Generic 128-bit Math API

Marta A Plantykow
Synopsys
Gdańsk, Poland
pmarta@synopsys.com

Milena Olech
Intel Technology Poland
Gdańsk, Poland
milena.olech@intel.com

Alex Lobakin
Intel Technology Poland
Gdańsk, Poland
alexandr.lobakin@intel.com

Abstract

This technology exists for other purposes despite the need to implement a 128-bit processor. In this work we propose a generic 128-bit math API for the Linux kernel.

Created API allows operating on 128-bit values using simple math operations in case of an addition, subtraction, and comparison, and more complex algorithms for division and multiplication.

Performance was tested on two functions that operate on more than 64-bit values. Tests proved that the developed API could deliver better results than the original functions, reducing the operation time.

The solution can be immediately used in Precision Time Protocol (PTP) implementation, in which precise calculations are crucial due to the rigorous phase-synchronization of telco requirements.

The proposed implementation is ready to be used for other purposes, such as Streaming SIMD [1] Extensions (SSE), in graphic accelerators or cryptography algorithms.

Introduction

As of 2022, no 128-bit computers are available on the market. (Groa) Moreover, such processors may never become available since there is no practical reason. 64-bit computers were introduced to solve the problems of Random Access Memory (RAM). 64-bit register can potentially manage up to 2^{64} bytes of RAM, 16 exabytes. (Groa) Nowadays, advanced servers designed for the most demanding applications support up to 6 TB of RAM (786GB). (RAM)

Even though there is no need to implement 128-bit operations, this technology exists for other purposes, such as hardware performance accelerators, graphic accelerators, or cryptography. Moreover, 128-bit-based variables allow performing calculations on large values with greater accuracy without the need for estimates.

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

The paper is divided into three main sections.

The Related work section describes 128-bit applications, focusing on Precision Time Protocol implementation in Linux kernel and mathematical theorem.

The following section describes Introduced changes and the analysis of their influence on PTP performance.

The third section describes Conclusions.

Finally, the last section describes the Future Work.

Related work

Precision Time Protocol

IEEE Standard 1588 defines a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. It 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. The clock at the top of the hierarchy determines the reference time for the entire system.

The standard provides administration-free operation and allows simple systems to be installed and operated without administrative attention. The protocol applies to both high-end and low-end devices. It operates by exchanging messages between the leader and the follower. (Gro20)

Streaming SIMD Extensions

Streaming SIMD Extensions (SSE) are a set of registers and instructions added to Intel Central Processing Units (CPU) to improve multimedia performance, such as video encoding and decoding. SSE works with all standard data types, including integers up to 128 bits. (Groc)

Graphic Accelerators

A graphics card is an integrated circuit that generates the video signal sent to a computer display. It contains a graphics processing unit (GPU), a digital-to-analog converter, and memory chips that store display data. (Bri) In some implementations, it has a pathway 128 bits wide between its onboard processor and memory. (Grob)

Cryptography

The Advanced Encryption Standard (AES) specifies a Federal Information Processing Standards (FIPS) approved cryptography algorithm used to protect electronic

[1] Single instruction, multiple data (SIMD) - the simplest method of parallelism (PA04)

data, encrypting and decryption information. The AES algorithm can use cryptography keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. (Dwo01)

Mathematical background

When the processor performs a system that supports 128-bit-based native operations, all mathematics and no manual implementation are needed. However, not every architecture does so. That is when fallback functions come to the rescue. Moreover, most of those architectures are 32-bit based, so it is crucial to implement fallback functions using 32-bit based mathematics. (kc)

The following subsections describe the theoretical background of 128-bit-based multiplication and division algorithms. Comparison, addition, and subtraction of two 128-bit values are basic mathematical operations that do not require complex algorithms. (Knu98)

In the case of multiplication and division, the following notation has been used:

$$(...a_3a_2a_1a_0a_{-1}a_{-2}...)_b = \quad (1)$$

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

The easiest 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.

Multiplication algorithm (Knu98)
Given nonnegative integers $(u_{m-1}...u_1u_0)_b$ and $(v_{n-1}...v_1v_0)_b$, this algorithm forms their radix-b product $(w_{m+n-1}...w_1w_0)_b$.

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 (Knu98)
Given nonnegative integers $u = (u_{m+n-1}...u_1u_0)_b$ and $v = (v_{n-1}...v_1v_0)_b$, where $v_{n-1} \neq 0$ and $n > 0$, we form the radix-b quotient $\lfloor \frac{u}{v} \rfloor = (q_m q_{m-1}...q_0)_b$ and the remainder $u \bmod v = (r_{n-1}...r_1r_0)_b$.

1. Normalize
   Set $d = \lfloor \frac{b-1}{v_{n-1}} \rfloor$. Then set $(u_{m+n}u_{m+n-1}...u_1u_0)_b$ equal to $(u_{m+n-1}...u_1u_0)_b$ times d. Similarly, set $(v_{n-1}...v_1v_0)_b$ equal to $(v_{n-1}...v_1v_0)_b$ 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 $\widehat{r} < b$.
4. Multiply and subtract
   Replace $(u_{j+n}u_{j+n-1}...u_j)_b$ by
   $$(u_{j+n}u_{j+n-1}...u_j)_b - \widehat{q}(v_{n-1}...v_1v_0)_b \quad (2)$$
   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.
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 $(v_{n-1}...v_1v_0)_b$ 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 $(q_m...q_1q_0)_b$ is the desired quotient, and the desired remainder may be obtained by dividing $(u_{n-1}...u_1u_0)_b$ by d.

Introduced changes

The proposed API defines a structure that represents unsigned 128bit-based variables along with load and store operations. It supports 32-bit and 64-bit-based architectures and use __int128 when supported by the platform (x86_64, ARM64, etc.). (kc)

```
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;
```

- u128_store

```
static inline __u128 u128_store(u64 high,
    u64 low)
{
  __u128 val = {
    .b127_64  = high,
    .b63_0    = low,
  };

  return val;
}
```

- u128_load

```
static inline u64 u128_load_high(__u128
    val)
{
  return val.b127_64;
}
static inline u64 u128_load_low(__u128 val
    )
{
  return val.b63_0;
}
```

A set of routines are defined:

- Comparison
  - *u*128_*et* checks if unsigned 128bit is equal to unsigned 128bit
  - *u*128_*gt* checks if unsigned 128bit is greater than unsigned 128bit
  - *u*128_*get* checks if unsigned 128bit is greater or equal to unsigned 128bit
  - *u*128_*lt* checks if unsigned 128bit is less than unsigned 128bit
  - *u*128_*let* checks if unsigned 128bit is less or equal to unsigned 128bit
- Addition
  - *add*_*u*128_*u*64 adds unsigned 64bit val to unsigned 128b val
  - *add*_*u*128_*u*128 adds unsigned 128bit val to unsigned 128b val
- Subtraction
  - *sub*_*u*128_*u*64 subtracts unsigned 64bit val from unsigned 128b val
  - *sub*_*u*128_*u*128 subtracts unsigned 128bit val from unsigned 128b val
- Multiplication
  - *mul*_*u*128_*u*64_*shr* unsigned 128bit multiplied by 64bit multiplier and shift right
  - *mul*_*u*128_*u*32_*shr* unsigned 128bit multiplied by 32bit multiplier and shift right
  - *mul*_*u*128_*u*128_*shr* unsigned 128bit multiplied with 128bit multiplier and shift right
  - *mul*_*u*128_*u*32 unsigned 128bit multiplied with 32bit multiplier
  - *mul*_*u*128_*u*32_*shr*_*fb* unsigned 128bit multiplied with 32bit multiplier and shift right
  - *mul*_*half*_*u*128_*u*64 unsigned 128bit multiplied with 64bit multiplier
  - *mul*_*u*128_*u*64 unsigned 128bit multiplied with 64bit multiplier
  - *mul*_*u*128_*u*64_*shr*_*fb* unsigned 128bit multiplied with 64bit multiplier and shift right
  - *mul*_*u*128_*u*128 unsigned 128bit multiplied with 128bit multiplier
- Division
  - *div*_*u*128_*u*32_*rem* unsigned 128bit division with 32bit divisor where a pointer to unsigned 128bit remainder is returned
  - *div*_*u*128_*u*32 unsigned 128bit division with 32bit divisor
  - *div*_*u*128_*u*64_*rem* unsigned 128bit division with 64bit divisor where a pointer to unsigned 128bit remainder is returned
  - *div*_*u*128_*u*64 unsigned 128bit division with 64bit divisor
  - *div*_*u*128_*u*128 unsigned 128bit division with 128bit divisor
  - *div*_*u*96_*u*96 unsigned 96bit division with 96bit divisor
  - *div*_*u*128_*u*96 unsigned 128bit division with 96bit divisor
  - *div*_*u*128_*rem* unsigned 128bit division with 128bit divisor where a pointer to unsigned 128bit remainder is returned
  - *div*_*u*128 unsigned 128bit division with 128bit divisor

Furthermore, a set of relevant kunit[2] tests is defined. Implemented tests can either run on kernel boot if built-in, or load as a module. KUnit follows the white-box testing approach. (doc) In such a scenario inputs for each of the functions are upfront defined in the tests along with expected outputs.

Methodology

Tests were performed to measure the efficiency of the introduced API.

[2]Kernel unit testing framework (KUnit) - provides a common framework for unit tests within Linux kernel. (doc)

As the first step, a function that operates on more than 64-bit values was chosen: *ice_ptp_adjfine* from the Intel ice driver of the 5.19.5 Linux kernel, later referred to as algorithm1. As a second, the same function was taken from the 6.0 Release Candidate kernel and the same tests were executed, referred as algorithm2. Finally, fallback implementations were compared against the native 128-bit ones, referred as algorithm3. The function is directly related to the Precision Time Protocol.

The test was divided into two parts - In the first part an operation was repeated 100 times and in the second part an operation was repeated 10000 times. Before and after each operation a timestamp was taken. Based on a time difference, expressed in nanoseconds, operation time was calculated. Measurements were taken with and without new API usage to determine the efficiency of the introduced API. Each test was repeated ten times to provide stability and predictability. It is important to mention that to reduce the possible noise interrupts were disabled while testing. Average values were calculated and compared. Outcome is presented in section Results.

Results

Table 1 and Table 2 show the results of performing algorithm 1 with and without 128bit usage for 100 and 10000 iterations respectively. Similarly, Table 3 and Table 4 show results of performing algorithm 2 with and without 128bit usage for 100 and 10000 iterations respectively, while Table 5 and 6 do that for the algorithm 3.

Table 1: Algorithm 1 - 100 iterations

	With 128	Without 128
Time[ns]	30986	36288
	30567	37174
	39572	35983
	30501	35531
	29645	37225
	30579	36753
	30662	36917
	30874	35918
	31076	36818
	30917	36694
Average[ns]	**30537,9**	**36529,1**
	Difference	**5991,2**

Table 2: Algorithm 1 - 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**

Table 3: Algorithm 2 - 100 iterations

	With 128	Without 128
Time[ns]	30986	30558
	30567	29817
	39572	29763
	30501	30862
	29645	30766
	30579	30920
	30662	31740
	30874	31549
	31076	31715
	30917	30536
Average[ns]	**30537,9**	**30822,6**
	Difference	**284,7**

In all tested cases, the developed API deliver better results. Although the major purpose of the API introduction was not to improve the performance, but to introduce generic API, this change did not negatively affect performance. Furthermore, operation time was reduced by up to 547,5 μs per 10,000 operations.

Conclusions

The developed solution provides an easy-to-use kernel API to support 128-bit operations. It may become a key feature considering the rapid development of Telco requirements. Created API allows operating on 128bit values using simple math operations in case of an addition, subtraction, and comparison and more complex algorithms for division and multiplication. Tests proved that introduced changes do not negatively influence analyzed functions' performance. Moreover, besides better performance, this change also improved the precision of calculations due to denomination elimination.

Table 4: Algorithm 2 - 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**

Table 5: Algorithm 3 - 100 iterations

	Native ops	Fallbacks
Time[ns]	30899	31509
	30532	29725
	30868	31896
	29578	31614
	30977	30931
	30681	31835
	29598	30471
	29701	31797
	31078	29881
	29634	31805
Average[ns]	**30354,6**	**31146,4**
	Difference	**791,8**

Future Work

Proposed API proves to be an effective method to perform 128-bit math calculations, so in the nearest future, the code will be submitted to the Linux kernel Mailing Lists. Later works may include treewide conversions and switching more drivers and subsystems (crypto etc.) to this solution.

Table 6: Algorithm 3 - 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**

References

[Bri] Encyclopedia Brittanica. Video card. `https://www.britannica.com/technology/video-card`. Accessed: 2022-8-29.

[doc] docs.kernel.org. Kunit - linux kernel unit testing. `https://docs.kernel.org/dev-tools/kunit/index.html`. Accessed: 2022-9-6.

[Dwo01] M.et all Dworkin. Advanced encryption standard (aes). *Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, [online]*, 2001.

[Groa] PCMag Digital Group. 128-bit computing. `https://www.pcmag.com/encyclopedia/term/128-bit-computing`. Accessed: 2022-8-29.

[Grob] PCMag Digital Group. 128-bit graphics accelerator. `https://www.pcmag.com/encyclopedia/term/128-bit-graphics-accelerator`. Accessed: 2022-8-29.

[Groc] PCMag Digital Group. Definition of sse. `https://www.pcmag.com/encyclopedia/term/sse`. Accessed: 2022-8-29.

[Gro20] Working Group PNCS Precise Networked Clock Synchronization Working Group. Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)*, pages 1–499, 2020.

[kc] The Linux kernel community. Elixir: config_arch_supports_int128. `https://elixir.bootlin.com/linux/v6.0-rc4/source/include/linux/math64.h#L157`. Accessed: 2022-9-6.

[Knu98] Donald E. Knuth. The art of computer programming. *Stanford University*, 1998.

[PA04] Wesley Petersen and Peter Arbenz. 85SIMD, Single Instruction Multiple Data. In *Introduction to Parallel Computing: A practical guide with examples in C*. Oxford University Press, 01 2004.

[RAM] Intel® xeon® platinum 8351n processor. `https://ark.intel.com/content/www/us/en/ark/products/212288/intel-xeon-platinum-8351n-processor-54m-cache-2-40-ghz.html`. Accessed: 2022-9-6.