

# Optimization 101

A simple example of the *process*  
of optimizing a function.

Mike Acton, Insomniac Games  
(2007)

# Optimization is Not Black Magic

- It is a systematic process
- It is an analytical process
- But you need to use your intuition
- Only better with experience
- Mistakes are made
- You don't have the answer in advance

# Signal Transformation

- Examine the source data
- Examine the destination data
- The “logical” process is not important
  - Slow code deals in what “should” be happening.
  - Fast code deals only in what actually is happening.
- Understand the transformation primitives (i.e. instruction set)

# The Example

- Let's look at the code...
- Let's look at the table...

# Logical Version

```
vector unsigned short
ReindexEdgesBlockCube( vector unsigned short edge_indices, uint32_t cube )
{
    union
    {
        qword          v;
        unsigned short us[8];
    }
    in_indices = { .v = (qword)edge_indices };

    union
    {
        qword          v;
        unsigned short us[8];
    }
    out_indices;

    for ( int i=0;i<8;i++ )
    {
        uint16_t in_index    = in_indices.us[i];
        uint16_t out_index   = _edge_map_table[ cube ][ in_index ];

        out_indices.us[ i ] = out_index;
    }

    return (vector unsigned short)out_indices.v;
}
```

# The Example

- Let's look at the code...
- Let's look at the table...

# \_edge\_map\_table

```
[0] 0x0200 0x0140 0x0240 0x0100 0x0201 0x0141 0x0241 0x0101 0x0000 0x0040 0x0050 0x0010
[1] 0x0201 0x0141 0x0241 0x0101 0x0202 0x0142 0x0242 0x0102 0x0001 0x0041 0x0051 0x0011
[2] 0x0202 0x0142 0x0242 0x0102 0x0203 0x0143 0x0243 0x0103 0x0002 0x0042 0x0052 0x0012
[3] 0x0203 0x0143 0x0243 0x0103 0x0204 0x0144 0x0244 0x0104 0x0003 0x0043 0x0053 0x0013
[4] 0x0204 0x0144 0x0244 0x0104 0x0205 0x0145 0x0245 0x0105 0x0004 0x0044 0x0054 0x0014
[5] 0x0205 0x0145 0x0245 0x0105 0x0206 0x0146 0x0246 0x0106 0x0005 0x0045 0x0055 0x0015
[6] 0x0206 0x0146 0x0246 0x0106 0x0207 0x0147 0x0247 0x0107 0x0006 0x0046 0x0056 0x0016
[7] 0x0207 0x0147 0x0247 0x0107 0x0208 0x0148 0x0248 0x0108 0x0007 0x0047 0x0057 0x0017
[8] 0x0208 0x0148 0x0248 0x0108 0x0209 0x0149 0x0249 0x0109 0x0008 0x0048 0x0058 0x0018
[9] 0x0209 0x0149 0x0249 0x0109 0x020a 0x014a 0x024a 0x010a 0x0009 0x0049 0x0059 0x0019
[10] 0x020a 0x014a 0x024a 0x010a 0x020b 0x014b 0x024b 0x010b 0x000a 0x004a 0x005a 0x001a
[11] 0x020b 0x014b 0x024b 0x010b 0x020c 0x014c 0x024c 0x010c 0x000b 0x004b 0x005b 0x001b
[12] 0x020c 0x014c 0x024c 0x010c 0x020d 0x014d 0x024d 0x010d 0x000c 0x004c 0x005c 0x001c
[13] 0x020d 0x014d 0x024d 0x010d 0x020e 0x014e 0x024e 0x010e 0x000d 0x004d 0x005d 0x001d
[14] 0x020e 0x014e 0x024e 0x010e 0x020f 0x014f 0x024f 0x010f 0x000e 0x004e 0x005e 0x001e
[15] 0x0240 0x0150 0x0280 0x0110 0x0241 0x0151 0x0281 0x0111 0x0010 0x0050 0x0060 0x0020
[16] 0x0241 0x0151 0x0281 0x0111 0x0242 0x0152 0x0282 0x0112 0x0011 0x0051 0x0061 0x0021
[17] 0x0242 0x0152 0x0282 0x0112 0x0243 0x0153 0x0283 0x0113 0x0012 0x0052 0x0062 0x0022
[18] 0x0243 0x0153 0x0283 0x0113 0x0244 0x0154 0x0284 0x0114 0x0013 0x0053 0x0063 0x0023
[19] 0x0244 0x0154 0x0284 0x0114 0x0245 0x0155 0x0285 0x0115 0x0014 0x0054 0x0064 0x0024
[20] 0x0245 0x0155 0x0285 0x0115 0x0246 0x0156 0x0286 0x0116 0x0015 0x0055 0x0065 0x0025
[21] 0x0246 0x0156 0x0286 0x0116 0x0247 0x0157 0x0287 0x0117 0x0016 0x0056 0x0066 0x0026
[22] 0x0247 0x0157 0x0287 0x0117 0x0248 0x0158 0x0288 0x0118 0x0017 0x0057 0x0067 0x0027
[23] 0x0248 0x0158 0x0288 0x0118 0x0249 0x0159 0x0289 0x0119 0x0018 0x0058 0x0068 0x0028
[24] 0x0249 0x0159 0x0289 0x0119 0x024a 0x015a 0x028a 0x011a 0x0019 0x0059 0x0069 0x0029
[25] 0x024a 0x015a 0x028a 0x011a 0x024b 0x015b 0x028b 0x011b 0x001a 0x005a 0x006a 0x002a
[26] 0x024b 0x015b 0x028b 0x011b 0x024c 0x015c 0x028c 0x011c 0x001b 0x005b 0x006b 0x002b
[27] 0x024c 0x015c 0x028c 0x011c 0x024d 0x015d 0x028d 0x011d 0x001c 0x005c 0x006c 0x002c
[28] 0x024d 0x015d 0x028d 0x011d 0x024e 0x015e 0x028e 0x011e 0x001d 0x005d 0x006d 0x002d
[29] 0x024e 0x015e 0x028e 0x011e 0x024f 0x015f 0x028f 0x011f 0x001e 0x005e 0x006e 0x002e
[30] 0x0280 0x0160 0x02c0 0x0120 0x0281 0x0161 0x02c1 0x0121 0x0020 0x0060 0x0070 0x0030
[31] 0x0281 0x0161 0x02c1 0x0121 0x0282 0x0162 0x02c2 0x0122 0x0021 0x0061 0x0071 0x0031
[32] 0x0282 0x0162 0x02c2 0x0122 0x0283 0x0163 0x02c3 0x0123 0x0022 0x0062 0x0072 0x0032
```

# \_edge\_map\_table

```
[33] 0x0283 0x0163 0x02c3 0x0123 0x0284 0x0164 0x02c4 0x0124 0x0023 0x0063 0x0073 0x0033
[34] 0x0284 0x0164 0x02c4 0x0124 0x0285 0x0165 0x02c5 0x0125 0x0024 0x0064 0x0074 0x0034
[35] 0x0285 0x0165 0x02c5 0x0125 0x0286 0x0166 0x02c6 0x0126 0x0025 0x0065 0x0075 0x0035
[36] 0x0286 0x0166 0x02c6 0x0126 0x0287 0x0167 0x02c7 0x0127 0x0026 0x0066 0x0076 0x0036
[37] 0x0287 0x0167 0x02c7 0x0127 0x0288 0x0168 0x02c8 0x0128 0x0027 0x0067 0x0077 0x0037
[38] 0x0288 0x0168 0x02c8 0x0128 0x0289 0x0169 0x02c9 0x0129 0x0028 0x0068 0x0078 0x0038
[39] 0x0289 0x0169 0x02c9 0x0129 0x028a 0x016a 0x02ca 0x012a 0x0029 0x0069 0x0079 0x0039
[40] 0x028a 0x016a 0x02ca 0x012a 0x028b 0x016b 0x02cb 0x012b 0x002a 0x006a 0x007a 0x003a
[41] 0x028b 0x016b 0x02cb 0x012b 0x028c 0x016c 0x02cc 0x012c 0x002b 0x006b 0x007b 0x003b
[42] 0x028c 0x016c 0x02cc 0x012c 0x028d 0x016d 0x02cd 0x012d 0x002c 0x006c 0x007c 0x003c
[43] 0x028d 0x016d 0x02cd 0x012d 0x028e 0x016e 0x02ce 0x012e 0x002d 0x006d 0x007d 0x003d
[44] 0x028e 0x016e 0x02ce 0x012e 0x028f 0x016f 0x02cf 0x012f 0x002e 0x006e 0x007e 0x003e
[45] 0x0210 0x0180 0x0250 0x0140 0x0211 0x0181 0x0251 0x0141 0x0040 0x0080 0x0090 0x0050
[46] 0x0211 0x0181 0x0251 0x0141 0x0212 0x0182 0x0252 0x0142 0x0041 0x0081 0x0091 0x0051
[47] 0x0212 0x0182 0x0252 0x0142 0x0213 0x0183 0x0253 0x0143 0x0042 0x0082 0x0092 0x0052
[48] 0x0213 0x0183 0x0253 0x0143 0x0214 0x0184 0x0254 0x0144 0x0043 0x0083 0x0093 0x0053
[49] 0x0214 0x0184 0x0254 0x0144 0x0215 0x0185 0x0255 0x0145 0x0044 0x0084 0x0094 0x0054
[50] 0x0215 0x0185 0x0255 0x0145 0x0216 0x0186 0x0256 0x0146 0x0045 0x0085 0x0095 0x0055
[51] 0x0216 0x0186 0x0256 0x0146 0x0217 0x0187 0x0257 0x0147 0x0046 0x0086 0x0096 0x0056
[52] 0x0217 0x0187 0x0257 0x0147 0x0218 0x0188 0x0258 0x0148 0x0047 0x0087 0x0097 0x0057
[53] 0x0218 0x0188 0x0258 0x0148 0x0219 0x0189 0x0259 0x0149 0x0048 0x0088 0x0098 0x0058
[54] 0x0219 0x0189 0x0259 0x0149 0x021a 0x018a 0x025a 0x014a 0x0049 0x0089 0x0099 0x0059
[55] 0x021a 0x018a 0x025a 0x014a 0x021b 0x018b 0x025b 0x014b 0x004a 0x008a 0x009a 0x005a
[56] 0x021b 0x018b 0x025b 0x014b 0x021c 0x018c 0x025c 0x014c 0x004b 0x008b 0x009b 0x005b
[57] 0x021c 0x018c 0x025c 0x014c 0x021d 0x018d 0x025d 0x014d 0x004c 0x008c 0x009c 0x005c
[58] 0x021d 0x018d 0x025d 0x014d 0x021e 0x018e 0x025e 0x014e 0x004d 0x008d 0x009d 0x005d
[59] 0x021e 0x018e 0x025e 0x014e 0x021f 0x018f 0x025f 0x014f 0x004e 0x008e 0x009e 0x005e
[60] 0x0250 0x0190 0x0290 0x0150 0x0251 0x0191 0x0291 0x0151 0x0050 0x0090 0x00a0 0x0060
[61] 0x0251 0x0191 0x0291 0x0151 0x0252 0x0192 0x0292 0x0152 0x0051 0x0091 0x00a1 0x0061
[62] 0x0252 0x0192 0x0292 0x0152 0x0253 0x0193 0x0293 0x0153 0x0052 0x0092 0x00a2 0x0062
[63] 0x0253 0x0193 0x0293 0x0153 0x0254 0x0194 0x0294 0x0154 0x0053 0x0093 0x00a3 0x0063
```

# \_edge\_map\_table

```
[64] 0x0254 0x0194 0x0294 0x0154 0x0255 0x0195 0x0295 0x0155 0x0054 0x0094 0x00a4 0x0064
[65] 0x0255 0x0195 0x0295 0x0155 0x0256 0x0196 0x0296 0x0156 0x0055 0x0095 0x00a5 0x0065
[66] 0x0256 0x0196 0x0296 0x0156 0x0257 0x0197 0x0297 0x0157 0x0056 0x0096 0x00a6 0x0066
[67] 0x0257 0x0197 0x0297 0x0157 0x0258 0x0198 0x0298 0x0158 0x0057 0x0097 0x00a7 0x0067
[68] 0x0258 0x0198 0x0298 0x0158 0x0259 0x0199 0x0299 0x0159 0x0058 0x0098 0x00a8 0x0068
[69] 0x0259 0x0199 0x0299 0x0159 0x025a 0x019a 0x029a 0x015a 0x0059 0x0099 0x00a9 0x0069
[70] 0x025a 0x019a 0x029a 0x015a 0x025b 0x019b 0x029b 0x015b 0x005a 0x009a 0x00aa 0x006a
[71] 0x025b 0x019b 0x029b 0x015b 0x025c 0x019c 0x029c 0x015c 0x005b 0x009b 0x00ab 0x006b
[72] 0x025c 0x019c 0x029c 0x015c 0x025d 0x019d 0x029d 0x015d 0x005c 0x009c 0x00ac 0x006c
[73] 0x025d 0x019d 0x029d 0x015d 0x025e 0x019e 0x029e 0x015e 0x005d 0x009d 0x00ad 0x006d
[74] 0x025e 0x019e 0x029e 0x015e 0x025f 0x019f 0x029f 0x015f 0x005e 0x009e 0x00ae 0x006e
[75] 0x0290 0x01a0 0x02d0 0x0160 0x0291 0x01a1 0x02d1 0x0161 0x0060 0x00a0 0x00b0 0x0070
[76] 0x0291 0x01a1 0x02d1 0x0161 0x0292 0x01a2 0x02d2 0x0162 0x0061 0x00a1 0x00b1 0x0071
[77] 0x0292 0x01a2 0x02d2 0x0162 0x0293 0x01a3 0x02d3 0x0163 0x0062 0x00a2 0x00b2 0x0072
[78] 0x0293 0x01a3 0x02d3 0x0163 0x0294 0x01a4 0x02d4 0x0164 0x0063 0x00a3 0x00b3 0x0073
[79] 0x0294 0x01a4 0x02d4 0x0164 0x0295 0x01a5 0x02d5 0x0165 0x0064 0x00a4 0x00b4 0x0074
[80] 0x0295 0x01a5 0x02d5 0x0165 0x0296 0x01a6 0x02d6 0x0166 0x0065 0x00a5 0x00b5 0x0075
[81] 0x0296 0x01a6 0x02d6 0x0166 0x0297 0x01a7 0x02d7 0x0167 0x0066 0x00a6 0x00b6 0x0076
[82] 0x0297 0x01a7 0x02d7 0x0167 0x0298 0x01a8 0x02d8 0x0168 0x0067 0x00a7 0x00b7 0x0077
[83] 0x0298 0x01a8 0x02d8 0x0168 0x0299 0x01a9 0x02d9 0x0169 0x0068 0x00a8 0x00b8 0x0078
[84] 0x0299 0x01a9 0x02d9 0x0169 0x029a 0x01aa 0x02da 0x016a 0x0069 0x00a9 0x00b9 0x0079
[85] 0x029a 0x01aa 0x02da 0x016a 0x029b 0x01ab 0x02db 0x016b 0x006a 0x00aa 0x00ba 0x007a
[86] 0x029b 0x01ab 0x02db 0x016b 0x029c 0x01ac 0x02dc 0x016c 0x006b 0x00ab 0x00bb 0x007b
[87] 0x029c 0x01ac 0x02dc 0x016c 0x029d 0x01ad 0x02dd 0x016d 0x006c 0x00ac 0x00bc 0x007c
[88] 0x029d 0x01ad 0x02dd 0x016d 0x029e 0x01ae 0x02de 0x016e 0x006d 0x00ad 0x00bd 0x007d
[89] 0x029e 0x01ae 0x02de 0x016e 0x029f 0x01af 0x02df 0x016f 0x006e 0x00ae 0x00be 0x007e
[90] 0x0220 0x01c0 0x0260 0x0180 0x0221 0x01c1 0x0261 0x0181 0x0080 0x00c0 0x00d0 0x0090
[91] 0x0221 0x01c1 0x0261 0x0181 0x0222 0x01c2 0x0262 0x0182 0x0081 0x00c1 0x00d1 0x0091
[92] 0x0222 0x01c2 0x0262 0x0182 0x0223 0x01c3 0x0263 0x0183 0x0082 0x00c2 0x00d2 0x0092
[93] 0x0223 0x01c3 0x0263 0x0183 0x0224 0x01c4 0x0264 0x0184 0x0083 0x00c3 0x00d3 0x0093
[94] 0x0224 0x01c4 0x0264 0x0184 0x0225 0x01c5 0x0265 0x0185 0x0084 0x00c4 0x00d4 0x0094
[95] 0x0225 0x01c5 0x0265 0x0185 0x0226 0x01c6 0x0266 0x0186 0x0085 0x00c5 0x00d5 0x0095
```

# \_edge\_map\_table

```

[96] 0x0226 0x01c6 0x0266 0x0186 0x0227 0x01c7 0x0267 0x0187 0x0086 0x00c6 0x00d6 0x0096
[97] 0x0227 0x01c7 0x0267 0x0187 0x0228 0x01c8 0x0268 0x0188 0x0087 0x00c7 0x00d7 0x0097
[98] 0x0228 0x01c8 0x0268 0x0188 0x0229 0x01c9 0x0269 0x0189 0x0088 0x00c8 0x00d8 0x0098
[99] 0x0229 0x01c9 0x0269 0x0189 0x022a 0x01ca 0x026a 0x018a 0x0089 0x00c9 0x00d9 0x0099
[100] 0x022a 0x01ca 0x026a 0x018a 0x022b 0x01cb 0x026b 0x018b 0x008a 0x00ca 0x00da 0x009a
[101] 0x022b 0x01cb 0x026b 0x018b 0x022c 0x01cc 0x026c 0x018c 0x008b 0x00cb 0x00db 0x009b
[102] 0x022c 0x01cc 0x026c 0x018c 0x022d 0x01cd 0x026d 0x018d 0x008c 0x00cc 0x00dc 0x009c
[103] 0x022d 0x01cd 0x026d 0x018d 0x022e 0x01ce 0x026e 0x018e 0x008d 0x00cd 0x00dd 0x009d
[104] 0x022e 0x01ce 0x026e 0x018e 0x022f 0x01cf 0x026f 0x018f 0x008e 0x00ce 0x00de 0x009e
[105] 0x0260 0x01d0 0x02a0 0x0190 0x0261 0x01d1 0x02a1 0x0191 0x0090 0x00d0 0x00e0 0x00a0
[106] 0x0261 0x01d1 0x02a1 0x0191 0x0262 0x01d2 0x02a2 0x0192 0x0091 0x00d1 0x00e1 0x00a1
[107] 0x0262 0x01d2 0x02a2 0x0192 0x0263 0x01d3 0x02a3 0x0193 0x0092 0x00d2 0x00e2 0x00a2
[108] 0x0263 0x01d3 0x02a3 0x0193 0x0264 0x01d4 0x02a4 0x0194 0x0093 0x00d3 0x00e3 0x00a3
[109] 0x0264 0x01d4 0x02a4 0x0194 0x0265 0x01d5 0x02a5 0x0195 0x0094 0x00d4 0x00e4 0x00a4
[110] 0x0265 0x01d5 0x02a5 0x0195 0x0266 0x01d6 0x02a6 0x0196 0x0095 0x00d5 0x00e5 0x00a5
[111] 0x0266 0x01d6 0x02a6 0x0196 0x0267 0x01d7 0x02a7 0x0197 0x0096 0x00d6 0x00e6 0x00a6
[112] 0x0267 0x01d7 0x02a7 0x0197 0x0268 0x01d8 0x02a8 0x0198 0x0097 0x00d7 0x00e7 0x00a7
[113] 0x0268 0x01d8 0x02a8 0x0198 0x0269 0x01d9 0x02a9 0x0199 0x0098 0x00d8 0x00e8 0x00a8
[114] 0x0269 0x01d9 0x02a9 0x0199 0x026a 0x01da 0x02aa 0x019a 0x0099 0x00d9 0x00e9 0x00a9
[115] 0x026a 0x01da 0x02aa 0x019a 0x026b 0x01db 0x02ab 0x019b 0x009a 0x00da 0x00ea 0x00aa
[116] 0x026b 0x01db 0x02ab 0x019b 0x026c 0x01dc 0x02ac 0x019c 0x009b 0x00db 0x00eb 0x00ab
[117] 0x026c 0x01dc 0x02ac 0x019c 0x026d 0x01dd 0x02ad 0x019d 0x009c 0x00dc 0x00ec 0x00ac
[118] 0x026d 0x01dd 0x02ad 0x019d 0x026e 0x01de 0x02ae 0x019e 0x009d 0x00dd 0x00ed 0x00ad
[119] 0x026e 0x01de 0x02ae 0x019e 0x026f 0x01df 0x02af 0x019f 0x009e 0x00de 0x00ee 0x00ae
[120] 0x02a0 0x01e0 0x02e0 0x01a0 0x02a1 0x01e1 0x02e1 0x01a1 0x00a0 0x00e0 0x00f0 0x00b0
[121] 0x02a1 0x01e1 0x02e1 0x01a1 0x02a2 0x01e2 0x02e2 0x01a2 0x00a1 0x00e1 0x00f1 0x00b1
[122] 0x02a2 0x01e2 0x02e2 0x01a2 0x02a3 0x01e3 0x02e3 0x01a3 0x00a2 0x00e2 0x00f2 0x00b2
[123] 0x02a3 0x01e3 0x02e3 0x01a3 0x02a4 0x01e4 0x02e4 0x01a4 0x00a3 0x00e3 0x00f3 0x00b3
[124] 0x02a4 0x01e4 0x02e4 0x01a4 0x02a5 0x01e5 0x02e5 0x01a5 0x00a4 0x00e4 0x00f4 0x00b4
[125] 0x02a5 0x01e5 0x02e5 0x01a5 0x02a6 0x01e6 0x02e6 0x01a6 0x00a5 0x00e5 0x00f5 0x00b5
[126] 0x02a6 0x01e6 0x02e6 0x01a6 0x02a7 0x01e7 0x02e7 0x01a7 0x00a6 0x00e6 0x00f6 0x00b6
[127] 0x02a7 0x01e7 0x02e7 0x01a7 0x02a8 0x01e8 0x02e8 0x01a8 0x00a7 0x00e7 0x00f7 0x00b7

```

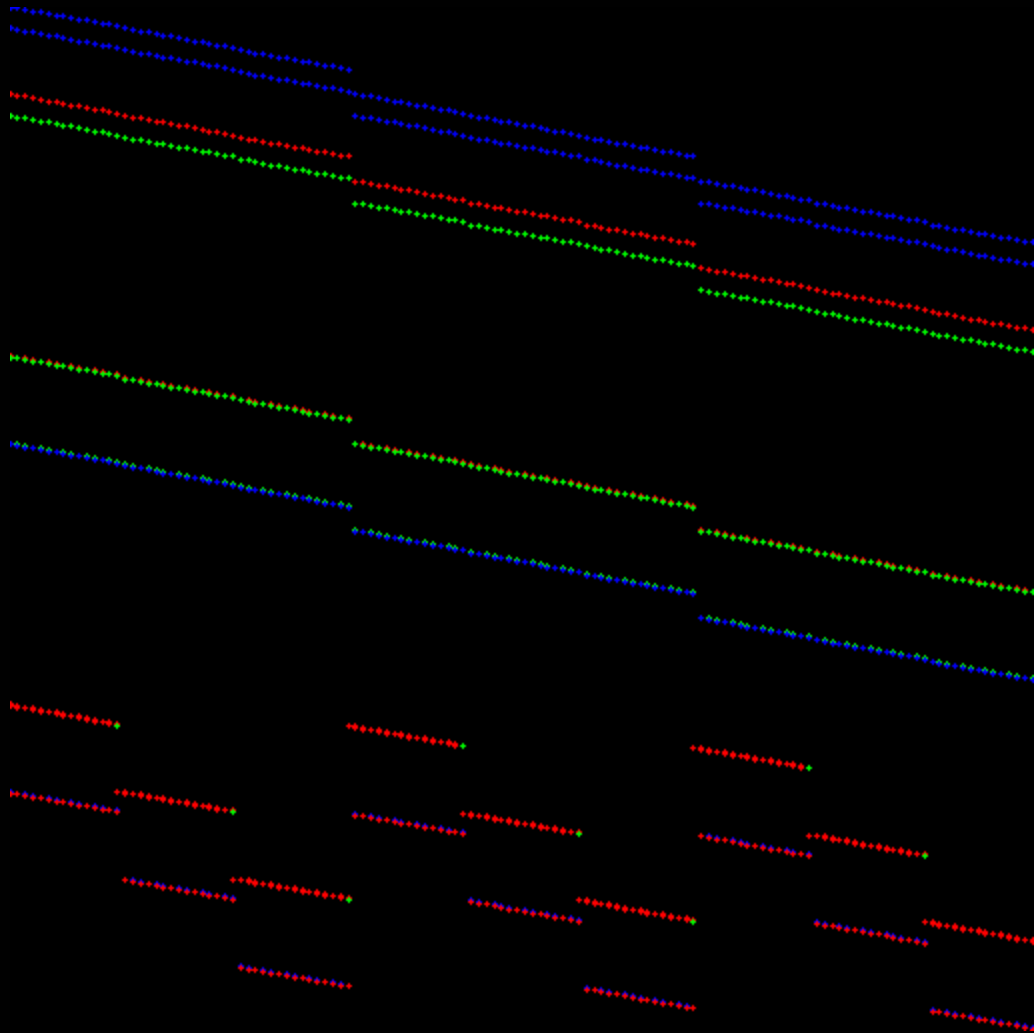
# \_edge\_map\_table

```
[128] 0x02a8 0x01e8 0x02e8 0x01a8 0x02a9 0x01e9 0x02e9 0x01a9 0x00a8 0x00e8 0x00f8 0x00b8  
[129] 0x02a9 0x01e9 0x02e9 0x01a9 0x02aa 0x01ea 0x02ea 0x01aa 0x00a9 0x00e9 0x00f9 0x00b9  
[130] 0x02aa 0x01ea 0x02ea 0x01aa 0x02ab 0x01eb 0x02eb 0x01ab 0x00aa 0x00ea 0x00fa 0x00ba  
[131] 0x02ab 0x01eb 0x02eb 0x01ab 0x02ac 0x01ec 0x02ec 0x01ac 0x00ab 0x00eb 0x00fb 0x00bb  
[132] 0x02ac 0x01ec 0x02ec 0x01ac 0x02ad 0x01ed 0x02ed 0x01ad 0x00ac 0x00ec 0x00fc 0x00bc  
[133] 0x02ad 0x01ed 0x02ed 0x01ad 0x02ae 0x01ee 0x02ee 0x01ae 0x00ad 0x00ed 0x00fd 0x00bd  
[134] 0x02ae 0x01ee 0x02ee 0x01ae 0x02af 0x01ef 0x02ef 0x01af 0x00ae 0x00ee 0x00fe 0x00be
```

# Step 1

- Is there a better way to generate these values?
  - that's faster, and..
  - that's smaller, and...
  - that's more SPU friendly.
- SIMD Unfriendly
  - Need to calculate in parallel each 16 bit entry. i.e. No lookup per entry.
- Visualize the data...

Visualize the data...



(Zoom in)



# Step 1

- Note on table:
  - Every column has a different starting value
  - Going down each column increments by one, until...
  - Every 15 rows the starting value is reset to  $N * (\text{row}/15)$ , where N is...
    - Different for each column,
    - Different at rows that are even multiples of 45.

# Basic Decomposition

- What are the extents?
  - Cube [0,134]
  - Edge Index [0,7]
- What do we need?
  - 8 bit unsigned integer divide by 15
  - 8 bit unsigned integer mod 15
  - 8 bit unsigned integer compare  $\geq 45$
  - 8 bit unsigned integer compare  $\geq 90$
  - Starting offsets
  - Step amounts

# Divide

- Divide by 15 = Divide by 3, Divide by 5
- Independent problem.
- Small test, fast iteration.

# Divide by 3

- Fractional Estimate

$$\frac{1}{3} x = \frac{5 \frac{1}{3}}{16} x = \frac{1}{16} x + \frac{4}{16} x + \frac{1}{48} x$$

$$\frac{1}{16} x = x \gg 4$$

$$\frac{4}{16} x = x \gg 2$$

$$\frac{x}{48} = ??$$

# Divide by 3

Estimate  $\frac{x}{48}$  with  $\frac{x}{64}$  (UNDER)

The Estimate:

$$\frac{1}{3} x \approx \frac{5 \frac{1}{3}}{16} x = \frac{1}{16} x + \frac{4}{16} x + \frac{1}{64} x$$

( Estimate \* 3 ) Represents the amount of the problem we've solved.

$$\begin{aligned} \text{Remainder} &= x - (\text{Estimate} * 3) \\ \text{Result} &= \text{Estimate} + (\text{Remainder}/3) \end{aligned}$$

# Divide by 3

Estimate	Remainder		11 * Remainder	
	-----	with	-----	(OVER)
	3		32	

11		1
--	>	-
32		3

( Remainder is a much smaller range of values than x. )

```
uint8_t
div3( uint8_t x )
{
    uint8_t y0 = x >> 2;    // 1/4
    uint8_t y1 = x >> 4;    // 1/16
    uint8_t y2 = y1 >> 4;   // 1/16 * 1/4
    uint8_t y3 = y0 + y1;   // 5/16
    uint8_t y4 = y3 + y2;   // 5/16 + 1/64

    uint8_t r0 = x - ( y4 * 3 );
    uint8_t r1 = ( r0 * 11 ) >> 5; // 11/32

    uint8_t q  = y4 + r1;

    return (q);
}
```

# Sanity Check

- Is the estimate good enough?
- Write a quick test on the PC.

```
int
main( void )
{
    uint8_t i;

    for (i=0;i<135;i++)
    {
        uint8_t a = div3( i );
        uint8_t b = i / 3;

        if ( a != b )
        {
            printf("%d/3 != %d (%d)\n",i,a,b);
        }
    }

    return (0);
}
```

# Divide by 5

- Fractional Estimate

$$\frac{1}{5} x = \frac{3 \frac{1}{5}}{16} x = \frac{1}{16} x + \frac{2}{16} x + \frac{1}{80} x$$

$$\frac{1}{16} x = x \gg 4$$

$$\frac{2}{16} x = x \gg 3$$

$$\frac{x}{80} = ??$$

# Divide by 3

Estimate  $\frac{x}{80}$  with  $\frac{x}{128}$  (UNDER)

The Estimate:

$$\frac{1}{5} x \approx \frac{3 \frac{1}{5}}{16} x = \frac{1}{16} x + \frac{2}{16} x + \frac{1}{128} x$$

( Estimate \* 5 ) Represents the amount of the problem we've solved.

$$\begin{aligned} \text{Remainder} &= x - (\text{Estimate} * 5) \\ \text{Result} &= \text{Estimate} + (\text{Remainder}/5) \end{aligned}$$

# Divide by 3

Estimate	Remainder		13 * Remainder	
	-----	with	-----	(OVER)
	5		64	

13		1
--	>	-
64		5

( Remainder is a much smaller range of values than x. )

```
uint8_t
div5( uint8_t x )
{
    uint8_t y0 = x >> 3;    // 1/8
    uint8_t y1 = x >> 4;    // 1/16
    uint8_t y2 = y1 >> 3;  // 1/128
    uint8_t y3 = y0 + y1;  // 3/16
    uint8_t y4 = y3 + y2;

    uint8_t r0 = x - ( y4 * 5 );
    uint8_t r1 = (r0 * 13) >> 6; // 13/64

    uint8_t q  = y4 + r1;

    return (q);
}
```

```
int
main( void )
{
    uint8_t i;

    for (i=0;i<135;i++)
    {
        uint8_t a = div5( i );
        uint8_t b = i / 5;

        if ( a != b )
        {
            printf("%d/5 != %d (%d)\n",i,a,b);
        }
    }

    return (0);
}
```

# Divide by 15

- $x / 15 = \text{div5}(\text{div3}(x))$
- Turned out a bit bigger than I thought.
- Reduced the div5, since it is always a much smaller range.
- Combined into div15 function.
- Moved to 16 bit.

```
uint16_t
div15( uint16_t n )
{
    uint16_t nd16    = n >> 4;
    uint16_t nd256   = n >> 8;
    uint16_t q       = nd16 + nd256;
    uint16_t r0      = q << 4;
    uint16_t r1      = r0 - q;
    uint16_t r       = n - r1;
    uint16_t qr0     = r << 3;
    uint16_t qr1     = qr0 + r;
    uint16_t qr      = qr1 >> 7;
    uint16_t result  = q + qr;

    return (result);
}
```

( Valid for range 0-239 )

# Divide by 15

- Still too bigger than I want.
- Try a new tactic.
- In 16 bit, take advantage of wide range.

```

uint16_t
div15( uint16_t n )
{
    uint16_t n0      = n << 4;    // 16x
    uint16_t n1      = n0 + n;    // 17x
    uint16_t n2      = n1 + 16;   // 17x + 16
    uint16_t r       = n2 >> 8;  // (17x + 16)/256

    return r;                ( Valid for range 0-239 )
}

```

$$\begin{array}{rcl}
 17 & & 16 & & 1 \\
 --- & \times & + & --- & \approx & -- & \times \\
 256 & & 256 & & & 15
 \end{array}$$

How did I pick these numbers?

$$\begin{array}{r}
 17 \\
 \text{---} \times \\
 256
 \end{array}
 +
 \begin{array}{r}
 16 \\
 \text{---} \\
 256
 \end{array}
 \approx
 \begin{array}{r}
 1 \\
 \text{--} \times \\
 15
 \end{array}$$

(1)

$$256 / 15 = 17.066666\dots$$

(2)

Is there a constant value (fudge-factor) that I can offset  $17x/256$  by to always get the correct result?

```
int
main( void )
{
    uint16_t x;
    uint16_t n;

    for (n=0;n<0xffff;n++)
    {
        int not_found = 0;

        for (x=0;x<135;x++)
        {
            uint16_t q_0 = x << 4;    // x * 16
            uint16_t q_1 = q_0 + x;   // x * 17
            uint16_t q_2 = q_1 + n;   // (x * 17) + n
            uint16_t q_3 = q_2 >> 8; // ((x * 17) + n) / 256

            if ( q_3 != ( x / 15 ) )
            {
                not_found = 1;
                break;
            }
        }

        if (!not_found)
        {
            printf("n=%d\n",n);
            return (0);
        }
    }

    return (-1);
}
```

```

uint16_t
mod15( uint16_t x, uint16_t x_div15 )
{
    uint16_t mod_0    = x_div15 << 4;    // div15*16
    uint16_t mod_1    = mod_0 - x_div15;  // div15*15
    uint16_t mod      = x - mod_1;        // x % 15

    return mod;
}

```

e.g.

```

x          = 17
x_div15    = 1
mod_0      = (1 * 16) -> 16
mod_1      = 16 - 1    -> 15
mod        = 17 - 15  -> 2

```

```

x%15      = 2

```

# Quick Scalar Version

- Started with a scalar version,
- Test the logic, bit arithmetic,
- Test the data,
- Test my sanity.



```
uint16_t cube_n0      = cube << 4;
uint16_t cube_n1      = cube_n0 + cube;
uint16_t cube_n2      = cube_n1 + 16;
uint16_t cube_q15     = cube_n2 >> 8;

uint16_t cube_m0      = cube_q15 << 4;
uint16_t cube_m1      = cube_m0 - cube_q15;
uint16_t cube_m15     = cube - cube_m1;

uint16_t is_cube_gt_90 = ( cube >= 90 )?0xffff:0;
uint16_t is_cube_gt_45 = ( cube >= 45 )?0xffff:0;

uint16_t cube_stride_ndx = cube_q15 - ( is_cube_gt_90 & 3 ) - ( is_cube_gt_45 & 3 );

union { vul6 v; uint16_t s[8]; } edge_indices_s  = {.v = edge_indices };
union { vul6 v; uint16_t s[8]; } result_indices_s;

uint8_t* edge_offset_lo;

if ( is_cube_gt_90 )
{
    edge_offset_lo = edge_offset_lo_0x90;
}
else if ( is_cube_gt_45 )
{
    edge_offset_lo = edge_offset_lo_0x45;
}
else
{
    edge_offset_lo = edge_offset_lo_0x00;
}
```

```
for(int i=0;i<8;i++)
{
    uint16_t edge_ndx          = edge_indices_s.s[i];

    uint16_t edge_start_offset_lo = (uint16_t)edge_offset_lo[ edge_ndx ];
    uint16_t edge_start_offset_hi = (uint16_t)edge_offset_hi[ edge_ndx ];

    uint16_t edge_start_offset  = edge_start_offset_lo | ( edge_start_offset_hi << 8 );

    uint16_t edge_stride        = (uint16_t)edge_stride_lo[ edge_ndx ];

    uint16_t edge_stride_offset = cube_stride_ndx * edge_stride;

    uint16_t edge_offset        = edge_start_offset + edge_stride_offset + cube_m15;

    result_indices_s.s[i]      = edge_offset;
}

return (result_indices_s.v);
}
```

# SPU Byte Shuffle

*Shuffle bytes. Each byte of register rc is used to select a byte from either register ra or register rb or a constant (0, 0x80, or 0xFF). The results are placed in the corresponding bytes of register rt.*

# SPU Byte Shuffle

```
vec4f spu_shuffle(PpuVec a, PpuVec b, PpuVec shuf)
{
    PpuVec res;

    for(u32 i = 0; i < 16; i++)
    {
        u32 s = shuf.m_u8[i];

        if ((s & 0xc0) == 0x80)          // &11000000 == 10000000 => 0x00
        {
            res.m_u8[i] = 0x00;
        }
        else if ((s & 0xe0) == 0xc0)    // &11100000 == 11000000 => 0xff
        {
            res.m_u8[i] = 0xff;
        }
        else if ((s & 0xe0) == 0xe0)    // &11100000 == 11100000 => 0x80
        {
            res.m_u8[i] = 0x80;
        }
        else if (s < 0x10)
        {
            res.m_u8[i] = a.m_u8[s];
        }
        else
        {
            res.m_u8[i] = b.m_u8[s & 0x0f];
        }
    }

    vec4f vres(res.m_f32[0], res.m_f32[1], res.m_f32[2], res.m_f32[3]);
    return vres;
}
```

# SPU Byte Shuffle

Example:

```
uint16_t edge_start_offset_lo = (uint16_t)edge_offset_lo[ edge_ndx ];
```

*edge\_ndx is from a quadword of 16 bit values in the range of [0,11]*

```
edges = { 0x0000, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007 }
```

*Only the low byte will have a non-zero value.*

*If we look up by edges, the correct value will be stored in the low byte.*

```
value = { 0xA0, 0xA1, 0xA2, 0xA3, 0xA4, 0xA5, 0xA6, 0xA7,
          0xA8, 0xA9, 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF };
```

```
spu_shuf( value, value, edges ) =
```

```
{ 0xA0A0, 0xA0A1, 0xA0A2, 0xA0A3, 0xA0A4, 0xA0A5, 0xA0A6, 0xA0A7 }
```

*The high byte is not what we want. Some choices:*

```
Mask with { 0x00ff, 0x00ff, 0x00ff, 0x00ff, 0x00ff, 0x00ff, 0x00ff, 0x00ff }
```

Shuffle with zero -

```
edges = edges | { 0x0010, 0x0010, 0x0010, 0x0010, 0x0010, 0x0010, 0x0010, 0x0010 }
```

```
edges = { 0x0010, 0x0011, 0x0012, 0x0013, 0x0014, 0x0015, 0x0016, 0x0017 }
```

```
spu_shuf( zero, value, edges ) = { 0x00A0, 0x00A1, 0x00A2, 0x00A3, 0x00A4, 0x00A5, 0x00A6, 0x00A7 }
```

*There are other options too.*

# Step 2: Initial SPU version

*Iteration and testing note: I use PS3 / Linux.*

*Only use si\_\* intrinsics.*

# Step 2: Initial SPU version

```
vector unsigned short
ReindexEdgesBlockCube( vector unsigned short v_edge_indices, uint16_t s_cube )
{
    const qword    cube_prefered    = si_from_ushort( s_cube );
    const qword    edge_indices     = (qword)v_edge_indices;

    // Broadcast cube
    const qword broadcast_halfword = si_ilh    ( 0x0203 );
    const qword cube                = si_shufb  ( cube_prefered, cube_prefered, broadcast_halfword );

    // cube_q15 = (cube / 15)

    const qword cube_q15_0          = si_shlhi  ( cube,          0x04 );
    const qword cube_q15_1          = si_ah     ( cube,          cube_q15_0 );
    const qword cube_q15_2          = si_ahi    ( cube_q15_1,    0x0f );
    const qword cube_q15            = si_rothmi ( cube_q15_2,    -0x08 );

    // cube_m15 = (cube % 15)

    const qword cube_m15_0          = si_shlhi  ( cube_q15,      0x04 );
    const qword cube_m15_1          = si_sfh    ( cube_q15,      cube_m15_0 );
    const qword cube_m15            = si_sfh    ( cube_m15_1,    cube );

    // cube_r90 = mask when (cube >= 90)
    // cube_r45 = mask when (cube >= 45) && (cube < 90)

    const qword cube_r90            = si_clgthi ( cube,          0x59 );
    const qword cube_r45_0          = si_clgthi ( cube,          0x2c );
    const qword cube_r45            = si_andc   ( cube_r45_0,    cube_r90 );

    // cube_stride_ndx

    const qword cube_stride_ndx_0   = si_andhi  ( cube_r90,      0x06 );
    const qword cube_stride_ndx_1   = si_andhi  ( cube_r45,      0x03 );
    const qword cube_stride_ndx_2   = si_sfh    ( cube_stride_ndx_0, cube_q15 );
    const qword cube_stride_ndx     = si_sfh    ( cube_stride_ndx_1, cube_stride_ndx_2 );
}
```



# Step 2: Initial SPU version

```
const qword zero                = si_ilh    ( 0x0000 );
const qword edge_indices_hi_mask = si_ilh    ( 0x1000 );

const qword edge_offset_lo_0     = si_selb   ( edge_offset_lo_0x00,      edge_offset_lo_0x45, cube_r45 );
const qword edge_offset_lo       = si_selb   ( edge_offset_lo_0,        edge_offset_lo_0x90, cube_r90 );
const qword edge_indices_lo      = si_or     ( edge_indices,            edge_indices_hi_mask );

const qword edge_start_offset_lo  = si_shufb  ( edge_offset_lo,          zero,    edge_indices_lo );
const qword edge_start_offset_hi_0 = si_shufb  ( edge_offset_hi,          zero,    edge_indices_lo );
const qword edge_start_offset_hi  = si_shlhi  ( edge_start_offset_hi_0,    0x08 );
const qword edge_start_offset     = si_or     ( edge_start_offset_lo,      edge_start_offset_hi );

const qword edge_stride           = si_shufb  ( edge_stride_lo,          zero,    edge_indices_lo );
const qword edge_stride_offset_hi_0 = si_mpyhhu ( edge_stride,                cube_stride_ndx );
const qword edge_stride_offset_hi  = si_shli  ( edge_stride_offset_hi_0,  0x10 );
const qword edge_stride_offset_lo  = si_mpyu  ( edge_stride,                cube_stride_ndx );
const qword edge_stride_offset     = si_or     ( edge_stride_offset_lo,    edge_stride_offset_hi );

const qword edge_offset_0         = si_ah     ( edge_start_offset,        edge_stride_offset );
const qword edge_offset           = si_ah     ( edge_offset_0,            cube_m15 );

return (vector unsigned short)( edge_offset );
}
```

# Step 2: Initial SPU version

```
vector unsigned short  
ReindexEdgesBlockCube( vector unsigned short v_edge_indices, uint16_t s_cube )  
{  
    const qword    cube_prefered    = si_from_ushort( s_cube );  
    const qword    edge_indices     = (qword)v_edge_indices;
```

*qword is the "native" 128 bit type for the SPU. There's no distinction on type of values stored.*

*There's only one register file ( 128 qwords ) and all instructions can use any register in the file.*

*si\_from\_ushort tells compiler to interpret uint16\_t as a qword. No actual code is generated (usually).*

# Step 2: Initial SPU version

```
// Broadcast cube
const qword broadcast_halfword = si_ilh    ( 0x0203 );
const qword cube                = si_shufb ( cube_prefered, cube_prefered, broadcast_halfword );
```

*ilh: Immediate load halfword. The value u16 is loaded into each of the 8 halfword elements of rt.*

*shuf: You already know...*

# Step 2: Initial SPU version

```
// cube_q15 = (cube / 15)

const qword cube_q15_0      = si_shlhi ( cube,          0x04 );
const qword cube_q15_1      = si_ah   ( cube,          cube_q15_0 );
const qword cube_q15_2      = si_ahi   ( cube_q15_1,    0x0f );
const qword cube_q15        = si_rothmi ( cube_q15_2,   -0x08 );
```

*Compares to scalar version:*

```
uint16_t cube_n0      = cube << 4;
uint16_t cube_n1      = cube_n0 + cube;
uint16_t cube_n2      = cube_n1 + 16;
uint16_t cube_q15     = cube_n2 >> 8;
```

*shlhi: Shift left halfword immediate. The contents of each halfword element register ra are shifted left according to unsigned value u5. The placed in the corresponding halfword elements of register rt.*

*ah: Add halfword. Each halfword element of register ra is added to the corresponding halfword element of register rb, and the results are placed in the corresponding halfword elements of register rt.*

*ahi: Add halfword immediate. The sign-extended immediate value s10 is added to each halfword element of register ra, and the results are placed in the corresponding halfword elements of register rt.*

*rothmi: Rotate and mask halfword immediate. The contents of each halfword element of register ra are right shifted according to the two's complement of the signed value s6. The results are placed in the corresponding halfword elements of register rt.*

# Step 2: Initial SPU version

```
// cube_m15 = (cube % 15)
```

```
const qword cube_m15_0      = si_shlhi ( cube_q15,      0x04 );  
const qword cube_m15_1      = si_sfh   ( cube_q15,      cube_m15_0 );  
const qword cube_m15        = si_sfh   ( cube_m15_1,    cube );
```

*Compares to scalar version:*

```
uint16_t cube_m0           = cube_q15 << 4;  
uint16_t cube_m1           = cube_m0 - cube_q15;  
uint16_t cube_m15         = cube - cube_m1;
```

*sfh: Subtract from halfword. Each halfword element of register ra is subtracted from the corresponding halfword element of register rb, and the results are placed in the corresponding word elements of register rt.*

# Step 2: Initial SPU version

```
// cube_r90 = mask when (cube >= 90)
// cube_r45 = mask when (cube >= 45) && (cube < 90)

const qword cube_r90          = si_clgthi ( cube,          0x59 );
const qword cube_r45_0       = si_clgthi ( cube,          0x2c );
const qword cube_r45         = si_andc   ( cube_r45_0,    cube_r90 );
```

*Compares to scalar code (mostly):*

```
uint16_t is_cube_gt_90  = ( cube >= 90 )?0xffff:0;
uint16_t is_cube_gt_45  = ( cube >= 45 )?0xffff:0;
```

```
cube_r45 = ( cube >= 45 ) && ( cube < 90 )
```

*clgthi: Compare logical greater than halfword immediate. Each halfword element of register ra is logically compared with the 16-bit sign-extended value s10. If the halfword in ra is greater than the value in s10, all ones are placed in the corresponding halfword element of register rt. Otherwise, if the halfword in ra is less than or equal to the value in s10, zeros are placed in the corresponding halfword element of register rt.*

*andc: And with complement. The value of register ra is logically ANDed with the complement of register rb, and the result is placed in register rt.*

# Step 2: Initial SPU version

```
// cube_stride_ndx

const qword cube_stride_ndx_0 = si_andhi ( cube_r90, 0x06 );
const qword cube_stride_ndx_1 = si_andhi ( cube_r45, 0x03 );
const qword cube_stride_ndx_2 = si_sfh ( cube_stride_ndx_0, cube_q15 );
const qword cube_stride_ndx = si_sfh ( cube_stride_ndx_1, cube_stride_ndx_2 );
```

*Compares to scalar version:*

```
uint16_t cube_stride_ndx = cube_q15 - ( is_cube_gt_90 & 3 ) - ( is_cube_gt_45 & 3 );
```

*andhi: And halfword immediate. The sign-extended immediate value s10 is logically ANDed with each halfword element of register ra, and the results are placed in the corresponding elements of register rt.*



# Step 2: Initial SPU version

```
const qword zero          = si_ilh    ( 0x0000 );
```

ilh: Immediate load halfword. The value u16 is loaded into each of the 8 halfword elements of rt.

# Step 2: Initial SPU version

```
const qword edge_offset_lo_0    = si_selb ( edge_offset_lo_0x00,    edge_offset_lo_0x45, cube_r45 );  
const qword edge_offset_lo     = si_selb ( edge_offset_lo_0,      edge_offset_lo_0x90, cube_r90 );
```

*Compares to scalar version:*

```
uint8_t* edge_offset_lo;  
  
if ( is_cube_gt_90 )  
{  
    edge_offset_lo = edge_offset_lo_0x90;  
}  
else if ( is_cube_gt_45 )  
{  
    edge_offset_lo = edge_offset_lo_0x45;  
}  
else  
{  
    edge_offset_lo = edge_offset_lo_0x00;  
}
```

*selb: Select bits. Each bit of register rc whose value is 0 selects the corresponding bit from register ra. A bit whose value is 1 selects the corresponding bit from register rb. The quadword result is placed in register rt.*

# Step 2: Initial SPU version

```
const qword edge_indices_hi_mask    = si_ilh    ( 0x1000 );
const qword edge_indices_lo        = si_or     ( edge_indices,          edge_indices_hi_mask );

const qword edge_start_offset_lo    = si_shufb ( edge_offset_lo,          zero,   edge_indices_lo );
const qword edge_start_offset_hi_0 = si_shufb ( edge_offset_hi,          zero,   edge_indices_lo );
const qword edge_start_offset_hi    = si_shlhi ( edge_start_offset_hi_0, 0x08 );
const qword edge_start_offset      = si_or     ( edge_start_offset_lo, edge_start_offset_hi );
```

*Compares with scalar version:*

```
for(int i=0;i<8;i++)
{
    uint16_t edge_ndx          = edge_indices_s.s[i];

    uint16_t edge_start_offset_lo = (uint16_t)edge_offset_lo[ edge_ndx ];
    uint16_t edge_start_offset_hi = (uint16_t)edge_offset_hi[ edge_ndx ];

    uint16_t edge_start_offset  = edge_start_offset_lo | ( edge_start_offset_hi << 8 );
}
```

# Step 2: Initial SPU version

```
const qword edge_stride           = si_shufb ( edge_stride_lo,           zero,  edge_indices_lo );
const qword edge_stride_offset_hi_0 = si_mpyhhu ( edge_stride,           cube_stride_ndx );
const qword edge_stride_offset_hi  = si_shli  ( edge_stride_offset_hi_0, 0x10 );
const qword edge_stride_offset_lo  = si_mpyu  ( edge_stride,           cube_stride_ndx );
const qword edge_stride_offset     = si_or    ( edge_stride_offset_lo,   edge_stride_offset_hi );
```

Compares with scalar version:

```
uint16_t edge_stride_offset = cube_stride_ndx * edge_stride;
```

*mpyhhu: Multiply high high unsigned. The unsigned 16 most significant bits of the word elements of registers ra and rb are multiplied, and the 32-bit products are then placed in the corresponding word elements of register rt.*

*mpyu: Multiply unsigned. The unsigned 16 least significant bits of the corresponding word elements of registers ra and rb are multiplied, and the 32-bit products are placed in the corresponding word elements of register rt.*

*shli: Shift left word immediate. The contents of each word element of register ra are shifted left according to the unsigned value u6. The results are placed in the corresponding word element of register rt.*

# Step 2: Initial SPU version

```
const qword edge_offset_0      = si_ah      ( edge_start_offset,      edge_stride_offset );  
const qword edge_offset      = si_ah      ( edge_offset_0,          cube_m15 );  
  
return (vector unsigned short)( edge_offset );
```

*Compares to scalar version:*

```
uint16_t edge_offset          = edge_start_offset + edge_stride_offset + cube_m15;  
result_indices_s.s[i]        = edge_offset;  
}  
  
return (result_indices_s.v);
```

# Step 3: Analyze the data in context

Think about the traces. What are the longest paths?

Quite a lot of work is done only to calculate `edge_stride_offset`

# Step 3: Analyze the data in context

*edge\_stride\_offset has only 6 unique results across all of the cube [0,134]*

```
0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x---- 0x---- 0x---- 0x----
0x0040 0x0010 0x0040 0x0010 0x0040 0x0010 0x0040 0x0010 0x0010 0x0010 0x0010 0x0010 0x---- 0x---- 0x---- 0x----
0x0080 0x0020 0x0080 0x0020 0x0080 0x0020 0x0080 0x0020 0x0020 0x0020 0x0020 0x0020 0x---- 0x---- 0x---- 0x----
0x00c0 0x0030 0x00c0 0x0030 0x00c0 0x0030 0x00c0 0x0030 0x0030 0x0030 0x0030 0x0030 0x---- 0x---- 0x---- 0x----
0x0100 0x0040 0x0100 0x0040 0x0100 0x0040 0x0100 0x0040 0x0040 0x0040 0x0040 0x0040 0x---- 0x---- 0x---- 0x----
0x0140 0x0050 0x0140 0x0050 0x0140 0x0050 0x0140 0x0050 0x0050 0x0050 0x0050 0x0050 0x---- 0x---- 0x---- 0x----
```

*Determined the old fashioned way:*

```
printf() +
sort +
uniq
```

# Step 3: Analyze the data in context

Create a small indirection table which tells us which of these to use based on cube.

- Index only requires 3 bits
- Use upper bits of cube to load quadword
- Use lower bits of cube to select byte with indirect offset
- Requires a 9 quadword table.

```

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01
0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x02 0x02
0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x03 0x03 0x03
0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x04 0x04 0x04 0x04
0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x05 0x05 0x05 0x05 0x05
0x05 0x05 0x05 0x05 0x05 0x05 0x05 0x05 0x05 0x05 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x01 0x01 0x01 0x01 0x01 0x01
0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02
0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x--
    
```

# Step 3: Analyze the data in context

*On the other hand,  $(edge\_stride\_offset+edge\_start\_offset)$  has only 9 unique results across all of the cube [0,134]*

```

0x0200 0x0140 0x0240 0x0100 0x0201 0x0141 0x0241 0x0101 0x0000 0x0040 0x0050 0x0010 0x---- 0x---- 0x---- 0x----
0x0220 0x01c0 0x0260 0x0180 0x0221 0x01c1 0x0261 0x0181 0x0080 0x00c0 0x00d0 0x0090 0x---- 0x---- 0x---- 0x----
0x0240 0x0150 0x0280 0x0110 0x0241 0x0151 0x0281 0x0111 0x0010 0x0050 0x0060 0x0020 0x---- 0x---- 0x---- 0x----
0x0260 0x01d0 0x02a0 0x0190 0x0261 0x01d1 0x02a1 0x0191 0x0090 0x00d0 0x00e0 0x00a0 0x---- 0x---- 0x---- 0x----
0x0280 0x0160 0x02c0 0x0120 0x0281 0x0161 0x02c1 0x0121 0x0020 0x0060 0x0070 0x0030 0x---- 0x---- 0x---- 0x----
0x02a0 0x01e0 0x02e0 0x01a0 0x02a1 0x01e1 0x02e1 0x01a1 0x00a0 0x00e0 0x00f0 0x00b0 0x---- 0x---- 0x---- 0x----
0x02c0 0x0170 0x0300 0x0130 0x02c1 0x0171 0x0301 0x0131 0x0030 0x0070 0x0080 0x0040 0x---- 0x---- 0x---- 0x----
0x0300 0x0180 0x0340 0x0140 0x0301 0x0181 0x0341 0x0141 0x0040 0x0080 0x0090 0x0050 0x---- 0x---- 0x---- 0x----
0x0340 0x0190 0x0380 0x0150 0x0341 0x0191 0x0381 0x0151 0x0050 0x0090 0x00a0 0x0060 0x---- 0x---- 0x---- 0x----

```

*This can save quite a bit of work. Remember:*

```

const qword edge_start_offset_lo      = si_shufb ( edge_offset_lo,      zero,  edge_indices_lo );
const qword edge_start_offset_hi_0    = si_shufb ( edge_offset_hi,      zero,  edge_indices_lo );
const qword edge_start_offset_hi      = si_shlhi ( edge_start_offset_hi_0, 0x08 );
const qword edge_start_offset        = si_or    ( edge_start_offset_lo, edge_start_offset_hi );

```

# Step 3: Analyze the data in context

*But for that we need to load:*

```
const qword edge_offset_hi      = (qword)(vector unsigned char){ 0x02, 0x01, 0x02, 0x01,
                                                                    0x02, 0x01, 0x02, 0x01,
                                                                    0x00, 0x00, 0x00, 0x00,
                                                                    0x00, 0x00, 0x00, 0x00 };

const qword edge_offset_lo_0x00 = (qword)(vector unsigned char){ 0x00, 0x40, 0x40, 0x00,
                                                                    0x01, 0x41, 0x41, 0x01,
                                                                    0x00, 0x40, 0x50, 0x10,
                                                                    0x00, 0x00, 0x00, 0x00 };

const qword edge_offset_lo_0x45 = (qword)(vector unsigned char){ 0x10, 0x80, 0x50, 0x40,
                                                                    0x11, 0x81, 0x51, 0x41,
                                                                    0x40, 0x80, 0x90, 0x50,
                                                                    0x00, 0x00, 0x00, 0x00 };

const qword edge_offset_lo_0x90 = (qword)(vector unsigned char){ 0x20, 0xc0, 0x60, 0x80,
                                                                    0x21, 0xc1, 0x61, 0x81,
                                                                    0x80, 0xc0, 0xd0, 0x90,
                                                                    0x00, 0x00, 0x00, 0x00 };
```

*And select:*

```
const qword edge_offset_lo_0      = si_selb ( edge_offset_lo_0x00,      edge_offset_lo_0x45, cube_r45 );
const qword edge_offset_lo        = si_selb ( edge_offset_lo_0,        edge_offset_lo_0x90, cube_r90 );
```

# Step 3: Analyze the data in context

*Adding the additional entries to the indirection table above would only require one extra bit for a total of four bits needed per entry.*

```

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01
0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x02 0x02
0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x02 0x03 0x03 0x03
0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x03 0x04 0x04 0x04 0x04
0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x04 0x05 0x05 0x05 0x05 0x05
0x05 0x05 0x05 0x05 0x05 0x05 0x05 0x05 0x05 0x05 0x05 0x06 0x06 0x06 0x06 0x06 0x06
0x06 0x06 0x06 0x06 0x06 0x06 0x06 0x06 0x06 0x06 0x07 0x07 0x07 0x07 0x07 0x07 0x07
0x07 0x07 0x07 0x07 0x07 0x07 0x07 0x07 0x08 0x08 0x08 0x08 0x08 0x08 0x08 0x08 0x08
0x08 0x08 0x08 0x08 0x08 0x08 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x--

```

*Interestingly, the table above represents ( cube / 15 ) – No point in doing the divide.*

*But there's still the mod.*

```

// cube_m15 = (cube % 15)
const qword cube_m15_0      = si_shlhi ( cube_q15,      0x04 );
const qword cube_m15_1      = si_sfh   ( cube_q15,      cube_m15_0 );
const qword cube_m15        = si_sfh   ( cube_m15_1,     cube );

```

*Mod 15 only requires four bits per entry to store.*

*There are four bits free in each entry.*

*Seems a natural fit for the upper 4 bits.*

```

0x00 0x10 0x20 0x30 0x40 0x50 0x60 0x70 0x80 0x90 0xa0 0xb0 0xc0 0xd0 0xe0 0x01
0x11 0x21 0x31 0x41 0x51 0x61 0x71 0x81 0x91 0xa1 0xb1 0xc1 0xd1 0xe1 0x02 0x12
0x22 0x32 0x42 0x52 0x62 0x72 0x82 0x92 0xa2 0xb2 0xc2 0xd2 0xe2 0x03 0x13 0x23
0x33 0x43 0x53 0x63 0x73 0x83 0x93 0xa3 0xb3 0xc3 0xd3 0xe3 0x04 0x14 0x24 0x34
0x44 0x54 0x64 0x74 0x84 0x94 0xa4 0xb4 0xc4 0xd4 0xe4 0x05 0x15 0x25 0x35 0x45
0x55 0x65 0x75 0x85 0x95 0xa5 0xb5 0xc5 0xd5 0xe5 0x06 0x16 0x26 0x36 0x46 0x56
0x66 0x76 0x86 0x96 0xa6 0xb6 0xc6 0xd6 0xe6 0x07 0x17 0x27 0x37 0x47 0x57 0x67
0x77 0x87 0x97 0xa7 0xb7 0xc7 0xd7 0xe7 0x08 0x18 0x28 0x38 0x48 0x58 0x68 0x78
0x88 0x98 0xa8 0xb8 0xc8 0xd8 0xe8 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x--

```

# Step 3: Analyze the data in context

*What if we stored the divide in the upper nibble and the mod in the lower instead?*

```

0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x10
0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x20 0x21
0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x30 0x31 0x32
0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x3a 0x3b 0x3c 0x3d 0x3e 0x40 0x41 0x42 0x43
0x44 0x45 0x46 0x47 0x48 0x49 0x4a 0x4b 0x4c 0x4d 0x4e 0x50 0x51 0x52 0x53 0x54
0x55 0x56 0x57 0x58 0x59 0x5a 0x5b 0x5c 0x5d 0x5e 0x60 0x61 0x62 0x63 0x64 0x65
0x66 0x67 0x68 0x69 0x6a 0x6b 0x6c 0x6d 0x6e 0x70 0x71 0x72 0x73 0x74 0x75 0x76
0x77 0x78 0x79 0x7a 0x7b 0x7c 0x7d 0x7e 0x80 0x81 0x82 0x83 0x84 0x85 0x86 0x87
0x88 0x89 0x8a 0x8b 0x8c 0x8d 0x8e 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x-- 0x--
  
```

*It's a counter that skips every fifteenth number!  
Seems so obvious in retrospect.*

*Just interesting... for now.*

# Step 4: Rewrite code and test

*Left with three tables:*

```
vector unsigned char divmod_table[] = {
    { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x10 },
    { 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x20, 0x21 },
    { 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x30, 0x31, 0x32 },
    { 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x40, 0x41, 0x42, 0x43 },
    { 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x50, 0x51, 0x52, 0x53, 0x54 },
    { 0x55, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x60, 0x61, 0x62, 0x63, 0x64, 0x65 },
    { 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76 },
    { 0x77, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87 },
    { 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98 } };
```

```
vector unsigned char edge_offset_table_hi[] = {
    { 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, },
    { 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, },
    { 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, },
    { 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, },
    { 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, },
    { 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, },
    { 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, },
    { 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, } };
```

```
vector unsigned char edge_offset_table_lo[] = {
    { 0x00, 0x40, 0x40, 0x00, 0x01, 0x41, 0x41, 0x01, 0x00, 0x40, 0x50, 0x10, 0xff, 0xff, 0xff, 0xff, },
    { 0x40, 0x50, 0x80, 0x10, 0x41, 0x51, 0x81, 0x11, 0x10, 0x50, 0x60, 0x20, 0xff, 0xff, 0xff, 0xff, },
    { 0x80, 0x60, 0xc0, 0x20, 0x81, 0x61, 0xc1, 0x21, 0x20, 0x60, 0x70, 0x30, 0xff, 0xff, 0xff, 0xff, },
    { 0x10, 0x80, 0x50, 0x40, 0x11, 0x81, 0x51, 0x41, 0x40, 0x80, 0x90, 0x50, 0xff, 0xff, 0xff, 0xff, },
    { 0x50, 0x90, 0x90, 0x50, 0x51, 0x91, 0x91, 0x51, 0x50, 0x90, 0xa0, 0x60, 0xff, 0xff, 0xff, 0xff, },
    { 0x90, 0xa0, 0xd0, 0x60, 0x91, 0xa1, 0xd1, 0x61, 0x60, 0xa0, 0xb0, 0x70, 0xff, 0xff, 0xff, 0xff, },
    { 0x20, 0xc0, 0x60, 0x80, 0x21, 0xc1, 0x61, 0x81, 0x80, 0xc0, 0xd0, 0x90, 0xff, 0xff, 0xff, 0xff, },
    { 0x60, 0xd0, 0xa0, 0x90, 0x61, 0xd1, 0xa1, 0x91, 0x90, 0xd0, 0xe0, 0xa0, 0xff, 0xff, 0xff, 0xff, },
    { 0xa0, 0xe0, 0xe0, 0xa0, 0xa1, 0xe1, 0xe1, 0xa1, 0xa0, 0xe0, 0xf0, 0xb0, 0xff, 0xff, 0xff, 0xff, } };
```

# Step 4: Rewrite code and test

```
vector unsigned short
ReindexEdgesBlockCube( vector unsigned short edge_indices, uint32_t cube )
{
    qword qcube                = si_from_uint( cube );
    qword byte_prefered       = si_ilh( 0x0300 );
    qword zero                 = si_ilh( 0x0000 );

    qword divmod_table_addr    = si_from_uint( (uintptr_t)divmod_table );
    qword edge_offset_table_hi_addr = si_from_uint( (uintptr_t)edge_offset_table_hi );
    qword edge_offset_table_lo_addr = si_from_uint( (uintptr_t)edge_offset_table_lo );

    qword cube_divmod_line     = si_lqx( divmod_table_addr, qcube );
    qword cube_all             = si_shufb( qcube, qcube, byte_prefered );
    qword cube_lo_allb        = si_andbi( cube_all, 0x0f );
    qword cube_lo_allh        = si_orhi( cube_lo_allb, 0x0010 );
    qword cube_divmod         = si_shufb( cube_divmod_line, zero, cube_lo_allh );

    qword cube_div            = si_rotmi( cube_divmod, -24 );
    qword edge_offset_lo      = si_lqx( edge_offset_table_lo_addr, cube_div );
    qword edge_offset_hi      = si_lqx( edge_offset_table_hi_addr, cube_div );

    qword cube_mod_0         = si_rotmi( cube_divmod, -8 );
    qword cube_mod           = si_andbi( cube_mod_0, 0x0f );

    qword edge_indices_hi_mask = si_ilh( 0x1000 );
    qword edge_indices_lo     = si_orhi( edge_indices, 0x0010 );
    qword edge_offset_lo_sorted_0 = si_shufb( zero, edge_offset_lo, edge_indices_lo );
    qword edge_offset_lo_sorted = si_ah( edge_offset_lo_sorted_0, cube_mod );
    qword edge_offset_hi_sorted_0 = si_shufb( zero, edge_offset_hi, edge_indices_lo );
    qword edge_offset_hi_sorted = si_shlhi( edge_offset_hi_sorted_0, 8 );

    qword edge_offset        = si_or( edge_offset_hi_sorted, edge_offset_lo_sorted );

    return (edge_offset);
}
```

# Step 4: Rewrite code and test

*Setup:*

```
qword qcube           = si_from_uint( cube );  
qword byte_prefered  = si_ilh( 0x0300 );  
qword zero           = si_ilh( 0x0000 );
```

# Step 4: Rewrite code and test

Prepare table addresses:

```
qword divmod_table_addr      = si_from_uint( (uintptr_t)divmod_table );
qword edge_offset_table_hi_addr = si_from_uint( (uintptr_t)edge_offset_table_hi );
qword edge_offset_table_lo_addr = si_from_uint( (uintptr_t)edge_offset_table_lo );
```

# Step 4: Rewrite code and test

*Load (cube/15 + cube%15) into each halfword (indexed by cube)*

```
qword cube_divmod_line      = si_lqx( divmod_table_addr, qcube );
qword cube_all              = si_shufb( qcube, qcube, byte_prefered );
qword cube_lo_allb         = si_andbi( cube_all, 0x0f );
qword cube_lo_allh         = si_orhi( cube_lo_allb, 0x0010 );
qword cube_divmod           = si_shufb( cube_divmod_line, zero, cube_lo_allh );
```

*lqx: Load quadword (x-form). A quadword is loaded into register rt from the effective address computed by the sum of registers ra and rb.*

*andbi: And byte immediate. The 8 least significant bits of s10 are logically ANDed with each byte element of register ra, and the results are placed in the corresponding elements of register rt.*

*orhi: Or halfword immediate. The sign-extended value s10 is logically ORed with each halfword element of register ra, and the results are placed in the corresponding elements of register rt.*

# Step 4: Rewrite code and test

*Load low and high bytes of edge offset (indexed by cube/15)*

```
qword cube_div          = si_rotmi( cube_divmod, -24 );  
qword edge_offset_lo   = si_lqx( edge_offset_table_lo_addr, cube_div );  
qword edge_offset_hi   = si_lqx( edge_offset_table_hi_addr, cube_div );
```

*Note: Within a word, cube\_divmod is here: 0x0n000000*

*We need it to be here: 0x0000000n*

*rotmi: Rotate and mask word immediate. The contents of each word element of register ra are right shifted according to the two's complement of the signed value s7. The results are placed in the corresponding word elements of register rt.*

# Step 4: Rewrite code and test

*Broadcast (cube%15) to halfwords*

```
qword cube_mod_0      = si_rotmi( cube_divmod, -8 );  
qword cube_mod        = si_andbi( cube_mod_0, 0x0f );
```

*Note: Within a half word, cube\_divmod is here: 0x0n00  
We need it to be here: 0x000n*

*And we need ONLY 0x000n*

# Step 4: Rewrite code and test

*Select and sort the 8 halfwords from edge\_indices*

```
qword edge_indices_hi_mask    = si_ilh( 0x1000 );
qword edge_indices_lo        = si_orhi( edge_indices, 0x0010 );
qword edge_offset_lo_sorted_0 = si_shufb( zero, edge_offset_lo, edge_indices_lo );
qword edge_offset_lo_sorted   = si_ah( edge_offset_lo_sorted_0, cube_mod );
qword edge_offset_hi_sorted_0 = si_shufb( zero, edge_offset_hi, edge_indices_lo );
qword edge_offset_hi_sorted   = si_shlhi( edge_offset_hi_sorted_0, 8 );
```

*Combine and return results*

```
qword edge_offset = si_or( edge_offset_hi_sorted, edge_offset_lo_sorted );

return (edge_offset);
```

# Step 5: Rearrange Deck Chairs

*High bytes of edge\_offset are always the same*

```
vector unsigned char edge_offset_table_hi[] ={
    { 0x02,0x01,0x02,0x01,0x02,0x01,0x02,0x01,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff, },
    { 0x02,0x01,0x02,0x01,0x02,0x01,0x02,0x01,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff, },
    { 0x02,0x01,0x02,0x01,0x02,0x01,0x02,0x01,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff, },
    { 0x02,0x01,0x02,0x01,0x02,0x01,0x02,0x01,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff, },
    { 0x02,0x01,0x02,0x01,0x02,0x01,0x02,0x01,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff, },
    { 0x02,0x01,0x02,0x01,0x02,0x01,0x02,0x01,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff, },
    { 0x02,0x01,0x02,0x01,0x02,0x01,0x02,0x01,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff, },
    { 0x02,0x01,0x02,0x01,0x02,0x01,0x02,0x01,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff, },
    { 0x02,0x01,0x02,0x01,0x02,0x01,0x02,0x01,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff, } };
```

*And it's easy to generate:*

```
const qword edge_offset_hi_0      = si_ilh      ( 0x0201 );
const qword edge_offset_hi      = si_shlqbyi ( edge_offset_hi_0,          0x08 );
```

*slqbyi: Shift left quadword by bytes immediate. The contents of register ra are shifted left by the number of bytes specified by the unsigned value u5. The result is placed in register rt.*

# Step 4: Rewrite code and test

*Re-work select and sort for less dependencies. From:*

```

qword edge_indices_hi_mask      = si_ilh( 0x1000 );
qword edge_indices_lo          = si_orhi( edge_indices, 0x0010 );
qword edge_offset_lo_sorted_0  = si_shufb( zero, edge_offset_lo, edge_indices_lo );
qword edge_offset_lo_sorted    = si_ah( edge_offset_lo_sorted_0, cube_mod );
qword edge_offset_hi_sorted_0  = si_shufb( zero, edge_offset_hi, edge_indices_lo );
qword edge_offset_hi_sorted    = si_shlhi( edge_offset_hi_sorted_0, 8 );

```

*To:*

```

qword edge_offset_lo_sorted_0  = si_shufb ( edge_offset_lo,          edge_offset_lo,  edge_indices );
qword edge_offset_lo_sorted_1  = si_ah   ( edge_offset_lo_sorted_0, cube_mod );
qword edge_offset_lo_sorted    = si_andhi ( edge_offset_lo_sorted_1, 0x00ff );
qword edge_offset_hi_sorted_0  = si_shufb ( edge_offset_hi,          edge_offset_hi,  edge_indices );
qword edge_offset_hi_sorted    = si_shlhi ( edge_offset_hi_sorted_0, 0x08 );
qword edge_offset              = si_or    ( edge_offset_hi_sorted,  edge_offset_lo_sorted );

```

*i.e.*

```
offset = ( ( lo + mod ) & 0x00ff ) | ( hi << 8 )
```

# Step 5: Look at the data again

```
vector unsigned char divmod_table[] = {
    { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x10 },
    { 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x20, 0x21 },
    { 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x30, 0x31, 0x32 },
    { 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x40, 0x41, 0x42, 0x43 },
    { 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x50, 0x51, 0x52, 0x53, 0x54 },
    { 0x55, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x60, 0x61, 0x62, 0x63, 0x64, 0x65 },
    { 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76 },
    { 0x77, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87 },
    { 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98 } };
```

Stared at this table.

There's something here...

Something obvious...

What is it...?

# Step 5: Look at the data again

*Look at x (i.e. cube)...*

x	x
15	0x0f
30	0x1e
45	0x2d
60	0x3c
75	0x4b
90	0x5a
105	0x69
120	0x78

# Step 5: Look at the data again

*What I want to do is make it divisible by 16.  
How far off is it?*

x	x	div16
---	----	-----
15	0x0f	+0x01
30	0x1e	+0x02
45	0x2d	+0x03
60	0x3c	+0x04
75	0x4b	+0x05
90	0x5a	+0x06
105	0x69	+0x07
120	0x78	+0x08

# Step 5: Look at the data again

*What I want to do is make it divisible by 16.  
How far off is it?*

x	x	div16
---	----	-----
15	0x0f	+0x01
30	0x1e	+0x02
45	0x2d	+0x03
60	0x3c	+0x04
75	0x4b	+0x05
90	0x5a	+0x06
105	0x69	+0x07
13	0x78	+0x08

*e.g.*

```
(104) 0x69
      +0x07
      -----
      0x70
```

```
0x70 >> 4 = 0x07
0x69 / 0x0f = 0x07
```

*I need a counter.*

```
<= 15, +0x01
<= 30, +0x02
<= 45, +0x03
```

*etc.*

# Step 5: Look at the data again

*Look at x again...*

x	x
15	0x0f
30	0x1e
45	0x2d
60	0x3c
75	0x4b
90	0x5a
105	0x69
120	0x78

*Just the top nibble...*

x	x
15	0x0-
30	0x1-
45	0x2-
60	0x3-
75	0x4-
90	0x5-
105	0x6-
25	0x7-

I have a counter. It's exactly what I want, only...

- (1) It's in the wrong place ( `x >> 4` )
- (2) It's one off ( `+ 0x01` )

# Step 5: Look at the data again

*Transforming x into the counter I want...*

x	(x >> 4)	(x >> 4)+1
15	0x00	0x01
30	0x01	0x02
45	0x02	0x03
60	0x03	0x04
75	0x04	0x05
90	0x05	0x06
105	0x06	0x07
12	0x07	0x08

*Look at the results...*

x	x	(x >> 4)+1	q_0	(q_0 >> 4)
15	0x0f + 0x01	0x10	0x10	0x01
30	0x1e + 0x02	0x20	0x20	0x02
45	0x2d + 0x03	0x30	0x30	0x03
60	0x3c + 0x04	0x40	0x40	0x04
75	0x4b + 0x05	0x50	0x50	0x05
90	0x5a + 0x06	0x60	0x60	0x06
105	0x69 + 0x07	0x70	0x70	0x07
120	0x78 + 0x08	0x80	0x80	0x08

# Step 5: Look at the data again

*Make sure to test it...*

```
int
main( void )
{
    uint16_t x;

    for (x=0;x<135;x++)
    {
        uint16_t z0 = x >> 4;
        uint16_t z1 = x + z0 + 1;
        uint16_t z2 = z1 >> 4;

        if ( z2 != (x/15) )
        {
            printf("%d/15 != %d (%d)\n",x,z2,x/15);
        }
    }

    return (0);
}
```

# Step 5: Look at the data again

Not coincidentally, I'm using  $x/15$  to look up into a quadword table.

- (1) Each quadword is 16 bytes, so...
- (2) The address will be offset by  $(x/15) \ll 4$ , so...

What I really want is  $q\_0$ . I can drop the final shift.

x	x	(x >> 4)+1	q_0	(q_0 >> 4)
---	-----	+	-----	= -> -----
15	0x0f	+ 0x01	0x10	0x01
30	0x1e	+ 0x02	0x20	0x02
45	0x2d	+ 0x03	0x30	0x03
60	0x3c	+ 0x04	0x40	0x04
75	0x4b	+ 0x05	0x50	0x05
90	0x5a	+ 0x06	0x60	0x06
105	0x69	+ 0x07	0x70	0x07
120	0x78	+ 0x08	0x80	0x08

This is the reason why the bottom 4 bits in the address of a quadword load are ignored.

```
offset = x + (x >> 4) + 1;
```

x	x	(x >> 4)+1	q_0	Address offset
---	-----	+	-----	= -> -----
30	0x1e	+ 0x02	0x20	0x20
31	0x1f	+ 0x02	0x21	0x20
32	0x20	+ 0x03	0x23	0x20
33	0x21	+ 0x03	0x24	0x20
34	0x22	+ 0x03	0x25	0x20
35	0x23	+ 0x03	0x26	0x20
36	0x24	+ 0x03	0x27	0x20
37	0x25	+ 0x03	0x28	0x20

# Step 5: Look at the data again

*New SPU version of divide by 15:*

```
const qword cube_div_0  = si_rotmi ( cube, -0x04 );  
const qword cube_div_1  = si_ai( cube_divmod_0, 0x01 );  
const qword cube_div    = si_a( cube, cube_divmod_1 );
```

*But even if I can calculate the divide fast, I still have the mod.*

*Without the divide, it now looks like:*

```
{ 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00 }  
{ 0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01 }  
{ 0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02 }  
{ 0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03 }  
{ 0x04,0x05,0x06,0x07,0x08,0x49,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04 }  
{ 0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04,0x05 }  
{ 0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04,0x05,0x06 }  
{ 0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07 }  
{ 0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08 }
```

# Step 5: Look at the data again

*First I considered just loading the first line and rotating it and modifying the last byte:*

```
{ 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00 } = ROT 0 + 0x00
{ 0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01 } = ROT 1 + 0x01
{ 0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02 } = ROT 2 + 0x02
{ 0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03 } = ROT 3 + 0x03
{ 0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04 } = ROT 4 + 0x04
{ 0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04,0x05 } = ROT 5 + 0x05
{ 0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04,0x05,0x06 } = ROT 6 + 0x06
{ 0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07 } = ROT 7 + 0x07
{ 0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08 } = ROT 8 + 0x08
```

*But I decided that was a lot of extra work for a load I was going to pay for anyway.  
So I dropped the table and went back to calculating it.*

```
mod15 = x - ( div15 * 15 );
        = x - (( div15 << 4 ) - div15);
```

*That's still a lot of work in practice though:*

```
qword cube_mod_0 = si_andi( cube_div, 0x00f0 ); // div15 << 4
qword cube_mod_1 = si_rotmi( cube_mod_0, -0x04 ); // div15
qword cube_mod_2 = si_sf( cube_mod_1, cube_mod_0 ); // ( div15 << 4 ) - div15
qword cube_mod_3 = si_sf( cube_mod_2, cube ); // x - (( div15 << 4 ) - div15)
qword cube_mod_4 = si_shlqbyi( cube_div, 0x02 ); // mod on opposite half words
qword cube_mod = si_or( cube_divmod, cube_mod_0 ); // mod splat on all half words
```

# Step 5: Look at the data again

*Then I stopped and thought for a second.*

*What am I using the mod for?*

# Step 5: Look at the data again

*To use as a counter to offset the results for each line between those divisible by 15:*

```
[15] 0x0240 0x0150 0x0280 0x0110 0x0241 0x0151 0x0281 0x0111 0x0010 0x0050 0x0060 0x0020 +0x00
[16] 0x0241 0x0151 0x0281 0x0111 0x0242 0x0152 0x0282 0x0112 0x0011 0x0051 0x0061 0x0021 +0x01
[17] 0x0242 0x0152 0x0282 0x0112 0x0243 0x0153 0x0283 0x0113 0x0012 0x0052 0x0062 0x0022 +0x02
[18] 0x0243 0x0153 0x0283 0x0113 0x0244 0x0154 0x0284 0x0114 0x0013 0x0053 0x0063 0x0023 +0x03
[19] 0x0244 0x0154 0x0284 0x0114 0x0245 0x0155 0x0285 0x0115 0x0014 0x0054 0x0064 0x0024 +0x04
[20] 0x0245 0x0155 0x0285 0x0115 0x0246 0x0156 0x0286 0x0116 0x0015 0x0055 0x0065 0x0025 +0x05
[21] 0x0246 0x0156 0x0286 0x0116 0x0247 0x0157 0x0287 0x0117 0x0016 0x0056 0x0066 0x0026 +0x06
[22] 0x0247 0x0157 0x0287 0x0117 0x0248 0x0158 0x0288 0x0118 0x0017 0x0057 0x0067 0x0027 +0x07
[23] 0x0248 0x0158 0x0288 0x0118 0x0249 0x0159 0x0289 0x0119 0x0018 0x0058 0x0068 0x0028 +0x08
[24] 0x0249 0x0159 0x0289 0x0119 0x024a 0x015a 0x028a 0x011a 0x0019 0x0059 0x0069 0x0029 +0x09
[25] 0x024a 0x015a 0x028a 0x011a 0x024b 0x015b 0x028b 0x011b 0x001a 0x005a 0x006a 0x002a +0x0a
[26] 0x024b 0x015b 0x028b 0x011b 0x024c 0x015c 0x028c 0x011c 0x001b 0x005b 0x006b 0x002b +0x0b
[27] 0x024c 0x015c 0x028c 0x011c 0x024d 0x015d 0x028d 0x011d 0x001c 0x005c 0x006c 0x002c +0x0c
[28] 0x024d 0x015d 0x028d 0x011d 0x024e 0x015e 0x028e 0x011e 0x001d 0x005d 0x006d 0x002d +0x0d
[29] 0x024e 0x015e 0x028e 0x011e 0x024f 0x015f 0x028f 0x011f 0x001e 0x005e 0x006e 0x002e +0x0e
```

# Step 5: Look at the data again

*I have such a counter already.*

*The cube index itself:*

```
[15] 0x0240 0x0150 0x0280 0x0110 0x0241 0x0151 0x0281 0x0111 0x0010 0x0050 0x0060 0x0020 -0x0f
[16] 0x0241 0x0151 0x0281 0x0111 0x0242 0x0152 0x0282 0x0112 0x0011 0x0051 0x0061 0x0021 -0x0f
[17] 0x0242 0x0152 0x0282 0x0112 0x0243 0x0153 0x0283 0x0113 0x0012 0x0052 0x0062 0x0022 -0x0f
[18] 0x0243 0x0153 0x0283 0x0113 0x0244 0x0154 0x0284 0x0114 0x0013 0x0053 0x0063 0x0023 -0x0f
[19] 0x0244 0x0154 0x0284 0x0114 0x0245 0x0155 0x0285 0x0115 0x0014 0x0054 0x0064 0x0024 -0x0f
[20] 0x0245 0x0155 0x0285 0x0115 0x0246 0x0156 0x0286 0x0116 0x0015 0x0055 0x0065 0x0025 -0x0f
[21] 0x0246 0x0156 0x0286 0x0116 0x0247 0x0157 0x0287 0x0117 0x0016 0x0056 0x0066 0x0026 -0x0f
[22] 0x0247 0x0157 0x0287 0x0117 0x0248 0x0158 0x0288 0x0118 0x0017 0x0057 0x0067 0x0027 -0x0f
[23] 0x0248 0x0158 0x0288 0x0118 0x0249 0x0159 0x0289 0x0119 0x0018 0x0058 0x0068 0x0028 -0x0f
[24] 0x0249 0x0159 0x0289 0x0119 0x024a 0x015a 0x028a 0x011a 0x0019 0x0059 0x0069 0x0029 -0x0f
[25] 0x024a 0x015a 0x028a 0x011a 0x024b 0x015b 0x028b 0x011b 0x001a 0x005a 0x006a 0x002a -0x0f
[26] 0x024b 0x015b 0x028b 0x011b 0x024c 0x015c 0x028c 0x011c 0x001b 0x005b 0x006b 0x002b -0x0f
[27] 0x024c 0x015c 0x028c 0x011c 0x024d 0x015d 0x028d 0x011d 0x001c 0x005c 0x006c 0x002c -0x0f
[28] 0x024d 0x015d 0x028d 0x011d 0x024e 0x015e 0x028e 0x011e 0x001d 0x005d 0x006d 0x002d -0x0f
[29] 0x024e 0x015e 0x028e 0x011e 0x024f 0x015f 0x028f 0x011f 0x001e 0x005e 0x006e 0x002e -0x0f
```

*... Can't I just offset the values stored in the table?*

# Step 5: Look at the data again

*The problem: It changes the high bytes.*

*The high bytes are not loaded from a table and I don't want to add another load.*

*Change the problem:*

*I need any byte where using a 16 bit add with cube I will get the correct value in the low byte.*

*i.e.*

*$(n + \text{cube}) \& 0x00ff = \text{offset\_lo}$*

*I could have calculated this byte for each element in the table, but...*

*I'm lazy. So, I created a little program to do it for me.*

# Step 5: Look at the data again

```
int
main( void )
{
    uint16_t i;
    uint16_t j;
    uint16_t k;

    printf("{\n");
    for (i=0;i<9;i++)
    {
        uint16_t cube = i * 15;

        printf("    { ");
        for (j=0;j<16;j++)
        {
            for (k=0;k<256;k++)
            {
                uint16_t n = (cube+k)&0x00ff;

                if ( n == lo[i][j] )
                {
                    printf("0x%02x,",k);
                    break;
                }
            }
        }
        printf(" },\n");
    }
    printf("};\n");
}
```

# Step 5: Look at the data again

*So now I have a new low table:*

```
{ 0x00,0x40,0x40,0x00,0x01,0x41,0x41,0x01,0x00,0x40,0x50,0x10,0x6d,0x61,0x63,0x74, },
{ 0x31,0x41,0x71,0x01,0x32,0x42,0x72,0x02,0x01,0x41,0x51,0x11,0x60,0x5f,0x31,0x5a, },
{ 0x62,0x42,0xa2,0x02,0x63,0x43,0xa3,0x03,0x02,0x42,0x52,0x12,0x50,0x55,0x51,0x4f, },
{ 0xe3,0x53,0x23,0x13,0xe4,0x54,0x24,0x14,0x13,0x53,0x63,0x23,0x41,0x3c,0x34,0x36, },
{ 0x14,0x54,0x54,0x14,0x15,0x55,0x55,0x15,0x14,0x54,0x64,0x24,0x2b,0x25,0x31,0x29, },
{ 0x45,0x55,0x85,0x15,0x46,0x56,0x86,0x16,0x15,0x55,0x65,0x25,0x28,0xe3,0x18,0x24, },
{ 0xc6,0x66,0x06,0x26,0xc7,0x67,0x07,0x27,0x26,0x66,0x76,0x36,0x13,0xc6,0xc6,0xc6, },
{ 0xf7,0x67,0x37,0x27,0xf8,0x68,0x38,0x28,0x27,0x67,0x77,0x37,0xb7,0xb7,0xb7,0xb7, },
{ 0x28,0x68,0x68,0x28,0x29,0x69,0x69,0x29,0x28,0x68,0x78,0x38,0xa8,0xa8,0xa8,0xa8, },
```

*I just need to:*

- (1) add cube,*
- (2) then mask off the top bits,*
- (3) then add the high bits as before.*

*i.e. I don't need cube%15*

# Step 6: Wrap up version

*The table that was just built:*

```
vector unsigned char _reindex_edges_block_cube_table[9] =
{
    { 0x00,0x40,0x40,0x00,0x01,0x41,0x41,0x01,0x00,0x40,0x50,0x10,0x6d,0x61,0x63,0x74 },
    { 0x31,0x41,0x71,0x01,0x32,0x42,0x72,0x02,0x01,0x41,0x51,0x11,0x60,0x5f,0x31,0x5a },
    { 0x62,0x42,0xa2,0x02,0x63,0x43,0xa3,0x03,0x02,0x42,0x52,0x12,0x50,0x55,0x51,0x4f },
    { 0xe3,0x53,0x23,0x13,0xe4,0x54,0x24,0x14,0x13,0x53,0x63,0x23,0x41,0x3c,0x34,0x36 },
    { 0x14,0x54,0x54,0x14,0x15,0x55,0x55,0x15,0x14,0x54,0x64,0x24,0x2b,0x25,0x31,0x29 },
    { 0x45,0x55,0x85,0x15,0x46,0x56,0x86,0x16,0x15,0x55,0x65,0x25,0x28,0xe3,0x18,0x24 },
    { 0xc6,0x66,0x06,0x26,0xc7,0x67,0x07,0x27,0x26,0x66,0x76,0x36,0x13,0xc6,0xc6,0xc6 },
    { 0xf7,0x67,0x37,0x27,0xf8,0x68,0x38,0x28,0x27,0x67,0x77,0x37,0xb7,0xb7,0xb7,0xb7 },
    { 0x28,0x68,0x68,0x28,0x29,0x69,0x69,0x29,0x28,0x68,0x78,0x38,0xa8,0xa8,0xa8,0xa8 },
};
```

# Step 6: Wrap up version

*New SPU version:*

```
qword
_reindex_edges_block_cube( qword edge_indices, qword cube, qword edge_offset_table_lo_addr )
{
    // cube_div = cube / 15
    //           = ( cube + (cube >> 4) + 1 ) [ >> 4 ]

    const qword cube_div_0      = si_rotmi ( cube,          -0x04 );
    const qword cube_div_1      = si_ai    ( cube_div_0,    0x01 );
    const qword cube_div        = si_a     ( cube,          cube_div_1 );

    // Broadcast cube to halfwords

    const qword cube_broadcast  = si_ilh   ( 0x0003 );
    const qword cube_offset     = si_shufb ( cube, cube, cube_broadcast );

    // Load low and high bytes of edge offset (indexed by cube/15)

    const qword edge_offset_lo  = si_lqx   ( edge_offset_table_lo_addr, cube_div );
    const qword edge_offset_hi_0 = si_ilh   ( 0x0201 );
    const qword edge_offset_hi  = si_shlqbyi ( edge_offset_hi_0, 0x08 );

    // Select and sort the 8 halfwords from edge_indices then add cube

    const qword edge_offset_lo_sorted_0 = si_shufb ( edge_offset_lo,          edge_offset_lo, edge_indices );
    const qword edge_offset_lo_sorted_1 = si_ah    ( edge_offset_lo_sorted_0, cube_offset );
    const qword edge_offset_lo_sorted   = si_andhi ( edge_offset_lo_sorted_1, 0x00ff );
    const qword edge_offset_hi_sorted_0 = si_shufb ( edge_offset_hi,          edge_offset_hi, edge_indices );
    const qword edge_offset_hi_sorted   = si_shlhi ( edge_offset_hi_sorted_0, 0x08 );
    const qword edge_offset             = si_or    ( edge_offset_hi_sorted,   edge_offset_lo_sorted );

    return (edge_offset);
}
```

# Step 6: Wrap up version

*Typed shell:*

```
vector unsigned short
ReindexEdgesBlockCube( vector unsigned short edge_indices, uint32_t cube )
{
    qword table_addr;
    __asm("ila %0, %1":"=r"(table_addr):"i"(_reindex_edges_block_cube_table));

    const qword qcube      = si_from_uint( cube );
    const qword qedge_indices = (qword)edge_indices;

    return (vector unsigned short)_reindex_edges_block_cube( qedge_indices, qcube, table_addr );
}
```

*ila: Immediate load address. The unsigned value u18 is loaded into each of the word elements of rt.*

# The Final Product

```

00000170 <_reindex_edges_block_cube>:
170: 0f 3f 02 02    rotmi    $2,$4,-4
174: 35 80 00 0f    hbr      1b0 <_reindex_edges_block_cube+0x40>,$0
178: 04 00 01 87    ori      $7,$3,0
17c: 30 81 02 03    lqa      $3,810 <sys_spu_thread_exit+0x10>
180: 43 80 01 86    ila      $6,196611    # 30003
184: 00 20 00 00    lnop
188: 1c 00 41 02    ai       $2,$2,1
18c: b0 c1 02 06    shufb    $6,$4,$4,$6
190: 18 01 01 02    a        $2,$2,$4
194: b0 60 c1 87    shufb    $3,$3,$3,$7
198: 38 80 82 85    lqx      $5,$5,$2
19c: 0f e2 01 83    shlhi    $3,$3,8
1a0: b0 a1 42 87    shufb    $5,$5,$5,$7
1a4: 19 01 82 85    ah       $5,$5,$6
1a8: 15 3f c2 85    andhi    $5,$5,255    # ff
1ac: 08 21 41 83    or       $3,$3,$5
1b0: 35 00 00 00    bi       $0
1b4: 00 20 00 00    lnop

```



The End