

# 어셈블리 분석 및 Hawk 최적화

<https://youtu.be/64FIW6fjCNg>

# 1. 어셈블리 분석

- 작성된 코드가 작동하는 과정

1. 전처리 과정

- C 코드가 컴파일 되기 전에 먼저 “전처리기”에 의한 전처리 과정을 수행함.
  - 전처리 과정은 `#include`, `#define`, `#ifdef` 와 같은 전처리 지시문들이 처리되는 것을 의미.
  - 이 과정에서는 여전히 C 코드인 상태

2. 컴파일 과정

- 전처리가 끝난 C 코드는 “컴파일러”에 의해서 어셈블리어로 변환됨.
  - 실행되는 환경(x86, arm 등)에 맞춰 어셈블리 코드로 변환됨

3. 어셈블리 과정

- 컴파일러에 의해 변환된 어셈블리 코드는 “어셈블러”에 의해서 기계어로 변환
  - 기계어는 실제 컴퓨터가 이해할 수 있는 이진 코드. 이렇게 생성된 파일이 오브젝트 파일(.o)

4. 링킹 과정

- 여러 개의 오브젝트 파일과 라이브러리 파일들이 “링커”에 의해 결합되는 과정
  - 프로그램 실행에 필요한 모든 기계어 코드를 결합하고, 함수와 변수의 주소를 해결함.
  - 이 과정에서 실행 파일이 생성됨(.exe)

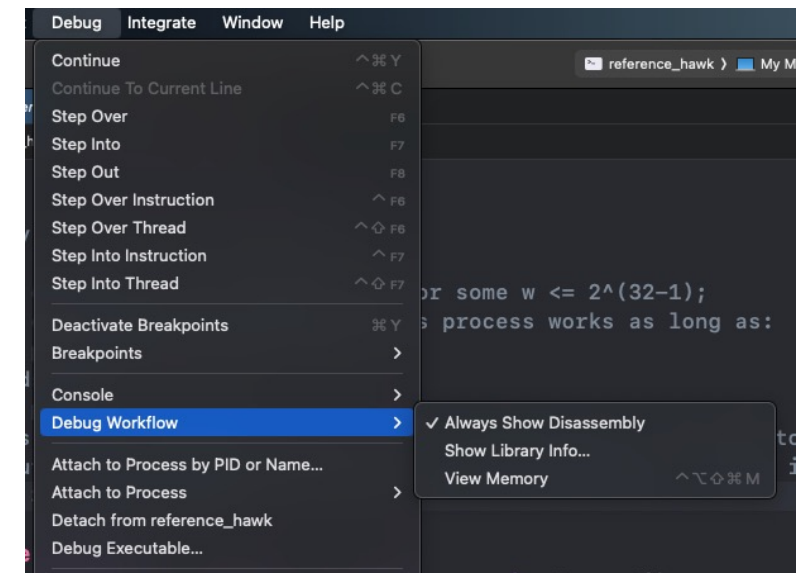
# 1. 어셈블리 과정

- 최적화를 위해서는 컴파일러가 C 코드를 어셈블리어로 변환하는 것보다 더 효율적인 어셈블리어로 구현되어야 함.
- 컴파일러는 여러 최적화 기법들이 적용되어 있어 자동으로 높은 효율성을 가진 어셈블리어로 변환을 함.
  - 벡터 레지스터는 잘 사용하지 않는 거 같음.
  - 따라서 벡터 레지스터를 활용하는 것이 아니라면, 컴파일러보다 더 좋은 성능을 보여주기 어려움
- 그렇기 때문에 컴파일러에 의해 변환되는 어셈블리 코드를 분석하는 것이 필요.

## 2. 디버그를 통해 어셈블리어 확인

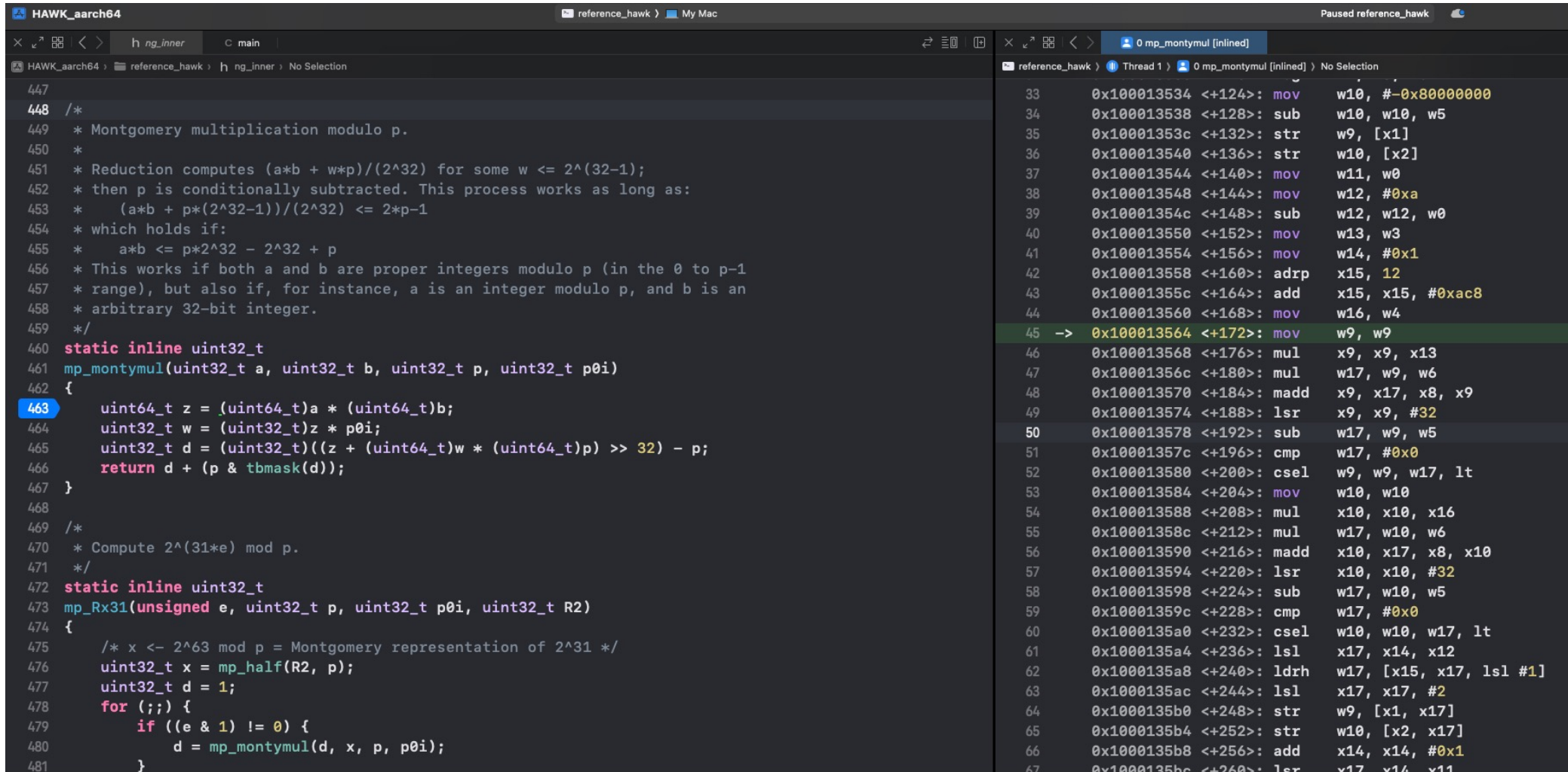
- 컴파일러 (gcc, clang)에서는 -S 옵션을 통해서 어셈블리 코드를 확인할 수 있음.
  - (.s) 파일로 코드를 확인할 수 있지만, 내가 원하는 부분의 코드를 확인하기 어려움.
- IDE의 디버깅 모드와 브레이크를 사용하면 원하는 코드줄의 어셈블리 코드를 확인하는 것이 쉬움.

```
460 static inline uint32_t
461 mp_montymul(uint32_t a, uint32_t b, uint32_t p, uint32_t p0i)
462 {
463     uint64_t z = (uint64_t)a * (uint64_t)b;
464     uint32_t w = (uint32_t)z * p0i;
465     uint32_t d = (uint32_t)((z + (uint64_t)w * (uint64_t)p) >> 32) - p;
466     return d + (p & tbmask(d));
467 }
468
```



## 2. 디버그를 통해 어셈블리어 확인

- Xcode에서 확인하기



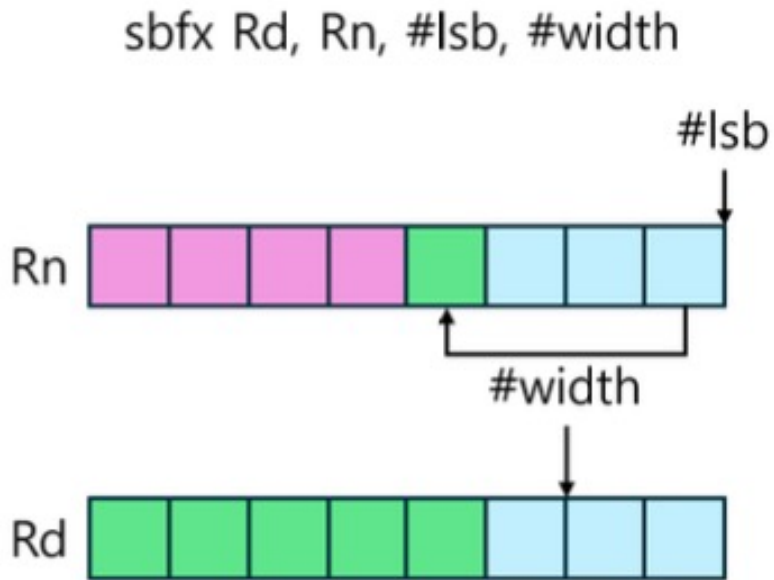
The screenshot displays the Xcode IDE interface. The left pane shows the C source code for a Montgomery multiplication function. The right pane shows the corresponding ARM assembly code for the same function, with line 45 highlighted in green.

```
447
448 /*
449  * Montgomery multiplication modulo p.
450  *
451  * Reduction computes (a*b + w*p)/(2^32) for some w <= 2^(32-1);
452  * then p is conditionally subtracted. This process works as long as:
453  *   (a*b + p*(2^32-1))/(2^32) <= 2*p-1
454  * which holds if:
455  *   a*b <= p*2^32 - 2^32 + p
456  * This works if both a and b are proper integers modulo p (in the 0 to p-1
457  * range), but also if, for instance, a is an integer modulo p, and b is an
458  * arbitrary 32-bit integer.
459  */
460 static inline uint32_t
461 mp_montymul(uint32_t a, uint32_t b, uint32_t p, uint32_t p0i)
462 {
463     uint64_t z = (uint64_t)a * (uint64_t)b;
464     uint32_t w = (uint32_t)z * p0i;
465     uint32_t d = (uint32_t)((z + (uint64_t)w * (uint64_t)p) >> 32) - p;
466     return d + (p & tbmask(d));
467 }
468
469 /*
470  * Compute 2^(31*e) mod p.
471  */
472 static inline uint32_t
473 mp_Rx31(unsigned e, uint32_t p, uint32_t p0i, uint32_t R2)
474 {
475     /* x <- 2^63 mod p = Montgomery representation of 2^31 */
476     uint32_t x = mp_half(R2, p);
477     uint32_t d = 1;
478     for (;;) {
479         if ((e & 1) != 0) {
480             d = mp_montymul(d, x, p, p0i);
481         }
```

```
33 0x100013534 <+124>: mov    w10, #-0x80000000
34 0x100013538 <+128>: sub    w10, w10, w5
35 0x10001353c <+132>: str    w9, [x1]
36 0x100013540 <+136>: str    w10, [x2]
37 0x100013544 <+140>: mov    w11, w0
38 0x100013548 <+144>: mov    w12, #0xa
39 0x10001354c <+148>: sub    w12, w12, w0
40 0x100013550 <+152>: mov    w13, w3
41 0x100013554 <+156>: mov    w14, #0x1
42 0x100013558 <+160>: adrp   x15, 12
43 0x10001355c <+164>: add    x15, x15, #0xac8
44 0x100013560 <+168>: mov    w16, w4
45 -> 0x100013564 <+172>: mov    w9, w9
46 0x100013568 <+176>: mul    x9, x9, x13
47 0x10001356c <+180>: mul    w17, w9, w6
48 0x100013570 <+184>: madd   x9, x17, x8, x9
49 0x100013574 <+188>: lsr    x9, x9, #32
50 0x100013578 <+192>: sub    w17, w9, w5
51 0x10001357c <+196>: cmp    w17, #0x0
52 0x100013580 <+200>: csel   w9, w9, w17, lt
53 0x100013584 <+204>: mov    w10, w10
54 0x100013588 <+208>: mul    x10, x10, x16
55 0x10001358c <+212>: mul    w17, w10, w6
56 0x100013590 <+216>: madd   x10, x17, x8, x10
57 0x100013594 <+220>: lsr    x10, x10, #32
58 0x100013598 <+224>: sub    w17, w10, w5
59 0x10001359c <+228>: cmp    w17, #0x0
60 0x1000135a0 <+232>: csel   w10, w10, w17, lt
61 0x1000135a4 <+236>: lsl    x17, x14, x12
62 0x1000135a8 <+240>: ldrh   w17, [x15, x17, lsl #1]
63 0x1000135ac <+244>: lsl    x17, x17, #2
64 0x1000135b0 <+248>: str    w9, [x1, x17]
65 0x1000135b4 <+252>: str    w10, [x2, x17]
66 0x1000135b8 <+256>: add    x14, x14, #0x1
67 0x1000135bc <+260>: lsr    w17, w14, w11
```

## 2. 디버그를 통해 어셈블리어 확인

- Hawk 알고리즘



---

### Algorithm 5 uint32\_t mp\_half

---

**Require:** uint32\_t \*a, uint32\_t p  
1: return (a + (p & -(a & 1))) >> 1;

---

Before(Using Branch)	After(Using 'sbfx' (Ours))
1: and w3, w0, #1 2: cbz w3, _even 3: add w0, w0, w1 4: _even: 5: lsr w0, w0, #1	1: sbfx w2, w0, #0, #1 2: and w1, w1, w2 3: add w0, w0, w1 4: lsr w0, w0, #1

## 2. 디버그를 통해 어셈블리어 확인

### • Hawk 알고리즘

#### Algorithm 4 uint32\_t zint\_mod\_small\_unsigned

**Require:** const uint32\_t \*d, size\_t len, size\_t stride, uint32\_t p, uint32\_t p0i, uint32\_t R2

**Ensure:** A 32-bit unsigned integer result.

```

1: uint32_t x = 0;
2: uint32_t z = mp_half(R2, p);
3: d += len * stride;
4: for uint32_t u = len; u > 0; u- do
5:   d -= stride;
6:   uint32_t w = *d - p;
7:   w += p & tbmask(w);
8:   x = mp_montymul(x, z, p, p0i);
9:   x = mp_add(x, w, p);
10: end for
11: return x;

```

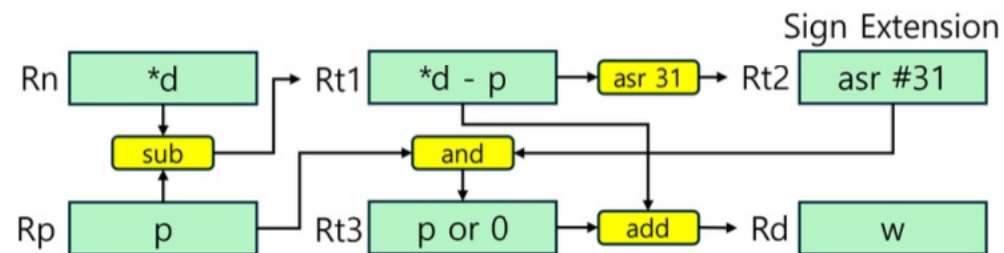


Figure 3. the  $w$  calculation process without using the 'csel' instruction.

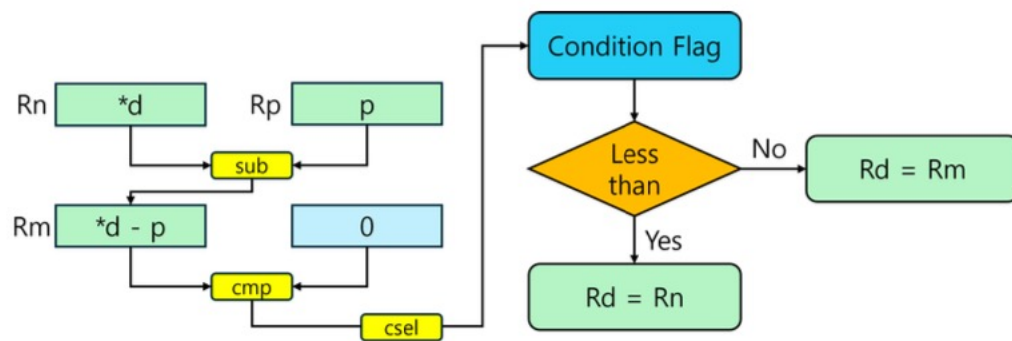


Figure 4. the  $w$  calculation process using the 'csel' instruction.

Before	After(Using 'csel' (Ours))
1: sub w0, w1, w2	1: sub w0, w1, w2
2: asr w3, w0, #31	2: cmp w0, #0
3: and w3, w3, w2	3: csel w0, w1, w0, lt
4: add w0, w0, w3	



## 2. 디버그를 통해 어셈블리어 확인

### • Hawk 알고리즘

#### Algorithm 6 void zint\_add\_mul\_small

**Require:** uint32\_t \*restrict x, size\_t len, size\_t xstride, const uint32\_t \*restrict y, uint32\_t s

**Ensure:** Updated x array.

```

1: uint32_t cc = 0;
2: for size_t u = 0; u < len; u++ do
3:   uint32_t xw, yw;
4:   uint64_t z;
5:   xw = *x;
6:   yw = y[u];
7:   z = (uint64_t)yw * (uint64_t)s + (uint64_t)xw + (uint64_t)cc;
8:   *x = (uint32_t)z & 0x7FFFFFFF;
9:   cc = (uint32_t)(z >> 31);
10:  x += xstride;
11: end for
12: *x = cc;
  
```

Table 10. Assembly code of Computation z(x0: xw, x1: yw, x2: s, x3: cc)

Before	After(Using 'madd' (Ours))
1: add x0, x0, x3 2: mul x1, x1, x2 3: add x0, x0, x1	1: add x0, x0, x3 2: madd x0, x1, x2, x0

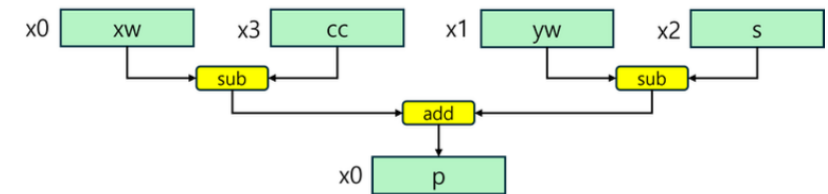


Figure 5. the z Computation process without using the 'madd' instruction.

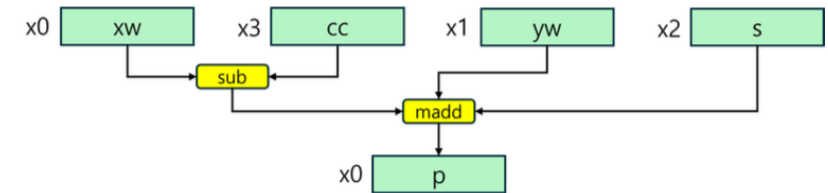


Figure 6. the z Computation process using the 'madd' instruction.



감 사 합 니 다