УДК 681.3.06:519.248.681

Е. Г. КАЧКО, канд. техн. наук, А. Е. ТАРАКАНОВ

ОПТИМИЗАЦИЯ ОПЕРАЦИЙ МНОГОКРАТНОЙ ТОЧНОСТИ

В настоящее время широко используются криптографические методы для организации защиты информации. Особенностью современных криптографических методов является то, что алгоритмы преобразований известны. Секретными являются ключи. В связи с этим для исключения атаки, связанной с полным перебором ключевых данных, их длины выбираются большими. На сегодняшний день стандартными размерами ключевых данных являются 256, 512 и 1024 бит (в некоторых системах используются более длинные ключи). Поэтому при построении систем защиты проблема работы с длинными числами является актуальной. Безусловно, крайне важным требованием к подобным операциям является их максимальная скорость выполнения. В настоящее время существует большое количество различных библиотек многократной точности, доступных, например, в Интернете. Но было бы интересно узнать, так ли они быстры, как говорится в их документации. Ведь скорость работы программы зависит не только от выбранного алгоритма, но и от его реализации на конкретной машине. Что же касается оптимизации кода, то в этой сфере современные процессоры предоставляют огромное поле для деятельности. В первую очередь, это новые технологии, реализуемые на аппаратном уровне, такие, как расширения SSE2 процессора Intel Pentium 4.

Данная статья посвящена проблеме уменьшения вычислительной сложности арифметических операций над числами многократной точности. В частности, рассматриваются алгоритмы, наиболее оптимальные для длин чисел, применяющихся в большинстве криптографических методов, а также описание их эффективной реализации на современных процессорах.

Обзор существующих алгоритмов

Рассмотрим последовательно арифметические операции и методы, применяемые в настоящее время для их реализации при работе с длинными числами [2].

Сложение и вычитание

Повсеместно использующимся способом для выполнения данных операций является хорошо известный школьный метод сложения или вычитания «в столбик» с попарным сложением или вычитанием соответствующих разрядов чисел и последовательным учетом переносов. Время выполнения данного алгоритма линейно зависит от количества цифр числа, т.е. $T(n) = O(n)$. Трудно предложить какой-то иной, более эффективный способ выполнения сложения или вычитания длинных чисел.

Умножение

Классическим методом для выполнения данной операции также является школьное умножение в столбик. Все цифры одного числа последовательно умножаются на каждую цифру другого числа, выполняется учет переносов и сложение с результатами умножений на предыдущие разряды числа. Очевидно, что между временем выполнения алгоритма и длиной числа существует квадратичная зависимость: $T(n) = O(n^2)$. Несмотря на такую асимптотическую оценку, данный метод является одним из самых эффективных алгоритмов умножения чисел, применяющихся в большинстве криптографических протоколов. Во все библиотеки многократной точности непременно входит процедура вычисления произведения длинных чисел, основанная именно на умножении «в столбик».

Для выполнения данной операции можно также предложить следующую модификацию классического метода. Вместо обычного умножения, учета переноса и сохранения очередной цифры результата, можно выполнять накопление результата по столбцам и только после

суммирования всего столбца сохранять очередную цифру ответа [5]. При использовании чисел многократной точности с длиной цифры, меньшей максимального размера слова компьютера, данный подход дает реальный выигрыш в скорости за счет уменьшения количества обращений к памяти и команд внутри главного цикла процедуры. Однако в случае равенства длины цифры и максимального слова процессора ситуация изменяется. Даже при условии, что ведется умножение чисел ограниченной длины, для хранения суммы столбца возникает потребность в некоторой ячейке тройной точности. А это связано с дополнительными обращениями к памяти, что негативно сказывается на производительности алгоритма.

Еще одним методом умножения является алгоритм, основанный на использовании следующей формулы:

$$U \cdot V = (2^{2n} + 2^n) \cdot U_1 \cdot V_1 + 2^n \cdot (U_1 - U_0) \cdot (V_0 - V_1) + (2^n + 1) \cdot U_0 \cdot V_0 .$$

Т.е. умножение двух $2n$-разрядных чисел сводится к трем операциям умножения n-разрядных чисел $U_1 \cdot V_1$, $(U_1 - U_0) \cdot (V_0 - V_1)$ и $U_0 \cdot V_0$, а также к нескольким операциям сложения и вычитания. Поскольку операция умножения является одной из самых медленных арифметических операций в компьютере, то уже замена четырех умножений на три теоретически должна давать преимущества. Важно то, что на основе данной формулы можно построить рекурсивный алгоритм, дающий время выполнения $T(n) = O(n^{\log 3}) = O(n^{1,585})$ вместо традиционного n^2 [1]. Однако при реализации данного метода возникают дополнительные расходы на рекурсию и взаимодействие с временной памятью, сводящие на-нет его преимущества на многих процессорах. Несмотря на это, на отдельных компьютерах (при определенной оптимизации кода процедуры) данный алгоритм дает выигрыш.

Что касается таких методов умножения, как алгоритм Тоома-Кука, быстрое преобразование Фурье [1], то, хотя их асимптотические оценки, несомненно, лучше, чем у алгоритмов, рассмотренных выше, фактически же их применение для чисел длиной не более 1024 бит не дает никаких преимуществ. Напротив, традиционные алгоритмы являются более эффективными.

Умножение на число однократной точности

Эта арифметическая операция является частным случаем операции умножения длинных чисел. Наиболее эффективным алгоритмом является все то же умножение «в столбик», дающее в данном случае линейную асимптотическую характеристику.

Деление

Основным алгоритмом, применяющимся для деления длинных чисел, является традиционный школьный метод, на каждом шаге которого на основании старших цифр делимого и делителя находится очередная цифра частного, которая умножается на все цифры делителя и вычитается из делимого. Время выполнения данного алгоритма дает квадратичную зависимость.

Для выполнения операции деления используется также метод, основанный на том, что $\frac{U}{V} = U \cdot \frac{1}{V}$, т.е. для того, чтобы разделить U на V, можно найти приближение к $\frac{1}{V}$ и умножить данную величину на U [1]. Для нахождения обратной величины используется метод Ньютона. Вычислительная сложность получения обратного элемента достаточно велика, и данный алгоритм может иметь смысл, если необходимо многократно выполнять деление на одно и тоже число. Недостатком данного способа является и то, что вычисление остатка от деления (данная операция необходима в криптографии) также затруднительно.

Приведенный анализ показывает, что в настоящее время не известны алгоритмы, позволяющие существенно увеличить скорость вычислений для арифметических операций по

сравнению с традиционными методами. В связи с этим, для уменьшения их вычислительной сложности необходимо максимально учитывать аппаратные особенности современных процессоров. При оптимизации необходимо исходить из того, что разработанный код должен быть наиболее эффективным для данного типа процессора и процессоров других типов.

Характеристика разработанной библиотеки

В ходе исследований была разработана библиотека функций, реализующих основные арифметические операции над числами многократной точности, а именно:

- сложение;
- вычитание;
- умножение;
- умножение на число однократной точности;
- деление с получением частного и остатка.

Код библиотеки написан исключительно на языке ассемблера для обеспечения максимальной эффективности. С целью достижения переносимости кода при разработке использовался компилятор NASM, позволяющий получать объектные файлы как для Windows, так и для Linux.

В библиотеке имеются две процедуры, реализующие операцию умножения. Во-первых, это функция, выполняющая умножение «в столбик». Во-вторых, это функция, использующая рекурсивный алгоритм, требующий трех операций умножения и нескольких операций сложения на каждом уровне рекурсии вместо обычных четырех. Как уже отмечалось, непосредственная реализация рекурсии и ряда циклов, имеющихся в алгоритме, приводит к тому, что скорость выполнения такой программы в несколько раз меньше скорости работы классического метода.

Однако существует иной подход. Используя, например, макросредства, поддерживаемые большинством существующих ассемблеров, и ориентируясь на некоторую определенную длину числа, можно заменить физические рекурсивные вызовы функций и циклические участки программы вызовами макросов. В итоге, после препроцессорной обработки может быть получена программа с полностью развернутыми циклами и рекурсивными обращениями. Несмотря на значительный размер кода такой функции (для чисел длиной 1024 бита – более 40 Кб), на современных процессорах (AMD Athlon XP, Intel Pentium 4) в ряде случаев был получен выигрыш в скорости примерно на 15-25% по сравнению с умножением «в столбик». Необходимо заметить, что применение чисел фиксированной длины оправдано тем, что современные криптографические стандарты предполагают работу именно с такими числами. Таковым, например, является ГОСТ 34.310-95.

Библиотеки, участвующие в тестировании

Анализ скорости выполнения реализованных операций проводился в сравнении с функциями существующих библиотек. В тестировании участвовали следующие библиотеки:

- Библиотека MIRACL фирмы Shamus Software. В ней реализованы практически все операции, используемые в современных криптографических системах. Библиотека написана на C/C++. Благодаря наличию ряда мер, направленных на оптимизацию ее функций (использование ассемблерных вставок, макросредств), считается одной из наиболее быстрых и эффективных библиотек.
- Библиотека многократной арифметики ММАТН АО «ИИТ». Представляет собой набор основных функций, применяемых в криптографии. Будучи написанной полностью на языке ассемблера, она является неплохим образцом для сравнения с процедурами разработанной библиотеки.

Результаты тестов

С целью анализа производительности разработанной библиотеки было выполнено сравнение скорости выполнения ее функций с аналогичными функциями библиотек MMATH и MIRACL. Конфигурации компьютеров, принимавших участие в тестировании, приведены в табл. 1. Далее подробно рассмотрены результаты тестирования каждой арифметической операции.

Таблица 1

Обозначение	Конфигурация
[1]	Intel Pentium MMX, 166 МГц, 16 Мб RAM, ОС Windows 98
[2]	AMD Athlon XP 1800+, 1533 МГц, 256 Мб RAM, ОС Windows 2000
[3]	Intel Pentium 4, 2 ГГц, 512 Мб RAM, ОС Windows XP

Сложение и вычитание

Проверка скорости выполнения данных операций выполнялась на случайным образом сгенерированных числах длинами 32*4 байт (первый операнд) и 8*4, 16*4, 24*4, 32*4 байт (второй операнд). Количество итераций цикла – 1000000 раз. Полученные результаты приведены в таблицах 2 (для операции сложения) и 3 (для операции вычитания).

Таблица 2

CPU	Длины операндов	Время выполнения (мс)		
		Разработанная библиотека	MMATH	MIRACL
[1]	32*4 × 8*4	1606	3219	5279
	32*4 × 16*4	1605	2829	7236
	32*4 × 24*4	1606	2926	9183
	32*4 × 32*4	1605	2926	11197
[2]	32*4 × 8*4	78	190	292
	32*4 × 16*4	88	201	389
	32*4 × 24*4	86	205	477
	32*4 × 32*4	89	205	567

Таблица 3

CPU	Длины операндов	Время выполнения (мс)		
		Разработанная библиотека	MMATH	MIRACL
[1]	32*4 × 8*4	1602	3354	4830
	32*4 × 16*4	1603	2944	6531
	32*4 × 24*4	1640	3033	8137
	32*4 × 32*4	1692	3063	9633
[2]	32*4 × 8*4	84	181	297
	32*4 × 16*4	97	191	421
	32*4 × 24*4	97	189	516
	32*4 × 32*4	105	194	592

Как следует из экспериментально полученных результатов, быстродействие функции сложения разработанной библиотеки в среднем в 1,9 – 5,1 раз (на CPU Intel Pentium MMX) и 2,4 – 5 раз (на CPU AMD Athlon XP) превышает быстродействие аналогичных функций рассмотренных библиотек.

Что касается операции вычитания разработанной библиотеки, то ее выигрыш в скорости составляет в среднем 1,9 – 4,4 раз (на CPU Intel Pentium MMX) и 2 – 4,7 раз (на CPU AMD Athlon XP).

Следует отметить, что такое отставание функций сложения и вычитания библиотеки MIRACL от процедур разработанной библиотеки (да и библиотеки MMATH) связано с тем, что их код написан полностью на C/C++. Значительное же преимущество разработанных процедур объясняется их тщательной низкоуровневой оптимизацией [3,4].

Умножение на число однократной точности

Анализ скорости выполнения данной операции осуществлялся на числах длиной 32*4 байт (первый операнд) и 4 байта (второй операнд). Число итераций цикла – 1000000 раз. В тесте принимала участие функция разработанной библиотеки, а также библиотеки MIRACL. Полученные результаты приведены в табл. 4.

Таблица 4

CPU	Длины операндов	Время выполнения (мс)	
		Разработанная библиотека	MIRACL
[1]	32*4 × 4*4	2943	5382
[2]	32*4 × 4*4	157	298

Из экспериментальных результатов можно заключить, что быстродействие разработанной процедуры в 1,8 раз (на CPU Intel Pentium MMX) и в 1,9 раз (на CPU AMD Athlon XP) превышает быстродействие функции библиотеки MIRACL, написание которой на C/C++ не лучшим образом сказалось на ее скорости выполнения.

Умножение

Производительность данной операции оценивалась на случайных числах длиной 32*4 байт (оба операнда). Число итераций цикла – 100000 раз. Таблица 5 демонстрирует полученные результаты.

Таблица 5

CPU	Время выполнения (мс)			
	Разработанная библиотека		MMATH	MIRACL
	«В столбик»	Макро		
[1]	9638	23600	10206	10358
[2]	473	423	553	497

Как следует из экспериментально полученных результатов, быстродействие функции умножения, реализованной на основе алгоритма умножения «в столбик», разработанной библиотеки на 6–7% (на CPU Intel Pentium MMX) и 5–15% (на CPU AMD Athlon XP) превышает быстродействие аналогичных функций MMATH и MIRACL.

Что касается процедуры умножения, основанной на полной раскрутке циклов и рекурсии, то, хотя ее производительность в 2,2 раза меньше на CPU Intel Pentium MMX, на процессоре AMD Athlon XP получен реальный выигрыш в скорости на 15-24%.

Необходимо отметить довольно тщательную проработку функции умножения библиотеки MIRACL – за счет ассемблерных вставок в ее главном цикле она продемонстрировала достойные скоростные характеристики (обогнать ее даже на 6-7% было не так легко).

Деление

Проверка скорости выполнения данной операции (также как и процедур сложения и вычитания) выполнялась на случайным образом сгенерированных числах длинами 32*4 байт (первый операнд) и 8*4, 16*4, 24*4, 32*4 байт (второй операнд). Количество итераций цикла – 100000 раз. Результаты тестирования приведены в табл. 6.

Таблица 6

CPU	Длины операндов	Время выполнения (мс)		
		Разработанная библиотека	MMATH	MIRACL
[1]	32*4 × 8*4	4185	9021	5515
	32*4 × 16*4	4705	8863	5687
	32*4 × 24*4	3481	6652	4988
	32*4 × 32*4	517	1812	448
[2]	32*4 × 8*4	295	459	335
	32*4 × 16*4	330	460	347
	32*4 × 24*4	241	347	325
	32*4 × 32*4	34	108	77

Из результатов эксперимента видно, что быстродействие функции деления разработанной библиотеки в среднем в 1,2–2,4 раз (на CPU Intel Pentium MMX) и 1,5–1,9 раз (на CPU AMD Athlon XP) превышает быстродействие аналогичных функций библиотек MMATH и MIRACL.

Все та же низкоуровневая оптимизация и тщательная проработка кода разработанной процедуры позволила добиться такого выигрыша.

Использование расширений SSE2

С выходом в свет процессора Intel Pentium 4, а точнее, его расширений SSE2 [3,4] появилась уникальная возможность увеличения скорости выполнения операций над длинными числами. В отличие от SIMD-расширений предыдущих процессоров (MMX, SSE), блок команд SSE2 позволяет выполнять одновременные операции над целыми числами длиной до 64 бит. Следует подчеркнуть, что большую ценность из себя представляют именно целочисленные команды SSE2. Благодаря их наличию, цифра числа многократной точности может рассматриваться как целое число длиной 32 бита для любых арифметических операций. Что касается расширений MMX, то их использование ограничивает размер цифры 16 битами. В расширения SSE же вообще не входят команды, оперирующие над целочисленными данными. Наиболее примечательными командами блока SSE2 являются инструкции, расширяющие размер операндов целочисленного блока MMX до 128 бит. Например, такие команды, как paddq, psubq выполняют одновременное сложение или вычитание двух 64-разрядных беззнаковых целых чисел, а команда pmuludq – умножение двух 32-разрядных беззнаковых целых чисел с получением двух 64-битных результатов.

Важной особенностью расширений SSE2 является наличие в их составе команд управления кэшированием. Использование этих команд совместно с потоковыми инструкциями пересылки данных позволяет достичь максимальной скорости выполнения программы.

В ходе исследований проверка производительности команд блока SSE2 осуществлялась на примере операции копирования памяти. Было разработано две программы: одна – на основе базовых инструкций процессора, другая – с использованием SSE2. В первом случае для копирования блока памяти применялась обычная строковая команда rep movsd, во втором имел место следующий алгоритм: страница памяти (4096 байт) предварительно помещалась в кэш второго уровня 32-мя командами prefetchnta (размер линейки кэша второго уровня – 128 байт) и далее последовательно перемещалась командами movdqa и movntdq с целью минимального загрязнения кэша. Проверка программы осуществлялась на блоке данных длиной 100 страниц (100*4096 байт) с числом итераций цикла 10000 раз. Следует заметить, что для более полного анализа скорости выполнения программы тестирование проводилось не только под управлением операционной системы, но и в «голом» защищенном режиме, а именно, с запрещенными аппаратными и программными прерываниями, flat-памятью и отключенной страничной адресацией. Результаты тестирования приведены в табл. 7.

Таблица 7

CPU	Среда	Время выполнения (с)	
		Базовые команды	SSE2
[2]	ОС	15,44	——
	РМ	16,26	——
[3]	ОС	9,9	3,83
	РМ	8,55	2,49

Как следует из результатов тестирования, операция копирования памяти, использующая команды SSE2, в среднем в 2,9 раз быстрее аналогичной операции, выполняемой на процессоре Intel Pentium 4, и в 5 раз быстрее операции, выполняемой на процессоре AMD Athlon XP. Интересно отметить тот факт, что время выполнения операции в непосредственном защищенном режиме на процессоре AMD Athlon XP больше времени выполнения программы под управлением ОС. Что касается Intel Pentium 4, то в этом случае наблюдается совершенно противоположная картина.

Результаты тестирования показали, что использование блока SSE2 (а точнее, команд управления кэшированием), несомненно, дает реальные преимущества при работе с большими объемами памяти. Но что можно сказать о применении расширений SSE2 собственно в программировании арифметических операций над числами многократной точности?

В разработанной библиотеке была реализована процедура умножения длинного числа на 32-битное число однократной точности, использующая целочисленные команды блока SSE2. В основу программы положен стандартный алгоритм умножения «в столбик». Следует отметить следующие ключевые моменты. Длина цифры числа многократной точности принимается равной 128 бит. Это означает, что в программе используются команды пересылки 128-битных данных (movdqa, movntdq), т.е. в четыре раза сокращается число обращений к памяти по сравнению с аналогичной процедурой, использующей базовые 32-битные команды. Умножение 128-битного числа (или четырех 32-битных чисел) на 32-битное с получением четырех 64-битных результатов выполняется двумя командами pmuludq вместо обычных четырех инструкций mul. Главной проблемой, возникшей при программировании процедуры, является учет переносов. Из-за отсутствия в блоке SSE2 встроенных средств контроля переполнения при выполнении арифметических операций, а также за неимением инструкции беззнакового сравнения по больше/меньше среди его команд, реализация учета переносов оказалась довольно громоздкой. Команды, осуществляющие данную функцию, составляют примерно 60% от общего числа команд главного цикла процедуры.

Программа была реализована в двух вариантах: с использованием команд управления кэшированием и без их использования. Первый вариант ориентирован на работу с длинными числами, размеры которых кратны странице (4096 байт): на каждой итерации цикла осуществляется перемещение текущей страницы в кэш второго уровня командами prefetchnta, после чего выполняется собственно умножение с сохранением каждой цифры результата командой movntdq.

Было произведено сравнение производительности данных программ и процедуры, реализованной на основе базовых инструкций. Первый вариант программы тестировался на числах длиной 4096*10 байт с количеством итераций цикла 100000 раз, второй вариант – на числах длиной 32*4 байт и числом итераций 1000000 раз. Анализ скорости выполнения осуществлялся как под управлением операционной системы, так и непосредственно в защищенном режиме. Полученные результаты приведены в табл. 8.

Таблица 8

CPU	Среда	Время выполнения (мс)		Время выполнения (с)		
		32*4 × 4*4		10*4096 × 4*4		
		Базовые команды	SSE2	Базовые команды	SSE2	SSE2 и управление кэшированием
[2]	ОС	157	——	4,62	——	——
	РМ	193	——	5,4	——	——
[3]	ОС	414	203	12,35	8,16	6,2
	РМ	249	202	7,17	7,91	6,15

Исходя из экспериментально полученных результатов, можно сделать следующие выводы. Во-первых, для размеров чисел 32*4 байт преимущества программы, основанной на SSE2, перед процедурой, выполненной на базе основных команд, очевидны. Что касается длин чисел 10*4096 байт, то здесь наблюдается та же картина: программа, использующая расширения SSE2 вместе с командами управления кэшированием, оказалась быстрее процедуры, реализованной только на базе арифметических команд SSE2, а та, в свою очередь, – обычной функции, включающей в себя основные инструкции процессора. Однако это справедливо лишь для программ, тестируемых под управлением ОС. В непосредственном защищенном режиме при выполнении тестов на числах длиной 32*4 байт традиционная процедура вплотную «подбирается» к программе, использующей SSE2, а на числах с размерами 10*4096 байт – даже ее обгоняет. Во-вторых, следует констатировать тот факт, что процессору Intel Pentium 4 так и не удалось обогнать AMD Athlon XP. Это касается как программы, использующей базовые команды, так и процедуры, выполненной на основе SSE2. Похоже, что давно высказываемое мнение о низкой производительности ядра процессора Intel Pentium 4, отвечающего за выполнение основных инструкций, является небезосновательным.

Таким образом, использование низкоуровневой оптимизации позволяет существенно уменьшить вычислительную сложность всех арифметических операций над длинными числами для наиболее распространенных типов современных процессоров.

Список литературы: 1. *Кнут Д.* Искусство программирования. Т. 2. М: Мир, 1977. 728 с. 2. *Качко Е.Г., Свинарев А..В., Горбенко ИД., Мельникова О.А.* Программирование операций многократной точности // Безопасность информации. К.: 1995. .№ 1. С. 18 – 22. 3. *Intel Corporation.* IA-32 Intel Architecture Software Developer's Manual. Vol. 1 – 3. 4. *Intel Corporation.* Intel Pentium 4 and Intel Xeon Processor Optimization. 5. *Gourdon X.* Arbitrary precision computation. http://numbers.computation.free.fr/Constants/constants.html.

Харьковский национальный университет радиоэлектроники

Поступила в редколлегию 29.04.2003