Fast basecases for arbitrary-size multiplication

Albin Ahlbäck[1] Fredrik Johansson[2]

[1]LIX, CNRS, École polytechnique

[2]Inria Bordeaux

31 January 2025

Outline

1. Basic multiple-precision arithmetic operations
2. Hardware overview and implementation
3. Results

Basic multiple-precision arithmetic operations

Let integers be on the form $a = \sum_{i=0}^{n-1} a_i \beta^i$ where $0 \leq a_i < \beta$.

Fundamentals are these naïve/schoolbook $\mathcal{O}(n)$ operations:

- Left and right shift: $r \leftarrow \lfloor a \cdot 2^e \rfloor$
- Addition and subtraction: $r \leftarrow a \pm b$
- $n \times 1$-multiplication: $r \leftarrow a \cdot b_0$ (`mul_1`)
- Addition of $n \times 1$-multiplication: $r \leftarrow r + a \cdot b_0$ (`addmul_1`)

Basecase multiplication

Full multiplication

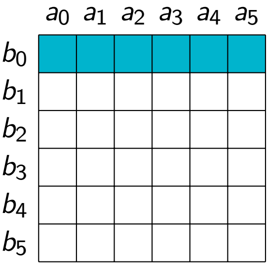
$$r \leftarrow a \cdot b = \sum_{i=0}^{n-1} a \cdot b_i \beta^i$$

can be carried out via

$r \leftarrow a \cdot b_0$ // `mul_1`
for $i \leftarrow 1$ *to* $n-1$ **do**
    $r \leftarrow r + (a \cdot b_i) \cdot \beta^i$ // `addmul_1`
end

where multiplication with β^i is trivial.

Visualizing 6×6 multiplication



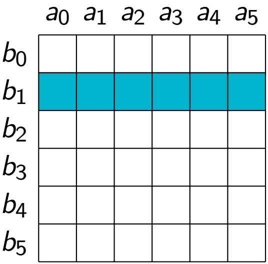
$r \leftarrow a \cdot b_0$

for $i \leftarrow 1$ *to* $n-1$ **do**

  $r \leftarrow r + (a \cdot b_i) \cdot \beta^i$

end

Visualizing 6×6 multiplication



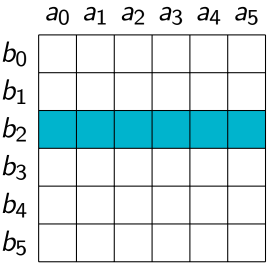
$r \leftarrow a \cdot b_0$

for $i \leftarrow 1$ *to* $n-1$ **do**

  $r \leftarrow r + (a \cdot b_i) \cdot \beta^i \quad (i = 1)$

end

Visualizing 6×6 multiplication



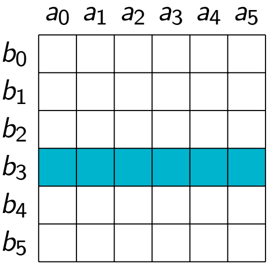
$r \leftarrow a \cdot b_0$

for $i \leftarrow 1$ *to* $n-1$ **do**

$\quad r \leftarrow r + (a \cdot b_i) \cdot \beta^i \quad (i = 2)$

end

Visualizing 6×6 multiplication



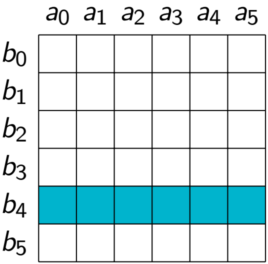
$r \leftarrow a \cdot b_0$

for $i \leftarrow 1$ *to* $n-1$ **do**

$\quad r \leftarrow r + (a \cdot b_i) \cdot \beta^i \quad (i = 3)$

end

Visualizing 6×6 multiplication



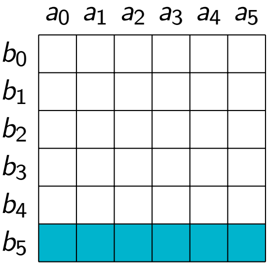
$r \leftarrow a \cdot b_0$

for $i \leftarrow 1$ *to* $n-1$ **do**

$\quad r \leftarrow r + (a \cdot b_i) \cdot \beta^i \quad (i = 4)$

end

Visualizing 6×6 multiplication



$r \leftarrow a \cdot b_0$

for $i \leftarrow 1$ *to* $n-1$ **do**

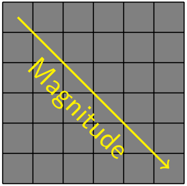
  $r \leftarrow r + (a \cdot b_i) \cdot \beta^i \quad (i = 5)$

end

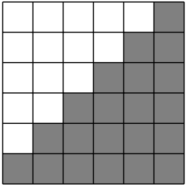
High multiplication

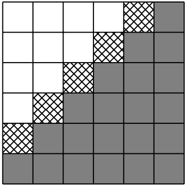
High multiplication is a multiplication where we scrap the lower part of the result, e.g. floating point arithmetic.

Typically, we want to compute the highest n words of an $n \times n$ product, where the full product would be contained in $2n$ words.



Naïve


Sloppy approximate


Precise approximate

 – high multiplication between two words u and v: $\lfloor uv/\beta \rfloor$

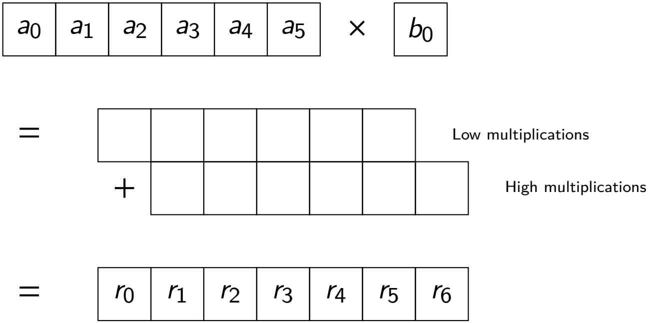
High multiplication

- Sloppy approximate yields an error of at most $(n-1)\beta^n$
- Precise approximate yields an error of at most $(2n-3)\beta^{n-1}$

Precise approximate contains only $n-1$ extra word-by-word high multiplications compared to sloppy approximate, but has far better precision!

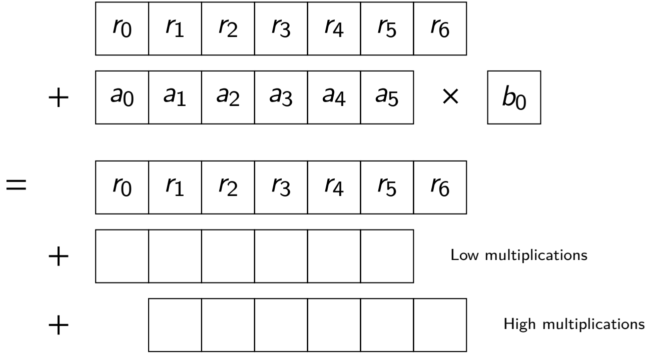
With precise approximate we can check if the upper n words are guaranteed to be correctly rounded.

Instructions for `mul_1`



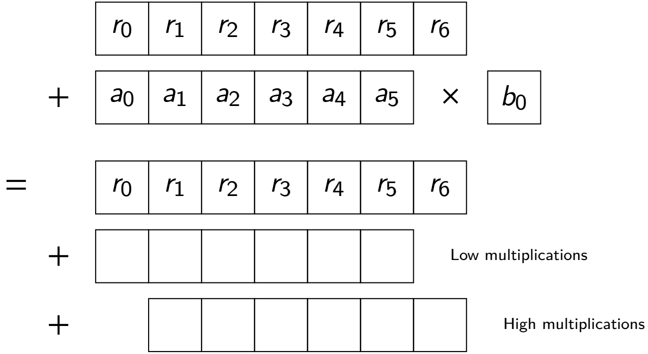
We need *low multiplication*, *high multiplication* and *addition with carry*.

Instructions for `addmul_1`



We need *low multiplication*, *high multiplication* and *addition with carry* (preferably with two separate carry flags).

Instructions for `addmul_1`



We need *low multiplication*, *high multiplication* and *addition with carry* (preferably with two separate carry flags).

Two separate carry flags $\Rightarrow$ lower bound of 1 cycle/n?

One carry flag $\Rightarrow$ lower bound of 2 cycles/n?

Architecture specifics

x86

- Two separate carry flags
- Has word-word full multiplication in one instruction

ARM

- One single carry flag
- Low multiplication and high multiplication are different instructions

$\Rightarrow$ ARM can only do one out of two carry chains in `addmul_1` at a time, while x86 do both at a time?

Simplified overview of CPU architecture

The main stages of a modern CPU:

1. Decoder
2. Branch prediction
3. Scheduler
4. Multiple units executing instructions

Simplified overview of CPU architecture

The main stages of a modern CPU:

1. Decoder
2. Branch prediction
3. Scheduler
4. Multiple units executing instructions

This enables:

- Scheduling instructions before branch is evaluated
- Out-of-order execution
- Concurrent execution of multiple instructions

Simplified overview of CPU architecture

The main stages of a modern CPU:

1. Decoder
2. Branch prediction
3. Scheduler
4. Multiple units executing instructions

This enables:

- Scheduling instructions before branch is evaluated
- Out-of-order execution
- Concurrent execution of multiple instructions

Things to be aware of:

- Dependency chains
- Overscheduled/bottlenecking units

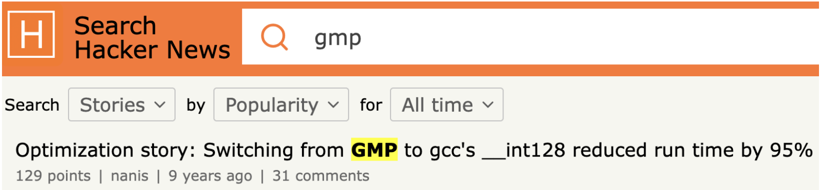
GMP versus MPFR versus FLINT

GMP's multiplication is loop-based, has handoptimized assembly code native to CPU, but lacks high multiplication.

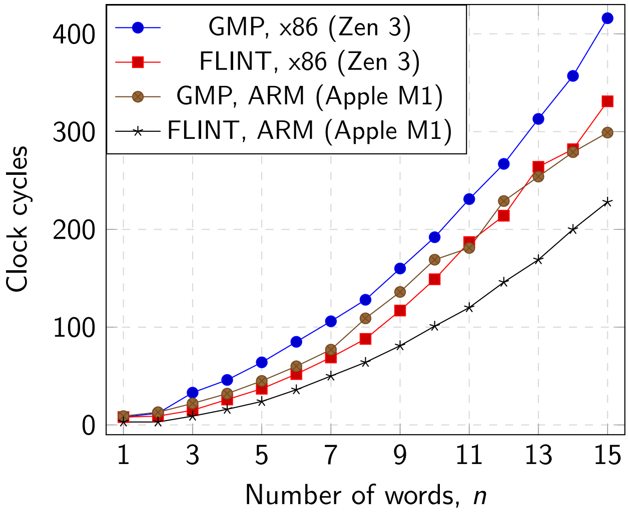
MPFR uses GMP as backend. It has sloppy approximate but not precise approximate.

FLINT low-level routines are mostly fully unrolled routines, implements both full multiplication and precise approximate.

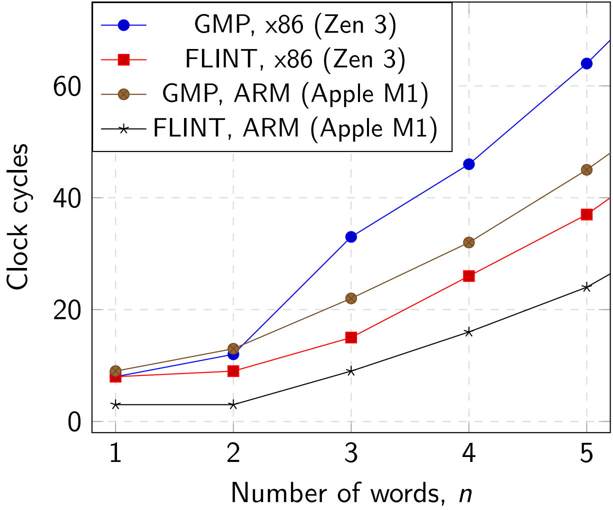
Funny headline



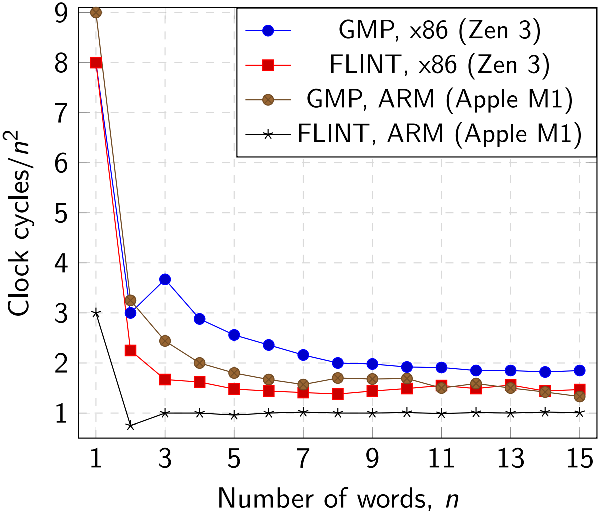
Results, full multiplication



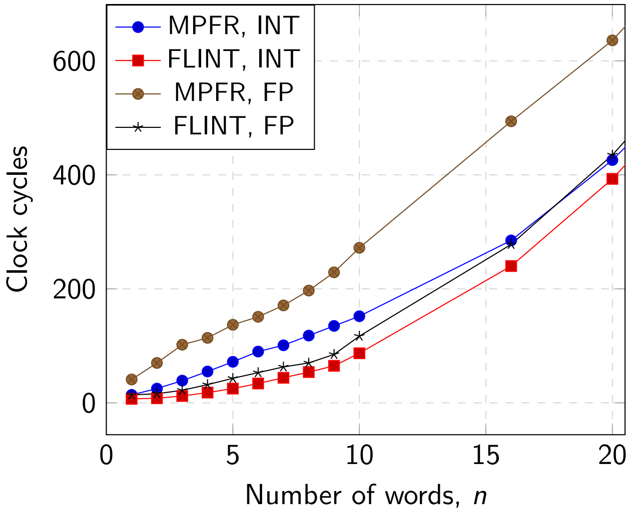
Results, full multiplication



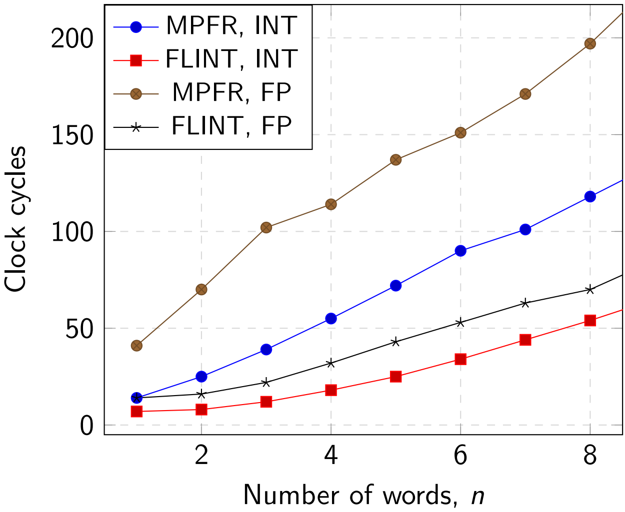
Results, full multiplication



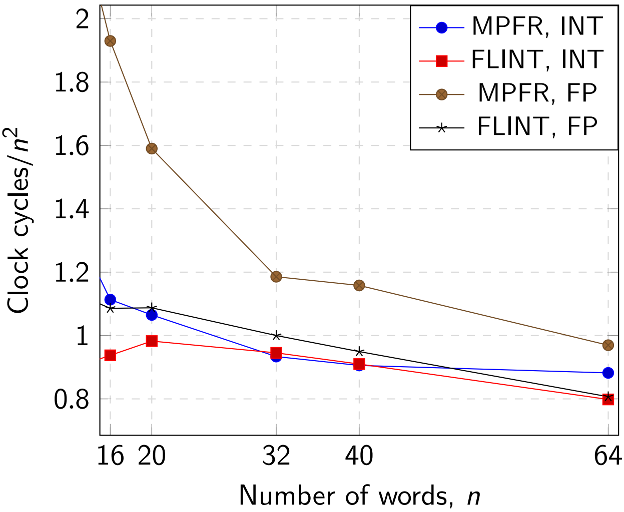
Results, high multiplication on Zen 3



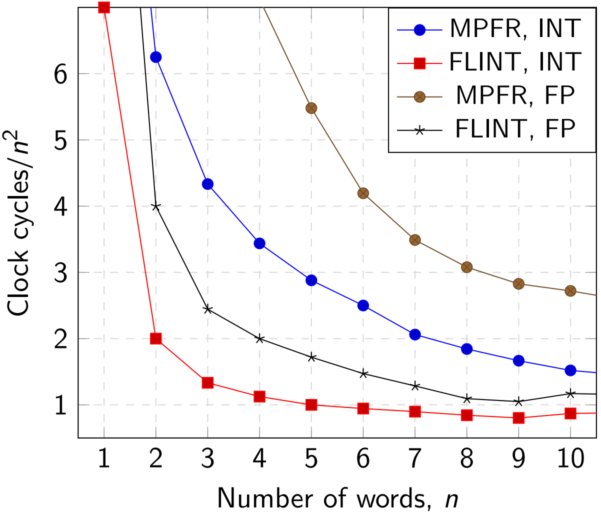
Results, high multiplication on Zen 3



Results, high multiplication on Zen 3



Results, high multiplication on Zen 3



Conclusions and thoughts

- Apple's ARM can actually perform multiple carry chains in parallel due to its scheduler

Conclusions and thoughts

- Apple's ARM can actually perform multiple carry chains in parallel due to its scheduler
- Straight line programs (SLPs) are important to reduce penalties when going from native data types to multiple precision arithmetic

Conclusions and thoughts

- Apple's ARM can actually perform multiple carry chains in parallel due to its scheduler
- Straight line programs (SLPs) are important to reduce penalties when going from native data types to multiple precision arithmetic
- Handwritten/"handgenerated" assembly remain important for multiple precision arithmetic due to poor compiler support