без их дублирования на бумаге.

Литература: 1. Шнайер Б. Прикладная криптография – М.: изд-во «Триумф», 2002. 2. ГОСТ Р 34.10-94. Информационная технология. Криптографическая защита информации. Процедуры выработки и проверкиэлектронной цифровой подписи на базе асимметричного криптографического алгоритма. – М.: Госстандарт России, 1994. 3. ГОСТ Р 34.11-94. Информационная технология. Криптографическая защита информации. Функция хэширования. – М.: Госстандарт России, 1994. 4. Материалы по НИР "Разработка проекта законодательного акта о юридическом статусе электронной документации (шифр "Статус"). – М.: Институт государства и права РАН, 1994.

УДК 004.056.53

МОДУЛЬНОЕ УМНОЖЕНИЕ В КРИПТОГРАФИИ РЕАЛЬНОГО ВРЕМЕНИ

Сергей Илюшин, Олег Павлов*
Предприятие "САЙКОМ"
**НТУУ "КПИ"*

Анотація: **Розглянуто алгоритми модульної арифметики, що застосовуються в криптографії. Проаналізовано базові операції. Запропановано узагальнюючий алгоритм, який названо Radix-Z. Знайдена залежність кількості операцій алгоритма Radix-Z від значення розщеплення Z. Наведено текст програми Radix-Z модульного множення мовою Сі. Надано характеристики асемблерної реалізації запропонованого алгоритма.**

Summary: **Efficient real-time cryptography modular multiplication algorithms are observed. Its basic operations are analyzes. A new common Radix-Z modular multiplication algorithm is proposed. A correlation between operation complicity and radix value is found. Source C code of Radix-Z modular multiplication is present. The typical cycle terms of ADSP-218x assembler release are given an example.**

Ключові слова: **Модульна арифметика, криптографія, RSA.**

I Введение

Защита информации начинает играть все более значимую роль в современных компьютерных и коммуникационных системах. Основными требованиями, предъявляемыми к устройствам защиты информации, являются обеспечение конфиденциальности, аутентичности, целостности и достоверности. Для решения этих задач в большинстве случаев используются либо симметричные ключевые алгоритмы, либо несимметричные — с открытым ключом. Наибольший интерес, на наш взгляд, представляют устройства защиты информации, в которых используется именно несимметричные криптографические алгоритмы.

Среди различных криптографических алгоритмов с открытым ключом, наиболее часто применяемых в устройствах защиты информации, можно выделить алгоритмы RSA, Диффи-Хеллмана, Эль-Гамаля, криптостойкость которых проверена лучшими криптоаналитиками мира. Неотъемлемой частью этих алгоритмов являются арифметические операции модульного умножения и модульного возведения в степень. Используемые в этих операциях числа имеют сотни и даже тысячи бит, что позволяет обеспечить высокую криптостойкость защищаемых данных. В то же время применение чисел с большим количеством бит приводит не только к улучшению криптостойкости, но и к замедлению работы криптографического устройства в целом. Данное обстоятельство является особенно критичным в случае необходимости обеспечения защиты данных в реальном времени, например, зашиты переговоров по открытым каналам связи. Поэтому оптимизация алгоритмов модульной арифметики в целом, а также с учетом характеристик и архитектуры конкретных вычислительных устройств на сегодняшний день является достаточно актуальной задачей. Можно ожидать, что эта задача не потеряет своей актуальности и через несколько лет, несмотря на стремительный прогресс в области создания сверхбыстрых процессоров.

Для решения данной задачи было затрачено немало усилий различными специалистами мира [1 – 13].

Мы надеемся, что наша публикация займет свое скромное место в этом достойном ряду.

II Анализ алгоритмов модульного умножения

Среди различных алгоритмов модульного умножения (умножения числа A на число B по модулю M, $(A \cdot B) \bmod (M)$, или еще более компактно $(A \cdot B)\%M$) в качестве исходных нами были отобраны

классический алгоритм модульного умножения и алгоритм Монтгомери (Montgomery). Последний по оценкам вычислительных затрат превосходит большинство других алгоритмов (требует меньшего числа операций) [1, 2].

Основной идеей обоих алгоритмов является циклическое частичное умножение с уменьшением длины промежуточных результатов. Классический подход к решению этой задачи состоит в том, что один из сомножителей, например число *A*, рассматривается в виде последовательности бит a_i,

$$A = a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + ... + a_1 \cdot 2^1 + a_0, \quad (1)$$

где *n* — количество бит в числе *A*. Это позволяет выполнить умножение числа *A* на число *B* побитно, т. е. $(...((a_{n-1} \cdot B) \cdot 2 + a_{n-2} \cdot B) \cdot 2 + ... + a_1 \cdot B) \cdot 2 + a_0 \cdot B$, с применением операции вычисления остатка по модулю *M* после каждого частичного (побитового) умножения:

$$(A \cdot B)\%M \equiv \left(...\left(\left(\left(a_{n-1} \cdot B\right)\%M\right) \cdot 2 + a_{n-2} \cdot B\right)\%M\right) \cdot 2 + ... + a_0 \cdot B\Big)\%M. \quad (2)$$

Такая последовательность побитного умножения получила название LR-двоичного метода (Left-to-Right binary method), поскольку умножения начинаются с самого старшего бита числа *A*.

После каждой операции частичного умножения в выражении (2), сводящейся к операции условного суммирования накапливаемого промежуточного результата с числом *B*, и вычисления остатка по модулю *M* получается промежуточный результат, который всегда меньше *M*.

В отличие от (2) в методе Монтгомери используется RL (Right-to-Left) последовательность побитового умножения, требующая не умножения промежуточных результатов на 2, а деления. Классический вариант такого побитового умножения приведен во многих работах, например в [3, 4], где рассматривается алгоритм Монтгомери в виде, показанном на рис. 1.

```
P = 0
For i = 0 to n-1 do
P = P + A_i*B
if P_0=1 then P = P + M
P = P div 2
If P > M then P = P – M
Result = P
```

Рисунок 1 — Алгоритм Монтгомери со сдвигом частичных результатов вправо

Непосредственное применение алгоритма рис. 1 позволяет вычислить значение $\left(A \cdot B \cdot r^{-1}\right)\mathrm{mod}(M)$, где $r = 2^n$ — автоматически появляющийся побочный множитель результата. Для устранения этого побочного множителя, т. е. для вычисления значения $\left(A \cdot B\right)\mathrm{mod}(M)$, алгоритм рис. 1 следует применить дважды: $T = \left(A \cdot B \cdot r^{-1}\right)\mathrm{mod}(M)$; $P = \left(T \cdot r^2 \cdot r^{-1}\right)\mathrm{mod}(M)$.

В работе [3] рассматриваются две разновидности алгоритма Монтгомери, основанные на применении операции CSA (carry save addition) — сложения с фиксацией суммы по модулю 2 и всех межразрядных переносов. Обе разновидности, рис. 2, позволяют существенно ускорить вычисления модульного произведения.

- Inputs: *X,Y,M* with $0 \le X,Y < M$
- Output: $P = (X \cdot Y \cdot 2^{-n}) \bmod M$
- *n*: number of bits in *X*,
- x_i *: i*-th bit of *X*
- s_0: LSB of *S*
- *1. S* := 0; *C* := 0;
- **2. for** *i*:=0 **to** *n*-1 **do**
- 3. S,C := S + C + x_i*Y;
- 4. S,C := S + C + s_0*M;
- *5. S* := *S* div 2; *C* := *C* div 2;
- 6. P := S + C
- **7. if** $P \ge M$ **then** *P* := *P* – *M*;

Рисунок 2 — Алгоритм Монтгомери с использованием CSA

В работе [5] рассматривается побитовый алгоритм (2), отягощенный, на наш взгляд, применением операций CSA и SET (Sign Estimation Technique — "техника оценки знака"), рис. 4.

В качестве альтернативы этому алгоритму авторами работы [5] был предложен алгоритм "Radix-4", рис. 5, в котором число *A* умножается на число *B* не побитно, а с использованием *Booth*-правил, кодирующих три последовательных бита числа *A*, рис. 6. Несмотря на то, что каждое *Booth*-правило основано на трех битах числа *A*, определяющими являются только два бита (в начале каждой итерации производится умножение предыдущего результата на 4, т. е. сдвиг влево на 2 бита).

- Inputs: X,Y,M with $0 \le X,Y < M$
- Output: $P = (X \cdot Y \cdot 2^{-n}) \bmod M$
- n: number of bits in X,
- x_i : i-th bit of X
- s_0 : LSB of S, c_0 : LSB of C, y_0 : LSB of Y
- R: precomputed value of $Y+M$
- *1.* $S := 0$; $C := 0$;
- **2. for** i:=0 **to** n-1 **do**
- **3. if** ($s_0 = c_0$) and not x_i **then** $I := 0$;
- **4. if** ($s_0 \ne c_0$) and not x_i **then** $I := M$;
- **5. if** not($s_0 \oplus c_0 \oplus y_0$) and x_i **then** $I := Y$;
- **6. if** ($s_0 \oplus c_0 \oplus y_0$) and x_i **then** $I := R$;
- 7. S,C := S + C + I;
- *8.* $S := S$ div 2; $C := C$ div 2;
- 9. P := S + C
- **10. if** $P \ge M$ **then** $P := P - M$;

Рисунок 3 — Ускоренный алгоритм Монтгомери с использованием CSA

1. Set $S^{(0)} = 0$, $C^{(0)} = 0$ and $M = -N$
2. Repeat Step 2a, 2b, and 2c for i = 1, 2, 3, ⋯, n
   2a. $(C^{(i)}, S^{(i)}) := 2(C^{(i-1)} + S^{(i-1)}) + A_{n-i}B$
   2b. $(\overline{C}^{(i)}, \overline{S}^{(i)}) := C^{(i)} + S^{(i)} + 2M$
   If $T(\overline{C}^{(i)}) + T(\overline{S}^{(i)}) \ge 0$ then set $C^{(i)} = \overline{C}^{(i)}$ and $S^{(i)} = \overline{S}^{(i)}$
   2c. $(\overline{C}^{(i)}, \overline{S}^{(i)}) := C^{(i)} + S^{(i)} + M$
   If $T(\overline{C}^{(i)}) + T(\overline{S}^{(i)}) \ge 0$ then set $C^{(i)} = \overline{C}^{(i)}$ *and* $S^{(i)} = \overline{S}^{(i)}$
3. $(\overline{C}^{(i)}, \overline{S}^{(i)}) := C^{(i)} + S^{(i)} + M$
4. Compute $P := C^{(n)} + S^{(n)}$ and $\overline{P} := \overline{C}^{(n)} + \overline{S}^{(n)}$
5. If $\overline{P} \ge 0$ then $P = \overline{P}$
6. Return P.

Рисунок 4 — Алгоритм (2) с применением CSA и SET

Третий бит введен в *Booth*-правила для работы со знаковыми числами и участвует в формировании правил дважды: на одной итерации как первый бит, а на следующей — как третий. На наш взгляд предложенные в работе [5] *Booth*-правила, рис. 6, несколько отягощают этот алгоритм, хотя при этом требуется меньшее число операций, чем в алгоритме рис. 4. Анализ сразу двух бит числа *A* позволяет вдвое уменьшить число итераций по сравнению с побитовым анализом. Что касается названия этого алгоритма, то, видимо, оно связано с числом возможных исходов для каждой анализируемой двухбитовой величины. В этом случае побитовый алгоритм, по аналогии, может быть назван алгоритмом "Radix-2". Отметим также еще одну особенность алгоритмов рис. 4 и рис. 5.

Пусть число *B*, участвующее в модульном умножении, всегда меньше модуля *M* (заметим попутно, что на число *A* не накладывается такого ограничения). Накапливаемый промежуточный результат, исходное значение которого в начале каждой итерации меньше модуля *M* (в алгоритме на рис. 5 модуль обозначен символом *N*), после умножения на 4 и прибавления значения BP_i, вычисленного по *Booth*-правилу, рис. 6, в самом худшем случае оказывается меньшим, чем 6*M* (6*N*). Поэтому, для вычисления остатка по модулю *M*, на каждой итерации делается попытка уменьшения этого результата на величину $k \cdot M$ ($k \cdot N$), где целое число

k, которое может принимать значения от 0 до 5, заранее неизвестно. В алгоритме рис. 4 число *k* ищется среди значений 0, 1, 2.

III Обобщенный алгоритм Radix-Z

Можно показать, что в основе алгоритма рис. 5 лежит идея представления числа *A*, в виде

$$A = (a_{n-1}a_{n-2})\cdot 2^{n-2} + (a_{n-3}a_{n-4})\cdot 2^{n-4} + \ldots + (a_1a_0)\cdot 2^0, \text{ если n — четное} \quad (3a)$$

$$A = (0a_{n-1})\cdot 2^{n-1} + (a_{n-2}a_{n-3})\cdot 2^{n-3} + \ldots + (a_1a_0)\cdot 2^0, \text{ если n — нечетное} \quad (3б)$$

или

$$A = (a_{n-1}a_{n-2})\cdot 4^{\frac{n-2}{2}} + (a_{n-3}a_{n-4})\cdot 4^{\frac{n-4}{2}} + \ldots + (a_1a_0)\cdot 4^0, \text{ если n — четное} \quad (4a)$$

1. Set $S^{(0)} = 0$ and $C^{(0)} = 0$
2. for $i = 1$ *to* $\frac{n}{2}+2$
   $(C^{(i)}, S^{(i)}) := 4(C^{(i-1)} + S^{(i-1)}) + BP_i$
   If $T(C^{(i)}) + T(S^{(i)}) \geq 0$ then
      $(\overline{C}^{(i)}, \overline{S}^{(i)}) := C^{(i)} + S^{(i)} - 2N$
         If $T(\overline{C}^{(i)}) + T(\overline{S}^{(i)}) \geq 0$ then $C^{(i)} = \overline{C}^{(i)}$ and $S^{(i)} = \overline{S}^{(i)}$
            $(\overline{C}^{(i)}, \overline{S}^{(i)}) := C^{(i)} + S^{(i)} - 2N$
               If $T(\overline{C}^{(i)}) + T(\overline{S}^{(i)}) \geq 0$ then $C^{(i)} = \overline{C}^{(i)}$ and $S^{(i)} = \overline{S}^{(i)}$
               end
         end
      $(\overline{C}^{(i)}, \overline{S}^{(i)}) := C^{(i)} + S^{(i)} - N$
         If $T(\overline{C}^{(i)}) + T(\overline{S}^{(i)}) \geq 0$ then $C^{(i)} = \overline{C}^{(i)}$ and $S^{(i)} = \overline{S}^{(i)}$
         end
   else
      $(C^{(i)}, S^{(i)}) := C^{(i)} + S^{(i)} + 2N$
      $(\overline{C}^{(i)}, \overline{S}^{(i)}) := C^{(i)} + S^{(i)} - N$
         If $T(\overline{C}^{(i)}) + T(\overline{S}^{(i)}) \geq 0$ then $C^{(i)} = \overline{C}^{(i)}$ and $S^{(i)} = \overline{S}^{(i)}$
         end
   end
3. $(\overline{C}^{(i)}, \overline{S}^{(i)}) := C^{(i)} + S^{(i)} - N$
4. Compute $P := C^{(n)} + S^{(n)}$ and $\overline{P} := \overline{C}^{(n)} + \overline{S}^{(n)}$
5. If $\overline{P} \geq 0$ then $P = \overline{P}$
6. Return P.

Рисунок 5 — алгоритм Radix-4 с применением CSA и SET

$$A = (0a_{n-1})\cdot 4^{\frac{n-1}{2}} + (a_{n-2}a_{n-3})\cdot 4^{\frac{n-3}{2}} + \ldots + (a_1a_0)\cdot 4^0, \text{ если n – нечетное.} \quad (4б)$$

Обобщая данный подход, число *A* можно представить в виде степенного многочлена

$$A = \alpha_{h-1}\cdot Z^{h-1} + \alpha_{h-2}\cdot Z^{h-2} + \ldots + \alpha_1\cdot Z^1 + \alpha_0\cdot Z^0, \quad (5)$$

где *Z* — основание выбранного представления, α_j — соответствующие коэффициенты такого представления, *h* — степень представления. Тогда, умножение числа *A* на число *B* по частям есть $(\ldots((\alpha_{h-1}\cdot B)\cdot Z + (\alpha_{h-2}\cdot B))\cdot Z + \ldots + (\alpha_1\cdot B))\cdot Z + (\alpha_0\cdot B)$, а применение операции вычисления остатка по модулю *M* после каждого частичного умножения есть

$$(A \cdot B)\%M \equiv \left(...\left(\left(\left(\alpha_{h-1} \cdot B\right)\%M\right) \cdot Z + \alpha_{h-2} \cdot B\right)\%M\right) \cdot Z + ... + \alpha_0 \cdot B\Big)\%M \,. \quad (6)$$

A_n	A_{n-1}	A_{n-2}	Recoded digit	Operation on B
0	0	0	0	0B
0	0	1	+1	+1B
0	1	0	+1	+1B
0	1	1	+2	+2B
1	0	0	-2	-2B
1	0	1	-1	-B
1	1	0	-1	-B
1	1	1	0	0B

Рисунок 6 — *Booth*-правила для алгоритма Radix-4

Алгоритм, соответствующий обобщенному выражению (6), назовем, по аналогии с работой [5], алгоритмом Radix-Z и рассмотрим его операции подробнее.

-Пусть число *B*, как и ранее, всегда меньше модуля *M*.

-В начале каждой итерации алгоритма Radix-Z накапливаемый промежуточный результат $\left(...\left(\alpha_j \cdot B\right)\%M\right)$ имеет значение, меньшее модуля *M*.

-После умножения его на основание *Z* будет получаться значение, всегда меньшее, чем $\left(Z \cdot M\right)$.

-После прибавления очередного частного произведения $\left(\alpha_{j-1} \cdot B\right)$ (которое всегда меньше, чем $\left(Z \cdot M\right)$, т. к. любой коэффициент α_j меньше основания представления *Z*, а число *B* должно быть меньше модуля *M* по условию) получим значение $\left(...\left(\alpha_j \cdot B\right)\%M\right) \cdot Z + \alpha_{j-1} \cdot B$, которое всегда меньше, чем $\left(2 \cdot Z \cdot M\right)$.

-Тогда, операция вычисления остатка по модулю *M* от полученного промежуточного результата в конце каждой итерации сводится к попытке уменьшения его на величину $k \cdot M$, где заранее неизвестное целое число *k* может принимать значения от 0 до $\left(2 \cdot Z - 1\right)$.

Последняя операция соответствует известной задаче оптимального "взвешивания" (замещения) некоторого значения некоторым набором мер (например, параллельное сравнение, последовательное приближение, комбинированная или ΣΔ-компенсация, а также поразрядное уравновешивание в теории измерений) с последующим нахождением неуравновешенного остатка.

В случае аппаратной реализации этой операции можно существенно ускорить вычисления и достичь потенциального быстродействия (ценой соответствующих аппаратных затрат), если применить параллельное сравнение замещаемой величины $\left(...\left(\alpha_j \cdot B\right)\%M\right) \cdot Z + \alpha_{j-1} \cdot B$ со всеми возможными значениями мер, т. е. со значениями $\left(2 \cdot Z - 1\right) \cdot M$, $\left(2 \cdot Z - 2\right) \cdot M$, $\left(2 \cdot Z - 3\right) \cdot M$, …, M.

В случае программной реализации лучшие результаты дает метод поразрядного уравновешивания. Поскольку в этом методе предполагается, что вес каждого последующего разряда (более младшего, т. к. первым уравновешивается самый старший разряд компенсирующего кода) должен быть вдвое меньше предыдущего, т. е. используется двоичная весовая система, то минимальное число актов уравновешивания получается в случае выбора $\left(2 \cdot Z\right) = 2^k$, откуда:

$$Z = 2^{k-1}, \quad (7)$$

где *k* — некоторое целое число.

Заметим, что выбор основания *Z* по правилу (7) очень удобен как для нахождения коэффициентов α_j в представлении (5), так и для выполнения умножения промежуточного результата $\left(...\left(\alpha_j \cdot B\right)\%M\right)$ на *Z*.

Заметим также, что в случае (7) набор двоичных мер будет соответствовать ряду

$$2^{k-1} \cdot M,\ 2^{k-2} \cdot M,\ ...,\ 2^1 \cdot M,\ 2^0 \cdot M \,. \quad (8)$$

Откуда следует, что число требуемых актов уравновешивания, равное числу двоичных разрядов в коде, эквивалентном ряду (8), равно *k*.

Формально алгоритм Radix-Z в терминах языка Си можно записать в виде, предоставленном рис. 7.

```
P=0;
for(i=h – 1; i>=0; i – –)
{
P=P * Z;
P=P + αi * B;
for(j=k – 1; j>=0; j– –) { T=P – (M<<j); if(T>=0) P=T; }
}
return(P);
```

Рисунок 7 — Обобщенный алгоритм Radix-Z

Заметим, что операция P=P*Z в алгоритме рис. 7 может быть заменена операцией P=P<<(k–1).

Для ускорения работы алгоритма рис. 7 значения (M<<j) можно выбирать из заранее созданной таблицы (поскольку очень часто модуль *M* не меняется от операции к операции, например при модульном возведении в степень). Тогда алгоритм рис. 7 можно переписать в виде рис. 8.

```
P=0;
for(i=h – 1; i>=0; i – –)
{
P=P<<(k – 1);
P=P + αi * B;
for(j=k – 1; j>=0; j– –) { T=P – Mtbl[j]; if(T>=0) P=T; }
}
return(P);
```

Рисунок 8 — Алгоритм Radix-Z, оптимизация 1

IV Характеристики алгоритма Radix-Z

Чтобы оценить эффективность алгоритма Radix-Z, рис. 8, нами были подсчитаны количества операций сдвига, операций частичного умножения и операций сложения для различных значений *Z*, и различного числа бит *n* (8, 16, …, 2048) в исходных сомножителях. При этом делались следующие допущения:

-затраты времени на организацию циклов ничтожно малы;

-сдвиг на любое число разрядов выполняется за одну операцию;

-затраты времени на получение значений коэффициентов α_i ничтожно малы;

-затраты времени на все проверки условий ничтожно малы по сравнению с затратами времени на операцию сложения;

-затраты времени на присвоение одной переменной значения другой переменной также ничтожно малы.

Полученные результаты приведены в табл. 1, где использованы следующие обозначения:

n — число бит в исходных сомножителях,

Z — расщепленность алгоритма Radix,

h — степень представления (5), или иначе — число итераций, $h = \mathrm{int}\left(0{,}9(9) + n \big/ \log_2 Z\right)$,

O_{shft} — число операций сдвига, $O_{shft} = h$,

O_{add} — число операций сложения, $O_{add} = \left(2 + \log_2 Z\right) \cdot h$,

O_{mul} — число операций частичного умножения, $O_{mul} = h$,

O_{alg} — общее число операций алгоритма рис. 8,

$$O_{alg} = O_{shft} + O_{add} + O_{mul} = \left(4 + \log_2 Z\right) \cdot h \approx \frac{4n}{\log_2 Z} + n,$$

O_{ini} — число операций инициализации таблицы Mtbl[j], $O_{ini} = \log_2 Z$,

O_{total} — общее число операций алгоритма рис. 8 с учетом числа операций инициализации таблицы Mtbl[j],

$$O_{total} = O_{alg} + O_{ini} = \left(4 + \log_2 Z\right) \cdot h + \log_2 Z \approx \frac{4n}{\log_2 Z} + n + \log_2 Z.$$

Таблица 1 — Характеристики алгоритма Radix-Z

	n=8							n=16						
Z	h	O_{shft}	O_{add}	O_{mul}	O_{alg}	O_{ini}	O_{total}	h	O_{shft}	O_{add}	O_{mul}	O_{alg}	O_{ini}	O_{total}
2	8	8	24	8	40	1	41	16	16	48	16	80	1	81
4	4	4	16	4	24	2	26	8	8	32	8	48	2	50
8	3	3	15	3	21	3	24	6	6	30	6	42	3	45
16	2	2	12	2	16	4	20	4	4	24	4	32	4	36
32	2	2	14	2	18	5	23	4	4	28	4	36	5	41
64	2	2	16	2	20	6	26	3	3	24	3	30	6	36
128	2	2	18	2	22	7	29	3	3	27	3	33	7	40
256	1	1	10	1	12	8	20	2	2	20	2	24	8	32
512	1	1	11	1	13	9	22	2	2	22	2	26	9	35
1024	1	1	12	1	14	10	24	2	2	24	2	28	10	38
2048	1	1	13	1	15	11	26	2	2	26	2	30	11	41
4096	1	1	14	1	16	12	28	2	2	28	2	32	12	44
8192	1	1	15	1	17	13	30	2	2	30	2	34	13	47
16384	1	1	16	1	18	14	32	2	2	32	2	36	14	50
32768	1	1	17	1	19	15	34	2	2	34	2	38	15	53
65536	1	1	18	1	20	16	36	1	1	18	1	20	16	36

	n=32							n=64						
Z	h	O_{shft}	O_{add}	O_{mul}	O_{alg}	O_{ini}	O_{total}	h	O_{shft}	O_{add}	O_{mul}	O_{alg}	O_{ini}	O_{total}
2	32	32	96	32	160	1	161	64	64	192	64	320	1	321
4	16	16	64	16	96	2	98	32	32	128	32	192	2	194
8	11	11	55	11	77	3	80	22	22	110	22	154	3	157
16	8	8	48	8	64	4	68	16	16	96	16	128	4	132
32	7	7	49	7	63	5	68	13	13	91	13	117	5	122
64	6	6	48	6	60	6	66	11	11	88	11	110	6	116
128	5	5	45	5	55	7	62	10	10	90	10	110	7	117
256	4	4	40	4	48	8	56	8	8	80	8	96	8	104
512	4	4	44	4	52	9	61	8	8	88	8	104	9	113
1024	4	4	48	4	56	10	66	7	7	84	7	98	10	108
2048	3	3	39	3	45	11	56	6	6	78	6	90	11	101
4096	3	3	42	3	48	12	60	6	6	84	6	96	12	108
8192	3	3	45	3	51	13	64	5	5	75	5	85	13	98
16384	3	3	48	3	54	14	68	5	5	80	5	90	14	104
32768	3	3	51	3	57	15	72	5	5	85	5	95	15	110
65536	2	2	36	2	40	16	56	4	4	72	4	80	16	96

	n=128							n=256						
Z	h	O_{shft}	O_{add}	O_{mul}	O_{alg}	O_{ini}	O_{total}	h	O_{shft}	O_{add}	O_{mul}	O_{alg}	O_{ini}	O_{total}
2	128	128	384	128	640	1	641	256	256	768	256	1280	1	1281
4	64	64	256	64	384	2	386	128	128	512	128	768	2	770
8	43	43	215	43	301	3	304	86	86	430	86	602	3	605
16	32	32	192	32	256	4	260	64	64	384	64	512	4	516
32	26	26	182	26	234	5	239	52	52	364	52	468	5	473
64	22	22	176	22	220	6	226	43	43	344	43	430	6	436
128	19	19	171	19	209	7	216	37	37	333	37	407	7	414
256	16	16	160	16	192	8	200	32	32	320	32	384	8	392
512	15	15	165	15	195	9	204	29	29	319	29	377	9	386
1024	13	13	156	13	182	10	192	26	26	312	26	364	10	374
2048	12	12	156	12	180	11	191	24	24	312	24	360	11	371
4096	11	11	154	11	176	12	188	22	22	308	22	352	12	364

8192	10	10	150	10	170	13	183	20	20	300	20	340	13	353

Продолжение таблицы 1

16384	10	10	160	10	180	14	194	19	19	304	19	342	14	356
32768	9	9	153	9	171	15	186	18	18	306	18	342	15	357
65536	8	8	144	8	160	16	176	16	16	288	16	320	16	336

	n=512							n=1024						
Z	h	O_{shft}	O_{add}	O_{mul}	O_{alg}	O_{ini}	O_{total}	h	O_{shft}	O_{add}	O_{mul}	O_{alg}	O_{ini}	O_{total}
2	512	512	1536	512	2560	1	2561	1024	1024	3072	1024	5120	1	5121
4	256	256	1024	256	1536	2	1538	512	512	2048	512	3072	2	3074
8	171	171	855	171	1197	3	1200	342	342	1710	342	2394	3	2397
16	128	128	768	128	1024	4	1028	256	256	1536	256	2048	4	2052
32	103	103	721	103	927	5	932	205	205	1435	205	1845	5	1850
64	86	86	688	86	860	6	866	171	171	1368	171	1710	6	1716
128	74	74	666	74	814	7	821	147	147	1323	147	1617	7	1624
256	64	64	640	64	768	8	776	128	128	1280	128	1536	8	1544
512	57	57	627	57	741	9	750	114	114	1254	114	1482	9	1491
1024	52	52	624	52	728	10	738	103	103	1236	103	1442	10	1452
2048	47	47	611	47	705	11	716	94	94	1222	94	1410	11	1421
4096	43	43	602	43	688	12	700	86	86	1204	86	1376	12	1388
8192	40	40	600	40	680	13	693	79	79	1185	79	1343	13	1356
16384	37	37	592	37	666	14	680	74	74	1184	74	1332	14	1346
32768	35	35	595	35	665	15	680	69	69	1173	69	1311	15	1326
65536	32	32	576	32	640	16	656	64	64	1152	64	1280	16	1296

n=2048							
Z	h	O_{shft}	O_{add}	O_{mul}	O_{alg}	O_{ini}	O_{total}
2	2048	2048	6144	2048	10240	1	10241
4	1024	1024	4096	1024	6144	2	6146
8	683	683	3415	683	4781	3	4784
16	512	512	3072	512	4096	4	4100
32	410	410	2870	410	3690	5	3695
64	342	342	2736	342	3420	6	3426
128	293	293	2637	293	3223	7	3230
256	256	256	2560	256	3072	8	3080
512	228	228	2508	228	2964	9	2973
1024	205	205	2460	205	2870	10	2880
2048	187	187	2431	187	2805	11	2816
4096	171	171	2394	171	2736	12	2748
8192	158	158	2370	158	2686	13	2699
16384	147	147	2352	147	2646	14	2660
32768	137	137	2329	137	2603	15	2618
65536	128	128	2304	128	2560	16	2576

На рис. 9 — 10 показаны графики зависимости O_{alg} — числа операций алгоритма рис. 8 (сокращенное обозначение O_a) и зависимости O_{total} — общего числа операций (сокращенное обозначение O_t) для некоторых значений *n* (соединяющие линии — условные).

Как видно из табл. 1 и рис. 9 значение O_{alg} — быстро уменьшается при увеличении *Z* от *Z*=2 (классическое модульное умножение) до *Z*=256 (коэффициенты α имеют длину 8 бит). Начиная с *Z*=256 выигрыш в числе операций становится не столь ощутимым.

Алгоритм Монтгомери, рис. 1, и ускоренный алгоритм Монтгомери, рис. 3, можно сравнить с алгоритмом Radix-Z, рис. 8, при *Z*=2, 4, поскольку все они используют однобитное частичное умножение. (Здесь мы будем рассматривать условное сложение в алгоритме рис. 1, как однобитовое частичное умножение — так же, как это рассматривалось для алгоритма рис. 8).

- Общее число операций алгоритма Radix-Z, рис. 8, при Z=2 есть $O_{alg} = 5n$.
- Общее число операций алгоритма Radix-Z, рис. 8, при Z=4 есть $O_{alg} = 3n$.
- Общее число операций алгоритма Монтгомери, рис. 1, есть $O_{alg} = 5n$.
- Общее число операций ускоренного алгоритма Монтгомери, рис. 3, есть $O_{alg} = 3n + 2$.

Алгоритм Монтгомери имеет такую же сложность, как и алгоритм Radix-Z при *Z*=2. Ускоренный алгоритм Монтгомери, рис. 3, практически не уступает алгоритму Radix-Z при *Z*=4.

Таким образом, предлагаемый нами алгоритм Radix-Z превосходит описанные выше алгоритмы уже при *Z*=8.

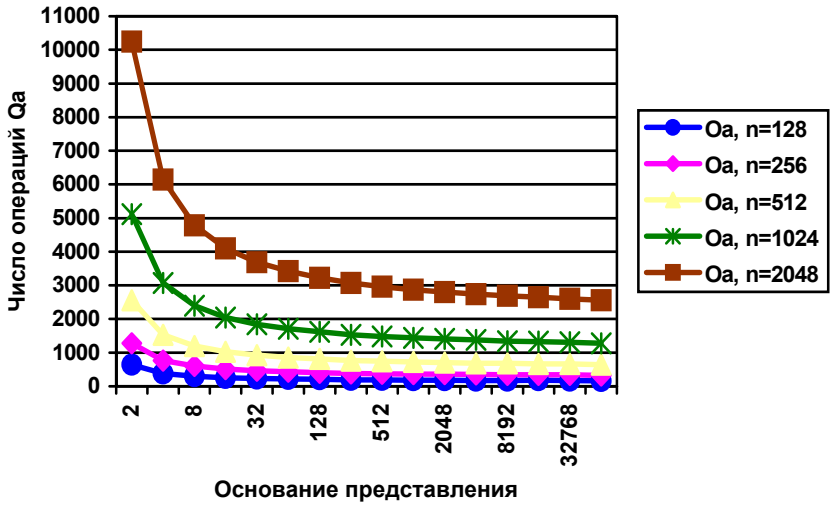

Рисунок 9 — Характеристики алгоритма Radix-Z

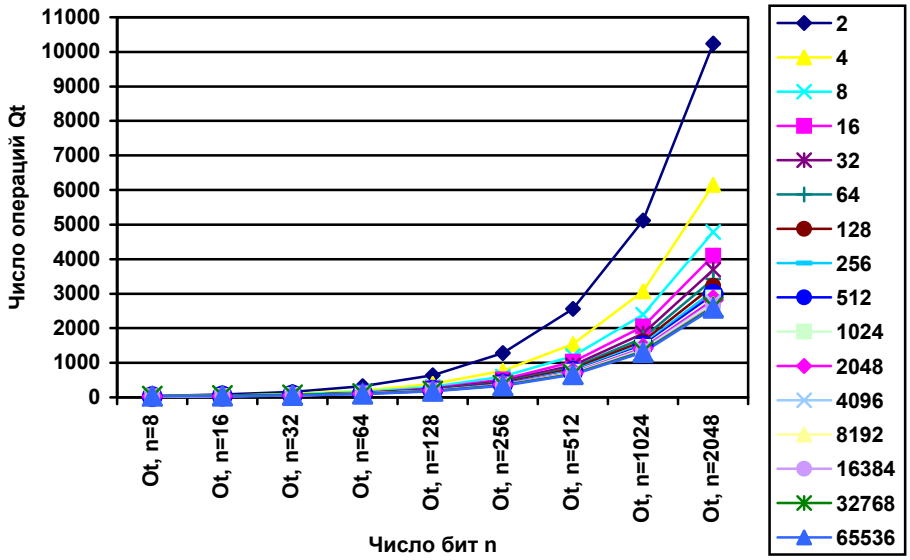

Рисунок 10 — Характеристики алгоритма Radix-Z

V Реализация алгоритма Radix-Z в реальном времени

Из сказанного ранее можно было бы сделать вывод, что увеличение значения основания Z позволяет уменьшить число операций алгоритма Radix-Z, рис.8, до значения $O_{alg} = 4 + n$, и, следовательно, время его выполнения. Однако тут не все так просто. Ранее, в качестве элементарных, рассматривались операции,

одним из операндов которых было длинное *n*-битовое число. При значениях *n* больших, чем разрядность применяемых арифметических устройств обязательно возникнут дополнительные затраты, связанные с реализацией операций над длинными числами. Эти затраты будут тем большими, чем меньше разрядность арифметико-логического устройства, умножителя и устройства сдвига применяемого процессора. Тем не менее, можно утверждать, что применение процессоров, арифметические устройства которых работают с 32-разрядными входными данными, не приведет к качественному выигрышу по сравнению с 16-разрядными устройствами, поскольку увеличение Z до значения $Z=2^{32}$ даст (для n=1024) $O_{alg} = \frac{4096}{32} + 1024 = 1152$, по сравнению с $O_{alg} = \frac{4096}{16} + 1024 = 1280$ при $Z=2^{16}$, т. е. выигрыш всего в 10%.

Для дальнейшего ускорения алгоритма рис. 8 можно рекомендовать:

-оперировать указателями на длинные числа, избегая копирования самих длинных чисел при выполнении операций присвоения;

-применять процессоры с модифицированной гарвардской архитектурой (например, сигнальные процессоры), позволяющие выбирать два операнда за одно обращение к памяти;

-использовать $Z=2^{k-1}$, где (k–1) — разрядность данных применяемого процессора;

-выполнять сокращенную проверку при уравновешивании промежуточного результата в конце каждой итерации.

Ускоренный вариант алгоритма Radix-Z приведен на рис. 11.

```
P=0;
for(i=h – 1; i>=0; i – –)
{
P=P<<(k – 1);
P=P + α_i * B;
for(j=k – 1; j>=0; j– –){ if(msw(P)>=msw(Mtbl[j])) P=P – Mtbl[j]; }
}
return(P);
```

Рисунок 11 — Алгоритм Radix-Z, оптимизация 2

Нами был реализован данный алгоритм на языке ассемблер сигнального процессора ADSP-218x со средним временем умножения 1024-битовых чисел, равным $107 \cdot 10^3$ тактов.

VI Выводы

Число операций, необходимых для реализации алгоритма модульного умножения Radix-Z, уменьшается с ростом основания представления Z в выражении (5) и стремится к значению $O_{alg} = 4 + n$ (при увеличении Z вплоть до $Z=2^n$). В случае аппаратной реализации операции взвешивания число операций алгоритма Radix-Z определяется выражением $O_{alg} = 3 \cdot h \approx \frac{3n}{\log_2 Z}$ и стремится к значению $O_{alg} = 3$ (при увеличении Z вплоть до $Z=2^n$). Основание представления Z следует выбирать с учетом (7). Число актов поразрядного уравновешивания промежуточных результатов равно $k = 1 + \log_2 Z$, а набор уравновешивающих мер соответствует (8). Для n=1024 бита удовлетворительные результаты (близкие к потенциально достижимым) получаются при использовании 16-разрядных процессоров.

Литература: ***1.*** *Peter L. Montgomery, "Modular Multiplication Without Trial Division", in Mathematics of Computation, volume 44, number 170, April 1985.* ***2.*** *Torbjorn Granlund and Peter L. Montgomery, "Division by Invariant Integers using Multiplication", in Proceedings of the SIGPLAN PLDI'94 Conference, June 1994.*