DOI：10.7652/xjtuxb201702008

# 面向嵌入式处理器的优化 Montgomery 模乘算法

李杨[1,2]，王劲林[1]，叶晓舟[1]，曾学文[1]
（1.中国科学院声学研究所国家网络新媒体工程技术研究中心，100190，北京；
2.中国科学院大学电子电气与通信工程学院，100049，北京）

**摘要**：针对嵌入式系统中频繁的内存存取影响 Montgomery 模乘算法效率的问题，提出了一种优化的分离连续操作数缓存算法。该算法基于连续操作数缓存算法并进行优化，应用于计算多精度乘法和约减两部分，将整个计算分块使得每块内操作数只被加载一次；为了不破坏操作数加载的连续性，在多精度乘法和约减之间采用分离集成的方式；通过动态地使用寄存器和有效的缓存操作数来减少嵌入式系统中算法使用内存存取操作的总量，实现提高模乘算法效率的目的。实验结果表明：在使用 MIPS64 架构的处理器上，当模数为 1 024 bit 时，与应用广泛的粗粒度集成操作数扫描算法相比，该算法的效率提高了 4.17%。在嵌入式系统中，可将该算法应用于公钥密码体系中的模乘运算，在提高模乘效率的同时提高公钥密码算法的运算效率。

**关键词**：Montgomery 模乘；嵌入式系统；连续操作数缓存算法
**中图分类号**：TP309.7 **文献标志码**：A **文章编号**：0253-987X(2017)02-0047-07

# An Optimized Montgomery Modular Multiplication Algorithm for Embedded Processors

LI Yang[1,2], WANG Jinlin[1], YE Xiaozhou[1], ZENG Xuewen[1]
(1. National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China; 2. School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, Beijing 100049, China)

**Abstract**: An improved separated consecutive operand caching algorithm is proposed to focus on the problem that frequent memory-access operations affect the efficiency of Montgomery modular multiplication on embedded processors. The algorithm carefully applies the general idea of consecutive operand caching and optimization to two calculation parts — multiplication and reduction. It separates the whole calculation into many blocks and loads operand in each block only once. The separately integrated mode is used between multiplication and reduction to keep the consecutiveness of operands. The number of memory-access operations on embedded processor is significantly reduced by dynamically using registers and efficient caching of operands. Experiments on processor with MIPS64 structure show that when the modulus is 1 024 bits, the proposed algorithm outperforms the coarsely integrated operand scanning algorithm by a factor of 4.17%. The proposed algorithm can be used for public-key cryptography to improve the efficiency of both the modular multiplication and the public-key algorithms.

**Keywords**: Montgomery modular multiplication; embedded processors; consecutive operand caching method

收稿日期：2016-06-20。 作者简介：李杨(1990—)，女，博士生；叶晓舟(通信作者)，男，研究员。 基金项目：中国科学院战略性先导科技专项课题资助项目(XDA06010302)；中国科学院声学研究所知识创新工程资助项目(Y154191601)。
网络出版时间：2016-12-30 网络出版地址：http://www.cnki.net/kcms/detail/61.1069.T.20161230.1629.008.html

公钥密码体系中目前使用最广泛的密码算法是公钥加密算法(RSA)[1]和椭圆曲线密码编码学算法(ECC)[2],其最重要的运算是有限域上的大数模乘。为了提高 ECC 和 RSA 算法的计算效率,有限域上大数模乘的优化成为研究的重点。Montgomery 模乘算法[3]是最有效的算法之一,许多研究都集中在如何实现 Montgomery 模乘算法上。Koc 等提出了 5 种实现算法,通过乘法、加法、加载、存储等简单操作实现模乘算法[4],通过优化这些操作数可以提高 Montgomery 模乘算法的效率。已有的算法由于反复加载同样的操作数带来了多次额外的时间开销,本文主要研究如何减少内存存取操作数。

由于细粒度集成的乘积扫描算法(FIPS)具有所需寄存器少、需加载和存储的中间结果少等优点,Chu、Großschäl、Liu 和 Zhang 等分别提出了最优素数域上优化的 FIPS 算法[5-9]。Seo 等在最优素数域上提出了 OPF-CIOC 模乘和 OPF-CISBD 模平方算法[10]。大部分的优化算法都针对最优素数域的 Montgomery 模乘实现算法。本文提出一种适用于任意素数域的算法,在连续操作数缓存算法(COC)[11]的基础上进行优化和改进,并提出分离连续操作数缓存算法(separated consecutive operand caching algorithm,SCOC),最后在使用 MIPS64 架构的 RISC 处理器上用汇编语言实现该算法并进行比较分析。

## 1 研究背景

对于 $P=AB \bmod N$($N$ 为 $m$ bit 二进制整数),Montgomery 模乘算法将模 $N$ 的运算转化为模 $R$ 的运算($R=2^m$)。计算 Montgomery 乘积主要包括两个关键步骤:计算多精度乘法 $T=AB$ 和 Montgomery 约减(简称约减)$P=(T+MN)/R$。Dusse 等提出 $r$ 进制 Montgomery 算法[12],用 $n_0'=-n_0^{-1} \bmod r$ 代替 $N'$。Koc 等在此基础上归纳了 5 种改进算法——SOS、CIOS、FIOS、FIPS 和 CIHS,其中最具代表意义的是 CIOS 和 FIPS 算法。S/CI/FI 代表多精度乘法和约减两部分之间的集成方式,其中 S 表示分离方式,即完全计算完多精度乘法后再计算约减;CI/FI 表示粗粒度和细粒度集成方式,即交替计算多精度乘法和约减。OS/PS 表示计算乘法 $AB$ 和 $MN$ 扫描方式,其中 OS 表示操作数扫描[13],PS 表示乘积扫描[14]。

下面分析 5 种 Montgomery 模乘算法内存存取复杂度。根据功能不同,分别统计加载(LD)和存储(ST)操作的数量。$L_{op}$ 表示加载操作数,$L_{int}$ 表示加载中间结果,$S_{int}$ 表示存储中间结果,$S_M$ 表示存储操作数,$S_{fin}$ 表示存储最终结果。表 1 给出了 5 种算法内存存取复杂度的分析统计。

表 1 几种算法内存存取复杂度分析

| 算法 | LD | | ST | | |
|---|---|---|---|---|---|
| | $L_{op}$ | $L_{int}$ | $S_{int}$ | $S_M$ | $S_{fin}$ |
| SOS | $2n^2+2n$ | $2n$ | $2n$ | 0 | $n+1$ |
| CIOS | $2n^2+2n$ | 0 | 0 | 0 | $n+1$ |
| FIOS | $4n^2$ | 0 | 0 | 0 | $n+1$ |
| FIPS | $4n^2$ | 0 | 0 | $n$ | $n+1$ |
| CIHS | $2.5n^2+1.5n$ | 0 | 0 | 0 | $n+1$ |

由表 1 可知,CIOS 算法需要最少内存存取操作。其通用算法使用 $n+4$ 个寄存器,当 $n+4$ 大于可用寄存器数时,不能使用此种算法。我们提出另一种实现算法,仅需 5 个寄存器,即 2 个存储操作数和 3 个存储中间结果,记这种算法为 CIOS-5reg 算法。CIOS-5reg 算法需要 $4n^2+4n-2$ 个 LD 操作和 $2n^2+3n-1$ 个 ST 操作。

下文使用以下符号:$A$、$B$ 表示 $m$ bit 多精度整数,$A=(A[n-1],\cdots,A[0])$,$B=(B[n-1],\cdots,B[0])$,$n=\lceil m/W \rceil$ 表示 $A$、$B$ 被表示字个数,其中 $W$ 表示处理器位数,则乘积 $C=AB=(C[2n-1],\cdots,C[0])$。为了形象化表示,Montgomery 模乘算法使用乘法结构和双菱形结构混合表示方法,如图 1 所示。图中:双菱形结构中上面的菱形表示多精度乘法 $C=AB$,下面的菱形表示约减 $C=(C+MN)/R$;菱形结构的每个点表示一个部分乘积 $A[i]B[j]$,从右到左表示了乘积从 $C[0]$ 到 $C[2n-1]$;乘法结构从上到下描述了部分乘积的计算顺序,多精度乘法用空心方框表示,约减部分用实心方框表示。

## 2 改进 Montgomery 模乘 SCOC 算法

本节提出一种计算 Montgomery 模乘的分离连续操作数缓存 SCOC 算法。基于 COC 算法[11]的基本思想进行优化和改进,在多精度乘法和约减两部分之间采用分离集成方式。

### 2.1 算法介绍

SCOC 算法在 PS 算法的基础上,通过将整个计算分块并适当增加寄存器,使每块内操作数只加载一遍来减少多余的 LD 开销。假设每块包含 $e$ 行,

整个计算有 $n$ 行，则被分成了 $r$ 块，$r=\lceil n/e \rceil$。通过 $n$、$e$ 大小不同，将 SCOC 算法分为两种情况：第一种情况 $n/e$ 是整数，记为 SCOC-a 算法；第二种情况 $n/e$不是整数，记为 SCOC-b 算法。

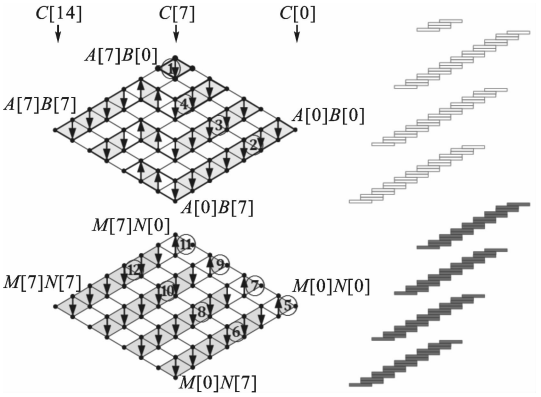



图 1　SCOC-a 算法结构图（$n=8$，$e=2$，$r=4$）

图 1 描述了 SCOC-a 算法结构（$n=8$，$e=2$，$r=4$），每块使用 $2e$ 个寄存器加载操作数和 3 个寄存器存储中间结果，寄存器总量为 $f=2e+3=7$。SCOC-a 算法采用分离集成方式，先计算多精度乘法，再计算约减，下面以图 1 为例分别对两部分进行详细分析。

2.1.1　SCOC-a 算法的多精度乘法　由图 1 中上面的菱形可以看出，多精度乘法被分成 $r=4$ 块，最上面的块①是大小为 $e$ 的菱形，称为 $b_t$，最下面的块②称为 $b_b$，中间的两块③④形状相同称为 $b_m$，计算按照 $b_t$、$b_b$、$b_m$ 的顺序进行。

$b_t$ 为最上面的块①，其内部使用 PS 算法进行计算。使用 $2e$ 个寄存器加载操作数，$e$ 个操作数 $A$ 和 $e$ 个 $B$ 仅在开始被加载一次即可，因此 LD 操作数是 $2e$。最终有 $2e$ 个计算结果需要被存储，则所需 ST 操作数为 $2e$。

$b_b$ 为最下面的块②。首先加载 $A[0]$到 $A[e-1]$至寄存器并缓存（$e$ 个 LD），操作数 $B$ 复用 $b_t$ 缓存的 $B[0]$到 $B[e-1]$；接下来依次加载 $B[e]$到 $B[n-1]$（$n-e$ 个 LD）。当计算完 $A[0]$到 $A[e-1]$的全部部分乘积后，使用 $B$ 缓存的 $B[n-e]$到 $B[n-1]$，依次加载 $A[e]$到 $A[2e-1]$（$e$ 个 LD），因此需要 LD 操作数为 $e+(n-e)+e=n+e$，又因为需要加载 $b_t$ 计算得到的 $2e$ 个中间结果，所以 LD 的总量是 $n+3e$；最终 $n+2e$ 个的结果需要被存储，所需的 ST 操作数为 $n+2e$。

$b_m$ 为包含中间剩余的所有块，计算按照从下到上的顺序。每块计算分 3 个阶段：每个阶段都复用上个阶段缓存的 $e$ 个操作数。图 2 形象地描述了块③3 个阶段的计算过程，图中方框标识的数表示需要重新加载到寄存器的操作数。第 1 阶段复用 $A[2]$和 $A[3]$，依次加载 $B[0]$到 $B[5]$；第 2 阶段复用 $B[4]$和 $B[5]$，依次加载 $A[4]$和 $A[5]$；第 3 阶段复用 $A[4]$和 $A[5]$，依次加载 $B[6]$和 $B[7]$；所以需要 LD 的操作数为 $n+e$，又因为需要加载上一块的 $n+e$ 个中间结果，所以 LD 操作数总量为 $2(n+e)$；最终 $n+2e$ 个结果被存储，所需 ST 的操作数为 $n+2e$。

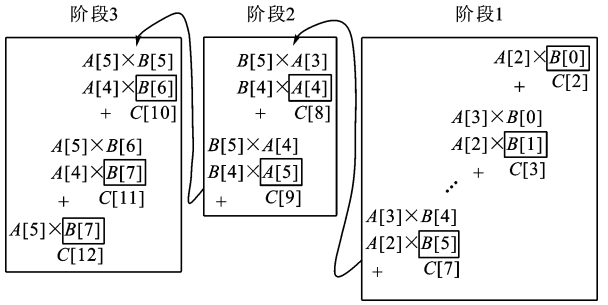



图 2　块③的计算流程图

2.1.2　SCOC-a 算法的约减　由图 1 中下面的菱形可以看出，约减被分成 $r=4$ 个相同的块 $b_M$，按照从下到上的顺序依次计算每块。$b_M$ 被分为两小块，最右面一小块先计算 $e$ 个 $M[i]$并缓存到寄存器（$e$ 个 LD），下面依次加载 $N[0]$到 $N[n-1]$至寄存器（$n$ 个 LD），因此 LD 操作数为 $n+e$。因为最终计算结果的低 $n$ 位为 0，所以每块计算得到的最低 $e$ 位固定为 0，不需要存储。每块加载上一块中间结果的数量由 $2n$ 减少到 $n+e$，存储中间结果的数量由 $2n-e$ 减少到 $n$，且每块比上一块减少 $e$。LD 总量为 $r(n+e)+(2n+\cdots+n+e)=2.5n^2/e+1.5n$；ST 总量为 $(2n-e+\cdots+n)r/2=1.5n^2/e-0.5n$。

图 3 描述了 SCOC-b 算法结构（$n=8$，$e=3$，$r=3$），所需寄存器数为 $f=2e+3=9$。同 SCOC-a 算法分析类似，对两部分进行详细分析。

2.1.3　SCOC-b 算法的多精度乘法　由图 3 可以

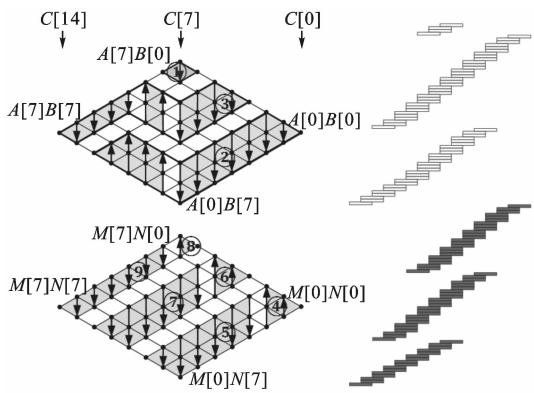



图 3　SCOC-b 算法结构图（$n=8$，$e=3$，$r=3$）

看出，多精度乘法被分成 $r=3$ 块，最上面和最下面的块仍称为 $b_t$ 和 $b_b$，而剩余块③同 SCOC-a 算法中 $b_m$ 不同，称为 $b_l$。这个例子中没有 $b_m$，但在 $n$ 和 $e$ 取一定值时 $b_m$ 仍存在($n=10$，$e=3$)。计算时按照 $b_t$、$b_b$、$b_m$、$b_l$ 的顺序进行。$b_m$ 同 SCOC-a 算法相同，不再赘述。

$b_t$ 同 SCOC-a 算法不同的是其大小为 $n-(r-1)e$，因此 LD 操作数为 $2[n-(r-1)e]$，ST 操作数为 $2[n-(r-1)e]$。

$b_b$ 同 SCOC-a 算法不同的是复用 $b_t$ 操作数由 $e$ 变为 $n-(r-1)e$，加载 $b_t$ 中间结果由 $2e$ 变为 $2[n-(r-1)e]$，所以 LD 操作数为 $n+2e+[n-(r-1)e]$。

$b_l$ 也分为 3 个阶段，第 1、2 阶段与 $b_m$ 相同，第 3 阶段复用操作数由($n-e$)个变为 $n-(r-1)e$ 个 $A$，即与 $b_m$ 相比，第 3 阶段变“瘦”了，所以 LD 操作数为 $2n+e+[n-(r-1)e]$，ST 操作数为 $n+e+[n-(r-1)e]$。

2.1.3 SCOC-b 算法的约减 由图 3 可以看出，约减被分成 $r=3$ 块，仍按照从下至上的顺序计算。最上面的块包括 $n-(r-1)e$ 行，记作 $b_{Mt}$。$b_{Mt}$ 中加载操作数为 $n+e$，加载中间结果数量是 $2n-(r-1)e$，故 LD 操作数为 $3n+(2-r)e$；ST 操作数为 $n$。下面剩余块与 SCOC-a 算法相同仍记为 $b_M$，故 LD 总量为 $(r-1)(n+e)+(2n+\cdots+2n-(r-2)e)=3(r-1)n+(2r-0.5r^2-1.5)e$；ST 总量为 $(2n-e+\cdots+2n-(r-1)e)=2(r-1)n+(0.5r-0.5r^2)e$。

表 2 和表 3 分别分析了 SCOC-a 算法和 SCOC-b 算法的内存存取复杂度，从中可以看出：SCOC-a 算法的内存存取总量为 $7n^2/e+n+e$；SCOC-b 算法的内存存取总量为 $(8r+5)n-(r^2+1.5r+1.5)e$。由于 $n/e<r<n/e+1$，故 SCOC-b 算法的内存存取总量小于 $7n^2/e+9.5n-4e$。当 $e$ 的取值越大时，SCOC 算法内存存取总量越小。

表 2 SCOC-a 算法内存存取复杂度分析

| 内存存取操作 | 多精度乘法 | | | 约减 |
|---|---|---|---|---|
| | $b_t$ | $b_b$ | $b_m$ | $b_M$ |
| LD | $2e$ | $n+3e$ | $2n+2e$ | $2.5n^2/e+1.5n$ |
| ST | $2e$ | $n+2e$ | $n+2e$ | $1.5n^2/e-0.5n$ |
| LD/ST 总量 | $4e$ | $2n+5e$ | $3n+4e$ | $4n^2/e+n$ |

表 3 SCOC-b 算法内存存取复杂度分析

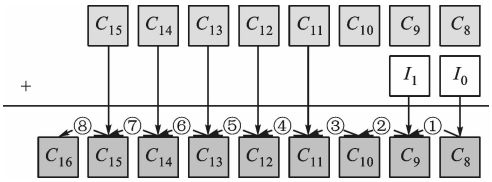
| 内存存取操作 | 多精度乘法 | | | | 约减 | |
|---|---|---|---|---|---|---|
| | $b_t$ | $b_b$ | $b_m$ | $b_l$ | $b_{Mt}$ | $b_M$ |
| LD | $2n-2(r-1)e$ | $2n+(3-r)e$ | $2n+2e$ | $3n+(2-r)e$ | $3n+(2-r)e$ | $3(r-1)n+(2r-0.5r^2-1.5)e$ |
| ST | $2n-2(r-1)e$ | $n+2e$ | $n+2e$ | $2n+(2-r)e$ | $n$ | $2(r-1)n+(0.5r-0.5r^2)e$ |
| LD/ST 总量 | $4n-4(r-1)e$ | $3n+(5-r)e$ | $3n+4e$ | $5n+(4-2r)e$ | $4n+(2-r)e$ | $5(r-1)n+(2.5r-r^2-1.5)e$ |

## 2.2 进位处理方式的改进

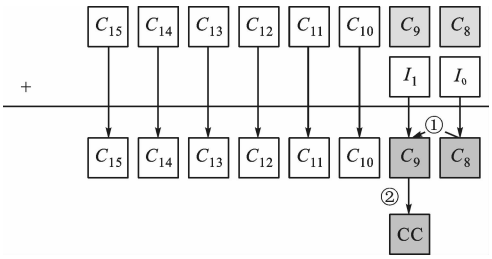
分析 SCOC 算法可知，计算约减时需加上多精度乘法得到的中间结果，因此每一块结束时产生进位都需要传递到中间结果的最高位，这就是进位传递问题。当 $n$ 较大时，进位传递问题会带来很大的开销。

下面以图 1 为例描述进位传递问题及改进后的进位处理方法。约减部分的每一大块结束时，最后一列仅包括一个点，即仅需计算 $M[i]N[j]+c$($c$ 为上一列进位)。由于处理器的位数为 $W$，操作数表述为 $r=2^W$ 进制，因此每个字取值范围为$[0,r-1]$，故 $M[i]N[j]+c\leqslant(r-1)(r-1)+(r-1)=r^2-r<r^2$，因此不会产生进位，可以用($I_1$，$I_0$)表示 $M[i]N[j]+c$ 的结果。

图 4 展示计算块⑤⑥的进位传递问题，$C_8\sim C_{15}$ 表示需要重新加载中间结果，$I_1$、$I_0$ 表示计算最后一列结果，CC 表示存储进位的寄存器。图 4a 表示正常进位处理方法，由图 4 可知，需要 8 次进位传递，且需加载 $C_8\sim C_{15}$ 到寄存器运算完后再存储到内存中。图 4b 描述了改进的进位处理办法，首先计算($R_1$，$R_0$)加上($I_1$，$I_0$)得到新的($R_1$，$R_0$)和一个进位，将($R_1$，$R_0$)存储到内存中，新的进位结果存储在寄存器 CC 中。在下一块计算到这列时，再将 CC 寄存器中的值加到最终结果上。改进后的方法有效减




(a)正常进位处理




(b)改进后进位处理

图 4　进位处理改进

少了进位传递和内存存取数，如图 4 所示，进位传递由 8 次减少到 2 次，内存存取由 17 次减少到 5 次，因此这种方法提高了 SCOC 算法的运算效率。

### 2.3　几种算法分析与比较

本节对 SCOC 算法与常用的几种 Montgomery 算法的内存存取复杂度进行对比，见表 4。比较几种现有算法，CIOS 算法内存存取数最少，但需要的寄存器数较多。当 $n$ 值较大、可用寄存器数小于 $n+4$ 时，就不能使用这种算法了。CIOS-5reg 和 FIPS 算法用到的寄存器数较小，但内存存取量较大。本文所提的 SCOC 算法可以通过可用寄存器数动态调整 $e$ 的大小，使得尽可能多的使用寄存器来减少内存存取的数量。由表 4 中对比可知，当 $e>3$ 时，SCOC 算法的内存存取小于 CIOS 算法。

表 4　几种 Montgomery 模乘算法内存存取复杂度对比

| 算法 | 加载操作数的寄存器数 | 累加寄存器数 | 寄存器数 | 加载操作数 | 存储操作数 | 内存存取操作数 |
|---|---|---|---|---|---|---|
| SOS | 2 | $n+2$ | $n+4$ | $2n^2+4n$ | $3n+1$ | $2n^2+7n+1$ |
| CIOS | 2 | $n+2$ | $n+4$ | $2n^2+2n$ | $n+1$ | $2n^2+3n+1$ |
| CIOS-5reg | 2 | 3 | 5 | $4n^2+4n-2$ | $2n^2+3n-1$ | $6n^2+7n-3$ |
| FIOS | 2 | $n+2$ | $n+4$ | $4n^2$ | $n+1$ | $4n^2+n+1$ |
| FIPS | 2 | 3 | 5 | $4n^2$ | $2n+1$ | $4n^2+2n+1$ |
| CIHS | 2 | $n+2$ | $n+4$ | $2.5n^2+1.5n$ | $n+1$ | $2.5n^2+2.5n+1$ |
| SCOC-a | $2e$ | 3 | $2e+3$ | $4.5n^2/e+0.5n+e$ | $2.5n^2/e+0.5n$ | $7n^2/e+n+e$ |
| SCOC-b | $2e$ | 3 | $2e+3$ | $(5r+3)n-(0.5r^2+r+0.5)e$ | $(3r+2)n-(0.5r^2+0.5r+1)e$ | $(8r+5)n-(r^2+1.5r+1.5)e$ |

## 3　仿真实现与分析

在 MIPS64 处理器上用汇编语言实现代表性算法 CIOS、CIOS-5reg、FIPS 和 SCOC，以及进位处理改进的 SCOC 算法（记为 SCOC-c 算法）。选取操作数从 192 到 1 024 位（$n=3/4/8/16$）。$e$ 的大小由 $n$ 及可用寄存器数 $f$ 来选取，在保证 $e\leqslant n$，$2e+3<f$ 的基础上尽可能大地选取 $e$。故 $n=3$ 时，选取 $e=3$；$n=4$ 时，选取 $e=4$；$n=8/16$ 时，选取 $e=6$。MIPS64 上乘积结果存在 HI/LO 中，且无带进位加法，故对比 2.3 节的理论值，需增加 2 个寄存器存储 HI/LO 数据，增加 1 个存储进位。$n=3,4$ 时，不存在进位传递问题，不使用 SCOC-c 算法。$n=16$ 时，CIOS 算法需要 23 个寄存器，大于可用寄存器数，所以不能使用。

表 5 对比了各种算法的指令数。由表 5 可知：$n$ 相同时乘法相关指令（DMULTU/MFLO/MFHI）基本相同；SCOC 算法的 LD/ST 指令数最少，尤其是 LD 指令，且随 $n$ 的增大优势更明显；SCOC 算法的 DADDU 指令数略高于其他算法。对于可以改进进位处理的情况，SCOC-c 算法与 SCOC 算法相比，增加了一个寄存器，但减少了 LD/ST 和加法指令。综上所述，SCOC 算法减少了内存存取指令数，但增加了寄存器数和加法指令数。

图 5 对比了各种算法的运行时钟周期，可见 $n$ 相同时 SCOC 算法用到的时钟周期最少。分析原因可知，SCOC 算法虽然增加了 DADDU 指令数，但明显减少了 LD/ST 操作数。LD/ST 和 DADDU 指令的执行时间相同，但 LD 操作有结果延迟。当寄存器较少时，会造成严重的结果延迟。SCOC 算法增加寄存器数并合理安排乘积的顺序，有效减少了 LD 指令数且避免延迟带来的开销，所以其效率

表 5 几种 Montgomery 模乘算法使用寄存器数及指令数对比

| 算法 | 寄存器数 | 指令数 | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | LD | ST | LD/ST | DADDU | DMULTU | MFLO | MFHI |
| CIOS (192 bit,$n$=3) | 10 | 24 | 4 | 28 | 69 | 21 | 21 | 18 |
| CIOS-5reg (192 bit,$n$=3) | 6 | 51 | 27 | 78 | 69 | 21 | 21 | 18 |
| FIPS (192 bit,$n$=3) | 7 | 36 | 7 | 43 | 80 | 21 | 21 | 18 |
| SCOC (192 bit,$n$=3,$e$=3) | 11 | 18 | 10 | 28 | 77 | 21 | 21 | 18 |
| CIOS (256 bit,$n$=4) | 11 | 40 | 5 | 45 | 124 | 36 | 36 | 32 |
| CIOS-5reg (256 bit,$n$=4) | 6 | 84 | 44 | 128 | 124 | 36 | 36 | 32 |
| FIPS (256 bit,$n$=4) | 7 | 64 | 9 | 73 | 148 | 36 | 36 | 32 |
| SCOC (256 bit,$n$=4,$e$=4) | 13 | 24 | 13 | 37 | 147 | 36 | 36 | 32 |
| CIOS (512 bit,$n$=8) | 15 | 144 | 9 | 153 | 504 | 136 | 136 | 128 |
| CIOS-5reg (512 bit,$n$=8) | 6 | 304 | 144 | 448 | 504 | 136 | 136 | 128 |
| FIPS (512 bit,$n$=8) | 7 | 256 | 17 | 273 | 620 | 136 | 136 | 128 |
| SCOC (512 bit,$n$=8,$e$=6) | 17 | 71 | 40 | 111 | 633 | 136 | 136 | 128 |
| SCOC-c (512 bit,$n$=8,$e$=6) | 18 | 69 | 37 | 106 | 634 | 136 | 136 | 128 |
| CIOS-5reg (1 024 bit,$n$=16) | 6 | 1 104 | 560 | 1 664 | 2 032 | 528 | 528 | 512 |
| FIPS (1 024 bit,$n$=16) | 7 | 1 024 | 33 | 1 057 | 2 524 | 528 | 528 | 512 |
| SCOC (1 024 bit,$n$=16,$e$=6) | 17 | 229 | 127 | 356 | 2 627 | 528 | 528 | 512 |
| SCOC-c (1 024 bit,$n$=16,$e$=6) | 18 | 215 | 111 | 326 | 2 619 | 528 | 528 | 512 |

要高于其他几种算法。$n$=3,$e$=3 时,SCOC 比 CIOS算法减少了 28 个时钟周期(7.52%),$n$=16,$e$=6时,减少了 368 个时钟周期(4.17%),随着 $n$ 的增大,SCOC 算法减少的时钟周期数增大。与 SCOC 算法相比,SCOC-c 算法的效率也有所提高。

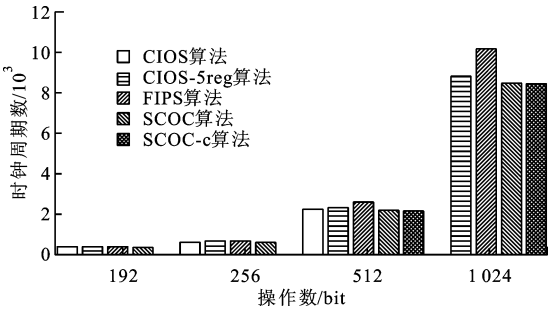



图 5 几种 Montgomery 算法运行时钟周期数的对比

## 4 结 论

本文提出一种面向嵌入式系统的 Montgomery 模乘 SCOC 算法,该算法在 COC 算法的基础上使用操作数缓存技术并进行了进位处理方式的改进,进一步减少了内存存取的数量。在基于 MIPS64 架构的 RISC 处理器上实现该算法和几种典型算法,结果表明该算法的运行效率明显优于其他几种算法。另外,由于 SCOC 算法的内部循环使用乘积扫描的方式,比较适用于支持乘积累加指令的处理器,因此进一步的工作可以在 MIPS64 指令集上扩展开发乘积累加指令,提高算法的实现效率。

**参考文献:**

[1] RIVEST R L, SHAMIR A, ADLEMAN L. A method for obtaining digital signatures and public-key cryptosystems [J]. Communications of the ACM, 1983, 26(2): 96-99.

[2] HANKERSON D, VANSTONE S, MENEZES A. Guide to elliptic curve cryptography [M]. Berlin, Germany: Springer-Verlag, 2004: 1-21.

[3] MONTGOMERY P L. Modular multiplication without trial division [J]. Mathematics of Computation, 1985, 44(170): 519-521.

[4] KOC C K, ACAR T, KALISKI B S. Analyzing and comparing Montgomery multiplication algorithms [J]. IEEE Micro, 1996, 16(3): 26-33.

[5] CHU D, GROBSCHADL J, LIU Zhe, et al. Twisted Edwards-form elliptic curve cryptography for 8-bit AVR-based sensor nodes [C] // Proceedings of the ACM Workshop on Asia Public-Key Cryptography. New York, USA: ACM, 2013: 39-44.

(下转第 127 页)