}
return 0;
}
```



*examples/gf\_example\_6.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_example_6.c
 *
 * Demonstrating altmap and extract_word
 */

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#include "gf_complete.h"
#include "gf_rand.h"

void usage(char *s)
{
    fprintf(stderr, "usage: gf_example_6\n");
    exit(1);
}

int main(int argc, char **argv)
{
    uint32_t *a, *b;
    int i, j;
    gf_t gf, gf_16;

    if (gf_init_hard(&gf_16, 16, GF_MULT_LOG_TABLE, GF_REGION_DEFAULT, GF_DIVIDE_DEFAULT,
                    0, 0, 0, NULL, NULL) == 0) {
        fprintf(stderr, "gf_init_hard (6) failed\n");
        exit(1);
    }

    if (gf_init_hard(&gf, 32, GF_MULT_COMPOSITE, GF_REGION_ALTMAP, GF_DIVIDE_DEFAULT,
                    0, 2, 0, &gf_16, NULL) == 0) {
        fprintf(stderr, "gf_init_hard (32) failed\n");
        exit(1);
    }

    a = (uint32_t *) malloc(200);
    b = (uint32_t *) malloc(200);

    a += 3;
    b += 3;

    MOA_Seed(0);

    for (i = 0; i < 30; i++) a[i] = MOA_Random_W(32, 1);

    gf.multiply_region.w32(&gf, a, b, 0x12345678, 30*4, 0);

    printf("a: 0x%lx      b: 0x%lx\n", (unsigned long) a, (unsigned long) b);

    for (i = 0; i < 30; i += 10) {
        printf("\n");
    }
}
```

*examples/gf\_example\_6.c lines 61 to 84*

```
    printf(" ");
    for (j = 0; j < 10; j++) printf(" %8d", i+j);
    printf("\n");

    printf("a:");
    for (j = 0; j < 10; j++) printf(" %08x", a[i+j]);
    printf("\n");

    printf("b:");
    for (j = 0; j < 10; j++) printf(" %08x", b[i+j]);
    printf("\n");
    printf("\n");
}

for (i = 0; i < 15; i++) {
    printf("Word %2d: 0x%08x * 0x12345678 = 0x%08x      ", i,
          gf.extract_word.w32(&gf, a, 30*4, i),
          gf.extract_word.w32(&gf, b, 30*4, i));
    printf("Word %2d: 0x%08x * 0x12345678 = 0x%08x\n", i+15,
          gf.extract_word.w32(&gf, a, 30*4, i+15),
          gf.extract_word.w32(&gf, b, 30*4, i+15));
}
return 0;
}
```



*examples/gf\_example\_7.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_example_7.c
 *
 * Demonstrating extract_word and Cauchy
 */

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#include "gf_complete.h"
#include "gf_rand.h"

void usage(char *s)
{
    fprintf(stderr, "usage: gf_example_7\n");
    exit(1);
}

int main(int argc, char **argv)
{
    uint8_t *a, *b;
    int i, j;
    gf_t gf;

    if (gf_init_hard(&gf, 3, GF_MULT_TABLE, GF_REGION_CAUCHY, GF_DIVIDE_DEFAULT, 0, 0, 0, NULL, NULL) == 0) {
        fprintf(stderr, "gf_init_hard failed\n");
        exit(1);
    }

    a = (uint8_t *) malloc(3);
    b = (uint8_t *) malloc(3);

    MOA_Seed(0);

    for (i = 0; i < 3; i++) a[i] = MOA_Random_W(8, 1);

    gf.multiply_region.w32(&gf, a, b, 5, 3, 0);

    printf("a: 0x%lx      b: 0x%lx\n", (unsigned long) a, (unsigned long) b);

    printf("\n");
    printf("a: 0x%02x 0x%02x 0x%02x\n", a[0], a[1], a[2]);
    printf("b: 0x%02x 0x%02x 0x%02x\n", b[0], b[1], b[2]);
    printf("\n");

    printf("a bits:");
    for (i = 0; i < 3; i++) {
        printf(" ");
        for (j = 7; j >= 0; j--) printf("%c", (a[i] & (1 << j)) ? '1' : '0');
    }
    printf("\n");
}
```

*examples/gf\_example\_7.c lines 61 to 75*

```
printf("b bits:");
for (i = 0; i < 3; i++) {
    printf(" ");
    for (j = 7; j >= 0; j--) printf("%c", (b[i] & (1 << j)) ? '1' : '0');
}
printf("\n");

printf("\n");
for (i = 0; i < 8; i++) {
    printf("Word %2d: %d * 5 = %d\n", i,
          gf.extract_word.w32(&gf, a, 3, i),
          gf.extract_word.w32(&gf, b, 3, i));
}
return 0;
}
```



*tools/gf\_add.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_add.c
 *
 * Adds two numbers in gf_2^w
 */

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

void usage(char *s)
{
    fprintf(stderr, "usage: gf_add a b w - does addition of a and b in GF(2^w)\n");
    fprintf(stderr, "      If w has an h on the end, treat a, b and the sum as hexadecimal (no 0x required)\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "      legal w are: 1-32, 64 and 128\n");
    fprintf(stderr, "      128 is hex only (i.e. '128' will be an error - do '128h')\n");

    if (s != NULL) fprintf(stderr, "%s", s);
    exit(1);
}

int read_128(char *s, uint64_t *v)
{
    int l, t;
    char save;

    l = strlen(s);
    if (l > 32) return 0;

    if (l > 16) {
        if (sscanf(s + (l-16), "%llx", (long long unsigned int *) &(v[1])) == 0) return 0;
        save = s[l-16];
        s[l-16] = '\0';
        t = sscanf(s, "%llx", (long long unsigned int *) &(v[0]));
        s[l-16] = save;
        return t;
    } else {
        v[0] = 0;
        return sscanf(s, "%llx", (long long unsigned int *)&(v[1]));
    }
    return 1;
}

void print_128(uint64_t *v)
{
    if (v[0] > 0) {
        printf("%llx", (long long unsigned int) v[0]);
        printf("%016llx", (long long unsigned int) v[1]);
    } else {
        printf("%llx", (long long unsigned int) v[1]);
    }
    printf("\n");
}
```

*tools/gf\_add.c lines 61 to 114*

```
int main(int argc, char **argv)
{
    int hex, w;
    uint32_t a, b, c, top;
    uint64_t a64, b64, c64;
    uint64_t a128[2], b128[2], c128[2];
    char *format;

    if (argc != 4) usage(NULL);
    if (sscanf(argv[3], "%d", &w) == 0) usage("Bad w\n");

    if (w <= 0 || (w > 32 && w != 64 && w != 128)) usage("Bad w");

    hex = (strchr(argv[3], 'h') != NULL);

    if (!hex && w == 128) usage(NULL);

    if (w <= 32) {
        format = (hex) ? "%x" : "%u";
        if (sscanf(argv[1], format, &a) == 0) usage("Bad a\n");
        if (sscanf(argv[2], format, &b) == 0) usage("Bad b\n");

        if (w < 32) {
            top = (w == 31) ? 0x80000000 : (1 << w);
            if (w != 32 && a >= top) usage("a is too large\n");
            if (w != 32 && b >= top) usage("b is too large\n");
        }

        c = a ^ b;
        printf(format, c);
        printf("\n");
    } else if (w == 64) {
        format = (hex) ? "%llx" : "%llu";
        if (sscanf(argv[1], format, &a64) == 0) usage("Bad a\n");
        if (sscanf(argv[2], format, &b64) == 0) usage("Bad b\n");
        c64 = a64 ^ b64;

        printf(format, c64);
        printf("\n");
    } else if (w == 128) {
        if (read_128(argv[1], a128) == 0) usage("Bad a\n");
        if (read_128(argv[2], b128) == 0) usage("Bad b\n");
        c128[0] = a128[0] ^ b128[0];
        c128[1] = a128[1] ^ b128[1];

        print_128(c128);
    }
    exit(0);
}
```



*tools/gf\_div.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_div.c
 *
 * Multiplies two numbers in gf_2^w
 */

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#include "gf_complete.h"
#include "gf_method.h"
#include "gf_general.h"

void usage(int why)
{
    fprintf(stderr, "usage: gf_div a b w [method] - does division of a and b in GF(2^w)\n");
    if (why == 'W') {
        fprintf(stderr, "Bad w.\n");
        fprintf(stderr, "Legal w are: 1 - 32, 64 and 128.\n");
        fprintf(stderr, "Append 'h' to w to treat a, b and the quotient as hexadecimal.\n");
        fprintf(stderr, "w=128 is hex only (i.e. '128' will be an error - do '128h')\n");
    }
    if (why == 'A') fprintf(stderr, "Bad a\n");
    if (why == 'B') fprintf(stderr, "Bad b\n");
    if (why == 'M') {
        fprintf(stderr, "Bad Method Specification: ");
        gf_error();
    }
    exit(1);
}

int main(int argc, char **argv)
{
    int hex, w;
    gf_t gf;
    gf_general_t a, b, c;
    char output[50];

    if (argc < 4) usage(' ');

    if (sscanf(argv[3], "%d", &w) == 0) usage('W');
    if (w <= 0 || (w > 32 && w != 64 && w != 128)) usage('W');

    hex = (strchr(argv[3], 'h') != NULL);
    if (!hex && w == 128) usage('W');

    if (argc == 4) {
        if (gf_init_easy(&gf, w) == 0) usage('M');
    } else {
        if (create_gf_from_argv(&gf, w, argc, argv, 4) == 0) usage('M');
    }

    if (!gf_general_s_to_val(&a, w, argv[1], hex)) usage('A');
```

*tools/gf\_div.c lines 61 to 68*

```
    if (!gf_general_s_to_val(&b, w, argv[2], hex)) usage('B');

    gf_general_divide(&gf, &a, &b, &c);
    gf_general_val_to_s(&c, w, output, hex);

    printf("%s\n", output);
    exit(0);
}
```



*tools/gf\_inline\_time.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_inline_time.c
 *
 * Times inline single multiplication when w = 4, 8 or 16
 */
```

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
```

```
#include "gf_complete.h"
#include "gf_rand.h"
```

```
void
timer_start (double *t)
{
    struct timeval  tv;

    gettimeofday (&tv, NULL);
    *t = (double)tv.tv_sec + (double)tv.tv_usec * 1e-6;
}
```

```
double
timer_split (const double *t)
{
    struct timeval  tv;
    double  cur_t;

    gettimeofday (&tv, NULL);
    cur_t = (double)tv.tv_sec + (double)tv.tv_usec * 1e-6;
    return (cur_t - *t);
}
```

```
void problem(char *s)
{
    fprintf(stderr, "Timing test failed.\n");
    fprintf(stderr, "%s\n", s);
    exit(1);
}
```

```
void usage(char *s)
{
    fprintf(stderr, "usage: gf_inline_time w seed #elts iterations - does timing of single multiplies\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "Legal w are: 4, 8 or 16\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "Use -1 for time(0) as a seed.\n");
    fprintf(stderr, "\n");
    if (s != NULL) fprintf(stderr, "%s\n", s);
    exit(1);
}
```

```
int main(int argc, char **argv)
```

*tools/gf\_inline\_time.c lines 61 to 120*

```
{
    int w, j, i, size, iterations;
    gf_t gf;
    double timer, elapsed, dnum, num;
    uint8_t *ra = NULL, *rb = NULL, *mult4, *mult8;
    uint16_t *ra16 = NULL, *rb16 = NULL, *log16, *alog16;
    time_t t0;

    if (argc != 5) usage(NULL);
    if (sscanf(argv[1], "%d", &w) == 0) usage("Bad w\n");
    if (w != 4 && w != 8 && w != 16) usage("Bad w\n");
    if (sscanf(argv[2], "%ld", &t0) == 0) usage("Bad seed\n");
    if (sscanf(argv[3], "%d", &size) == 0) usage("Bad #elts\n");
    if (sscanf(argv[4], "%d", &iterations) == 0) usage("Bad iterations\n");
    if (t0 == -1) t0 = time(0);
    MOA_Seed(t0);

    num = size;

    gf_init_easy(&gf, w);

    printf("Seed: %ld\n", t0);

    if (w == 4 || w == 8) {
        ra = (uint8_t *) malloc(size);
        rb = (uint8_t *) malloc(size);

        if (ra == NULL || rb == NULL) { perror("malloc"); exit(1); }
    } else if (w == 16) {
        ra16 = (uint16_t *) malloc(size*2);
        rb16 = (uint16_t *) malloc(size*2);

        if (ra16 == NULL || rb16 == NULL) { perror("malloc"); exit(1); }
    }

    if (w == 4) {
        mult4 = gf_w4_get_mult_table(&gf);
        if (mult4 == NULL) {
            printf("Couldn't get inline multiplication table.\n");
            exit(1);
        }
        elapsed = 0;
        dnum = 0;
        for (i = 0; i < iterations; i++) {
            for (j = 0; j < size; j++) {
                ra[j] = MOA_Random_W(w, 1);
                rb[j] = MOA_Random_W(w, 1);
            }
            timer_start(&timer);
            for (j = 0; j < size; j++) {
                ra[j] = GF_W4_INLINE_MUULTDIV(mult4, ra[j], rb[j]);
            }
            dnum += num;
            elapsed += timer_split(&timer);
        }
        printf("Inline mult:      %10.6lf s      Mops: %10.3lf      %10.3lf Mega-ops/s\n",
            elapsed, dnum/1024.0/1024.0, dnum/1024.0/1024.0/elapsed);
    } else if (w == 8) {
        mult8 = gf_w8_get_mult_table(&gf);
```



*tools/gf\_inline\_time.c lines 121 to 170*

```
    if (mult8 == NULL) {
        printf("Couldn't get inline multiplication table.\n");
        exit(1);
    }
    elapsed = 0;
    dnum = 0;
    for (i = 0; i < iterations; i++) {
        for (j = 0; j < size; j++) {
            ra[j] = MOA_Random_W(w, 1);
            rb[j] = MOA_Random_W(w, 1);
        }
        timer_start(&timer);
        for (j = 0; j < size; j++) {
            ra[j] = GF_W8_INLINE_MULTDIV(mult8, ra[j], rb[j]);
        }
        dnum += num;
        elapsed += timer_split(&timer);
    }
    printf("Inline mult:      %10.6lf s      Mops: %10.3lf      %10.3lf Mega-ops/s\n",
           elapsed, dnum/1024.0/1024.0, dnum/1024.0/1024.0/elapsed);
} else if (w == 16) {
    log16 = gf_w16_get_log_table(&gf);
    alog16 = gf_w16_get_mult_alog_table(&gf);
    if (log16 == NULL) {
        printf("Couldn't get inline multiplication table.\n");
        exit(1);
    }
    elapsed = 0;
    dnum = 0;
    for (i = 0; i < iterations; i++) {
        for (j = 0; j < size; j++) {
            ra16[j] = MOA_Random_W(w, 1);
            rb16[j] = MOA_Random_W(w, 1);
        }
        timer_start(&timer);
        for (j = 0; j < size; j++) {
            ra16[j] = GF_W16_INLINE_MULT(log16, alog16, ra16[j], rb16[j]);
        }
        dnum += num;
        elapsed += timer_split(&timer);
    }
    printf("Inline mult:      %10.6lf s      Mops: %10.3lf      %10.3lf Mega-ops/s\n",
           elapsed, dnum/1024.0/1024.0, dnum/1024.0/1024.0/elapsed);
}
free (ra);
free (rb);
free (ra16);
free (rb16);
return 0;
}
```

*tools/gf\_methods.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_methods.c
 *
 * Lists supported methods (incomplete w.r.t. GROUP and COMPOSITE)
 */

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#include "gf_complete.h"
#include "gf_method.h"
#include "gf_int.h"

#define BNMULTS (8)
static char *BMULTS[BNMULTS] = { "CARRY_FREE", "GROUP48",
                                "TABLE", "LOG", "SPLIT4", "SPLIT8", "SPLIT88", "COMPOSITE" };

#define NMULTS (17)
static char *MULTS[NMULTS] = { "SHIFT", "CARRY_FREE", "CARRY_FREE_GK", "GROUP44", "GROUP48", "BYTWO_p", "BYTWO_b",
                                "TABLE", "LOG", "LOG_ZERO", "LOG_ZERO_EXT", "SPLIT2",
                                "SPLIT4", "SPLIT8", "SPLIT16", "SPLIT88", "COMPOSITE" };

/* Make sure CAUCHY is last */

#define NREGIONS (7)
static char *REGIONS[NREGIONS] = { "DOUBLE", "QUAD", "LAZY", "SSE", "NOSSE",
                                    "ALTMAP", "CAUCHY" };

#define BNREGIONS (4)
static char *BREGIONS[BNREGIONS] = { "DOUBLE", "QUAD", "ALTMAP", "CAUCHY" };

#define NDIVS (2)
static char *divides[NDIVS] = { "MATRIX", "EUCLID" };

void usage(char *s)
{
    fprintf(stderr, "usage: gf_methods w -BADC -LUMDRB\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "      w can be 1-32, 64, 128\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "-B lists basic methods that are useful\n");
    fprintf(stderr, "-A does a nearly exhaustive listing\n");
    fprintf(stderr, "-D adds EUCLID and MATRIX division\n");
    fprintf(stderr, "-C adds CAUCHY when possible\n");
    fprintf(stderr, "Combinations are fine.\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "-L Simply lists methods\n");
    fprintf(stderr, "-U Produces calls to gf_unit\n");
    fprintf(stderr, "-M Produces calls to time_tool.sh for single multiplications\n");
    fprintf(stderr, "-D Produces calls to time_tool.sh for single divisions\n");
    fprintf(stderr, "-R Produces calls to time_tool.sh for region multiplications\n");
    fprintf(stderr, "-B Produces calls to time_tool.sh for the fastest region multiplications\n");
    fprintf(stderr, "Cannot combine L, U, T.\n");
    if (s != NULL) {
        fprintf(stderr, "\n");
    }
}
```



*tools/gf\_methods.c lines 61 to 120*

```
        fprintf(stderr, "%s\n", s);
    }
    exit(1);
}

int main(int argc, char *argv[])
{
    int m, r, d, w, i, sa, j, k, reset, ok;
    int nregions;
    int nmults;
    char **regions;
    char **mults;
    int exhaustive = 0;
    int divide = 0;
    int cauchy = 0;
    int listing;
    char *gf_argv[50], *x;
    gf_t gf;
    char ls[10];
    char * w_str;

    if (argc != 4) usage(NULL);
    w = atoi(argv[1]);
    ok = (w >= 1 && w <= 32);
    if (w == 64) ok = 1;
    if (w == 128) ok = 1;
    if (!ok) usage("Bad w");

    if (argv[2][0] != '-' || argv[3][0] != '-' || strlen(argv[2]) == 1 || strlen(argv[3]) != 2) {
        usage(NULL);
    }
    for (i = 1; argv[2][i] != '\0'; i++) {
        switch(argv[2][i]) {
            case 'B': exhaustive = 0; break;
            case 'A': exhaustive = 1; break;
            case 'D': divide = 1; break;
            case 'C': cauchy = 1; break;
            default: usage("Bad -BADC");
        }
    }

    if (strchr("LUMDRB", argv[3][1]) == NULL) { usage("Bad -LUMDRB"); }
    listing = argv[3][1];

    if (listing == 'U') {
        w_str = "../test/gf_unit %d A -1";
    } else if (listing == 'L') {
        w_str = "w=%d:";
    } else {
        w_str = strdup("sh time_tool.sh X %d");
        x = strchr(w_str, 'X');
        *x = listing;
    }

    gf_argv[0] = "-";
    if (create_gf_from_argv(&gf, w, 1, gf_argv, 0) > 0) {
        printf(w_str, w);
        printf("- \n");
        gf_free(&gf, 1);
    } else if (_gf_errno == GF_E_DEFAULT) {
```

*tools/gf\_methods.c lines 121 to 180*

```
    fprintf(stderr, "Unlabeled failed method: w=%d: -\n", 2);
    exit(1);
}
```

```
nregions = (exhaustive) ? NREGIONS : BNREGIONS;
if (!cauchy) nregions--;
regions = (exhaustive) ? REGIONS : BREGIONS;
mults = (exhaustive) ? MULTS : BMULTS;
nmults = (exhaustive) ? NMULTS : BNMULTS;
```

```
for (m = 0; m < nmults; m++) {
    sa = 0;
    gf_argv[sa++] = "-m";
    if (strcmp(mults[m], "GROUP44") == 0) {
        gf_argv[sa++] = "GROUP";
        gf_argv[sa++] = "4";
        gf_argv[sa++] = "4";
    } else if (strcmp(mults[m], "GROUP48") == 0) {
        gf_argv[sa++] = "GROUP";
        gf_argv[sa++] = "4";
        gf_argv[sa++] = "8";
    } else if (strcmp(mults[m], "SPLIT2") == 0) {
        gf_argv[sa++] = "SPLIT";
        sprintf(ls, "%d", w);
        gf_argv[sa++] = ls;
        gf_argv[sa++] = "2";
    } else if (strcmp(mults[m], "SPLIT4") == 0) {
        gf_argv[sa++] = "SPLIT";
        sprintf(ls, "%d", w);
        gf_argv[sa++] = ls;
        gf_argv[sa++] = "4";
    } else if (strcmp(mults[m], "SPLIT8") == 0) {
        gf_argv[sa++] = "SPLIT";
        sprintf(ls, "%d", w);
        gf_argv[sa++] = ls;
        gf_argv[sa++] = "8";
    } else if (strcmp(mults[m], "SPLIT16") == 0) {
        gf_argv[sa++] = "SPLIT";
        sprintf(ls, "%d", w);
        gf_argv[sa++] = ls;
        gf_argv[sa++] = "16";
    } else if (strcmp(mults[m], "SPLIT88") == 0) {
        gf_argv[sa++] = "SPLIT";
        gf_argv[sa++] = "8";
        gf_argv[sa++] = "8";
    } else if (strcmp(mults[m], "COMPOSITE") == 0) {
        gf_argv[sa++] = "COMPOSITE";
        gf_argv[sa++] = "2";
        gf_argv[sa++] = "-";
    } else {
        gf_argv[sa++] = mults[m];
    }
}
reset = sa;
```

```
for (r = 0; r < (1 << nregions); r++) {
    sa = reset;
    for (k = 0; k < nregions; k++) {
        if (r & (1 << k)) {
```



*tools/gf\_methods.c lines 181 to 228*

```
        gf_argv[sa++] = "-r";
        gf_argv[sa++] = regions[k];
    }
}
gf_argv[sa++] = "-";

/* printf("Hmmm. %s", gf_argv[0]);
for (j = 0; j < sa; j++) printf(" %s", gf_argv[j]);
printf("\n"); */

if (create_gf_from_argv(&gf, w, sa, gf_argv, 0) > 0) {
    printf(w_str, w);
    for (j = 0; j < sa; j++) printf(" %s", gf_argv[j]);
    printf("\n");
    gf_free(&gf, 1);
} else if (_gf_errno == GF_E_DEFAULT) {
    fprintf(stderr, "Unlabeled failed method: w=%d:", w);
    for (j = 0; j < sa; j++) fprintf(stderr, " %s", gf_argv[j]);
    fprintf(stderr, "\n");
    exit(1);
}
sa--;
if (divide) {
    for (d = 0; d < NDIVS; d++) {
        gf_argv[sa++] = "-d";
        gf_argv[sa++] = divides[d];
        /* printf("w=%d:", w);
           for (j = 0; j < sa; j++) printf(" %s", gf_argv[j]);
           printf("\n"); */
        gf_argv[sa++] = "-";
        if (create_gf_from_argv(&gf, w, sa, gf_argv, 0) > 0) {
            printf(w_str, w);
            for (j = 0; j < sa; j++) printf(" %s", gf_argv[j]);
            printf("\n");
            gf_free(&gf, 1);
        } else if (_gf_errno == GF_E_DEFAULT) {
            fprintf(stderr, "Unlabeled failed method: w=%d:", w);
            for (j = 0; j < sa; j++) fprintf(stderr, " %s", gf_argv[j]);
            fprintf(stderr, "\n");
            exit(1);
        }
        sa-=3;
    }
}
}
}
return 0;
}
```

*tools/gf\_mult.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_mult.c
 *
 * Multiplies two numbers in gf_2^w
 */

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#include "gf_complete.h"
#include "gf_method.h"
#include "gf_general.h"

void usage(int why)
{
    fprintf(stderr, "usage: gf_mult a b w [method] - does multiplication of a and b in GF(2^w)\n");
    if (why == 'W') {
        fprintf(stderr, "Bad w.\n");
        fprintf(stderr, "Legal w are: 1 - 32, 64 and 128.\n");
        fprintf(stderr, "Append 'h' to w to treat a, b and the product as hexadecimal.\n");
        fprintf(stderr, "w=128 is hex only (i.e. '128' will be an error - do '128h')\n");
    }
    if (why == 'A') fprintf(stderr, "Bad a\n");
    if (why == 'B') fprintf(stderr, "Bad b\n");
    if (why == 'M') {
        fprintf(stderr, "Bad Method Specification: ");
        gf_error();
    }
    exit(1);
}

int main(int argc, char **argv)
{
    int hex, w;
    gf_t gf;
    gf_general_t a, b, c;
    char output[50];

    if (argc < 4) usage(' ');

    if (sscanf(argv[3], "%d", &w) == 0) usage('W');
    if (w <= 0 || (w > 32 && w != 64 && w != 128)) usage('W');

    hex = (strchr(argv[3], 'h') != NULL);
    if (!hex && w == 128) usage('W');

    if (argc == 4) {
        if (gf_init_easy(&gf, w) == 0) usage('M');
    } else {
        if (create_gf_from_argv(&gf, w, argc, argv, 4) == 0) usage('M');
    }

    if (!gf_general_s_to_val(&a, w, argv[1], hex)) usage('A');
```



*tools/gf\_mult.c lines 61 to 68*

```
    if (!gf_general_s_to_val(&b, w, argv[2], hex)) usage('B');

    gf_general_multiply(&gf, &a, &b, &c);
    gf_general_val_to_s(&c, w, output, hex);

    printf("%s\n", output);
    exit(0);
}
```

*tools/gf\_poly.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_poly.c - program to help find irreducible polynomials in composite fields,
 * using the Ben-Or algorithm.
 *
 * (This one was written by Jim)
 *
 * Please see the following paper for a description of the Ben-Or algorithm:
 *
 * author      S. Gao and D. Panario
 * title       Tests and Constructions of Irreducible Polynomials over Finite Fields
 * booktitle   Foundations of Computational Mathematics
 * year        1997
 * publisher   Springer Verlag
 * pages       346-361
 *
 * The basic technique is this.  You have a polynomial f(x) whose coefficients are
 * in a base field GF(2^w).  The polynomial is of degree n.  You need to do the
 * following for all i from 1 to n/2:
 *
 * Construct x^(2^w)^i modulo f.  That will be a polynomial of maximum degree n-1
 * with coefficients in GF(2^w).  You construct that polynomial by starting with x
 * and doubling it w times, each time taking the result modulo f.  Then you
 * multiply that by itself i times, again each time taking the result modulo f.
 *
 * When you're done, you need to "subtract" x -- since addition = subtraction =
 * XOR, that means XOR x.
 *
 * Now, find the GCD of that last polynomial and f, using Euclid's algorithm.  If
 * the GCD is not one, then f is reducible.  If it is not reducible for each of
 * those i, then it is irreducible.
 *
 * In this code, I am using a gf_general_t to represent elements of GF(2^w).  This
 * is so that I can use base fields that are GF(2^64) or GF(2^128).
 *
 * I have two main procedures.  The first is x_to_q_to_i_minus_x, which calculates
 * x^(2^w)^i - x, putting the result into a gf_general_t * called retval.
 *
 * The second is gcd_one, which takes a polynomial of degree n and a second one
 * of degree n-1, and uses Euclid's algorithm to decide if their GCD == 1.
 *
 * These can be made faster (e.g. calculate x^(2^w) once and store it).
 */
```

```
#include "gf_complete.h"
#include "gf_method.h"
#include "gf_general.h"
#include "gf_int.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
```

```
char *BM = "Bad Method: ";
```

```
void usage(char *s)
{
```



*tools/gf\_poly.c lines 61 to 120*

```
fprintf(stderr, "usage: gf_poly w(base-field) method power:coef [ power:coef .. ]\n");
fprintf(stderr, "\n");
fprintf(stderr, "    use - for the default method.\n");
fprintf(stderr, "    use 0x in front of the coefficient if it's in hex\n");
fprintf(stderr, "\n");
fprintf(stderr, "    For example, to test whether x^2 + 2x + 1 is irreducible\n");
fprintf(stderr, "    in GF(2^16), the call is:\n");
fprintf(stderr, "\n");
fprintf(stderr, "    gf_poly 16 - 2:1 1:2 0:1\n");
fprintf(stderr, "\n");
fprintf(stderr, "    See the user's manual for more information.\n");
if (s != NULL) {
    fprintf(stderr, "\n");
    if (s == BM) {
        fprintf(stderr, "%s", s);
        gf_error();
    } else {
        fprintf(stderr, "%s\n", s);
    }
}
exit(1);
}
```

```
int gcd_one(gf_t *gf, int w, int n, gf_general_t *poly, gf_general_t *prod)
{
```

```
    gf_general_t *a, *b, zero, factor, p;
    int i, j, da, db;

    gf_general_set_zero(&zero, w);

    a = (gf_general_t *) malloc(sizeof(gf_general_t) * n+1);
    b = (gf_general_t *) malloc(sizeof(gf_general_t) * n);
    for (i = 0; i <= n; i++) gf_general_add(gf, &zero, poly+i, a+i);
    for (i = 0; i < n; i++) gf_general_add(gf, &zero, prod+i, b+i);
```

```
    da = n;
    while (1) {
        for (db = n-1; db >= 0 && gf_general_is_zero(b+db, w); db--) ;
        if (db < 0) return 0;
        if (db == 0) return 1;
        for (j = da; j >= db; j--) {
            if (!gf_general_is_zero(a+j, w)) {
                gf_general_divide(gf, a+j, b+db, &factor);
                for (i = 0; i <= db; i++) {
                    gf_general_multiply(gf, b+i, &factor, &p);
                    gf_general_add(gf, &p, a+(i+j-db), a+(i+j-db));
                }
            }
        }
        for (i = 0; i < n; i++) {
            gf_general_add(gf, a+i, &zero, &p);
            gf_general_add(gf, b+i, &zero, a+i);
            gf_general_add(gf, &p, &zero, b+i);
        }
    }
}
```

```
}
```

```
void x_to_q_to_i_minus_x(gf_t *gf, int w, int n, gf_general_t *poly, int logq, int i, gf_general_t *retval)
{
```

*tools/gf\_poly.c lines 121 to 180*

```
gf_general_t x;
gf_general_t *x_to_q;
gf_general_t *product;
gf_general_t p, zero, factor;
int j, k, lq;

gf_general_set_zero(&zero, w);
product = (gf_general_t *) malloc(sizeof(gf_general_t) * n*2);
x_to_q = (gf_general_t *) malloc(sizeof(gf_general_t) * n);
for (j = 0; j < n; j++) gf_general_set_zero(x_to_q+j, w);
gf_general_set_one(x_to_q+1, w);

for (lq = 0; lq < logq; lq++) {
    for (j = 0; j < n*2; j++) gf_general_set_zero(product+j, w);
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            gf_general_multiply(gf, x_to_q+j, x_to_q+k, &p);
            gf_general_add(gf, product+(j+k), &p, product+(j+k));
        }
    }
    for (j = n*2-1; j >= n; j--) {
        if (!gf_general_is_zero(product+j, w)) {
            gf_general_add(gf, product+j, &zero, &factor);
            for (k = 0; k <= n; k++) {
                gf_general_multiply(gf, poly+k, &factor, &p);
                gf_general_add(gf, product+(j-n+k), &p, product+(j-n+k));
            }
        }
    }
    for (j = 0; j < n; j++) gf_general_add(gf, product+j, &zero, x_to_q+j);
}
for (j = 0; j < n; j++) gf_general_set_zero(retval+j, w);
gf_general_set_one(retval, w);

while (i > 0) {
    for (j = 0; j < n*2; j++) gf_general_set_zero(product+j, w);
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            gf_general_multiply(gf, x_to_q+j, retval+k, &p);
            gf_general_add(gf, product+(j+k), &p, product+(j+k));
        }
    }
    for (j = n*2-1; j >= n; j--) {
        if (!gf_general_is_zero(product+j, w)) {
            gf_general_add(gf, product+j, &zero, &factor);
            for (k = 0; k <= n; k++) {
                gf_general_multiply(gf, poly+k, &factor, &p);
                gf_general_add(gf, product+(j-n+k), &p, product+(j-n+k));
            }
        }
    }
    for (j = 0; j < n; j++) gf_general_add(gf, product+j, &zero, retval+j);
    i--;
}

gf_general_set_one(&x, w);
gf_general_add(gf, &x, retval+1, retval+1);

free(product);
free(x_to_q);
```



*tools/gf\_poly.c lines 181 to 240*

```
}

int main(int argc, char **argv)
{
    int w, i, power, n, ap, success;
    gf_t gf;
    gf_general_t *poly, *prod;
    char *string, *ptr;
    char buf[100];

    if (argc < 4) usage(NULL);

    if (sscanf(argv[1], "%d", &w) != 1 || w <= 0) usage("Bad w.");
    ap = create_gf_from_argv(&gf, w, argc, argv, 2);

    if (ap == 0) usage(BM);

    if (ap == argc) usage("No powers/coefficients given.");

    n = -1;
    for (i = ap; i < argc; i++) {
        if (strchr(argv[i], ':') == NULL || sscanf(argv[i], "%d:", &power) != 1) {
            string = (char *) malloc(sizeof(char)*(strlen(argv[i])+100));
            sprintf(string, "Argument '%s' not in proper format of power:coefficient\n", argv[i]);
            usage(string);
        }
        if (power < 0) {
            usage("Can't have negative powers\n");
        } else {
            n = power;
        }
    }
    // in case the for-loop header fails
    assert (n >= 0);

    poly = (gf_general_t *) malloc(sizeof(gf_general_t)*(n+1));
    for (i = 0; i <= n; i++) gf_general_set_zero(poly+i, w);
    prod = (gf_general_t *) malloc(sizeof(gf_general_t)*n);

    for (i = ap; i < argc; i++) {
        sscanf(argv[i], "%d:", &power);
        ptr = strchr(argv[i], ':');
        ptr++;
        if (strncmp(ptr, "0x", 2) == 0) {
            success = gf_general_s_to_val(poly+power, w, ptr+2, 1);
        } else {
            success = gf_general_s_to_val(poly+power, w, ptr, 0);
        }
        if (success == 0) {
            string = (char *) malloc(sizeof(char)*(strlen(argv[i])+100));
            sprintf(string, "Argument '%s' not in proper format of power:coefficient\n", argv[i]);
            usage(string);
        }
    }

    printf("Poly:");
    for (power = n; power >= 0; power--) {
        if (!gf_general_is_zero(poly+power, w)) {
            printf("%s", (power == n) ? " " : " + ");
            if (!gf_general_is_one(poly+power, w)) {
```

*tools/gf\_poly.c lines 241 to 275*

```
gf_general_val_to_s(poly+power, w, buf, 1);
if (n > 0) {
    printf("(0x%s)", buf);
} else {
    printf("0x%s", buf);
}
}
if (power == 0) {
    if (gf_general_is_one(poly+power, w)) printf("1");
} else if (power == 1) {
    printf("x");
} else {
    printf("x^%d", power);
}
}
}
printf("\n");

if (!gf_general_is_one(poly+n, w)) {
    printf("\n");
    printf("Can't do Ben-Or, because the polynomial is not monic.\n");
    exit(0);
}

for (i = 1; i <= n/2; i++) {
    x_to_q_to_i_minus_x(&gf, w, n, poly, w, i, prod);
    if (!gcd_one(&gf, w, n, poly, prod)) {
        printf("Reducible.\n");
        exit(0);
    }
}

printf("Irreducible.\n");
exit(0);
}
```



*tools/gf\_time.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_time.c
 *
 * Performs timing for gf arithmetic
 */

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>

#include "gf_complete.h"
#include "gf_method.h"
#include "gf_rand.h"
#include "gf_general.h"

void
timer_start (double *t)
{
    struct timeval  tv;

    gettimeofday (&tv, NULL);
    *t = (double)tv.tv_sec + (double)tv.tv_usec * 1e-6;
}

double
timer_split (const double *t)
{
    struct timeval  tv;
    double  cur_t;

    gettimeofday (&tv, NULL);
    cur_t = (double)tv.tv_sec + (double)tv.tv_usec * 1e-6;
    return (cur_t - *t);
}

void problem(char *s)
{
    fprintf(stderr, "Timing test failed.\n");
    fprintf(stderr, "%s\n", s);
    exit(1);
}

char *BM = "Bad Method: ";

void usage(char *s)
{
    fprintf(stderr, "usage: gf_time w tests seed size(bytes) iterations [method [params]] - does timing\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "does unit testing in GF(2^w)\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "Legal w are: 1 - 32, 64 and 128\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "Tests may be any combination of:\n");
}
```

*tools/gf\_time.c lines 61 to 120*

```
fprintf(stderr, "      A: All\n");
fprintf(stderr, "      S: All Single Operations\n");
fprintf(stderr, "      R: All Region Operations\n");
fprintf(stderr, "      M: Single: Multiplications\n");
fprintf(stderr, "      D: Single: Divisions\n");
fprintf(stderr, "      I: Single: Inverses\n");
fprintf(stderr, "      G: Region: Buffer-Constant Multiplication\n");
fprintf(stderr, "      0: Region: Doing nothing, and bzero()\n");
fprintf(stderr, "      1: Region: Memcpy() and XOR\n");
fprintf(stderr, "      2: Region: Multiplying by two\n");
fprintf(stderr, "\n");
fprintf(stderr, "Use -1 for time(0) as a seed.\n");
fprintf(stderr, "\n");
if (s == BM) {
    fprintf(stderr, "%s", BM);
    gf_error();
} else if (s != NULL) {
    fprintf(stderr, "%s\n", s);
}
exit(1);
}

int main(int argc, char **argv)
{
    int w, it, i, size, iterations, xor;
    char tests[100];
    char test;
    char *single_tests = "MDI";
    char *region_tests = "G012";
    char *tstrings[256];
    void *tmetholds[256];
    gf_t gf;
    double timer, elapsed, ds, di, dnum;
    int num;
    time_t t0;
    uint8_t *ra, *rb;
    gf_general_t a;

    if (argc < 6) usage(NULL);

    if (sscanf(argv[1], "%d", &w) == 0){
        usage("Bad w[-pp]\n");
    }

    if (sscanf(argv[3], "%ld", &t0) == 0) usage("Bad seed\n");
    if (sscanf(argv[4], "%d", &size) == 0) usage("Bad size\n");
    if (sscanf(argv[5], "%d", &iterations) == 0) usage("Bad iterations\n");
    if (t0 == -1) t0 = time(0);
    MOA_Seed(t0);

    ds = size;
    di = iterations;

    if ((w > 32 && w != 64 && w != 128) || w < 0) usage("Bad w");
    if ((size * 8) % w != 0) usage ("Bad size -- must be a multiple of w*8\n");

    if (!create_gf_from_argv(&gf, w, argc, argv, 6)) usage(BM);
```



*tools/gf\_time.c lines 121 to 180*

```
strcpy(tests, "");
for (i = 0; argv[2][i] != '\0'; i++) {
    switch(argv[2][i]) {
        case 'A': strcat(tests, single_tests);
                  strcat(tests, region_tests);
                  break;
        case 'S': strcat(tests, single_tests); break;
        case 'R': strcat(tests, region_tests); break;
        case 'G': strcat(tests, "G"); break;
        case '0': strcat(tests, "0"); break;
        case '1': strcat(tests, "1"); break;
        case '2': strcat(tests, "2"); break;
        case 'M': strcat(tests, "M"); break;
        case 'D': strcat(tests, "D"); break;
        case 'I': strcat(tests, "I"); break;
        default: usage("Bad tests");
    }
}

tstrings['M'] = "Multiply";
tstrings['D'] = "Divide";
tstrings['I'] = "Inverse";
tstrings['G'] = "Region-Random";
tstrings['0'] = "Region-By-Zero";
tstrings['1'] = "Region-By-One";
tstrings['2'] = "Region-By-Two";

tmethods['M'] = (void *) gf.multiply.w32;
tmethods['D'] = (void *) gf.divide.w32;
tmethods['I'] = (void *) gf.inverse.w32;
tmethods['G'] = (void *) gf.multiply_region.w32;
tmethods['0'] = (void *) gf.multiply_region.w32;
tmethods['1'] = (void *) gf.multiply_region.w32;
tmethods['2'] = (void *) gf.multiply_region.w32;

printf("Seed: %ld\n", t0);

ra = (uint8_t *) malloc(size);
rb = (uint8_t *) malloc(size);

if (ra == NULL || rb == NULL) { perror("malloc"); exit(1); }

for (i = 0; i < 3; i++) {
    test = single_tests[i];
    if (strchr(tests, test) != NULL) {
        if (tmethods[(int)test] == NULL) {
            printf("No %s method.\n", tstrings[(int)test]);
        } else {
            elapsed = 0;
            dnum = 0;
            for (it = 0; it < iterations; it++) {
                gf_general_set_up_single_timing_test(w, ra, rb, size);
                timer_start(&timer);
                num = gf_general_do_single_timing_test(&gf, ra, rb, size, test);
                dnum += num;
                elapsed += timer_split(&timer);
            }
            printf("%14s:          %10.6lf s    Mops: %10.3lf      %10.3lf Mega-ops/s\n",
                  tstrings[(int)test], elapsed,
                  dnum/1024.0/1024.0, dnum/1024.0/1024.0/elapsed);
        }
    }
}
```

*tools/gf\_time.c lines 181 to 212*

```
    }  
  }  
}  
  
for (i = 0; i < 4; i++) {  
  test = region_tests[i];  
  if (strchr(tests, test) != NULL) {  
    if (tmethoeds[(int)test] == NULL) {  
      printf("No %s method.\n", tstrings[(int)test]);  
    } else {  
      if (test == '0') gf_general_set_zero(&a, w);  
      if (test == '1') gf_general_set_one(&a, w);  
      if (test == '2') gf_general_set_two(&a, w);  
  
      for (xor = 0; xor < 2; xor++) {  
        elapsed = 0;  
        for (it = 0; it < iterations; it++) {  
          if (test == 'G') gf_general_set_random(&a, w, 1);  
          gf_general_set_up_single_timing_test(8, ra, rb, size);  
          timer_start(&timer);  
          gf_general_do_region_multiply(&gf, &a, ra, rb, size, xor);  
          elapsed += timer_split(&timer);  
        }  
        printf("%14s: XOR: %d      %10.6lf s      MB: %10.3lf      %10.3lf MB/s\n",  
              tstrings[(int)test], xor, elapsed,  
              ds*di/1024.0/1024.0, ds*di/1024.0/1024.0/elapsed);  
      }  
    }  
  }  
}  
}  
return 0;  
}
```



*test/gf\_unit.c lines 1 to 60*

```
/*
 * GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic
 * James S. Plank, Ethan L. Miller, Kevin M. Greenan,
 * Benjamin A. Arnold, John A. Burnum, Adam W. Disney, Allen C. McBride.
 *
 * gf_unit.c
 *
 * Performs unit testing for gf arithmetic
 */

#include <stdio.h>
#include <getopt.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>

#include "gf_complete.h"
#include "gf_int.h"
#include "gf_method.h"
#include "gf_rand.h"
#include "gf_general.h"

#define REGION_SIZE (16384)
#define RMASK (0x00000000ffffffffLL)
#define LMASK (0xffffffff00000000LL)

void problem(char *s)
{
    fprintf(stderr, "Unit test failed.\n");
    fprintf(stderr, "%s\n", s);
    exit(1);
}

char *BM = "Bad Method: ";

void usage(char *s)
{
    fprintf(stderr, "usage: gf_unit w tests seed [method] - does unit testing in GF(2^w)\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "Legal w are: 1 - 32, 64 and 128\n");
    fprintf(stderr, "           128 is hex only (i.e. '128' will be an error - do '128h')\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "Tests may be any combination of:\n");
    fprintf(stderr, "      A: All\n");
    fprintf(stderr, "      S: Single operations (multiplication/division)\n");
    fprintf(stderr, "      R: Region operations\n");
    fprintf(stderr, "      V: Verbose Output\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "Use -1 for time(0) as a seed.\n");
    fprintf(stderr, "\n");
    if (s == BM) {
        fprintf(stderr, "%s", BM);
        gf_error();
    } else if (s != NULL) {
        fprintf(stderr, "%s\n", s);
    }
    exit(1);
}
```

*test/gf\_unit.c lines 61 to 120*

```
void SigHandler(int v)
{
    fprintf(stderr, "Problem: SegFault!\n");
    fflush(stdout);
    exit(2);
}

int main(int argc, char **argv)
{
    signal(SIGSEGV, SigHandler);

    int w, i, verbose, single, region, top;
    int s_start, d_start, bytes, xor, alignment_test;
    gf_t gf, gf_def;
    time_t t0;
    gf_internal_t *h;
    gf_general_t *a, *b, *c, *d;
    uint8_t a8, b8, c8, *mult4 = NULL, *mult8 = NULL;
    uint16_t a16, b16, c16, *log16 = NULL, *alog16 = NULL;
    char as[50], bs[50], cs[50], ds[50];
    uint32_t mask = 0;
    char *ra, *rb, *rc, *rd, *target;
    int align;

    if (argc < 4) usage(NULL);

    if (sscanf(argv[1], "%d", &w) == 0) {
        usage("Bad w\n");
    }

    if (sscanf(argv[3], "%ld", &t0) == 0) usage("Bad seed\n");
    if (t0 == -1) t0 = time(0);
    MOA_Seed(t0);

    if (w > 32 && w != 64 && w != 128) usage("Bad w");

    if (create_gf_from_argv(&gf, w, argc, argv, 4) == 0) {
        usage(BM);
    }

    printf("Args: ");
    for (i = 1; i < argc; i++) {
        printf ("%s ", argv[i]);
    }
    printf("/ size (bytes): %d\n", gf_size(&gf));

    for (i = 0; i < strlen(argv[2]); i++) {
        if (strchr("ASRV", argv[2][i]) == NULL) usage("Bad test\n");
    }

    h = (gf_internal_t *) gf.scratch;
    a = (gf_general_t *) malloc(sizeof(gf_general_t));
    b = (gf_general_t *) malloc(sizeof(gf_general_t));
    c = (gf_general_t *) malloc(sizeof(gf_general_t));
    d = (gf_general_t *) malloc(sizeof(gf_general_t));

    //15 bytes extra to make sure it's 16byte aligned
    ra = (char *) malloc(sizeof(char)*REGION_SIZE+15);
```



*test/gf\_unit.c lines 121 to 180*

```
rb = (char *) malloc(sizeof(char)*REGION_SIZE+15);
rc = (char *) malloc(sizeof(char)*REGION_SIZE+15);
rd = (char *) malloc(sizeof(char)*REGION_SIZE+15);
```

```
//this still assumes 8 byte aligned pointer from malloc
//(which is usual on 32-bit machines)
```

```
ra += (uint64_t)ra & 0xf;
rb += (uint64_t)rb & 0xf;
rc += (uint64_t)rc & 0xf;
rd += (uint64_t)rd & 0xf;
```

```
if (w <= 32) {
    mask = 0;
    for (i = 0; i < w; i++) mask |= (1 << i);
}
```

```
verbose = (strchr(argv[2], 'V') != NULL);
single = (strchr(argv[2], 'S') != NULL || strchr(argv[2], 'A') != NULL);
region = (strchr(argv[2], 'R') != NULL || strchr(argv[2], 'A') != NULL);
```

```
if (!gf_init_hard(&gf_def, w, GF_MULT_DEFAULT, GF_REGION_DEFAULT, GF_DIVIDE_DEFAULT,
    (h->mult_type != GF_MULT_COMPOSITE) ? h->prim_poly : 0, 0, 0, NULL, NULL))
    problem("No default for this value of w");
```

```
if (w == 4) {
    mult4 = gf_w4_get_mult_table(&gf);
} else if (w == 8) {
    mult8 = gf_w8_get_mult_table(&gf);
} else if (w == 16) {
    log16 = gf_w16_get_log_table(&gf);
    alog16 = gf_w16_get_mult_alog_table(&gf);
}
```

```
if (verbose) printf("Seed: %ld\n", t0);
```

```
if (single) {
```

```
    if (gf.multiply.w32 == NULL) problem("No multiplication operation defined.");
    if (verbose) { printf("Testing single multiplications/divisions.\n"); fflush(stdout); }
    if (w <= 10) {
```

```
        top = (1 << w) * (1 << w);
    } else {
        top = 1024*1024;
    }
```

```
    for (i = 0; i < top; i++) {
        if (w <= 10) {
            a->w32 = i % (1 << w);
            b->w32 = (i >> w);
```

```
        //Allen: the following conditions were being run 10 times each. That didn't seem like nearly enough to
        //me for these special cases, so I converted to doing this mod stuff to easily make the number of times
        //run both larger and proportional to the total size of the run.
```

```
    } else {
        switch (i % 32)
        {
            case 0:
                gf_general_set_zero(a, w);
                gf_general_set_random(b, w, 1);
                break;
            case 1:
```



*test/gf\_unit.c lines 181 to 240*

```
gf_general_set_random(a, w, 1);
gf_general_set_zero(b, w);
break;
case 2:
gf_general_set_one(a, w);
gf_general_set_random(b, w, 1);
break;
case 3:
gf_general_set_random(a, w, 1);
gf_general_set_one(b, w);
break;
default:
gf_general_set_random(a, w, 1);
gf_general_set_random(b, w, 1);
}
}

//Allen: the following special cases for w=64 are based on the code below for w=128.
//These w=64 cases are based on Dr. Plank's suggestion because some of the methods for w=64
//involve splitting it in two. I think they're less likely to give errors than the 128-bit case
//though, because the 128 bit case is always split in two.
//As with w=128, I'm arbitrarily deciding to do this sort of thing with a quarter of the cases
if (w == 64) {
switch (i % 32)
{
case 0: if (!gf_general_is_one(a, w)) a->w64 &= RMASK; break;
case 1: if (!gf_general_is_one(a, w)) a->w64 &= LMASK; break;
case 2: if (!gf_general_is_one(a, w)) a->w64 &= RMASK; if (!gf_general_is_one(b, w)) b->w64 &= RMASK; break;
case 3: if (!gf_general_is_one(a, w)) a->w64 &= RMASK; if (!gf_general_is_one(b, w)) b->w64 &= LMASK; break;
case 4: if (!gf_general_is_one(a, w)) a->w64 &= LMASK; if (!gf_general_is_one(b, w)) b->w64 &= RMASK; break;
case 5: if (!gf_general_is_one(a, w)) a->w64 &= LMASK; if (!gf_general_is_one(b, w)) b->w64 &= LMASK; break;
case 6: if (!gf_general_is_one(b, w)) b->w64 &= RMASK; break;
case 7: if (!gf_general_is_one(b, w)) b->w64 &= LMASK; break;
}
}

//Allen: for w=128, we have important special cases where one half or the other of the number is all
//zeros. The probability of hitting such a number randomly is 1^-64, so if we don't force these cases
//we'll probably never hit them. This could be implemented more efficiently by changing the set-random
//function for w=128, but I think this is easier to follow.
//I'm arbitrarily deciding to do this sort of thing with a quarter of the cases
if (w == 128) {
switch (i % 32)
{
case 0: if (!gf_general_is_one(a, w)) a->w128[0] = 0; break;
case 1: if (!gf_general_is_one(a, w)) a->w128[1] = 0; break;
case 2: if (!gf_general_is_one(a, w)) a->w128[0] = 0; if (!gf_general_is_one(b, w)) b->w128[0] = 0; break;
case 3: if (!gf_general_is_one(a, w)) a->w128[0] = 0; if (!gf_general_is_one(b, w)) b->w128[1] = 0; break;
case 4: if (!gf_general_is_one(a, w)) a->w128[1] = 0; if (!gf_general_is_one(b, w)) b->w128[0] = 0; break;
case 5: if (!gf_general_is_one(a, w)) a->w128[1] = 0; if (!gf_general_is_one(b, w)) b->w128[1] = 0; break;
case 6: if (!gf_general_is_one(b, w)) b->w128[0] = 0; break;
case 7: if (!gf_general_is_one(b, w)) b->w128[1] = 0; break;
}
}

gf_general_multiply(&gf, a, b, c);

/* If w is 4, 8 or 16, then there are inline multiplication/division methods.
Test them here. */
```



*test/gf\_unit.c lines 241 to 300*

```
if (w == 4 && mult4 != NULL) {
    a8 = a->w32;
    b8 = b->w32;
    c8 = GF_W4_INLINE_MULTDIV(mult4, a8, b8);
    if (c8 != c->w32) {
        printf("Error in inline multiplication. %d * %d.  Inline = %d.  Default = %d.\n",
            a8, b8, c8, c->w32);
        exit(1);
    }
}

if (w == 8 && mult8 != NULL) {
    a8 = a->w32;
    b8 = b->w32;
    c8 = GF_W8_INLINE_MULTDIV(mult8, a8, b8);
    if (c8 != c->w32) {
        printf("Error in inline multiplication. %d * %d.  Inline = %d.  Default = %d.\n",
            a8, b8, c8, c->w32);
        exit(1);
    }
}

if (w == 16 && log16 != NULL) {
    a16 = a->w32;
    b16 = b->w32;
    c16 = GF_W16_INLINE_MULT(log16, alog16, a16, b16);
    if (c16 != c->w32) {
        printf("Error in inline multiplication. %d * %d.  Inline = %d.  Default = %d.\n",
            a16, b16, c16, c->w32);
        printf("%d %d\n", log16[a16], log16[b16]);
        top = log16[a16] + log16[b16];
        printf("%d %d\n", top, alog16[top]);
        exit(1);
    }
}

/* If this is not composite, then first test against the default: */
if (h->mult_type != GF_MULT_COMPOSITE) {
    gf_general_multiply(&gf_def, a, b, d);

    if (!gf_general_are_equal(c, d, w)) {
        gf_general_val_to_s(a, w, as, 1);
        gf_general_val_to_s(b, w, bs, 1);
        gf_general_val_to_s(c, w, cs, 1);
        gf_general_val_to_s(d, w, ds, 1);
        printf("Error in single multiplication (all numbers in hex):\n\n");
        printf("  gf.multiply(gf, %s, %s) = %s\n", as, bs, cs);
        printf("  The default gf multiplier returned %s\n", ds);
        exit(1);
    }
}

/* Now, we also need to double-check by other means, in case the default is wanky,
   and when we're performing composite operations. Start with 0 and 1, where we know
   what the result should be. */
if (gf_general_is_zero(a, w) || gf_general_is_zero(b, w) ||
    gf_general_is_one(a, w) || gf_general_is_one(b, w)) {
    if ((gf_general_is_zero(a, w) || gf_general_is_zero(b, w)) && !gf_general_is_zero(c, w)) ||
```

*test/gf\_unit.c lines 301 to 360*

```
        (gf_general_is_one(a, w) && !gf_general_are_equal(b, c, w)) ||
        (gf_general_is_one(b, w) && !gf_general_are_equal(a, c, w))) {
    gf_general_val_to_s(a, w, as, 1);
    gf_general_val_to_s(b, w, bs, 1);
    gf_general_val_to_s(c, w, cs, 1);
    printf("Error in single multiplication (all numbers in hex):\n\n");
    printf("    gf.multiply(gf, %s, %s) = %s, which is clearly wrong.\n", as, bs, cs);
    exit(1);
}
}

/* Dumb check to make sure that it's not returning numbers that are too big: */
if (w < 32 && (c->w32 & mask) != c->w32) {
    gf_general_val_to_s(a, w, as, 1);
    gf_general_val_to_s(b, w, bs, 1);
    gf_general_val_to_s(c, w, cs, 1);
    printf("Error in single multiplication (all numbers in hex):\n\n");
    printf("    gf.multiply.w32(gf, %s, %s) = %s, which is too big.\n", as, bs, cs);
    exit(1);
}

/* Finally, let's check to see that multiplication and division work together */
if (!gf_general_is_zero(a, w)) {
    gf_general_divide(&gf, c, a, d);
    if (!gf_general_are_equal(b, d, w)) {
        gf_general_val_to_s(a, w, as, 1);
        gf_general_val_to_s(b, w, bs, 1);
        gf_general_val_to_s(c, w, cs, 1);
        gf_general_val_to_s(d, w, ds, 1);
        printf("Error in single multiplication/division (all numbers in hex):\n\n");
        printf("    gf.multiply(gf, %s, %s) = %s, but gf.divide(gf, %s, %s) = %s\n", as, bs, cs, cs, as, ds);
        exit(1);
    }
}
}

if (region) {
    if (verbose) { printf("Testing region multiplications\n"); fflush(stdout); }
    for (i = 0; i < 1024; i++) {
        //Allen: changing to a switch thing as with the single ops to make things proportional
        switch (i % 32)
        {
            case 0:
                gf_general_set_zero(a, w);
                break;
            case 1:
                gf_general_set_one(a, w);
                break;
            case 2:
                gf_general_set_two(a, w);
                break;
            default:
                gf_general_set_random(a, w, 1);
        }
    }
    MOA_Fill_Random_Region(ra, REGION_SIZE);
    MOA_Fill_Random_Region(rb, REGION_SIZE);
}
```



*test/gf\_unit.c lines 361 to 420*

```
xor = (i/32)%2;
align = w/8;
if (align == 0) align = 1;
if (align > 16) align = 16;

/* JSP - Cauchy test. When w < 32 & it doesn't equal 4, 8 or 16, the default is
   equal to GF_REGION_CAUCHY, even if GF_REGION_CAUCHY is not set. We are testing
   three alignments here:

   1. Anything goes -- no alignment guaranteed.
   2. Perfect alignment. Here src and dest must be aligned wrt each other,
      and bytes must be a multiple of 16*w.
   3. Imperfect alignment. Here we'll have src and dest be aligned wrt each
      other, but bytes is simply a multiple of w. That means some XOR's will
      be aligned, and some won't.
*/

if ((h->region_type & GF_REGION_CAUCHY) || (w < 32 && w != 4 && w != 8 && w != 16)) {
    alignment_test = (i%3);

    s_start = MOA_Random_W(5, 1);
    if (alignment_test == 0) {
        d_start = MOA_Random_W(5, 1);
    } else {
        d_start = s_start;
    }

    bytes = (d_start > s_start) ? REGION_SIZE - d_start : REGION_SIZE - s_start;
    bytes -= MOA_Random_W(5, 1);
    if (alignment_test == 1) {
        bytes -= (bytes % (w*16));
    } else {
        bytes -= (bytes % w);
    }

    target = rb;

/* JSP - Otherwise, we're testing a non-cauchy test, and alignment
   must be more strict. We have to make sure that the regions are
   aligned wrt each other on 16-byte pointers. */

} else {
    s_start = MOA_Random_W(5, 1) * align;
    d_start = s_start;
    bytes = REGION_SIZE - s_start - MOA_Random_W(5, 1);
    bytes -= (bytes % align);

    if (h->mult_type == GF_MULT_COMPOSITE && (h->region_type & GF_REGION_ALTMAP)) {
        target = rb;
    } else {
        target = (i/64)%2 ? rb : ra;
    }
}

memcpy(rc, ra, REGION_SIZE);
memcpy(rd, target, REGION_SIZE);
gf_general_do_region_multiply(&gf, a, ra+s_start, target+d_start, bytes, xor);
gf_general_do_region_check(&gf, a, rc+s_start, rd+d_start, target+d_start, bytes, xor);
}
```

*test/gf\_unit.c lines 421 to 432*

```
    free(a);  
    free(b);  
    free(c);  
    free(d);  
    free(ra);  
    free(rb);  
    free(rc);  
    free(rd);  
  
    return 0;  
}
```