ATHOS: A Hybrid Accelerator for PQC CRYSTALS-Algorithms exploiting new CV-X-IF Interface

Alessandra Dolmeta

This document is supplementary material to the primary study on accelerating the Kyber algorithm, focusing on the hardware implementation details of the five accelerators introduced in the main paper. The accelerators are tailored to optimize specific computational tasks required by Kyber, including Montgomery and Barrett reduction, matrix generation, centered binomial distribution (CBD) operations, and polynomial arithmetic.

This supplementary material provides the detailed hardware design for each accelerator, outlining the specific operations implemented and the instruction-level optimizations made. Additionally, this document includes a comparative analysis of the original C code and the modified versions that incorporate the custom RISC-V instructions.
This material complements the performance evaluations discussed in the main study by demonstrating the practical application of the accelerators and providing insights into the hardware design.

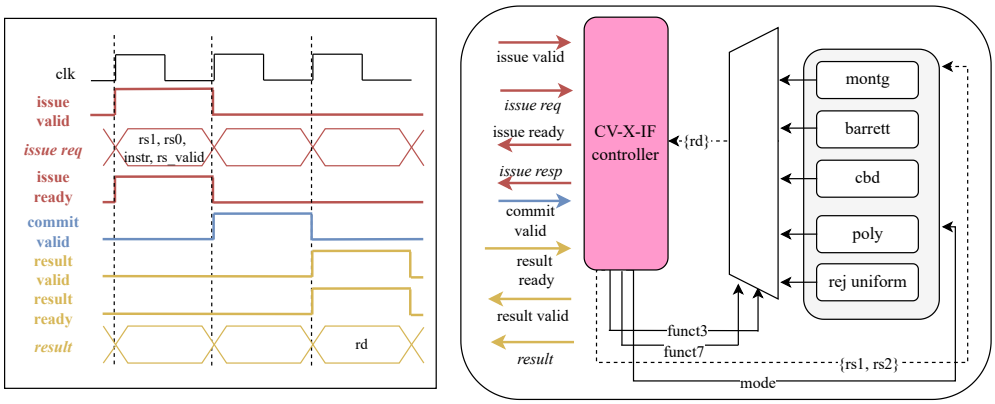

Figure 1: CV-X-IF protocol and ATHOS-tightly architecture.

1 ATHOS-Tightly

1.1 Barrett

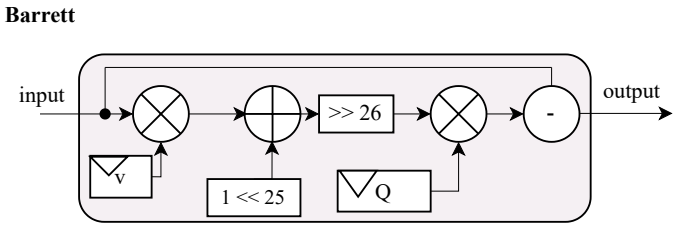

Figure 2

Original function:

```
int16_t PQCLEAN_KYBER512_CLEAN_barrett_reduce(int16_t a) {
    int16_t t;
    const int16_t v = ((1 << 26) + KYBER_Q / 2) / KYBER_Q;
    t  = ((int32_t)v * a + (1 << 25)) >> 26;
    t *= KYBER_Q;
    return a - t;
}
```

Optimized function:

```
asm volatile (".insn r 0x0b, 0x004, 0, %[dst], %[src], x0\r\n" : [dst] "=r" (a) :
    [src] "r" (a) : );
```

The original function `PQCLEAN_KYBER512_CLEAN_barrett_reduce` implements a reduction operation, commonly used in cryptographic algorithms to ensure that a value remains within a specific range, typically modulo a prime number. Below is a breakdown of the original code:

- **Constants and Variables:**
  - `v` is a precomputed constant used in the Barrett reduction algorithm. It is calculated based on the modulus `KYBER_Q` and is designed to scale and reduce the input `a`.
  - `t` is an intermediate variable used to store the scaled and reduced value.
- **Steps:**
  - The value `a` is multiplied by the constant `v` and then adjusted by adding a fixed value `(1 << 25)` before performing a right shift by 26 bits. This effectively performs a division operation.
  - The result `t` is then scaled back by multiplying with `KYBER_Q`, giving a value that is close to but less than `a`.
  - Finally, the original value `a` is reduced by subtracting `t`, ensuring that the result is within the range $[0, \texttt{KYBER\_Q})$.

The optimized function replaces the entire reduction process with a single custom assembly instruction. The line:

```
asm volatile (".insn r 0x0b, 0x004, 0, %[dst], %[src], x0\r\n" : [dst] "=r" (a) :
    [src] "r" (a) : );
```

does the following:

- **Custom Instruction:**
  - The `.insn` directive in assembly allows for the insertion of custom instructions directly into the code.
  - The operands and the instruction encoding are specified in the parameters:
    - `0x0b` indicates the opcode.
    - `0x004` and `0` are the function and subfunction fields, respectively.
    - `%[dst]` and `%[src]` refer to the destination and source registers, both of which are the same in this case (`a`).
    - `x0` is a zero register, often used in RISC architectures as a constant zero.
- **Single Instruction Optimization:**
  - The custom instruction likely performs a combined multiply, shift, and subtract operation similar to the original function, but does so in one cycle. This reduces the overhead of multiple arithmetic operations and register accesses in the original C code.
  - By using a single instruction, the operation becomes much faster, leveraging hardware acceleration or specific processor features that are not available in standard C code.

In summary, this optimized function uses a custom assembly instruction to perform the equivalent of the Barrett reduction in a single, highly efficient operation. The specific instruction encodes the logic of the original function, but instead of multiple steps, it is reduced to a single, potentially hardware-accelerated operation, which greatly improves performance.

1.2 Montgomery

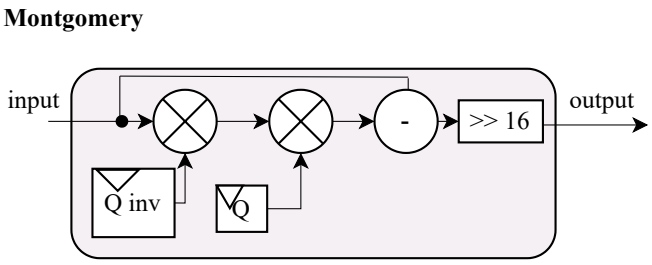

Figure 3

Original function:

```
int16_t PQCLEAN_KYBER512_CLEAN_montgomery_reduce(int32_t a) {
    int16_t t;
    t = (int16_t)a * QINV;
    t = (a - (int32_t)t * KYBER_Q) >> 16;
    return t;
}
```

Optimized function:

```
    asm volatile (".insn r 0x0b, 0x003, 2, %[dst], %[src], x0\r\n" : [dst] "=r" (t)
        : [src] "r" (a) : );
```

The original function `PQCLEAN_KYBER512_CLEAN_montgomery_reduce` implements the Montgomery reduction operation, which is a method used in modular arithmetic, especially within cryptographic algorithms, to efficiently compute modular reductions.

- **Variables:**
  - `t` is an intermediate variable used to store the partial reduction result.
- **Steps:**
  - The value `a` is first multiplied by `QINV`, which is the modular inverse of `KYBER_Q` modulo 2^{16}. This multiplication keeps the result within a manageable range.
  - The product is then multiplied by `KYBER_Q`, and this product is subtracted from `a`.
  - The result is then shifted right by 16 bits, effectively performing a division by 2^{16}, and the result is stored in `t`.

The optimized function replaces the above steps with a single custom assembly instruction. The line:

```
asm volatile (".insn r 0x0b, 0x003, 2, %[dst], %[src], x0\r\n" : [dst] "=r" (t) :
    [src] "r" (a) : );
```

does the following:

- **Custom Instruction:**
  - The `.insn` directive in assembly allows the insertion of a custom instruction directly into the code.
  - The operands and the instruction encoding are specified as follows:

    - `0x0b` indicates the opcode.
    - `0x003` specifies the function code.
    - `2` indicates the immediate value used in the operation.
    - `%[dst]` and `%[src]` refer to the destination and source registers, corresponding to `t` and `a`, respectively.
    - `x0` is a zero register used as a constant zero.

- **Single Instruction Optimization:**
  - This custom instruction likely performs the Montgomery reduction in a single, combined operation, effectively replacing the multiple arithmetic and logical operations from the original function.
  - The use of a single assembly instruction significantly reduces the overhead, making the operation much faster by leveraging hardware features or processor-specific optimizations.

In summary, the optimized function uses a custom assembly instruction to perform the Montgomery reduction in one step, enhancing the efficiency and speed of the operation by taking advantage of specialized hardware capabilities.

1.3 cbd

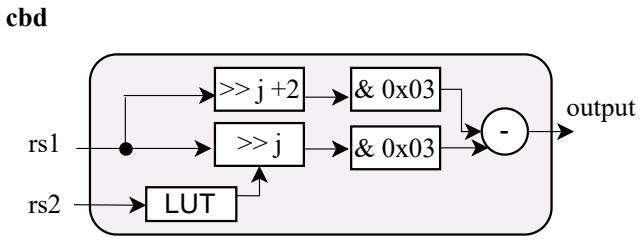

Figure 4

Original function:

```
static void cbd2(poly *r, const uint8_t buf[2 * KYBER_N / 4]) {
    unsigned int i, j;
    uint32_t t, d;
    int16_t a, b;

    for (i = 0; i < KYBER_N / 8; i++) {
        t  = load32_littleendian(buf + 4 * i);
        d  = t & 0x55555555;
        d += (t >> 1) & 0x55555555;
        for (j = 0; j < 8; j++) {
            a = (d >> (4 * j + 0)) & 0x3;
            b = (d >> (4 * j + 2)) & 0x3;
            r->coeffs[8 * i + j] = a - b;
        }}}
```

Optimized function:

```
static void cbd2(poly *r, const uint8_t buf[2 * KYBER_N / 4]) {
    unsigned int i, j;
    uint32_t t, d;
    int16_t a, b;

    for (i = 0; i < KYBER_N / 8; i++) {
        asm volatile (".insn i 0x2b, 2, %[dst], %[src], 0\r\n" : [dst] "=r" (d) :
            [src] "r" (buf + 4 * i) : );

         for (j = 0; j < 8; j++) {
            asm volatile (".insn r 0x0b, 0x5, 0x9, %[dst], %[src], %[x]\r\n" : [dst]
                "=r" (r->coeffs[8 * i + j]) : [src] "r" (d), [x] "r" (j) : );
    }}}
```

The function `cbd2` computes a polynomial from an array of uniformly random bytes. The coefficients of the resulting polynomial are distributed according to a centered binomial distribution with parameter $\eta = 2$. This function is used in Kyber cryptographic schemes to generate polynomial coefficients from random bytes.

- **Inputs:**
  - `r`: Pointer to the output polynomial.
  - `buf`: Pointer to the input byte array, which is assumed to be uniformly random.

- **Steps:**
  - For each 32-bit chunk of the input buffer (8 chunks total per iteration of the outer loop):
    - Load 32 bits from the buffer and store in `t`.
    - Compute `d` by isolating and summing parts of `t` to get values distributed according to the binomial distribution.
    - Extract and compute polynomial coefficients `a` and `b` from `d`.
    - Store the difference $a - b$ in the polynomial coefficients array.

The use of custom assembly instructions optimizes the loading and calculation operations, reducing the number of explicit C operations and leveraging hardware-specific features to improve performance.

`cbd3`: Similar to `cbd2`, but for Kyber-512 only. It computes a polynomial with coefficients distributed according to a centered binomial distribution with parameter $\eta = 3$. It has similar principles but involves different parameter values and computations.

1.4 rej_uniform

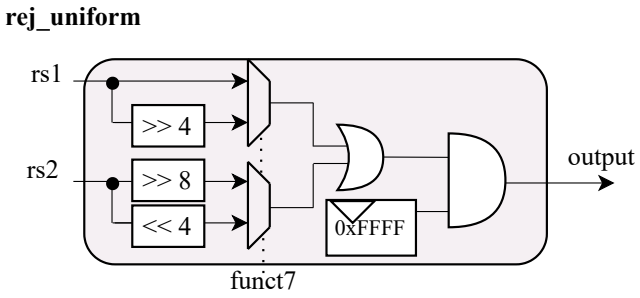

Figure 5

In the original function, the following lines extract 12-bit values from the byte array `buf`:

```
val0 = ((buf[pos + 0] >> 0) | ((uint16_t)buf[pos + 1] << 8)) & 0xFFF;
val1 = ((buf[pos + 1] >> 4) | ((uint16_t)buf[pos + 2] << 4)) & 0xFFF;
```

These operations combine adjacent bytes and apply bit-shifting and masking to produce 12-bit values `val0` and `val1`.

In the optimized function, the equivalent operations are performed using custom assembly instructions:

```
asm volatile (".insn r 0x0b, 0x1, 40, %[dst], %[src], %[src2]\r\n" : [dst] "=r"
    (val0) : [src] "r" (buf[pos + 0]), [src2] "r" (buf[pos + 1]) : );
asm volatile (".insn r 0x0b, 0x1, 41, %[dst], %[src], %[src2]\r\n" : [dst] "=r"
    (val1) : [src] "r" (buf[pos + 1]), [src2] "r" (buf[pos + 2]) : );
```

Explanation:

- The custom instructions use assembly to directly combine and shift the values of the adjacent bytes (`buf[pos+0]`, `buf[pos+1]`, and `buf[pos+2]`) into the 12-bit values `val0` and `val1`.
- This optimization replaces manual bit manipulation with efficient hardware-level operations, improving performance.

1.5 poly_compress

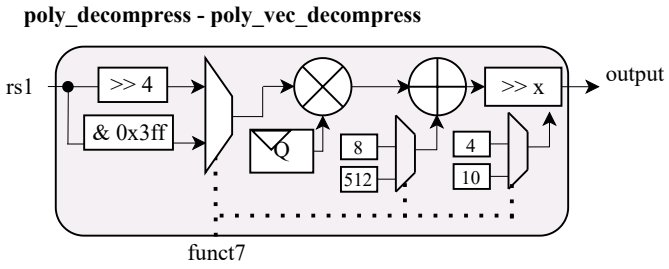

Figure 6

In the original `PQCLEAN_KYBER512_CLEAN_poly_compress` function, coefficients of a polynomial are compressed into 4 bytes at a time. The inner loop iterates over groups of 8 coefficients and transforms each coefficient to map it into 4-bit values.

```
void PQCLEAN_KYBER512_CLEAN_poly_compress(uint8_t r[KYBER_POLYCOMPRESSEDBYTES],
    const poly *a) {
    unsigned int i, j;
    int32_t u;
    uint32_t d0;
    uint8_t t[8];

    for (i = 0; i < KYBER_N / 8; i++) {
        for (j = 0; j < 8; j++) {
            // map to positive standard representatives
            u  = a->coeffs[8 * i + j];
            u += (u >> 15) & KYBER_Q;
            d0 = u << 4;
            d0 += 1665;
            d0 *= 80635;
            d0 >>= 28;
            t[j] = d0 & 0xf;
        }

        r[0] = t[0] | (t[1] << 4);
        r[1] = t[2] | (t[3] << 4);
        r[2] = t[4] | (t[5] << 4);
        r[3] = t[6] | (t[7] << 4);
        r += 4;
    }
}
```

The function compresses the polynomial coefficients into 4-bit values and stores them in the output array `r`.
In the inner loop, each coefficient `u` is mapped into a positive representative and then compressed using a series of shifts, additions, and multiplications. The result is stored in the temporary array `t`, where each element holds a 4-bit compressed value.

In the optimized function, the compression steps for each coefficient are replaced with custom assembly instructions.

```
asm volatile (".insn r 0x0b, 0x6, 17, %[dst], %[src], %[x]\r\n" : [dst] "=r" (t[0])
    : [src] "r" (a->coeffs[8 * i]), [x] "r" (0) : );
```

```
asm volatile (".insn r 0x0b, 0x6, 17, %[dst], %[src], %[x]\r\n" : [dst] "=r" (t[1])
    : [src] "r" (a->coeffs[8 * i + 1]), [x] "r" (0) : );
asm volatile (".insn r 0x0b, 0x6, 17, %[dst], %[src], %[x]\r\n" : [dst] "=r" (t[2])
    : [src] "r" (a->coeffs[8 * i + 2]), [x] "r" (0) : );
asm volatile (".insn r 0x0b, 0x6, 17, %[dst] "=r" (t[3]) : [src] "r" (a->coeffs[8 *
    i + 3]), [x] "r" (0) : );
```

Each assembly instruction (`.insn r`) processes a single coefficient `a->coeffs[8 * i + j]`. The custom instruction efficiently compresses the coefficients into 4-bit values, storing the result directly in `t[j]`.
By using hardware-specific instructions, the compression step is significantly accelerated, reducing the computational overhead of the transformation.

1.6 poly_decompress

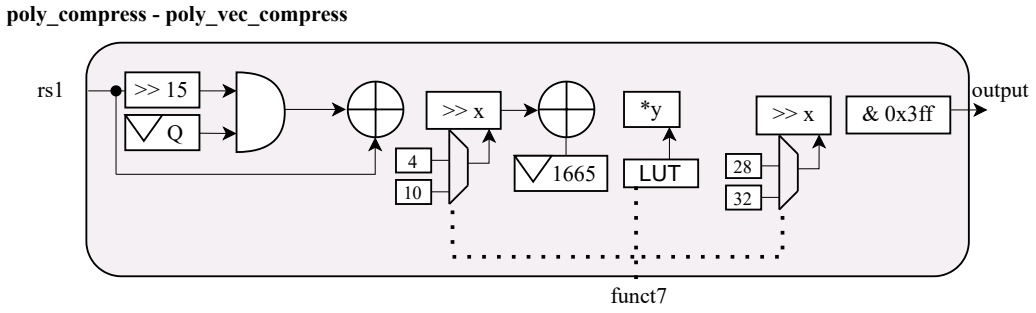

Figure 7

In the original `PQCLEAN_KYBER512_CLEAN_poly_decompress` function, the coefficients of the compressed polynomial are decompressed and stored into the output polynomial `r`:

```
void PQCLEAN_KYBER512_CLEAN_poly_decompress(poly *r, const uint8_t
    a[KYBER_POLYCOMPRESSEDBYTES]) {
    size_t i;

    for (i = 0; i < KYBER_N / 2; i++) {
        r->coeffs[2 * i + 0] = (((uint16_t)(a[0] & 15) * KYBER_Q) + 8) >> 4;
        r->coeffs[2 * i + 1] = (((uint16_t)(a[0] >> 4) * KYBER_Q) + 8) >> 4;
        a += 1;
    }
}
```

This function decompresses the polynomial coefficients from the compressed array `a`.

- The lower 4 bits of each byte `a[0]` are decompressed into `r->coeffs[2 * i + 0]`.
- The upper 4 bits of the byte are decompressed into `r->coeffs[2 * i + 1]`.
- These values are scaled by `KYBER_Q`, adjusted with an offset, and right-shifted to fit into the polynomial.

In the optimized function, this decompression process is accelerated using custom assembly instructions:

```
void PQCLEAN_KYBER512_CLEAN_poly_decompress(poly *r, const uint8_t
    a[KYBER_POLYCOMPRESSEDBYTES]) {
    size_t i;

    for (i = 0; i < KYBER_N / 2; i++) {
        asm volatile (".insn r 0x0b, 0x006, 18, %[dst], %[src], %[x]\r\n" : [dst]
            "=r" (r->coeffs[2 * i + 0]) : [src] "r" (a[0]), [x] "r" (0) : );
        asm volatile (".insn r 0x0b, 0x006, 19, %[dst], %[src], %[x]\r\n" : [dst]
            "=r" (r->coeffs[2 * i + 1]) : [src] "r" (a[0]), [x] "r" (0) : );
        a += 1;
    }
}
```

- The decompression of `r->coeffs[2 * i + 0]` and `r->coeffs[2 * i + 1]` is handled using custom instructions: `.insn r 0x0b`.

- The first assembly instruction extracts and decompresses the lower 4 bits of `a[0]`, and the second instruction processes the upper 4 bits.

This approach uses hardware-specific instructions to replace the manual bit extraction and scaling, offering a more efficient decomposition of the polynomial coefficients.

1.7 poly_from_msg

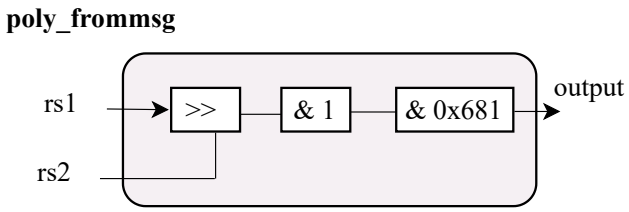

Figure 8

In the original `PQCLEAN_KYBER512_CLEAN_poly_frommsg` function, the polynomial `r` is constructed from the message `msg`:

```
void PQCLEAN_KYBER512_CLEAN_poly_frommsg(poly *r, const uint8_t
    msg[KYBER_INDCPA_MSGBYTES]) {
    size_t i, j;
    int16_t mask;

    for (i = 0; i < KYBER_N / 8; i++) {
        for (j = 0; j < 8; j++) {
            mask = -(int16_t)((msg[i] >> j) & 1);
            r->coeffs[8 * i + j] = mask & ((KYBER_Q + 1) / 2);
        }
    }
}
```

This function converts a message `msg` into a polynomial `r`.

- For each byte in `msg`, the bits are extracted and transformed.
- The bitwise value of `msg[i] >> j` is used to determine the mask, which is applied to generate the corresponding polynomial coefficient.
- The resulting polynomial coefficients are scaled by `(KYBER_Q + 1) / 2`.

In the optimized function, custom assembly instructions are used to accelerate the inner loop by processing the coefficients in a single operation:

```
asm volatile (".insn r 0x0b, 0x006, 27, %[dst], %[src], %[src2]\r\n" : [dst] "=r"
    (r->coeffs[8*i+j]) : [src] "r" (msg[i]), [src2] "r" (j) : );
```

Each coefficient of `r->coeffs` is computed using the custom assembly instruction `.insn r 0x0b`, which efficiently processes the mask and scaling operations that were originally done in C. The register `msg[i]` is passed as input to generate the corresponding coefficients, iterating through each bit using different values of `src2` (0 through 7).
By using these hardware-specific instructions, the need for manual bit manipulation and masking is eliminated, leading to more efficient polynomial construction.

1.8 poly_to_msg

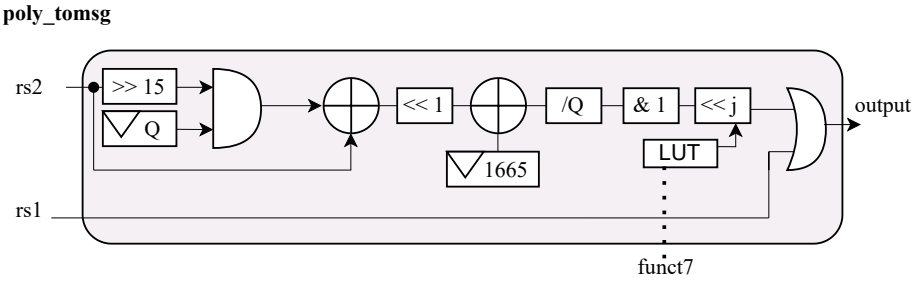

Figure 9

In the original `PQCLEAN_KYBER512_CLEAN_poly_tomsg` function, the polynomial `a` is converted into a message `msg`:

```
void PQCLEAN_KYBER512_CLEAN_poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const
    poly *a) {
    unsigned int i, j;
    uint32_t t;

    for (i = 0; i < KYBER_N / 8; i++) {
        msg[i] = 0;
        for (j = 0; j < 8; j++) {
            t  = a->coeffs[8 * i + j];
            t += ((int16_t)t >> 15) & KYBER_Q;
            t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
            msg[i] |= t << j;
        }
    }
}
```

This function converts a polynomial `a` back into a message `msg`.

- For each group of 8 coefficients in `a`, the function first adjusts the value to fit within the desired range.
- The function then determines whether the coefficient is closer to 0 or `KYBER_Q`, and sets the corresponding bit in `msg[i]`.

In the optimized function, assembly instructions are used to perform the same steps more efficiently:

```
asm volatile (".insn r 0x0b, 0x006, 28, %[dst], %[src], %[src2]\r\n" : [dst] "=r"
    (msg[i]) : [src] "r" (a->coeffs[8 * i + j]), [src2] "r" (msg[i]) : );
```

Instead of manually adjusting and checking each coefficient, the assembly instruction `.insn r 0x0b, 0x006, 28` directly handles the transformation from the coefficient in `a->coeffs` to the appropriate bit in `msg[i]`. The instruction takes `a->coeffs[8 * i + j]` as input, processes it, and updates the message `msg[i]` by setting the corresponding bit based on the coefficient value.

1.9 poly_from_bytes

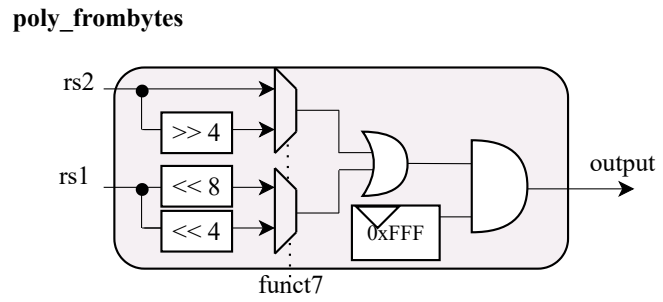

Figure 10

In the original `PQCLEAN_KYBER512_CLEAN_poly_frombytes` function, the byte array `a` is converted into the polynomial `r`:

```
void PQCLEAN_KYBER512_CLEAN_poly_frombytes(poly *r, const uint8_t
    a[KYBER_POLYBYTES]) {
    size_t i;
    for (i = 0; i < KYBER_N / 2; i++) {
        r->coeffs[2 * i] = ((a[3 * i + 0] >> 0) | ((uint16_t)a[3 * i + 1] << 8)) &
            0xFFF;
        r->coeffs[2 * i + 1] = ((a[3 * i + 1] >> 4) | ((uint16_t)a[3 * i + 2] << 4))
            & 0xFFF;
    }}
```

This function converts a byte array `a` into a polynomial `r`.

- For every two coefficients in the polynomial `r`, the function processes three bytes from the array `a`.
- The lower 12 bits of the first two bytes are combined to form `r->coeffs[2 * i]`, while the upper 12 bits of the second and third bytes are combined to form `r->coeffs[2 * i + 1]`.

In the optimized function, assembly instructions are used to achieve the same result more efficiently:

```
void PQCLEAN_KYBER512_CLEAN_poly_frombytes(poly *r, const uint8_t
    a[KYBER_POLYBYTES]) {
    size_t i;
    for (i = 0; i < KYBER_N / 2; i++) {
        asm volatile (".insn r 0x0b, 0x006, 25, %[dst], %[src], %[src2]\r\n" : [dst]
            "=r" (r->coeffs[2 * i] ) : [src] "r" (a[3 * i + 0]), [src2] "r" (a[3 * i
            + 1]) : );
        asm volatile (".insn r 0x0b, 0x006, 26, %[dst], %[src], %[src2]\r\n" : [dst]
            "=r" (r->coeffs[2 * i + 1] ) : [src] "r" (a[3 * i + 1]), [src2] "r" (a[3
            * i + 2]) : );
    }}
```

The assembly instructions `.insn r 0x0b, 0x006, 25` and `.insn r 0x0b, 0x006, 26` efficiently compute the same operations as the original function. These instructions combine the bytes from the array `a` and directly store the results in `r->coeffs`, minimizing manual bit manipulation.

This optimization reduces the number of steps and improves the overall performance, particularly for large input sizes.

1.10 poly_to_bytes

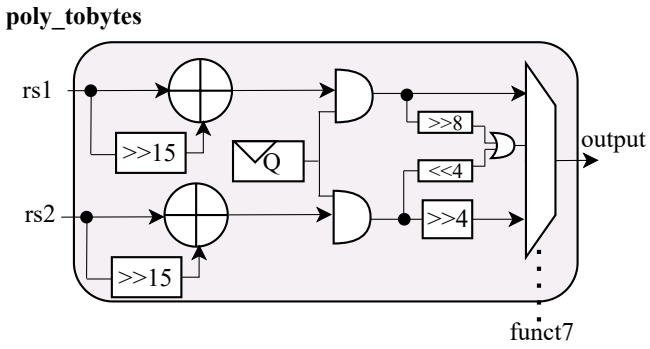

Figure 11

In the original `PQCLEAN_KYBER512_CLEAN_poly_tobytes` function, the polynomial `a` is converted into a byte array `r`:

```
void PQCLEAN_KYBER512_CLEAN_poly_tobytes(uint8_t r[KYBER_POLYBYTES], const poly *a)
    {
    size_t i;
    uint16_t t0, t1;
    for (i = 0; i < KYBER_N / 2; i++) {
        t0  = a->coeffs[2 * i];
        t0 += ((int16_t)t0 >> 15) & KYBER_Q;
        t1 = a->coeffs[2 * i + 1];
        t1 += ((int16_t)t1 >> 15) & KYBER_Q;
        r[3 * i + 0] = (uint8_t)(t0 >> 0);
        r[3 * i + 1] = (uint8_t)((t0 >> 8) | (t1 << 4));
        r[3 * i + 2] = (uint8_t)(t1 >> 4);
    }}
```

This function converts a polynomial `a` into a byte array `r`.

- For every two coefficients in the polynomial `a`, the function generates three bytes for the array `r`.
- The lower 12 bits of the two coefficients `t0` and `t1` are split across three bytes.

In the optimized function, assembly instructions are used to perform the same operation in a more efficient manner:

```
void PQCLEAN_KYBER512_CLEAN_poly_tobytes(uint8_t r[KYBER_POLYBYTES], const poly *a)
    {
    size_t i;
    for (i = 0; i < KYBER_N / 2; i++) {
        asm volatile (".insn r 0x0b, 0x006, 22, %[dst], %[src], %[src2]\r\n" : [dst]
            "=r" (r[3 * i + 0]) : [src] "r" (a->coeffs[2 * i]), [src2] "r"
            (a->coeffs[2 * i + 1]) : );
        asm volatile (".insn r 0x0b, 0x006, 23, %[dst], %[src], %[src2]\r\n" : [dst]
            "=r" (r[3 * i + 1]) : [src] "r" (a->coeffs[2 * i]), [src2] "r"
            (a->coeffs[2 * i + 1]) : );
        asm volatile (".insn r 0x0b, 0x006, 24, %[dst], %[src], %[src2]\r\n" : [dst]
            "=r" (r[3 * i + 2]) : [src] "r" (a->coeffs[2 * i]), [src2] "r"
            (a->coeffs[2 * i + 1]) : );
```

```
}}
```

The assembly instructions `.insn r 0x0b, 0x006, 22`, `23`, and `24` are used to handle the byte conversion more efficiently. These instructions split the 12-bit values of the two coefficients and store them in the byte array `r`.

This optimization eliminates manual bit manipulations, resulting in a more efficient implementation for converting the polynomial to the byte array.