A Short Note on Intel GFNI

Guido Moerkotte

September 10, 2026

Contents

1 Introduction

This paper clarifies the semantics of `fg2p8arrine` intrinsics and shows how to use them to implement various useful functions.

2 GFNI Intrinsics: Overview

2.1 GFNI Intrinsics

As we will not go into assembler programming, we take a look at the intrinsics provided by GFNI (Galois Field New Instructions). They are:

```
__m128i _mm_gf2p8affine_epi64_epi8 (__m128i x, __m128i A, int b)
__m128i _mm_mask_gf2p8affine_epi64_epi8 (__m128i src, __mmask16 k,
                                         __m128i x, __m128i A, int b)
__m128i _mm_maskz_gf2p8affine_epi64_epi8 (__mmask16 k, __m128i x,
                                          __m128i A, int b)
__m256i _mm256_gf2p8affine_epi64_epi8 (__m256i x, __m256i A, int b)
__m256i _mm256_mask_gf2p8affine_epi64_epi8 (__m256i src, __mmask32 k,
                                            __m256i x, __m256i A, int b)
__m256i _mm256_maskz_gf2p8affine_epi64_epi8 (__mmask32 k, __m256i x,
                                             __m256i A, int b)
__m512i _mm512_gf2p8affine_epi64_epi8 (__m512i x, __m512i A, int b)
__m512i _mm512_mask_gf2p8affine_epi64_epi8 (__m512i src, __mmask64 k,
                                            __m512i x, __m512i A, int b)
__m512i _mm512_maskz_gf2p8affine_epi64_epi8 (__mmask64 k, __m512i x,
                                             __m512i A, int b)
__m128i _mm_gf2p8affineinv_epi64_epi8 (__m128i x, __m128i A, int b)
__m128i _mm_mask_gf2p8affineinv_epi64_epi8 (__m128i src, __mmask16 k,
                                            __m128i x, __m128i A, int b)
__m128i _mm_maskz_gf2p8affineinv_epi64_epi8 (__mmask16 k, __m128i x,
                                             __m128i A, int b)
__m256i _mm256_gf2p8affineinv_epi64_epi8 (__m256i x, __m256i A, int b)
__m256i _mm256_mask_gf2p8affineinv_epi64_epi8 (__m256i src, __mmask32 k,
```

```
                                                __m256i x, __m256i A, int b)
__m256i _mm256_maskz_gf2p8affineinv_epi64_epi8 (__mmask32 k, __m256i x,
                                                 __m256i A, int b)
__m512i _mm512_gf2p8affineinv_epi64_epi8 (__m512i x, __m512i A, int b)
__m512i _mm512_mask_gf2p8affineinv_epi64_epi8 (__m512i src, __mmask64 k,
                                               __m512i x, __m512i A, int b)
__m512i _mm512_maskz_gf2p8affineinv_epi64_epi8 (__mmask64 k, __m512i x,
                                                 __m512i A, int b)
__m128i _mm_gf2p8mul_epi8 (__m128i a, __m128i b)
__m128i _mm_mask_gf2p8mul_epi8 (__m128i src, __mmask16 k,
                                __m128i a, __m128i b)
__m128i _mm_maskz_gf2p8mul_epi8 (__mmask16 k, __m128i a, __m128i b)
__m256i _mm256_gf2p8mul_epi8 (__m256i a, __m256i b)
__m256i _mm256_mask_gf2p8mul_epi8 (__m256i src, __mmask32 k,
                                   __m256i a, __m256i b)
__m256i _mm256_maskz_gf2p8mul_epi8 (__mmask32 k, __m256i a, __m256i b)
__m512i _mm512_gf2p8mul_epi8 (__m512i a, __m512i b)
__m512i _mm512_mask_gf2p8mul_epi8 (__m512i src, __mmask64 k,
                                   __m512i a, __m512i b)
__m512i _mm512_maskz_gf2p8mul_epi8 (__mmask64 k, __m512i a, __m512i b)
```

They differ in the vector length, there are masked and unmasked versions but only three different kinds of operations:

1. gf2p8affine
2. gf2p8affineinv
3. gf2p8mul

Here, we are concerned with the `gf2p8affine` variants. Note that these all have an `epi64` in their name. That is, they (roughly) work on `uint64_t` operands returning a `uint64_t` result. To be more specific, we want to clarify what the operation does on a single lane, that is what it does 2, 4 or 8 times depending on the simd-register width of 128, 256, or 512 bits.

2.2 The `gf2p8affine` Intrinsics

The `gf2` stands for GF(2) meaning a Galois Field with two elements. A Field is a set of elements with two operations + and $*$. The neutral element w.r.t + is named '0' and the neutral element w.r.t $*$ is named '1'. Since in GF(2) there are only two elements, we only have 0 and 1, i.e., one bit. The

operations + and $*$ are always taken modulo 2. Thus, the result of + and $*$ is always 0 or 1. More specifically, we have that

$$\begin{aligned} 0+0 &= 0 \\ 1+0 &= 1 \\ 0+1 &= 1 \\ 1+1 &= 0 \end{aligned}$$

and

$$\begin{aligned} 0*0 &= 0 \\ 1*0 &= 0 \\ 0*1 &= 0 \\ 1*1 &= 1 \end{aligned}$$

Thus, + in GF(2) is the same as *bit exclusive or* and $*$ is the same as *bit and*.

The `p8` stands for 8x8 matrices with elements from GF(2). Note that 8*8 = 64 and thus an 8x8 bit matrix can be stored in one `uint64_t`. The *affine* means in the mathematical sense an affine transformation which is essentially a linear function. For A, B matrices and b a vector, $AB + b$ is an affine transformation. However, if we look at the Intel Intrinsics Guide[1], we see that the operations are not described in terms of matrix multiplication. Instead, ignoring vectorization and masking, `gf2p8affine` can be understood as an operation

`gf2p8affine`(uint64_t X, uint64_t A, uint8_t b) $\rightarrow$ uint64_t

Since we look at a single `uint64_t` as an 8x8 bit matrix, which is 8 bytes, we will use the overlay

```
union uint64_ut {
  uint64_t _ui64;
  uint8_t  _ui8[8];
};
```

We can implement an emulation of the `gf2p8affine` intrinsic as an operation `gfni_mul_emu` which uses the two subroutines `parity` and `affine_byte`. This code is a simple translation of the code given on Intel's intrinsics guide to C++.

[1] `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`

```
inline uint8_t
parity(const uint8_t x) {
  uint8_t r = 0;
  for(uint i = 0; i < 8; ++i) {
    r ^= (x >> i);   // '^'
  }
  return (r & 0x1);
}
// calculate one of the eight output bytes
// given one byte x of X
inline uint8_t
affine_byte(const uint64_ut A,
            const uint8_t   x,
            const uint8_t   b) {
  uint8_t r = 0;
  for(uint i = 0; i < 8; ++i) {
    r |= (parity(A._ui8[7 - i] & x) << i);  // '&'
  }
  r ^= b;
  return r;
}
// compared to original intrinsic
// the first two arguments are reversed here
inline uint64_ut
gfni_mul_emu(const uint64_ut A,
             const uint64_ut X,
             const uint8_t   b) {
  uint64_ut R;
  for(uint i = 0; i < 8; ++i) {
    R._ui8[i] = affine_byte(A, X._ui8[i], b);
  }
  return R;
}
```

Note that the first two arguments of `gfni_mul_emu` are reversed compared to the `gf2p8affine` intrinsics. Otherwise, even if you don't have a machine with AVX512, you can do experiments with the intrinsic by simply using the above code. However, it might be a little slow.

This code maybe quite easy to understand but what it really does may not be clear at a first sight. Thus, the rest of this short note will provide

the reader with an intuition of this operation. Just a few remarks on the code so that you can relate it to the GF(2) operations:

- `parity`(x) returns 1 if the number of bits set in x is odd, 0 otherwise
- Observe the *bit xor* operation in `parity`, i.e., a $+$, which means that `parity` calculates a sum modulo 2.
- Oberve the *bit and* operation in `affine_byte`, i.e., a multiplication in GF(2).

In order to start out, we will ignore the argument b by setting it to zero. The remaining arguments A and X correspond to 8x8 bit matrices. and the operation is a kind of a matrix multiplication whose precise semantics will become clear throughout the rest of this note. In order to avoid the long name of the intrinsic, we define

$$A \circledast X := \texttt{gfni\_mul\_emu}(A, X, 0) \tag{1}$$

Besides clarifying the semantics of $\circledast$, we will show how it can be used to implement a pletora of bit manipulating and matrix operations. We further provide a simple guide on designing the arguments in order to implement new operations.[2]

We start out by looking at matrices and operations on matrices as well as different ways to store 8x8 bit matrices in 64 bit unsigned integers (Sec. 4). This prepares us to express the semantics of $A \circledast X$ in terms of matrix operations (Sec. 5).

3 Matrix Storage Layout

The Intel GFNI intrinsics gf2p8affine_epi64_epi8 implement vectorized versions of a binary operation taking two `uint64_t`, i.e., two times 64 bits[3]. Each `uint64_t` is interpreted as an 8x8 bit matrix. Consider again

```
union uint64_ut {
  uint64_t _ui64;
  uint8_t  _ui8[8];
};
```

Then, we have a couple of choices of what to store in every of the eight `_ui8` elements:

[2] Code: [`src/GFNI`]

[3] We intentially ignore masked versions and the extra input byte.

1. We could store either a row (row major) or a column (column major) of a 8x8 bit matrix in one of the bytex in $\_ui8[i]$.

2. We could store the first row (or column) in $\_ui8[0]$ or in $\_ui8[7]$.

3. We could use the least significant bit (lsb) of $\_ui8[i]$ or the most significant bit of $\_ui8[i]$ to store the first entry of or a row (or column).

We use the following abbreviations to denote these choices:

rm/cm row major vs. column major

i/r store first row (or column) in _ui8[0] (i) or _ui8[7] (r)

i/r store the first bit of a row (or column) in the lsb (i) or msb (r)

The result are 8 different storage layouts:

row major	column major
`rm_ii`, `rm_ri`, `rm_ir`, `rm_rr`	`cm_ii`, `cm_ri`, `cm_ir`, `cm_rr`

where `rm`, `cm` denote row or column major, the first `i` or `r` denotes the byte order and the second `i` or `r` denotes the bit order.

To clarify this even further, assume we have 8 classes to represent an 8x8 bit matrix, one for each storage layout. Each class has a member `_data` of type `uint64_ut`. We give the essence of accessors like

uint `operator()(const uint i, const uint j)`)

which retrieve a bit (i, j) where i denotes the row index and j denotes the column index of the represented 8x8 bit matrix:

```
Bm8x8_rm_ii(i,j): (_data._ui[i]   >>      j)  & 0x1
Bm8x8_rm_ri(i,j): (_data._ui[7-i] >>      j)  & 0x1
Bm8x8_rm_ir(i,j): (_data._ui[i]   >> (7 - j)) & 0x1
Bm8x8_rm_rr(i,j): (_data._ui[7-i] >> (7 - j)) & 0x1

Bm8x8_cm_ii(i,j): (_data._ui[j]   >>      i)  & 0x1
Bm8x8_cm_ri(i,j): (_data._ui[7-j] >>      i)  & 0x1
Bm8x8_cm_ir(i,j): (_data._ui[j]   >> (7 - i)) & 0x1
Bm8x8_cm_rr(i,j): (_data._ui[7-j] >> (7 - i)) & 0x1
```

4 Matrix Transformations and Layout Reinterpretation

4.1 Matrix Transformations

Let A be a $m \times n$ matrix. Then, besides the well-known transposition transformation A^t, we define the *reverse row* A^r, the *reverse column* A^c and the *perpose* A^p transformation:

$$\begin{aligned} A^t(i,j) &= A(j,i) && (2)\\ A^r(i,j) &= A(m-1-i,j) && (3)\\ A^c(i,j) &= A(i,n-1-j) && (4)\\ A^p(i,j) &= A(m-1-i,n-1-j) && (5) \end{aligned}$$

where we start indexing with 0. Fig. 1 illustrates these transformations for some example matrix A. A matrix A with $A = A^t$ is called *symmetric* and with $A = A^p$ is called *persymmetric*.

Applying all of these transformations as often as we wish, we can generate for any matrix A only eight possibly different matrices:

$$A^t, A^r, A^c, A^p, (A^t)^r, (A^t)^c, (A^t)^p.$$

Let I be the $n \times n$ identity matrix, i.e.,

$$I = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ & & \dots & & \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}$$

and

$$R = \begin{pmatrix} 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & \dots & 1 & 0 \\ & & \dots & & \\ 0 & 1 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \end{pmatrix}$$

the matrix with only ones on the counter diagonal, i.e., $R = I^r = I^c$. Obviously,

$$\begin{aligned} II &= I \\ IR &= R \\ RI &= R \\ RR &= I \end{aligned}$$

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad A^r = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$A^c = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \quad A^p = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$A^t = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Figure 1: Examples for Matrix Transformations

We further have that

$$\begin{aligned} I^t &= I \\ I^r &= R \\ I^c &= R \\ I^p &= I \\ R^t &= R \\ R^r &= I \\ R^c &= I \\ R^p &= R \end{aligned}$$

Note that R is a permutation matrix[4] that reverses either rows or columns depending on its multiplication from the left or right. Thus, some transformations can be expressioned by multiplication with R:

$$\begin{aligned} A^r &= RA \\ A^c &= AR \\ A^p &= RAR \end{aligned}$$

When looking at two applications of any of our four transformations, we see

[4] A permutation matrix is a matrix with exactly one 1 in every row and every column.

that

$$\begin{aligned}
(A^t)^t &= A \\
(A^t)^c &= (A^r)^t \\
(A^t)^r &= (A^c)^t \\
(A^t)^p &= (A^p)^t \\
(A^r)^t &= (A^t)^c \\
(A^r)^r &= A \\
(A^r)^c &= A^p \\
(A^r)^p &= A^c \\
(A^c)^t &= (A^t)^r \\
(A^c)^r &= A^p \\
(A^c)^c &= A \\
(A^c)^p &= A^r \\
(A^p)^t &= (A^t)^p \\
(A^p)^r &= A^c \\
(A^p)^c &= A^r \\
(A^p)^p &= A
\end{aligned}$$

Thus

$$A^p = (A^r)^c = (A^c)^r = RAR.$$

When applying one of our four transformations to the product of two matrices, we get that

$$\begin{aligned}
(AB)^t &= B^t A^t \\
(AB)^p &= A^p B^p \\
(AB)^r &= A^r B \\
(AB)^c &= AB^c
\end{aligned}$$

4.2 Matrix Layout Reinterpretation

A conversion of some matrix layout into another one can be expressed by our matrix transformations. Table 1 has for each input layout a row which tells us to which transformation(s) a reinterpretation in any layout corresponds (a reinterpretation means `B._data = A._data` for two matrices A and B in (possibly) different layouts.).

A in	rm_ii	rm_ir	rm_ri	rm_rr	cm_ii	cm_ir	cm_ri	cm_rr
rm_ii	A	A^c	A^r	A^p	A^t	$(A^t)^c$	$(A^t)^r$	$(A^t)^p$
rm_ir	A^c	A	A^p	A^r	$(A^t)^c$	A^t	$(A^t)^p$	$(A^t)^r$
rm_ri	A^r	A^p	A	A^c	$(A^t)^r$	$(A^t)^p$	A^t	$(A^t)^c$
rm_rr	A^p	A^r	A^c	A	$(A^t)^p$	$(A^t)^r$	$(A^t)^c$	A^t
cm_ii	A^t	$(A^t)^r$	$(A^t)^c$	$(A^t)^p$	A	A^r	A^c	A^p
cm_ir	$(A^t)^r$	A^t	$(A^t)^p$	$(A^t)^c$	A^r	A	A^p	A^c
cm_ri	$(A^t)^c$	$(A^t)^p$	A^t	$(A^t)^r$	A^c	A^p	A	A^r
cm_rr	$(A^t)^p$	$(A^t)^c$	$(A^t)^r$	A^t	A^p	A^c	A^r	A

Table 1: Relating storage layout reinterpretation to matrix transformations

5 The Semantics of $\circledast$

We start with the code of multiplying to 8x8 matrices A and X with some number type `Tmatrix::item_type` in their elements:

```
template<typename Tmatrix>
Tmatrix
matrix_mul(const Tmatrix& A, const Tmatrix& X) {
  Tmatrix lRes;
  for(uint i = 0; i < 8; ++i) {
    for(uint j = 0; j < 8; ++j) {
      Tmatrix::item_type lSum = 0;
      for(uint k = 0; k < 8; ++k) {
        lSum += A(i,k) * X(k,j);
      }
      lRes(i,j) = lSum;
    }
  }
  return lRes;
}
```

We first rewrite this code such that it becomes matrix multiplication in GF(2). Then, we rewrite it further and fill in some missing details until we reach a form of the code that is equivalent to the code we gave for `gfni_mu_emu`. This allows us to see that `gfni_mu_emu` indeed performs multiplication of two GF(2) matrices under the conditions indicated by the filled-in details.

For GF(2), + becomes *bit xor* and $*$ becomes *bit and* resulting in

```
template<typename Tbm>
Tbm
bm_mul(const Tbm& A, const Tbm& X) {
  Tbm lRes;
  for(uint i = 0; i < 8; ++i) {
    for(uint j = 0; j < 8; ++j) {
      uint lSum = 0;
      for(uint k = 0; k < 8; ++k) {
        lSum ^= A(i,k) & X(k,j);
      }
      lRes.set(i, j, lSum);
    }
  }
  return lRes;
}
```

We rewrite this code using `uint64_ut` to represent the matrices A and X. Thereby, we assume that A is represented in row major format using layout `rm_ii` but X is represented in column major using layout `cm_ii`. Then, the above code is equivalent to

```
uint64_ut
bm_mul(const uint64_ut A, const uint64_ut X) {
  uint64_ut lRes;
  for(uint i = 0; i < 8; ++i) {
    for(uint j = 0; j < 8; ++j) {
      uint lSum = 0;
      for(uint k = 0; k < 8; ++k) {
        lSum ^= (A._ui8[i] >> k) & (X._ui8[j] >> k) & 0x1
      }
      lRes.set(i, j, lSum);
    }
  }
  return lRes;
}
```

Now, the innermost loop over k does the same thing for every bit: *bit anding* it. We can do so in one operation resulting in

```
uint64_ut
bm_mul(const uint64_ut A, const uint64_ut X) {
```

```
  uint64_ut lRes;
  for(uint i = 0; i < 8; ++i) {
    for(uint j = 0; j < 8; ++j) {
      uint8_t a_and_b = A._ui8[i] & X._ui8[j];
      uint lSum = 0;
      for(uint k = 0; k < 8; ++k) {
        lSum ^= (a_and_b >> k) & 0x1
      }
      lRes.set(i, j, lSum);
    }
  }
  return lRes;
}
```

The innermost loop over k implements the `parity` function. Thus the above code is equivalent to

```
uint64_ut
bm_mul(const uint64_ut A, const uint64_ut X) {
  uint64_ut lRes;
  for(uint i = 0; i < 8; ++i) {
    for(uint j = 0; j < 8; ++j) {
      uint8_t a_and_b = A._ui8[i] & X._ui8[j];
      uint lSum = parity(a_and_b);
      lRes.set(i, j, lSum);
    }
  }
  return lRes;
}
```

We haven't specified the result layout yet. Let us assume that `lRes` is stored column major using storage layout `cm_ir`. Thus, we find the matrix element `lRes`(i, j) in `(_ui8[j] >> (7 - i)) 0x1`. Then, the above is equivalent to

```
uint64_ut
bm_mul(const uint64_ut A, const uint64_ut X) {
  uint64_ut lRes;
  for(uint i = 0; i < 8; ++i) {
    for(uint j = 0; j < 8; ++j) {
      uint8_t a_and_b = A._ui8[i] & X._ui8[j];
      uint lSum = parity(a_and_b);
```

```
      lRes._ui8[j] |= lSum << (7 - i);
    }
  }
  return lRes._data._ui64;
}
```

We observe that within the loop over j all j bytes of `lRes` are touched at one position. The code becomes better, if we calculate the `lRes` byte-wise. Thus, we interchange the loops:

```
uint64_ut
bm_mul(const uint64_ut A, const uint64_ut X) {
  uint64_ut lRes;
  for(uint j = 0; j < 8; ++j) {
    for(uint i = 0; i < 8; ++i) {
      uint8_t a_and_b = A._ui8[i] & X._ui8[j];
      uint lSum = parity(a_and_b);
      lRes._ui8[j] |= lSum << (7 - i);
    }
  }
  return lRes._data._ui64;
}
```

and rename i to j and j to i:

```
uint64_ut
bm_mul(const uint64_ut A, const uint64_ut X) {
  uint64_ut lRes;
  for(uint i = 0; i < 8; ++i) {
    for(uint j = 0; j < 8; ++j) {
      uint8_t a_and_b = A._ui8[j] & X._ui8[i];
      uint lSum = parity(a_and_b);
      lRes._ui8[i] |= lSum << (7 - j);
    }
  }
  return lRes._data._ui64;
}
```

Now, `X._ui[i]` is constant within the inner loop and we can introduce a variable x for it. Further, we introduce a variable r to hold the calculations of the inner loop. Last but not least, we remove the variables `a_and_b` and `lSum`. This gives us

```
uint64_ut
bm_mul(const uint64_ut A, const uint64_ut X) {
  uint64_ut lRes;
  for(uint i = 0; i < 8; ++i) {
    const uint8_t x = X._ui8[i];
    uint8_t r = 0;
    for(uint j = 0; j < 8; ++j) {
      r |= parity(A._ui8[j] & x) << (7 - j);
    }
    lRes._ui8[i] = r;
  }
  return lRes._data._ui64;
}
```

Now, the line

```
r |= parity(A._ui8[j] & x) << (7 - j);
```

is equivalent to

```
r |= parity(A._ui8[7 - j] & x) << j;
```

since it exchanges only the order in which the bits are calculated. Doing this exchange results in

```
uint64_ut
bm_mul(const uint64_ut A, const uint64_ut X) {
  uint64_ut lRes;
  for(uint i = 0; i < 8; ++i) {
    const uint8_t x = X._ui8[i];
    uint8_t r = 0;
    for(uint j = 0; j < 8; ++j) {
      r |= parity(A._ui8[7 - j] & x) << j;
    }
    lRes._ui8[i] = r;
  }
  return lRes._data._ui64;
}
```

Finally, it is easy to see that this code is equivalent to the code of `gfni_mul_emu` in case $b = 0$. Thus, we have shown that `gfni_mul_emu`(A, X, 0) calculates AX in case A is given in `rm_ii`, X is given in `cm_ii`, and the result is

A	B		$A \circledast B$
`rm_ii`	`cm_ii`	$\rightarrow$	`cm_ir`
`rm_ii`	`cm_ri`	$\rightarrow$	`cm_rr`
`rm_ir`	`cm_ir`	$\rightarrow$	`cm_ir`
`rm_ir`	`cm_rr`	$\rightarrow$	`cm_rr`
`rm_ri`	`cm_ii`	$\rightarrow$	`cm_ii`
`rm_ri`	`cm_ri`	$\rightarrow$	`cm_ri`
`rm_rr`	`cm_ir`	$\rightarrow$	`cm_ii`
`rm_rr`	`cm_rr`	$\rightarrow$	`cm_ri`

Table 2: Storage layouts for which $A \circledast B = AB$

interpreted as `cm_ir`. In order to explicitly associate storage layouts and matrices, we add the storage layouts as subscripts to the matrices. Remembering that $A \circledast X$ abbreviated `gfni_mul_emu`$(A, X, 0)$, we can specify the semantics of $\circledast$ as

$$A_{\texttt{rm\_ii}} \circledast X_{\texttt{cm\_ii}} = (A_{\texttt{rm\_ii}} X_{\texttt{cm\_ii}})_{\texttt{cm\_ir}} \tag{6}$$

There are more combinations of layouts such that $A \circledast B = AB$. Table 2 contains a complete list where this is the case.

What happens if we have or expect other storage layouts? My personal favorite is `rm_ii` for *all* matrices A, B and $A \circledast B$. What then is the semantics of $A \circledast B$? To clarify this, we consider the first line and observe that for A we are already fine. To express some equalities, we add the storage layout of a matrix as its subscript. Then, the first line of Table 2 reads

$$A_{\texttt{rm\_ii}} \circledast B_{\texttt{cm\_ii}} = (A_{\texttt{rm\_ii}} B_{\texttt{cm\_ii}})_{\texttt{cm\_ir}}$$

Table 1 shows that reinterpreting a matrix $B_{\texttt{rm\_ii}}$ as a matrix $B_{\texttt{cm\_ii}}$ corresponds to transposition. Thus, we can derive

$$A_{\texttt{rm\_ii}} \circledast B_{\texttt{rm\_ii}} = (A_{\texttt{rm\_ii}} (B_{\texttt{rm\_ii}})^t)_{\texttt{cm\_ir}}$$

Finally, if we are given a matrix $X_{\texttt{cm\_ir}}$ and reinterpret it as a matrix in storage layout $X_{\texttt{rm\_ii}}$, Table 1 shows us that this corresponds to $((X_{\texttt{cm\_ir}})^t)^c$. Thus

$$A_{\texttt{rm\_ii}} \circledast B_{\texttt{rm\_ii}} = (((A_{\texttt{rm\_ii}} (B_{\texttt{rm\_ii}})^t)^t)^c)_{\texttt{rm\_ii}} \tag{7}$$

which we can rewrite to

$$A_{\texttt{rm\_ii}} \circledast B_{\texttt{rm\_ii}} = (((A_{\texttt{rm\_ii}} (B_{\texttt{rm\_ii}})^t)^r)^t)_{\texttt{rm\_ii}} \tag{8}$$

or, dropping the storage layout and assuming that all matrices are given in storage layout `rm_ii`, we have that

$$A \circledast B = ((AB^t)^r)^t = ((AB^t)^t)^c = (BA^t)^c \tag{9}$$

where there are some possibilities to rewrite the expression on the right-hand side.

Let us see what happens if one of A and B is either I or R:

$$\begin{aligned} I \circledast A &= A^c && (10)\\ A \circledast I &= (A^t)^c && (11)\\ R \circledast A &= A && (12)\\ A \circledast R &= (A^t)^p = (A^p)^t && (13) \end{aligned}$$

Using Eq. 9, these are pretty easy to derive:

$$\begin{aligned} I \circledast A &= (AI^t)^c = A^c\\ A \circledast I &= (IA^t)^c = (A^t)^c\\ R \circledast A &= (AR^t)^c = (AR)^c = (A^c)^c = A\\ A \circledast R &= (RA^t)^c = ((A^t)^r)^c = (A^t)^p \end{aligned}$$

6 Manipulating Bits within Bytes (AD-Opcodes)

6.1 General Principles

The goal of this section is to design matrices that perform bit manipulation operations (by specifying some matrix D, called opcode) on every of the eight bytes (of some argument X) in a uint64_t using gf2p8affine_epi64_-epi8.

For matrices X and D in `rm_ii` format, we have that the Intel GFNI intrinsic gf2p8affine_epi64_epi8 $(X, D, 0)$ calculates

$$D \circledast X := ((DX^t)^r)^t \tag{14}$$

(Remember the reversal of the first two arguments in gf2p8affine_epi64_epi8 $(X, D, 0)$.)

For the purpose of this section, we rewrite this to

$$\begin{aligned} \hat{D} \circledast X &= ((\hat{D}X^t)^r)^t\\ &= (\hat{D}^r X^t)^t\\ &= X(\hat{D}^r)^t\\ &= XD \end{aligned}$$

where

$$D = (\hat{D}^r)^t. \tag{15}$$

If D is a permutation matrix, then XD permutes the columns in X. Since X is in `rm_ii` format, each row of X corresponds to a single byte. Thus, XD permutes the bits in every byte of X (in the same way).

Let D be a permutation matrix, i.e., a matrix with exactly one 1 in every row and in every column. Let us denote by e_i the i-th unit column vector which contains a 1 is row i and is 0 everywhere else. Assume we have a byte x

$$x = [\text{lsb}]b_0, b_1, \ldots, b_7[\text{msb}]$$

and we want to permute it via a permutation $\pi : [0, 7] \to [0, 7]$. That is, we want to get

$$[\text{lsb}]b_{\pi(0)}, b_{\pi(1)}, \ldots, b_{\pi(7)}[\text{msb}]$$

then this can be achieved by defining the permutation matrix D_π as

$$D_\pi = \begin{pmatrix} e_{\pi(0)} & e_{\pi(1)} & \ldots & e_{\pi(7)} \end{pmatrix} \tag{16}$$

Many operations can be expressed by some form of permutation matrix, sometimes with additional modifications to clear certain bits.

The following subsections contain examples of how to design different D for various bit manipulations within a byte. Since $\hat{D}$ and D are central to this, we give them names. We call $\hat{D}$ the *real* opcode and D the `rm_ii`-opcode. To get the real opcode, we must calculate $\hat{D}$ from a given D. Since $D = (\hat{D}^r)^t$, we can solve for $\hat{D}$ resulting in

$$\hat{D} = ((D)^t)^r \tag{17}$$

Further, $\hat{D}$ must be given as a `uint64_t`. However, we will represent $\hat{D}$ always as a matrix. For convienience, Table 3 gives real opcodes in hex for some of the bit manipulation operations we consider later on.

Assume we have designed a series of `rm_ii`-opcode for several bit manipulations D_1, D_1, ..., D_k and we want to execute them in sequence one after the other as in

$$(\ldots((XD_1)D_1)\ldots)D_k$$

then associativity of matrix multiplication tells us that this is equal to

$$X(D_0D_1 \ldots D_k)$$

Thus, we can collapse all the bit manipulations into one operation with the product of the `rm_ii`-opcode, which of course has to be translated to the

real opcode. After we have done so, we can execute k bit manipulations on 64 bytes (using 512 bit simd registers) in *one* instruction!

The conversion of the `rm_ii`-opcode to the real opcode is performed using the simple procedure (excuse the German function name)

```
using bm8x8_rm_ii_vt = std::vector<Bm8x8_rm_ii>;
uint64_t
hex_hex(const bm8x8_rm_ii_vt& v) {
  Bm8x8_rm_ii lRes = bm_identity<Bm8x8_rm_ii>();
  for(uint i = 0; i < v.size(); ++i) {
    lRes = bm_mul(lRes, v[i]);
  }
  lRes = lRes.transpose().reverse_row();
  return lRes.data();
}
```

by calling

$$\texttt{hex\_hex}(\{D_0, D_1, ..., D_k\}).$$

In this note, we assume that we have fixed series of operations we want to perform. To dynamically select the operations to be executed, a `shuffle` operation can be used.

Another sometimes useful way to combine `rm_ii`-opcodes exploites the distribution law for matrices:

$$AB_1 + AB_2 = A(B_1 + B_2). \tag{18}$$

For our applications, it is (mostly) important the two matrices B_1 and B_2 are *disjoint* which is defined such that there exist no i, j with $B_1(i, j) = B_2(i, j) = 1$.

6.2 Example[rev8]

In this subsection, we want to reverse the bits in every byte, that is,

$$[lsb]b_0, \ldots, b_7[msb]$$

becomes

$$[lsb]b_7, \ldots, b_0[msb].$$

It is clear that $D = R$ does the job. In order to apply $\circledast$, we must calculate $\hat{D}$ from $D = (\hat{D}^r)^t$ using

$$\hat{D} = ((D)^t)^r. \tag{19}$$

```
constexpr uint64_t gIdentityLE = 0x80'40'20'10'08'04'02'01ul;
constexpr uint64_t gReverseLE  = 0x01'02'04'08'10'20'40'80ul;
constexpr uint64_t Dmsk0xAA    = 0x01'00'04'00'10'00'40'00ul;
constexpr uint64_t Dmsk0x55    = 0x00'02'00'08'00'20'00'80ul;
constexpr uint64_t Drotl8_1    = 0x80'01'02'04'08'10'20'40ul;
constexpr uint64_t Drotl8_2    = 0x40'80'01'02'04'08'10'20ul;
constexpr uint64_t Drotl8_3    = 0x20'40'80'01'02'04'08'10ul;
constexpr uint64_t Drotl8_4    = 0x10'20'40'80'01'02'04'08ul;
constexpr uint64_t Drotl8_5    = 0x08'10'20'40'80'01'02'04ul;
constexpr uint64_t Drotl8_6    = 0x04'08'10'20'40'80'01'02ul;
constexpr uint64_t Drotl8_7    = 0x02'04'08'10'20'40'80'01ul;
constexpr uint64_t Drotr8_1    = 0x02'04'08'10'20'40'80'01ul;
constexpr uint64_t Drotr8_2    = 0x04'08'10'20'40'80'01'02ul;
constexpr uint64_t Drotr8_3    = 0x08'10'20'40'80'01'02'04ul;
constexpr uint64_t Drotr8_4    = 0x10'20'40'80'01'02'04'08ul;
constexpr uint64_t Drotr8_5    = 0x20'40'80'01'02'04'08'10ul;
constexpr uint64_t Drotr8_6    = 0x40'80'01'02'04'08'10'20ul;
constexpr uint64_t Drotr8_7    = 0x80'01'02'04'08'10'20'40ul;
constexpr uint64_t Dshiftl8_1  = 0x00'01'02'04'08'10'20'40ul;
constexpr uint64_t Dshiftl8_2  = 0x00'00'01'02'04'08'10'20ul;
constexpr uint64_t Dshiftl8_3  = 0x00'00'00'01'02'04'08'10ul;
constexpr uint64_t Dshiftl8_4  = 0x00'00'00'00'01'02'04'08ul;
constexpr uint64_t Dshiftl8_5  = 0x00'00'00'00'00'01'02'04ul;
constexpr uint64_t Dshiftl8_6  = 0x00'00'00'00'00'00'01'02ul;
constexpr uint64_t Dshiftl8_7  = 0x00'00'00'00'00'00'00'01ul;
constexpr uint64_t Dshiftr8_1  = 0x02'04'08'10'20'40'80'00ul;
constexpr uint64_t Dshiftr8_2  = 0x04'08'10'20'40'80'00'00ul;
constexpr uint64_t Dshiftr8_3  = 0x08'10'20'40'80'00'00'00ul;
constexpr uint64_t Dshiftr8_4  = 0x10'20'40'80'00'00'00'00ul;
constexpr uint64_t Dshiftr8_5  = 0x20'40'80'00'00'00'00'00ul;
constexpr uint64_t Dshiftr8_6  = 0x40'80'00'00'00'00'00'00ul;
constexpr uint64_t Dshiftr8_7  = 0x80'00'00'00'00'00'00'00ul;
constexpr uint64_t Dsepoddevn  = 0x01'04'10'40'02'08'20'80ul;
constexpr uint64_t Dsr1x1      = 0x03'00'0c'00'30'00'c0'00ul;
```

Table 3: Real opcodes for bit manipulationg operations

$$X = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad I \circledast X = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2: Example `rev8`

Thus, for our special case $D = I$, we get that

$$\hat{D} = ((D)^t)^r = (R^t)^r = I.$$

Thus, let us define $\hat{D}_{rev} := I$. Then, $I \circledast X$ reverses the bits in every byte and we can conclude that

$$X^c = \hat{D}_{\mathrm{rev}} \circledast X = I \circledast X \tag{20}$$

Figure 2 contains an example.

6.3 Example[`rotl8/rotr8`]

We take a look at the cyclic shift (also called rotate) to the left or right. We start with finding $D_{\mathrm{rot}l}()$ for rotating a byte one position to the left:

$$\begin{array}{ll} [msb]b_7, b_6, \ldots b_1, b_0[lsb] & \rightarrow \\ [msb]b_6, b_5, \ldots b_0, b_7[lsb] & \end{array}$$

Then the lsb bit in the result becomes the msb of the input, the second bit in the result becomes the first bit of the input and so on. Figure 3 contains examples for $D_{\mathrm{rot}l}(i)$ for $i = 1, 2, 3$ and for $\hat{D}_{\mathrm{rot}l}(i)$. Rotating to the right is similar. The matrices for virtual opcodes $D_{\mathrm{rot}r}(i)$ for $i = 1, 2, 3, 4$ are shown in Figure 4.

6.4 Example[`shiftl8/shiftr8`]

The shift operation is very similar to the rotate operation except that we fill in zeros from the left if we shift left and fill in zeros from the right if we shift right. Clearing a bit (i.e., setting is to zero) is achieved by setting the according column in the `rm_ii`-opcode to zero for all its entries. Figure 5 contains three examples for the shifting to the left ($D_{shiftl}(i)$) and shifting to the right ($D_{shiftr}(i)$)

$$D_{rotl}(1) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{D}_{rotl}(1) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$D_{rotl}(2) = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{D}_{rotl}(2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$D_{rotl}(3) = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{D}_{rotl}(3) = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 3: Examples for real and `rm_ii` opcodes for `rotl8`

$$D_{rotr}(1) = \begin{pmatrix} 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0 \end{pmatrix} \quad D_{rotr}(2) = \begin{pmatrix} 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0 \end{pmatrix}$$

$$D_{rotr}(3) = \begin{pmatrix} 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0 \end{pmatrix} \quad D_{rotr}(4) = \begin{pmatrix} 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0 \end{pmatrix}$$

Figure 4: Examples for `rm_ii` opcodes $D_{rotr}(i)$

6.5 Example[msk8]

Task: given an 8-bit mask m, calculate $x\&m$ via $\circledast$. Clearly $D_{\mathrm{msk}}(m) := \mathrm{diag}(m)$ does the job, where $\mathrm{diag}(m)$ is 0 outside the diagonal and $\mathrm{diag}(i,i) = m[i]$. Figure 6 contains two examples.

6.6 Example[bextr8(p,k)]

The operation `bextr8`(x,p,k) selects k bits starting at position p and puts these bits at the beginning of the byte. All other $8-k$ bits are set to zero For example, if $x = [msb]00010100[lsb]$ then

$$\texttt{bextr8}(x,2,3) = [msb]00000101[lsb].$$

This can be implemented by two shift operations:

```
(x << (8 - (p + k))) >> (8 - k)
```

The first shift sets the bits following the range $[p : p+k-1]$ to zero, the second shift moves the pattern of interest to the right and draws in zeros.

From the above it is clear that we can use

$$D_{\mathrm{ext}}(p,k) = D_{shiftl}(8-(p+k))D_{\mathrm{s}hiftr}(8-k)$$

to implement a `bextr8`(x,p,k) on every row (byte) of a `rm_ii` matrix X.

$$D_{shiftl}(1) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad D_{shiftr}(1) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$D_{shiftl}(2) = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad D_{shiftr}(2) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

$$D_{shiftl}(3) = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad D_{shiftr}(3) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 5: `rm_ii`-opcodes for shift operations

$$D_{\mathrm{msk}}(0x55) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \hat{D}_{\mathrm{msk}}(0x55) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_{\mathrm{msk}}(0xAA) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \hat{D}_{\mathrm{msk}}(0xAA) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 6: Example opcodes for masking with $D_{\mathrm{msk}}(m) = \mathrm{diag}(m)$

For $p = 2$ and $k = 3$, we get that

$$D_{shiftl}(3) = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad D_{shiftr}(5) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$D_{\mathrm{ext}}(2,3) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

As an example consider

$$\begin{pmatrix} 0&1&1&0&0&0&0&1\\ 1&1&0&0&0&1&1&1\\ 1&0&1&1&0&0&0&1\\ 0&1&1&1&1&0&0&0\\ 1&0&1&1&1&1&1&0\\ 1&1&0&1&1&1&1&0\\ 0&0&0&0&1&0&1&1\\ 1&0&1&0&0&0&0&0 \end{pmatrix} \begin{pmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{pmatrix} = \begin{pmatrix} 1&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 1&1&0&0&0&0&0&0\\ 1&1&1&0&0&0&0&0\\ 1&1&1&0&0&0&0&0\\ 0&1&1&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 1&0&0&0&0&0&0&0 \end{pmatrix}$$

6.7 Example [pext8, pdep8]

The `pext8` and `pdep8` instructions take two arguments: a byte x on which the operation is to be performed and a mask m Let $0 \leq i_1 < \ldots < i_k < 8$ be the bit positions where m is set to 1. Then $b[i_k], \ldots, b[i_1]$ are the bits of m which are set to 1. All other bits are 0. The `pext8` instruction collects the bits of x which have a corresponding 1 in the mask m at the lsb end:

$$\texttt{pext8}(x, m) = [msb]0 \ldots 0x[i_k] \ldots x[i_1][lsb]$$

The instruction `pdep8` does the opposite. It deposits the k bits at the lsb end of x to the positions indicated by the 1s in the mask m:

$$\texttt{pdep8}(x, m) = [msb]0^{8-i_k-1}x[k]0^{i_k-i_{k-1}-1}x[k-1]0^{i_{k-1}-i^{k-2}-1}x[k-2] \ldots x[2]0^{i_2-i_1-1}x[1]0^{i_1}[lsb]$$

where 0^i denotes i zero bits. For example:

$$\begin{aligned} \texttt{pext8}(0'1'000'10'0, 01000110) &= 00000'110' \\ \texttt{pdep8}(00000'110, 01001010) &= 0'1'00'1'0'0'0 \end{aligned}$$

Thus, it should be clear that the two operations are inverse to each other if they use the same mask m and if we only look at the bits set in m. More specifically, $\texttt{pdep8}(\texttt{pext8}(x, m), m) = x\&m$.

There are instruction for `pext` and `pdep` for 32 bits and 64 bits. They are very useful in many applications, for example enumerating subsets (which itself is useful in, e.g., plan generation via dynamic programming) or working with Z-curves. Here, we show how `pext8` and `pdep8` can be implemented using $\circledast$.

To start, we first consider a mask m with bits set only at positions $i_1 < \ldots < i_k$. The first bit in the result of $\texttt{pext8}(x, m)$ is bit $x[i_1]$. The second bit in the result is $x[i_2]$ and so forth. Thus, we can define

$$D_{\text{pext}}(m) = \begin{pmatrix} e_{i_1} & e_{i_2} & \ldots & e_{i_k} & 0 \ldots & 0 \end{pmatrix}$$

where e_n is the n-th unit (column) vector containing only a single 1 at position n. For $m = [lsb]01100101[msb]$, we get

$$D_{\text{pext}}(m) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

For `pdep8` we have that

$$D_{\text{pdep}}(m) = D_{\text{pext}}(m)^t$$

For the example mask $m = [lsb]01100101[msb]$, we get that

$$D_{\text{pdep}}(m) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Further,

$$\begin{aligned} D_{\text{pext}}(m)D_{\text{pdep}}(m) &= D_{\text{msk}}(m) \\ D_{\text{pdep}}(m)D_{\text{pext}}(m) &= \text{diag}(1, \ldots 1, 0, \ldots 0) \end{aligned}$$

where the number of 1s on the right-hand side of the second equation equals the number of bits set in m.

6.8 Example[sepoddevn8]

The operation `sepoddevn8` collects the odd bits into the lower nibble and the even bits into the upper nibble. Thus, for a single byte x, this operation can be defined as

```
sepoddevn8(x) := pext8(x, 0x55) | (pext8(x, 0xAA) << 4)
```

In matrix form, we get for an input matrix X

$$\begin{aligned} & XD_{\text{pext}}(0x55) + ((XD_{\text{pext}}(0xAA))D_{shiftl}(4)) \\ = \; & XD_{\text{pext}}(0x55) + (X(D_{\text{pext}}(0xAA)D_{shiftl}(4))) \\ = \; & XD_{\text{pext}}(0x55) + (X(D_{\text{pext}}(0xAA)D_{shiftl}(4))) \\ = \; & X(D_{\text{pext}}(0x55) + (D_{\text{pext}}(0xAA)D_{shiftl}(4))) \end{aligned}$$

Thus, we can define

$$D_{\text{sepoddevn}} := D_{\text{pext}}(0x55) + D_{\text{pext}}(0xAA)D_{shiftl}(4) \tag{21}$$

where the two matrices which are added are disjoint.

The following illustrates this process:

$$D_{\text{pext}}(0xAA) = \begin{pmatrix} 0&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \end{pmatrix} \qquad D_{shiftl}(4) = \begin{pmatrix} 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$D_{\text{pext}}(0xAA)D_{shiftl}(4) = \begin{pmatrix} 0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1 \end{pmatrix} \qquad D_{\text{pext}}(0x55) = \begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$D_{\text{sepoddevn}} = \begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&1 \end{pmatrix}$$

As an example consider

$$\begin{pmatrix} 0&1&1&0&0&1&0&1\\ 0&1&1&1&0&1&0&1\\ 0&1&1&0&1&1&1&1\\ 0&1&1&0&1&1&1&1\\ 0&0&1&1&1&0&0&0\\ 1&0&0&1&1&0&0&0\\ 1&0&1&1&0&1&1&0\\ 1&0&0&1&0&0&1&1 \end{pmatrix} \begin{pmatrix} 1&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0&1 \end{pmatrix} = \begin{pmatrix} 0&1&0&0&1&0&1&1\\ 0&1&0&0&1&1&1&1\\ 0&1&1&1&1&0&1&1\\ 0&1&1&1&1&0&1&1\\ 0&1&1&0&0&1&0&0\\ 1&0&1&0&0&1&0&0\\ 1&1&0&1&0&1&1&0\\ 1&0&0&1&0&1&0&1 \end{pmatrix}$$

6.9 Example[sr1x1]

So far, we considered only cases with at most one bit set in every column of the `rm_ii`-opcode. Now, we consider an example with two bits set in some columns.

A typical pattern I often write is

```
((x >> 1) ^ x) & 0x55
```

For two bits b_1 and b_2, we have that

```
b_1 ^ b2 = (b_1 & 1) ^ (b_2 & 1) = (b_1, b_2) (1,1)^t
```

where we interpret $(1,1)$ as a row vector that thus needs to be transposed. From this, it might be clear to some readers that the `rm_ii`-opcode in Fig. 7 does the job. A more formal way to derive D_{sr1x1} uses the distributivity law

$$AB_1 + AB_2 = A(B_1 + B_2)$$

Then, remembering that + is the bitwise xor in GF(2),

```
((x >> 1) ^ x) & 0x55
```

becomes

$$\begin{aligned} (XD_{shiftr}(1)) + X)D_{\text{msk}}(0x55) &= (X(D_{shiftr}(1) + I))D_{\text{msk}}(0x55)\\ &= X((D_{shiftr}(1) + I)D_{\text{msk}}(0x55)\\ &= XD_{\text{sr1x1}} \end{aligned}$$

where $D_{\text{sr1x1}} = ((D_{shiftr}(1) + I)D_{\text{msk}}(0x55)$. Fig. 7 illustrates this process.

$$D_{shiftr}(1) = \begin{pmatrix} 0&0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0 \end{pmatrix} \quad D_{shiftr}(1)+I = \begin{pmatrix} 1&0&0&0&0&0&0&0\\ 1&1&0&0&0&0&0&0\\ 0&1&1&0&0&0&0&0\\ 0&0&1&1&0&0&0&0\\ 0&0&0&1&1&0&0&0\\ 0&0&0&0&1&1&0&0\\ 0&0&0&0&0&1&1&0\\ 0&0&0&0&0&0&1&1 \end{pmatrix}$$

$$D_{\mathrm{msk}}(0x55) = \begin{pmatrix} 1&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0 \end{pmatrix} \quad D_{\mathrm{sr1x1}} = \begin{pmatrix} 1&0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1&0 \end{pmatrix}$$

Figure 7: D_{sr1x1}

6.10 Example [ctz8]

The `ctz8` instruction counts the number of trailing zeroes. Consider for example

$$x = [msb]01001100[lsb]$$

Then $\texttt{ctz8}(x) = 2$ since there are two zeroes at the end [lsb] of x. It is well known that any of the following expressions turn any x into a form x' such that it contains a '1' for each of the trailing zeroes and a '0' everywhere else (see Hacker's Delight p.11):

```
x' = ~x & (x-1)
x' = ~(x | x-1)
x' = (x & -x) -1
```

Any of these three lines can easily be implemented using standard intrinsics. For x as above, the result would be

$$x' = [msb]00000011[lsb]$$

In order to implement `ctz8`, we now need to count the number of bits set in x'. Instead of using some `popcnt_epi8` instruction, we use $\circledast$ with a `rm_ii` opcode D_{pcnt}. To find the correct D_{pcnt}, we observe the following. The first bit in the result of $\texttt{popcnt}(x')$ is 1 if and only if the number of bits set in

x' is odd. Thus, we can take the sum (in GF(2)) of all bits in x' to do so. This results in only 1s in the first column of D_{pcnt}. Then, the second bit in the result of `popcnt`(x') is 1 if and only if the number of 1s in odd positions (starting indexing with 0) is odd. Thus, the second column of D_{pcnt} contains a 1 in all odd positions. The third bit in the result is 1 if and only if x' contains an odd number of bits set to 1 at positions p with $p+1 \mod 4 = 0$. Finally, The fourth bit of the result is 1 if and only if x' contains an odd number of bits set at positions p with $p+1 \mod 8 = 0$. Since there are no more than 8 bits in a byte, the remaining columns of D_{pcnt} contain only zeroes. Thus

$$D_{\text{pcnt}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

counts the numbers of bit set for every byte (row) in a matrix X in `rm_ii` where each byte (row) x of X must be of the form

$$x = [msb]00\ldots011\ldots1[lsb]$$

via XD_{pcnt}. For example

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

6.11 Example[dkny8]

Here, we implement

```
(0x0 == ((x >> k) & 0x1)) ? x : ~x
```

Thus, we need an operation that if $x[k] = 0$ does nothing and if $x[k] = 1$ inverts all bits. First, we note that for any bit b we have that

```
0 ^ b = b
```

and

```
1 ^ b = ~b
```

Thus, we can use $x[k]$ as the b. Then, we must bitwise xor every bit in a byte x by $x[k]$. Special care has to be taken to $x[k]$ which has to be cleared if it is one.

We first define matrices $D_{\text{row}}(k)$ for $0 \leq k < 8$ to contain only 1s in the k-th row and 0s everywhere else:

$$D_{\text{row}}(k)(i,j) = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{else} \end{cases} \tag{22}$$

Then, we define $D_{\text{dkny}}(k) = I + D_{\text{row}}(k)$. For $k = 3$ we then get

$$D_{\text{dkny}}(3) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

As an example application (again $k = 3$) consider

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

6.12 General Method to Derive AD-Opcodes

Assume we want to derive an opcode for a (partial) function $f : [0, 256] \to [0, 256]$ for n input/output value pairs. Then, we can store the n input values in a $n \times 8$ matrix A and all n output values in a $n \times 8$ matrix D. The AD-opcode, if it exists is a solution to the equation

$$AX = D \tag{23}$$

Denote by $A^- \in A\{1\}$ some $\{1\}$-inverse of A. Then, a solution exists if and only if

$$AA^-D = D \tag{24}$$

and if a solution exists, it can be derived as

$$X = A^-D + Y + A^-AY \tag{25}$$

for any Y. For $Y = 0$ this simplifies to

$$X = A^-D.$$

(see [`src/GFNI/BinaryMatrix.hh`] for the derivation of a $\{1\}$-inverse by transforming A into Hermite normal form and [`src/GFNI/main_binary_matrix.cc`] for the check for the existence of a solution and its derivation.)

7 Matrix Operations with GFNI

In this section, we assume that all matrices are in storage layout `rm_ii`.

7.1 Matrix Transformations

Let us first implement the matrix transformations tranpose, reverse rows, reverse columns and perpose using $\circledast$:

$$\begin{aligned}
A^t &= I \circledast (A \circledast I) && (26)\\
A^c &= I \circledast A && (27)\\
A^r &= (A \circledast R) \circledast I && (28)\\
A^p &= (A \circledast I) \circledast I && (29)\\
(A^t)^c &= A \circledast I && (30)\\
(A^t)^r &= I \circledast (A \circledast R) && (31)\\
(A^t)^p &= A \circledast R && (32)
\end{aligned}$$

Eq. 27 is the same as Eq. 10, Eq. 30 is the same as Eq. 11, Eq. 32 is the same as Eq. 13,

Eq. 30 followed by an addition reversal of columns (Eq. 27) gives Eq. 26. Eq. 31 follows from Eq.32 and the observation that $(A^t)^r = ((A^t)^p)^c$.

Next is the perpose transformation. Since $A \circledast I = (A^t)^c$, we have that

$$\begin{aligned}
(A \circledast I) \circledast I &= (((A^t)^c)^t)^c\\
&= (((A^r)^t)^t)^c\\
&= (A^r)^c\\
&= A^p
\end{aligned}$$

Last, we consider the reverse row transformation.

$$\begin{aligned}(A \circledast R) \circledast I &= (A^t)^p \circledast I \\ &= (((A^t)^p)^t)^c \\ &= (((A^t)^t)^p)^c \\ &= A^r\end{aligned}$$

Thus, we need two instructions for it. However, the reverse row transformation for a matrix in `rm_ii` layout corresponds to a `bswap64` instruction. But I do not know of a vectorized version of it. The only obvious solution instead of using the above would be to apply shuffle. This way, $A^r_{\texttt{rm\_ii}}$ can be implemented in a single instruction.

7.2 Matrix Multiplication

7.2.1 General Matrix Multiplication

In order to calculate $A_{\texttt{rm\_ii}} B_{\texttt{rm\_ii}}$ we use

$$A_{rm\_ir} B_{cm\_rr} = (A_{rm\_ir} \circledast B_{cm\_rr})_{cm\_rr}$$

Then, we have to convert

- A_{rm_ii} into A_{rm_ir}, which can be done via $I \circledast A$,
- B_{rm_ii} into B_{cm_rr}, which can be done via $B \circledast R$, and
- $X := (A_{rm\_ir} \circledast B_{cm\_rr})_{cm\_rr}$ into $(A_{rm\_ir} \circledast B_{cm\_rr})_{rm\_ii}$ which can be done via $X \circledast R$.

Putting things together, we get

$$AB = ((I \circledast A) \circledast (B \circledast R)) \circledast R \tag{33}$$

for A and B in `rm_ii`, requiring 4 $\circledast$ instructions.

To calulcate a series of matrix multiplications $A_1 \dots A_k$, where all A_i are in `rm_ii` ... [todo]

7.2.2 Special Cases AA^t and A^tA

In order to calculate AA^t, we use

$$A_{\texttt{rm\_ii}} B_{\texttt{cm\_ii}} = (A_{\texttt{rm\_ii}} \circledast B_{\texttt{cm\_ii}})_{\texttt{cm\_ir}}$$

Using $(A_{\texttt{rm\_ii}})^t = A_{\texttt{cm\_ii}}$, we see that we only have to correct the result layout. Thus

$$AA^t = I \circledast (A \circledast A) \tag{34}$$

In order to calculate A^tA, we could use

$$A_{\texttt{rm\_rr}} B_{\texttt{cm\_rr}} = (A_{\texttt{rm\_rr}} \circledast B_{\texttt{cm\_rr}})_{\texttt{cm\_ri}}$$

Then we need for the first argument $((A_{\texttt{rm\_ii}})^t)_{\texttt{rm\_rr}}$ which is $(A^t)^p$. Further, this also converts $A_{\texttt{rm\_ii}}$ into $A_{\texttt{cm\_rr}}$, which is required for the second argument. Finally, we have to convert the output layout `cm_ri` to `rm_ii` using Eqn. 31. This results in

$$\begin{aligned} A^tA &= ((A \circledast R) \circledast (A \circledast R))^r \\ &= (((A \circledast R) \circledast (A \circledast R)) \circledast R) \circledast I \\ &= (((A \circledast R) \circledast (A \circledast R)) \circledast I) \end{aligned}$$

The last step uses the fact that (A^tA) is symmetric, i.e. $(A^tA)^t = A^tA$ which allows for a simplified conversion using Eqn. 30 instead of Eqn. 31.