Perfect Hash Slider Lookup Explained

By: Aria Kafie

Say there is a bishop on 'e4'. We want a bitmask that represents all the squares that this bishop can legally travel to.

Ignoring piece color for now, and considering the board state at left, such a bitmask would look like this:

Most significant bit

0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	1
0	0	1	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0

Least significant bit

When read from left to right and top to bottom, and as an unsigned 64-bit integer, this value reads 41221400142040 in hexadecimal.

The bit pattern of this mask is entirely dependent upon the occupancy of the squares along the bishop's diagonals, its 'attack rays'. The initial, naive approach would be to feed a mask containing all the occupancy information of these squares into a function, perform some calculations with a loop (or two), and return the above mask.

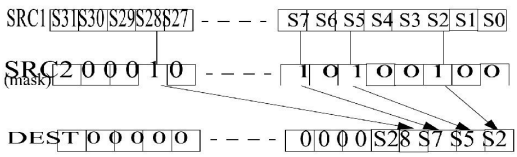
But this is slow, and for something as low level as move generation, constant time operations are needed to ensure that enough time is left for searching the game tree, and evaluating millions of positions.

So, let's consider a hash function. Rather than writing a function to process input directly into output, we want the computer to mindlessly convert each input into a key using a simpler algorithm, then use that key to index an enormous table of precomputed attack masks.

A hash function is needed because, although the raw occupancy bitmask is already an integer, and can therefore be used to directly index an array, its value is too large. In the above example, the occupancy of the squares surrounding e4 would be represented by the decimal number 18,014,398,509,744,128. This alone would require hundreds of quadrillions of bytes of RAM to access directly by array, so clearly a better solution is possible.

To achieve this with zero collisions (i.e. to achieve 'perfect' hashing), we must use some operation that takes the bit string representing square occupancy, extracts from it only the relevant bits (those bits along the diagonal rays emanating from e4), and packs them tightly into an equally unique bitstring at the lower order digits of the output number. This way, the resulting integer is small enough to use directly as an index into an array of precomputed attack masks.

For this operation, the hardware instruction PEXT (parallel bits extract) is exactly what we need.



Because of its implementation at the hardware level, PEXT not only checks all our functional requirements, but is an extremely cheap option.

All that's left is to fill a large table with precomputed attacks, indexed by their corresponding PEXT index. This requires translating an index's binary representation to its corresponding PEXT inverse, or the mask from which it will be derived at runtime. The following code I've written does just that:

```
uint64_t generate_occupancy(uint64_t mask, int iteration) {

        #define popcnt(x) _mm_popcnt_u64(x) // counts the '1' bits in a 64 bit integer
        #define tzcnt(x) _tzcnt_u64(x)      // returns the bit index of the least significant set bit
        #define blsr(x) _blsr_u64(x)        // pops the least significant set bit

        int bitcount = bitcount(mask);
        uint64_t occupancy = 0ull;
        for (int bitpos = 0; bitpos < bitcount; bitpos++) {
                int lsb_index = tzcnt(mask);
                if (iteration & (1 << bitpos))
                        occupancy |= 1ull << lsb_index;
                mask = blsr(mask);
        }
        return occupancy;

}
```