Generic 128-bit Math API

Marta Plantykow, Milena Olech, Alex Lobakin

Netdev 0x16

October 24, 2022

Agenda

Introduction

At this moment no 128-bit computer architecture exists. However, 128-bit operations exists for different purposes.

When such operations exist - CPU performs them natively

However, not every architecture does so and we need a fallback

Introduction

In this work, we propose a generic 128b Math API for the Linux kernel ready to be used in Precision Time Protocol (PTP) implementation.

128-bit-based variables allow performing calculations on large values with greater accuracy without the need for estimates.

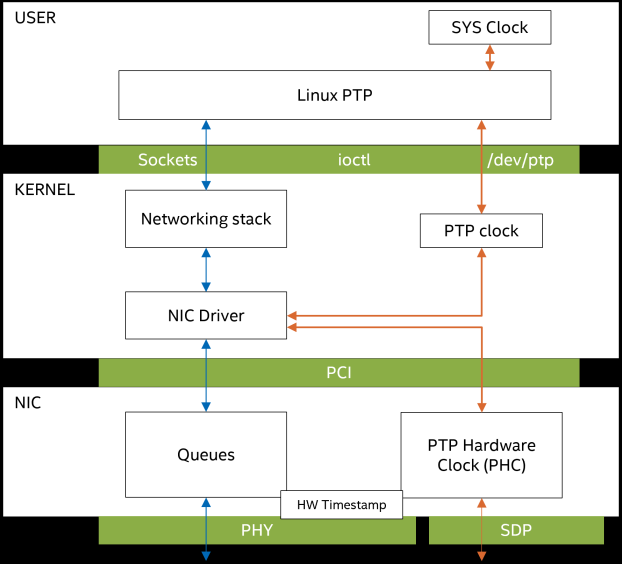
128-bit based applications

- Hardware performance accelerators - Streaming SIMD Extensions (SSE) - registers and instructions added to Intel (CPU) to improve video encoding and decoding.
- Graphic accelerators - In some implementations, it has a pathway 128 bits wide between its onboard processor and memory.
- Cryptography - The Advanced Encryption Standard (AES) algorithm can use cryptography keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.
- MD5 hashes produce 128-bit results
- ZFS is 128-bit filesystem
- IPv6 operates on 128-bit range of addresses

128-bit based applications

- Precision Time Protocol (IEEE 1588)
  - Defines a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems
  - Supports system-wide synchronization in the sub-microsecond range putting minimal requirements on network and local computing resources
  - The clocks within a system are organized into a leader-follower hierarchy, in which the clock located at the top of the hierarchy determines the reference time for the entire system
  - The protocol applies to both high-end and low-end devices

128-bit based applications



Mathematical background

- If the processor supports 128-bit-based native operations, no manual implementation is required
- Some architectures do not support 128-bit operations
- Most of them are 32-bit based, so it is crucial to implement fallback functions using 32-bit based mathematics
- 128-bit comparison, addition, and subtraction do not require complex algorithms

Mathematical background

128-bit processors are used for addressing up to 2^{128} (over 3.40×10^{38}) bytes.

This number is greater than the total data captured, created, or replicated on Earth as of 2018 which was approximated to be around 33 zettabytes (33×10^{21}).

Mathematical background

- Unsigned integer
  From 0 to
  $340,282,366,920,938,463,463,374,607,431,768,211,455$
- Signed integer
  From
  $-170,141,183,460,469,231,731,687,303,715,884,105,728$
  to
  $170,141,183,460,469,231,731,687,303,715,884,105,727$

128-bit multiplication and division

In case of division and multiplication, the following notation has been used [Knuth, 98]:

$$(...a_3a_2a_1a_0.a_{-1}a_{-2}...)_b = \tag{1}$$

$$... + a_3b^3 + a_2b^2 + a_1b^1 + a_0 + a_{-1}b^{-1} + a_{-2}b^{-2} + ... \tag{2}$$

The most straightforward generalizations of the decimal number system are received when we take b to be an integer greater than one and when $a's$ are required to be integers in the range of $0 \leq a_k < b$.

This gives the standard binary ($b = 2$), ternary ($b = 3$), quaternary ($b = 4$) number systems.

128-bit multiplication and division

$$(...a_3 a_2 a_1 a_0.a_{-1} a_{-2}...)_b = \tag{3}$$

$$... + a_3 b^3 + a_2 b^2 + a_1 b^1 + a_0 + a_{-1} b^{-1} + a_{-2} b^{-2} + ... \tag{4}$$

- The dot between a_0 and a_{-1} is called the *radix point*
- The $a's$ in equation 3 are called *digit of representation*
- The rightmost digit is called *least significant digit*
- The leftmost digit is called *most significant digit*

128-bit multiplication and division

Let's assume that we have two numbers $u = (u_{m+n-1} ... u_1 u_0)_b$ and $v = (v_{n-1} ... v_1 v_0)_b$.
The most crucial part is understanding of **radix-b notation** where b is the computer word size.

If we have an integer that fills 10 words on the computer whose word size is 10 ^{10} we receive:

1. 100 decimal digit
2. 10-place number to the base 10^{10}

Multiplication Algorithm

Given nonnegative integers $(u_{m-1}...u_1 u_0)_b$ and $(v_{n-1}...v_1 v_0)_b$, this algorithm forms their radix-b product $(w_{m+n-1}...w_1 w_0)_b$.

1. Initialize
   Set $w_{m-1}, w_{m-2}, ..., w_0$ all to 0. Set $j = 0$
2. Zero multiplier?
   If $v_j = 0$, set $w_{j+m} = 0$ and go to step 6.
3. Initialize i
   Set $i = 0$, $k = 0$
4. Multiply and add
   Set $t = u_i \times v_j + w_{i+j} + k$; then set $w_{j+k} = t \bmod b$ and $k = \lfloor \frac{t}{b} \rfloor$
5. Loop on i
   Increase i by one. Now, if $i < m$, go back to step 4; otherwise, set $w_{j+m} = k$
6. Loop on j
   Increase j by one. Now, if $j < n$, go back to step 2;, the algorithm terminates.

Division Algorithm

The difference between the algorithm and "pencil and paper method" is that this method creates partial products of $(u_{m-1}...u_1 u_0)_b \times v_j$ for $0 \leq j < n$ and adds these products at the end with appropriate scale factors.

Introduced algorithm does addition and multiplication simultaneously.

Division Algorithm

Given nonnegative integers $u = (u_{m+n-1} \ldots u_1 u_0)_b$ and $v = (v_{n-1} \ldots v_1 v_0)_b$, where $v_{n-1} \neq 0$ and $n > 0$, we form the radix-b quotient $\lfloor \frac{u}{v} \rfloor = (q_m q_{m-1} \ldots q_0)_b$ and the remainder $u \bmod v = (r_{n-1} \ldots r_1 r_0)_b$.

1. Normalize
   Set $d = \lfloor \frac{b-1}{v_{n-1}} \rfloor$. Then set $(u_{m+n} u_{m+n-1} \ldots u_1 u_0)_b$ equal to $(u_{m+n-1} \ldots u_1 u_0)_b$ times d. Similarly, set $(v_{n-1} \ldots v_1 v_0)_b$ equal to $(v_{n-1} \ldots v_1 v_0)_b$ times d.
2. Initialize j
   Set $j = m$.
3. Calculate $\widehat{q}$
   Set $\widehat{q} = \lfloor \frac{(u_{j+n} b + u_{j+n-1})}{v_n - 1} \rfloor$ and let $\widehat{r}$ be the remainder $(u_{j+n} b + u_{j+n-1}) \bmod v_{n-1}$. Not test if $\widehat{q} = b$ or $\widehat{q} v_{n-2} > b \widehat{r} + u_{j+n-2}$. If so, decrease $\widehat{q}$ by 1, increase $\widehat{r}$ by v_{n-1}, and repeat this test if $\widehat{r} < b$.

Division Algorithm

4. Multiply and subtract
   Replace $(u_{j+n}u_{j+n-1}...u_j)_b$ by

$$(u_{j+n}u_{j+n-1}...u_j)_b - \widehat{q}(v_{n-1}...v_1v_0)_b \tag{5}$$

   This computation consists of a simple multiplication by a one-place number combined with a subtraction. The digits $(u_{j+n}, u_{j+n-1}, ..., u_j)$ should be kept positive. If the result of this step is negative, $(u_{n+j}u_{j+n-1}...u_j)_b$ should be left as the actual value plus b^{n+1}, namely as the $b's$ complement of the actual value, and borrow to the left should be remembered.

Division Algorithm

5. Test remainder
   Set $q_j = \widehat{q}$. If the result of step 4 was negative, go to step 6. Otherwise, go on to step 7.
6. Add back
   Decrease q_j by 1, and add $(v_{n-1}...v_1 v_0)_b$ to $(u_{n+j} u_{j+n-1}...u_{j+1} u_j)_b$
7. Loop on j
   Decrease j by one. Now if $j \geq 0$, go back to 3.
8. Unnormalize
   Now $(q_m...q_1 q_0)_b$ is the desired quotient, and the desired remainder may be obtained by dividing $(u_{n-1}...u_1 u_0)_b$ by d.

Introduced API

The proposed API defines a structure that represents unsigned 128bit-based variables.

```
typedef union {
#ifdef __BIG_ENDIAN
  struct {
    u32   b127_96;
    u32   b95_64;
    u32   b63_32;
    u32   b31_0;
  };
  struct {
    u64   b127_64;
    u64   b63_0;
  };
#else /* __LITTLE_ENDIAN */
  struct {
    u32   b31_0;
    u32   b63_32;
    u32   b95_64;
    u32   b127_96;
  };
  struct {
    u64   b63_0;
    u64   b127_64;
  };
#endif /* __LITTLE_ENDIAN */
#ifdef __HAVE_INT128
  unsigned __int128 b127_0;
#endif /* __HAVE_INT128 */
} __u128;
```

Introduced API

Introduced functions are divided into following groups:

- Comparison
- Addition
- Subtraction
- Multiplication
- Divison

Introduced API

Division of unsigned 128bit dividend by 128bit divisor

$u64 \quad dividend\_high = 0x6767676721212121;$
$u64 \quad dividend\_low = 0x1243252265375421;$
$u64 \quad divisor\_high = 0x1111143454354354;$
$u64 \quad divisor\_low = 0x1111111114325342;$
$u128 \quad remainder;$
$u128 \quad result;$

$result = div\_u128\_u128(u128\_store(dividend\_high, dividend\_low),$
u128_store(divisor_high, divisor_low),
&remainder);

Performance Test

To measure the performance of introduced API, several tests were performed.

Following functions were chosen to be examined:

1. A function that operates on more than 64-bit values *ice_ptp_adjfine* from the Intel ice driver of the 5.19.5 Linux kernel (*algorithm1*)
2. The same function (*ice_ptp_adjfine*) from the 6.0 Release Candidate (*algorithm2*)
3. The native 128-bit function directly related to the PTP (*algorithm3*)

Performance Test

Test procedure:

- Each operation was repeated 10000 times
- Before and after each operation, the timestamp was taken
- Based on the time difference, expressed in nanoseconds, operation time was calculated
- Measurements were taken with and without the new API usage
- Each test was repeated ten times to provide stability and predictability
- To reduce the possible noise, interrupts were disabled while testing
- Average values were calculated and compared

Test results

Results for *algorithm1* with and without using 128bit API for 10000 iterations

	With 128	Without 128
Time[ns]	2910762	3479241
	2889556	3458588
	2898945	3456600
	2885530	3464868
	2885966	3456716
	2884493	3466790
	2888336	3468363
	2904135	3493585
	2886087	3457316
	2884718	3462869
Average[ns]	**2891852,8**	**3466413,6**
	Difference	**574560,8**

Test results

Results for *algorithm2* with and without using 128bit API for 10000 iterations

	With 128	Without 128
Time[ns]	2910762	2884022
	2889556	2886298
	2898945	2905804
	2885530	2884171
	2885966	2900811
	2884493	2905661
	2888336	2897499
	2904135	2887431
	2886087	2910105
	2884718	2885615
Average[ns]	**2891852,8**	**2894741,7**
	Difference	**2888,9**

Test results

Results for *algorithm3* with and without using 128bit API for 10000 iterations

	Native ops	Fallbacks
Time[ns]	2893146	2910706
	2894902	2882109
	2903383	2906288
	2891043	2899066
	2890052	2908561
	2885330	2900073
	2888230	2886179
	2884972	2887796
	2905913	2887784
	2888076	2891369
Average[ns]	**2892504,7**	**2895993,1**
	Difference	**3488,4**

Test results

∗ 128-bit API delivers better results in all tested scenarios.

∗ Although the primary goal of the API introduction was not to improve the performance, but to introduce generic API, this change did not negatively affect performance.

∗ Operation time was reduced by up to 547,5 μs per 10,000 operations.

Future work

1. The code will be submitted to the Linux kernel Mailing Lists.
2. Later works may include tree-wide conversions and switching more drivers and subsystems (crypto etc.) to this solution.

Summary

- Proposed solution is an **easy-to-use kernel API** for 128-bit operations
- For addition and subtraction **basic math operations** are used
- **Multiplication and division** require **dedicated algorithms**
- Tests prove that introduced API **does not degrade** analyzed functions' performance
- The major benefit of introduced API is **improvement of the calculations precision**

References

Donald E. Knuth (1998)
The art of computer programming
Stanford University

Q&A