Assembly for medium precision arithmetic

Albin Ahlbäck

MPFR/MPC/MPFI/ARB Developers Meeting
Institut de Mathématiques de Bordeaux, 18 June 2024

The most basic operations

The most basic operations for multiple-precision arithmetic:

- Addition and subtraction
- Left and right shift
- Schoolbook multiplication

For faster computer arithmetic, we can combine certain operations such as *add a to* $2^k \cdot b$ *and store result in* r.

Other important basics

- Schoolbook squaring
- Low, mid and high schoolbook multiplication/squaring
- Granlund-Möller approximate reciprocal $\left\lfloor \frac{\beta^{2n}-1}{d} \right\rfloor - \beta^n$ (see [1])
- Division via approximate reciprocal [1]
- Greatest common divisor

The GNU MP Library (GMP)

Provides fast implementations for basic multiple precision arithmetic for many processor models.
Has:

- Extremely fast asymptotics for basecase algorithms [2]
- Well-written C and assembly code

Fast Library for Number Theory (FLINT)

Has started to extend GMP's low-level interface by incorporating instruction set specific assembly code for 64-bit ARM and 64-bit x86.

- Implementations of low and high multiplication exist.
- Tries to provide routines that outperforms GMP, with the caveat that we do not care about binary size.

GMP's `mpn`-routines for Apple M1

GMP provides superior asymptotics for this model for many routines:

Function	Cycles per limb
`mpn_add_n`	1
`mpn_sub_n`	1
`mpn_addlsh1_n`	1
`mpn_mul_1`	1
`mpn_addmul_1`	5/4

The first four routines have *optimal* asymptotics.

`mpn_addmul_1` has optimal asymptotic for how it is written.

Dissecting `mpn_addmul_1`

GMP's `mpn_addmul_1` performs the operation

$$r \leftarrow r + a \cdot b_0,$$

where r, a and b_0 are non-negative integers and $a < \beta^n$ and $b < \beta$. This is done via the basecase/schoolbook/naïve algorithm.

Overview of Apple M1's Firestorm-unit

Dougall Johnson has done great research on Apple's M1 architecture [3]. In the following slide, we will show a scheme that gives a good oversight for possible lower bounds of the number of cycles per limb for `mpn_mul_1`.

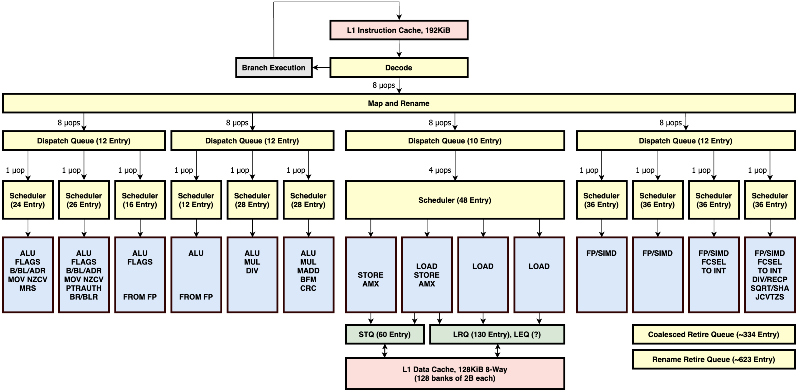

Figure: Overview of Apple M1's Firestorm-unit [3].

GMP's mpn_addmul_1 on Apple M1

```
L(top):
  ldp   a0, a1, [ap], #16
  ldp   a2, a3, [ap], #16
  ldp   r0, r1, [rp]
  ldp   r2, r3, [rp, #16]

  mul   t0, a0, b0
  umulh a0, a0, b0
  mul   t1, a1, b0
  umulh a1, a1, b0
  mul   t2, a2, b0
  umulh a2, a2, b0
  mul   t3, a3, b0
  umulh a3, a3, b0
```

```
  adds  t0, r0, t0
  adcs  a0, r1, a0
  adcs  a1, r2, a1
  adcs  a2, r3, a2
  cinc  a3, a3, cs

  adds  t0, t0, CY
  adcs  a0, a0, t1
  adcs  a1, a1, t2
  adcs  a2, a2, t3
  cinc  CY, a3, cs

  stp   t0, a0, [rp], #16
  stp   a1, a2, [rp], #16
  sub   n, n, #1
  cbnz  n, L(top)
```

Notes

1. We want a full schedule for optimal performance.

Notes

1. We want a full schedule for optimal performance.
2. Every instruction sequence forms a dependency chains.

Notes

1. We want a full schedule for optimal performance.
2. Every instruction sequence forms a dependency chains.
3. Out-of-order execution and keeping check on dependency chains is important.

Notes

1. We want a full schedule for optimal performance.
2. Every instruction sequence forms a dependency chains.
3. Out-of-order execution and keeping check on dependency chains is important.
4. Number of ports for vital instructions is important (in this case, ports for `mul` and `umulh`).

Notes

1. We want a full schedule for optimal performance.
2. Every instruction sequence forms a dependency chains.
3. Out-of-order execution and keeping check on dependency chains is important.
4. Number of ports for vital instructions is important (in this case, ports for `mul` and `umulh`).
5. Carry-chains has a lower bound of one clock cycle per limb.

This list, however, is incomplete. Agner Fog provides a good overview, specialized for recent x86 processors [4].

However, `mpn_mul` and `mpn_mul_basecase` both has overhead and more branches than wanted when doing small to medium sized multiple precision arithmetic.

New ARM schoolbook multiplication implementation

Instead of GMP's `mpn_mul_1` + `mpn_addmul_1` sequence forming a schoolbook multiplication algorithm, we could instead half-hardcode routines representing the action of

$$r \leftarrow a \cdot b,$$

where $a \in \mathbb{Z}_{\geq 0}$ and b is an n-limbed whole number for some *fix* n.

mpn_mul/flint_mpn_mul on Apple M1

m/n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	4.69														
2	4.64	3.60													
3	4.00	2.96	2.67												
4	2.85	2.35	2.20	2.08											
5	3.01	2.14	1.91	1.96	1.93										
6	2.58	1.91	2.28	1.92	2.04	1.97									
7	2.31	1.72	1.89	1.62	1.76	1.78	1.80								
8	2.21	1.70	1.53	1.51	1.66	1.70	1.77	1.78							
9	1.94	1.62	1.57	1.57	1.58	1.64	1.65	1.69	1.77						
10	1.82	1.49	1.49	1.47	1.47	1.58	1.56	1.67	1.72	1.74					
11	1.75	1.43	1.41	1.45	1.42	1.47	1.47	1.53	1.56	1.52	1.55				
12	1.64	1.35	1.36	1.44	1.46	1.55	1.66	1.76	1.64	1.59	1.58	1.57			
13	1.60	1.32	1.34	1.34	1.43	1.46	1.51	1.47	1.45	1.46	1.48	1.51	1.54		
14	1.53	1.25	1.30	1.29	1.34	1.40	1.40	1.39	1.39	1.39	1.49	1.47	1.46	1.49	
15	1.52	1.30	1.30	1.30	1.34	1.36	1.32	1.34	1.34	1.35	1.34	1.35	1.36	1.36	1.36

Some notes on this result:

1. GMP does not provide a native `mpn_mul_basecase` for ARM, only native `mpn_mul_1` and `mpn_addmul_1`.
2. We only implement routines for $n < 16$.
3. Using this implementation with Karatsuba ($\mathcal{O}(n^{1.58})$), we outperform GMP until $n \approx 483$ on Apple M1. For reference, GMP starts using Toom-Cook 6.5 ($\mathcal{O}(n^{1.39})$) at $n = 446$ on Apple M1.

New x86 schoolbook multiplication implementation

While ARM enjoys having a three-argumented instructions, x86 does not have this luxury. Because of this, one of the most viable options to outperforming GMP is to fully hardcode every case schoolbook multiplication.

mpn_mul/flint_mpn_mul on Intel Skylake

m/n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	3.20															
2	3.41	2.50														
3	4.17	3.00	2.72													
4	3.72	2.25	2.53	2.15												
5	2.91	2.02	1.94	1.85	1.84											
6	2.43	1.86	1.74	1.60	1.57	1.54										
7	2.20	1.82	1.63	1.55	1.54	1.55	1.51									
8	2.01	1.78	1.64	1.56	1.55	1.53	1.54	1.50								
9	1.89	1.80	1.64	1.60	1.71	1.78	1.69	1.75	1.55							
10	1.90	1.79	1.61	1.62	1.75	1.78	1.71	1.75	1.53	1.44						
11	1.89	1.83	1.63	1.67	1.78	1.83	1.78	1.76	1.57	1.47	1.38					
12	1.83	1.74	1.61	1.67	1.78	1.83	1.74	1.78	1.56	1.45	1.40	1.32				
13	1.79	1.64	1.60	1.64	1.77	1.84	1.77	1.81	1.59	1.49	1.41	1.35	1.32			
14	1.85	1.58	1.53	1.62	1.74	1.83	1.75	1.80	1.58	1.45	1.39	1.36	1.31	1.26		
15	1.86	1.57	1.54	1.61	1.74	1.83	1.76	1.78	1.60	1.47	1.40	1.34	1.31	1.27	1.42	
16	1.85	1.58	1.55	1.65	1.79	1.85	1.76	1.81	1.61	1.49	1.40	1.34	1.29	1.26	1.29	1.35

Some notes on this result:

1. We still perform significantly better than GMP, even when it has a native `mpn_mul_basecase`.
2. We implement for $n \leq m \leq 16$.
3. Karatsuba with this implementation performs better than GMP up until $n \approx 230$ on Zen 3. On Zen 3, GMP starts using Toom-Cook 4 ($\mathcal{O}(n^{1.40})$) when $n > 130$.

Prospects

- Implement more (half-)hardcoded assembly routines.

Prospects

- Implement more (half-)hardcoded assembly routines. Perhaps use AI + superoptimizer to automate the process?

Prospects

- Implement more (half-)hardcoded assembly routines. Perhaps use AI + superoptimizer to automate the process?
- Think of more powerful routines that could be more performant in assembly versions. With more limbs: approximate reciprocals, division via approximate reciprocals, GCD?

I want to thank Andreas Enge and Fredrik Johansson for inviting me here, and also thank Andreas again for organizing this.
And thanks to Adélie Linux at `cfarm` for letting me benchmark on their Apple M1.

[1] Niels Möller and Torbjörn Granlund. "Improved Division by Invariant Integers". In: *IEEE Transactions on Computers* 60.2 (2011), pp. 165–175. DOI: 10.1109/TC.2010.143.

[2] Torbjörn Granlund. *GMP assembly chart*. URL: https://gmplib.org/devel/asm (visited on 06/15/2024).

[3] Dougall Johnson. *Firestorm Overview*. URL: https://dougallj.github.io/applecpu/firestorm.html (visited on 06/16/2024).

[4] Agner Fog. *Optimizing subroutines in assembly language*. 2023. URL: https://agner.org/optimize/optimizing_assembly.pdf (visited on 06/16/2024).