# Modular Exponentiation Algorithm Analysis
## for Energy Consumption and Performance

Lin Zhong
lzhong@princeton.edu
Technical Report CE-01-ZJL
Dept. of Electrical Engineering
Princeton University

**Abstract**

Modular Exponentiation for very large integers is the core of many modern cryptographic algorithms. Figure 0 shows how modular exponentiation is implemented in the essence of most algorithms. The basic building blocks are 1) modular algorithm, 2) multiplication of short integers and 3) addition of short integers. In this project, how the building blocks build up the modular exponentiation in popular algorithms is studied in terms of their complexity, parallelism and latency. Insights are found for tradeoff between energy consumption and performance for cryptosystem implementation. Moreover, the analyses shed insights for ISA and micro-architecture researches for cryptosystem.

## 1. Standard Multiplication

Suppose $a$ and $b$ are integers of $N$ bits. $a_i$ is the $ik$ bit to $(i+1)k-1$ bit of $a$. $b_i$ is the $ik$ bit to $(i+1)k-1$ bit of $b$. Therefore, $a_i$ and $b_j$ are integers of $k$ bits. The standard multiplication algorithm implements $N$ x $N$-bit multiplication by dividing N-bit integers into $\frac{N}{k}$ k-bit integers and doing k x k-bit multiplications/additions.

### 1.1 the Algorithm and Complexity

1.2 The algorithm is showed in Algorithm 1. Let $s = \frac{N}{k}$.

Input: a, b
Output: t = a - b
**0.** Initially $t_i$ : = 0 for all i = 0; 1,.., 2s-1.
**1.** for i = 0 to s - 1
**2.** C := 0
**3.** for j = 0 to s- 1
**4.** (C, S) :=$t_{i+j}$ + $a_{j*}$ $b_i$ + C
**5.** $t_{i+j}$ := S
**6.** $t_{i+s}$ := C
**7.** return ($t_{2s-1}t_{2s-2}$ ··· $t_0$)

**Algorithm 1. The Standard Multiplication Algorithm**

For line 4, we have four arithmetic operations which are shown in the Table 1.1. Let's suppose all the operands are assigned to k-bit registers.

| | |
|---|---|
| +1 $t_{i+j}$ +C => (C, S) | *1 $a_j * b_i$ =>(H, L) |
| +2 H+C => C | +3 L+S =>(C', S) |
| +4 C'+C => C | |

**Table 1.1**

It should be noted that H+C will not overflow. The data flow graph for line 4 is showed in Figure 1.1.

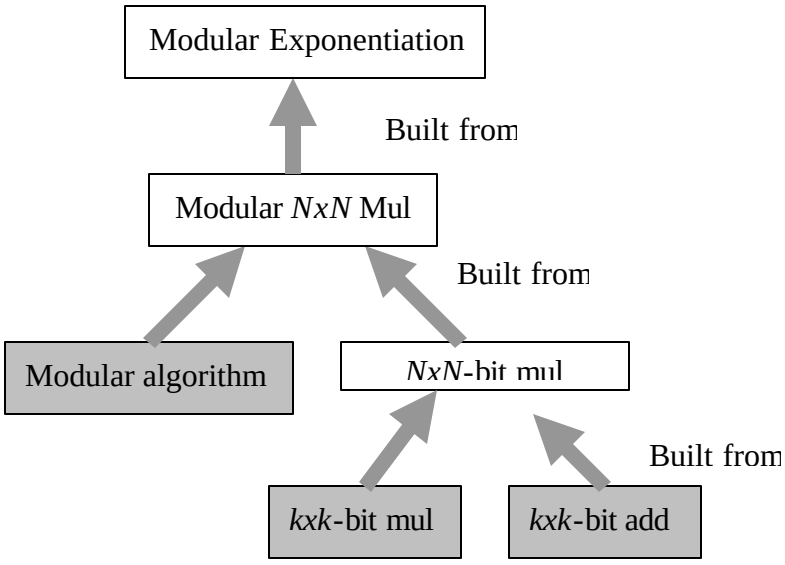



**Figure 0. What're the building blocks?**

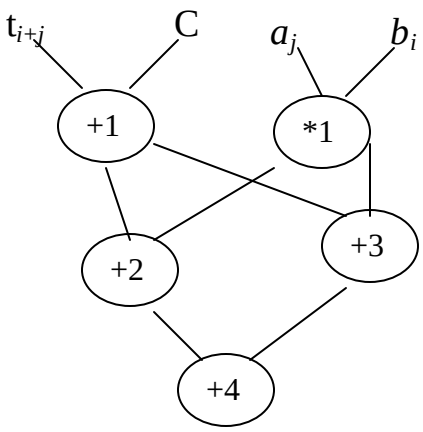



**Figure 1.1**

For Algorithm 1, we have $4(\frac{N}{k})^2$ *k x k*-bit additions and $(\frac{N}{k})^2$ k x k-bit multiplications. The complexity is

$$T(N) = 4(\frac{N}{k})^2 Add(k) + (\frac{N}{k})^2 Mul(k) \qquad (1)$$

Standard multiplication can be reduced if ***a=b,*** which is true in the modular exponentiation detailed in section 4. For $a=b$, we have

$$T(N) \approx 4(\frac{N}{k})^2 Add(k) + \frac{1}{2}(\frac{N}{k})^2 Mul(k) + \frac{1}{2}(\frac{N}{k})^2 Shift(k,1)$$

**1.2 Parallelism Analysis**

**1.2.1 Without Consideration of Carry-Ins**

Now we are going to analyze how much operational parallelism there is. Figure 1.2 shows the case when $s = \frac{N}{k} = 4$.

All the multiplication can be carried out in parallel. To get $t_i$ for $i = 0,..,2s-1$, we have to add 3 *k*-bit integers for $t_1$ and $t_6$, 5 *k*-bit integers for $t_2$ and $t_5$, 7 *k*-bit integers for $t_3$ and $t_4$.

We can generalize that we have to add $2i+1$ *k*-bit integers for $t_i$ for $i = 0,..,\frac{N}{k}-1$ and add $2(\frac{2N}{k} - i - 1) + 1$ *k*-bit integers for $t_i$ for $i = \frac{N}{k},..,\frac{2N}{k}-1$.

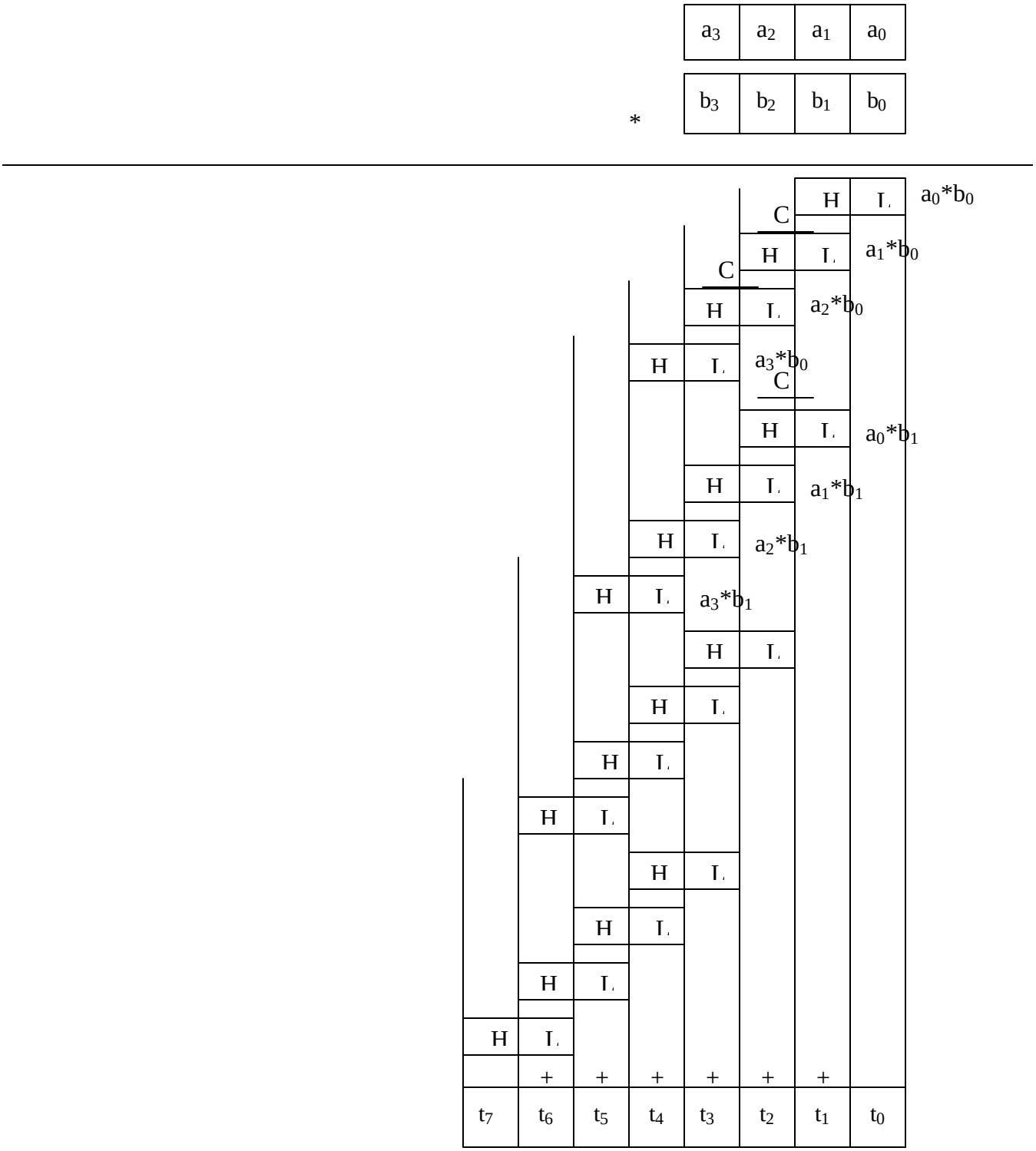

a3 a2 a1 a0
*
b3 b2 b1 b0
a0*b0
a1*b0
a2*b0
a3*b0
a0*b1
a1*b1
a2*b1
a3*b1
+ + + + + + +
t7 t6 t5 t4 t3 t2 t1 t0

**Figure 1.2**

Suppose our adder hardware adds two *k*-bit integers together each operation and output the carry-in and sum. The additions of those integers for $t_i$ can be done in a parallel way. The latency-optimized way is to implement them in a sequence of a tree structure. For example, Figure 1.3 shows to add 5 k-bit integers together in a tree sequence. It's optimal w.r.t the latency which is proportional to the number of levels the tree has.

More generally, to add $w$ k-bit integers, the tree will have $w-1$ nodes and $\log_2 w+1$ levels. If we relax the number of operands for the adder hardware to $m$ instead of 2, the latency-optimized way is the *m*-ary tree implementation. The tree will have $\left\lceil \frac{w-1}{m-1} \right\rceil$ nodes and $\log_m w+1$ levels. The number of nodes is the indicator of energy consumption while the number of the levels is the indicator of latency/performance.

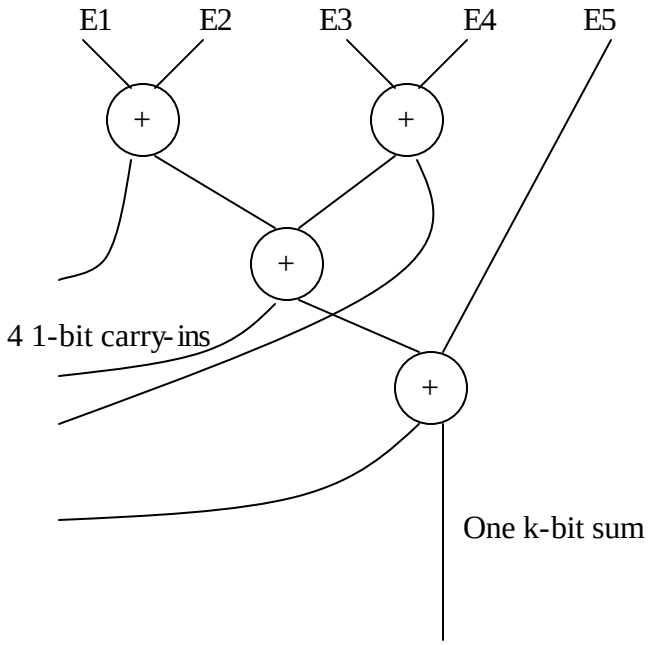



**Figure 1.3 Binary Tree for a Series of Additions**

There is one way to implement Algorithm 1 so that the parallel is more utilized. That is, we can compute $t_i, i = 0,..,\frac{2N}{k}-1,$ consecutively. If we implement the addition in the tree way, the latency is $Add(k)\{1+\log_2(2i+1)\}$ for $t_i$ for $i = 0,..,\frac{N}{k}-1$ and $Add(k)[1+\log\{2(\frac{2N}{k}-i-1)+1\}]$ for $t_i$, $i = \frac{N}{k},..,\frac{2N}{k}-1$. The total latency due to addition is $2Add(k)\sum_{i=0}^{\frac{N}{k}-1}\{1+\log_2(2i+1)\} = 2Add(k)\log_2\prod_{i=0}^{\frac{N}{k}-1}(2i+1)+2\frac{N}{k}Add(k)$.

On the other hand, we have

$$\log_2 \prod_{i=0}^{\frac{N}{k}-1}(2i+1) = \log_2 \prod_{i=1}^{\frac{2N}{k}-1} i - \log_2 \prod_{i=1}^{\frac{N}{k}-1} 2i = \log_2 \prod_{i=1}^{\frac{2N}{k}-1} i - \log_2 \prod_{i=1}^{\frac{N}{k}-1} i - \frac{N}{k} + 1 = \log_2\left(\frac{2N}{k}-1\right)! - \log_2\left(\frac{N}{k}-1\right)! - \frac{N}{k}$$

Applying ***Stirling's approximation of n!***, we have

$$\log_2 \prod_{i=0}^{\frac{N}{k}-1}(2i+1) = \log_2\left(\frac{2N}{k}-1\right)! - \log_2\left(\frac{N}{k}-1\right)! - \frac{N}{k} + 1$$

$$\approx \left(\frac{2N}{k}-\frac{1}{2}\right)\log_2\left(\frac{2N}{k}-1\right) - (\log_2 e)\left(\frac{2N}{k}-1\right) - \left(\frac{N}{k}-\frac{1}{2}\right)\log_2\left(\frac{N}{k}-1\right) + (\log_2 e)\left(\frac{N}{k}-1\right) - \frac{N}{k} + 1$$

$$= \left(\frac{2N}{k}-\frac{1}{2}\right)\log_2\left(\frac{2N}{k}\right) - \left(\frac{N}{k}-\frac{1}{2}\right)\log_2\left(\frac{N}{k}\right) - (\log_2 e + 1)\frac{N}{k} + 3$$

$$= \left(\frac{N}{k}-\frac{1}{2}\right)\log_2\left(\frac{N}{k}\right) + \left(\frac{2N}{k}-\frac{1}{2}\right) - (\log_2 2e)\frac{N}{k} + 3 \approx \frac{N}{k}(\log_2 \frac{N}{k} - \log_2 \frac{e}{2}) = \frac{N}{k}\log_2 \frac{2N}{ek}$$

Then the latency due to addition is $2Add(k)\frac{N}{k}(1+\log_2 \frac{2N}{ek}) = 2Add(k)\frac{N}{k}\log_2 \frac{4N}{ek}$

Moreover, all the *k*-bit multiplication can be done in parallel if we have enough hardware sources. Therefore, the latency for the implementation of Algorithm 1 in this way will be

$$Latency(N) = 2Add(k)\frac{N}{k}\log_2 \frac{4N}{ek} + Mul(k) \qquad (2)$$

(2) is significantly less than (1). This means if we utilize the parallelism in Algorithm 1, the performance could be significantly better. However, as we didn't reduce the number of operations in the parallel implementation, (2) still holds for the energy consumption of Algorithm 1.

### 1.2.2 Problem of Carry-In

If the adder adds two integers together each time, the carry-in is always 1-bit. Carry-ins can be taken into consideration with a conditional increment operation (We'll talk about this in the conclusion).

If(C =1) increase S by 1.

We showed that we have to add $2i+1$ k-bit integers for $t_i$ for $i = 0,..,\frac{N}{k}-1$ and add $2(\frac{2N}{k}-i-1)+1$ k-bit integers for $t_i$, $i = \frac{N}{k},..,\frac{2N}{k}-1$. The number of carry-ins for $t_i$, $i = 1,..,\frac{N}{k}-1$, is $2(i-1)$. That for $t_i, i = \frac{N}{k},..,\frac{2N}{k}-1$, is $2(\frac{2N}{k}-i)$.

If we take all the carry-ins into consideration and treat them as additions, the latency will at most double for the implementation proposed in 1.2.1.

# 2. Karatsuba-Ofman Algorithm

Karatsuba-Ofman algorithm keeps on dividing the long integer into two shorter ones of equal sizes until their lengths are *k*. Karatsuba-Ofman algorithm gets the multiplication of two long integers by doing multiplications and additions on their divided parts of the half length.

### 2.1 the Algorithm and Complexity

Let $a_1$ and $a_0$ denote the higher and the lower halves of *a*, respectively. Let $b_1$ and $b_0$ denote the higher and the lower halves of *b*, respectively.

KORMA(a, b)
**1. if(a and b are of more than 2k bits) do**
**2.** $t_0$ : = KORMA($a_0$, $b_0$) // $T(\frac{N}{2})$
**3.** $t_2$ : = KORMA($a_1$, $b_1$) // $T(\frac{N}{2})$
**4.** $u_0$ : = KORMA($a_1$ + $a_0$, $b_1$ + $b_0$) // $T(\frac{N}{2}+1)+2Add(\frac{N}{2})$
**5.** $t_1$ : = $u_0$ - $t_0$ - $t_2$ // $2Add(N)$
**6. else do**
**7.** $t_0$ : = $a_0$*$b_0$
**8.** $t_2$ : = $a_1$*$b_1$
**9.** $u_0$ : = ($a_1$ + $a_0$)*($b_1$ + $b_0$)
**10.** $t_1$ : = $u_0$ - $t_0$ - $t_2$
**11.** return ( $2^N t_2 + 2^{\frac{N}{2}} t_1 + t_0$ ) // $Shift(N,N)+Shift(N,\frac{N}{2})+Add(2N,N)$

**Algorithm 2. Karatsuba-Ofman Algorithm**

Let T(*N*) denote the arithmetic/logic operations needed for *NxN*-bit multiplication. Then we have

$$T(N) = 2T(\frac{N}{2}) + T(\frac{N}{2}+1) + 2Add(N) + 2Add(\frac{N}{2}) + Add(2N,N) + Shift(N,N) + Shift(N,\frac{N}{2})$$

$$= 2T(\frac{N}{2}) + T(\frac{N}{2}+1) + 5Add(N) + Shift(N,N) + Shift(N,\frac{N}{2})$$

Where *Add(N)* denotes the *N x N* bit operations. Note that line 11 only requires one *2N x N* addition, which can be achieved with 2 consecutive *NxN* additions.

On the other hand, if we implement (N/2+1)x(N/2+1) in the following way.

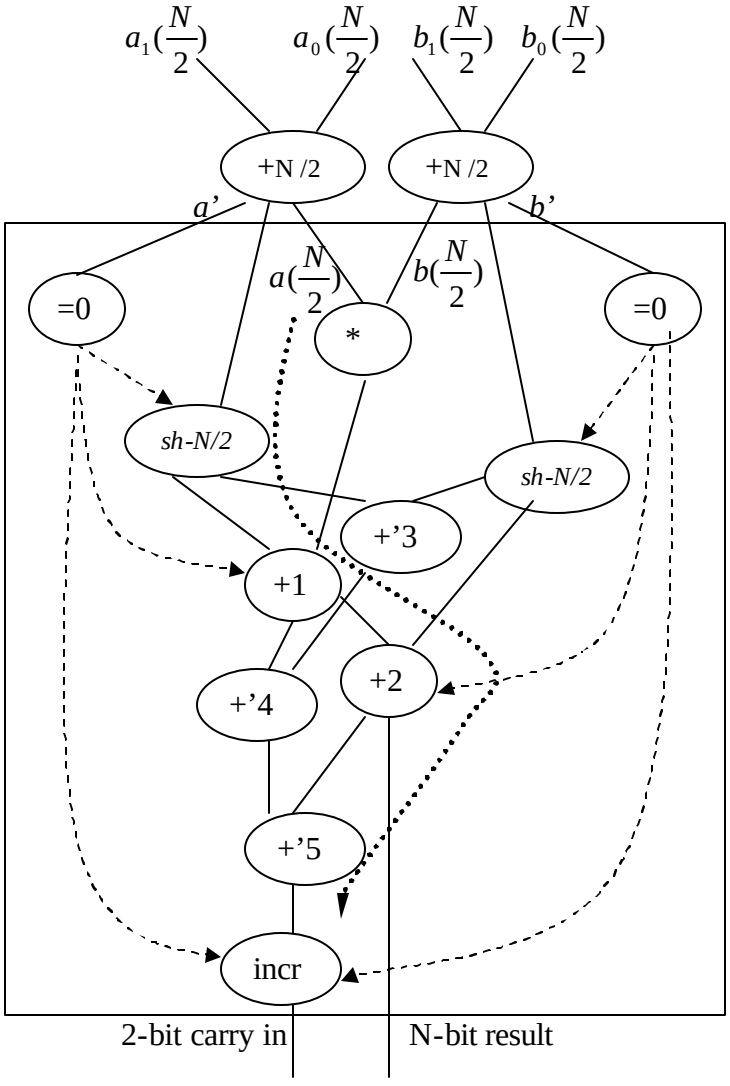
$$a(\frac{N}{2}+1) * b(\frac{N}{2}+1) = \{a'*2^{\frac{N}{2}} + a(\frac{N}{2})\} * \{b'*2^{\frac{N}{2}} + b(\frac{N}{2})\}$$

$$= a'*b(\frac{N}{2})*2^{\frac{N}{2}} + b'*a(\frac{N}{2})*2^{\frac{N}{2}} + a(\frac{N}{2})*b(\frac{N}{2}) + a'*b'*2^{N}$$

where a' and b' are one bit, we have

$$T(\frac{N}{2}+1) = T(\frac{N}{2}) + 2Shift(\frac{N}{2},\frac{N}{2}) + 3condition + 2Add(N) + 4Add(2)$$

The control-data flow graph for line 4 of Algorithm 2 and computing $Add(\frac{N}{2}+1)$ is showed in Figure 2.1. The critical path is showed with the thick dashed arrow.




KORMA(N/2+1)

**Figure 2.1 +' is 2-bit addition**

Then we have

$$T(N) = 3T(\frac{N}{2}) + 7Add(N) + 4Add(2) + Shift(N,N) + Shift(N,\frac{N}{2}) + 2Shift(\frac{N}{2},\frac{N}{2}) + 3condition \quad (3)$$

If the recurrence stops when *a* and *b* are *k* bits, we have

$$T(N)=3^{\log(\frac{N}{k})}T(k)+7\sum_{i=1}^{\log(\frac{N}{k})}3^{i-1}Add(\frac{2N}{2^i})+4*3^{\log(\frac{N}{k})-1}Add(2)$$

$$+Shift(N,N)+\sum_{i=1}^{\log(\frac{N}{k})}3^{i-1}Shift(\frac{2N}{2^i},\frac{N}{2^i})+5\sum_{i=1}^{\log(\frac{N}{k})-1}3^{i-1}Shift(\frac{N}{2^i},\frac{N}{2^i})+2*3^{\log(\frac{N}{k})}Shift(k,k)+3^{\log(\frac{N}{k})}condition$$

$$=\left(\frac{N}{k}\right)^{\log 3}T(k)+7\sum_{i=1}^{\log(\frac{N}{k})}3^{i-1}Add(\frac{2N}{2^i})+\frac{4}{3}*\left(\frac{N}{k}\right)^{\log 3}Add(2)$$

$$+Shift(N,N)+\sum_{i=1}^{\log(\frac{N}{k})}3^{i-1}Shift(\frac{2N}{2^i},\frac{N}{2^i})+5\sum_{i=1}^{\log(\frac{N}{k})-1}3^{i-1}Shift(\frac{N}{2^i},\frac{N}{2^i})+2*\left(\frac{N}{k}\right)^{\log 3}Shift(k,k)+\left(\frac{N}{k}\right)^{\log 3}conditio$$

$$=\left(\frac{N}{k}\right)^{\log 3}\{Mul(k,k)+\frac{4}{3}Add(2)+2Shift(k,k)+conidtion\}+$$

$$7\sum_{i=1}^{\log(\frac{N}{k})}3^{i-1}Add(\frac{2N}{2^i})+\sum_{i=1}^{\log(\frac{N}{k})}3^{i-1}Shift(\frac{2N}{2^i},\frac{N}{2^i})+5\sum_{i=1}^{\log(\frac{N}{k})-1}3^{i-1}Shift(\frac{N}{2^i},\frac{N}{2^i})+Shift(N,N)$$

Let's assume that $Add(N)=2Add(\frac{N}{2})$ ( indeed, the NxN addition can be implemented as two *N/2 x N/2* additions in sequence), then we have $Add(\frac{2N}{2^i})=2^{\log(\frac{N}{k})-i+1}Add(k)$ . Therefore, we have

$$T(N)\approx\left(\frac{N}{k}\right)^{\log 3}\{Mul(k,k)+\frac{4}{3}Add(2)+2Shift(k,k)+conidtion\}+7\sum_{i=1}^{\log(\frac{N}{k})}3^{i-1}2^{\log(\frac{N}{k})-i+1}Add(k)$$

$$\approx\left(\frac{N}{k}\right)^{\log 3}\{Mul(k,k)+\frac{4}{3}Add(2)+2Shift(k,k)+conidtion\}+14\left(\frac{N}{k}\right)^{\log 3}Add(k)$$

$$\approx\left(\frac{N}{k}\right)^{\log 3}Mul(k,k)+14\left(\frac{N}{k}\right)^{\log 3}Add(k)$$

(4)

Table 2.1 shows the ratio of complexities for Algorithm 1 and Algorithm 2 for *N* and *k* of typical values (The complexity of *Mul(k)* is assumed to be 3 times that of *Add(k)*).

| $\frac{T_{STD}(N)}{T_{KO}(N)}$ | N=256 | N=512 | N=1024 | N=2048 |
|---|---|---|---|---|
| k=16 | 1.30 | 1.74 | 2.31 | 3.08 |
| k=32 | 0.98 | 1.30 | 1.74 | 2.31 |
| k=64 | 0.73 | 0.98 | 1.30 | 1.74 |

**Table 2.1**

**2.2 Parallelism Analysis**

Algorithm 2 is very desirable for parallel implementation due to its recurrent nature. As in section 1, let's first forget about the carry-ins. All the k-bit multiplication in Algorithm 1 can be done in parallel if we have enough hardware resources. All the *W x W*-bit multiplications can be done in parallel with the results from all the *W/2 x W/2*-bit multiplications. In line 4 of Algorithm 2, the two additions can be done in parallel.

Figure 2.2 shows the data-flow graph for one recurrent level for Algorithm 2. The critical path is showed with the dashed arrow.

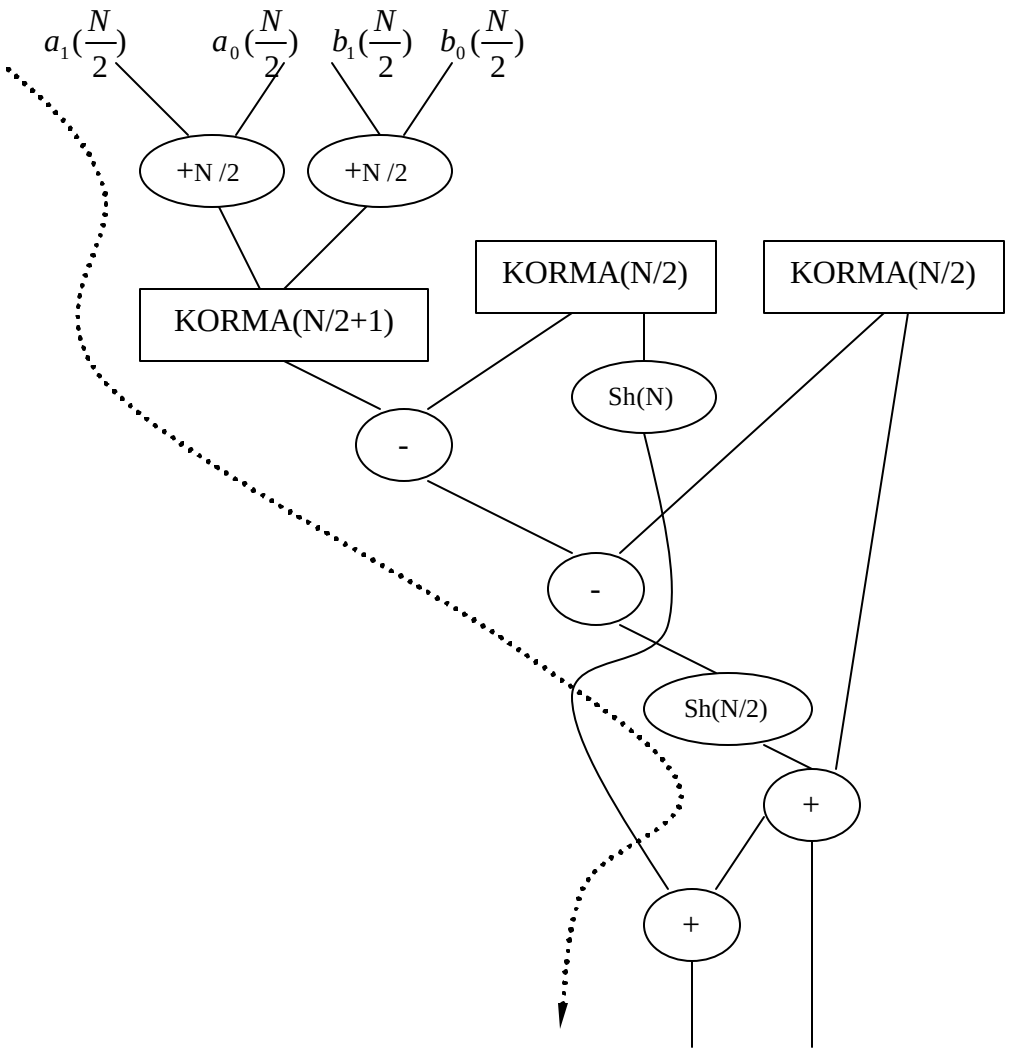



**Figure 2.2 DFG for one recurrence of Algorithm 2 and the critical path**

According to Figure 2.2, the latency for KORMA(W) is

$$Latency(W) \approx Latency(\frac{W}{2}+1) + Add(\frac{W}{2}) + 4Add(W) + Shift(W, \frac{W}{2})$$

According to Figure 2.1, the latency for KORMA(W/2+1) is

$$Latency(\frac{W}{2}+1) = Latency(\frac{W}{2}) + 2Add(W) + Add(2) + Incr(2)$$

Therefore

$$Latency(W) \approx Latency(\frac{W}{2}) + 6Add(W) + Add(\frac{W}{2}) \approx Latency(\frac{W}{2}) + 13Add(\frac{W}{2})$$

$$\approx Latency(\frac{W}{2}) + \frac{13W}{2k} Add(k)$$

Therefore we have

$$Latency(N) \approx Mul(k) + \frac{13}{2k} Add(k) \sum_{i=\log k+1}^{\log N} 2^i = Mul(k) + \frac{13}{2k}(2N - 2k)Add(k) \approx Mul(k) + 13\frac{N}{k} Add(k)$$

(5)

(5) is significantly less than (4). This means if we utilize the parallelism in Algorithm 2, the performance could be significantly better. However, as we didn't reduce the number of operations in the parallel implementation, (4) still holds for the energy consumption of Algorithm 2.

If we take all the carry-ins into consideration, the latency will at most double.

For Algorithm 1, we have

$$Latency(N) \approx 2\frac{N}{k}\log_2 \frac{4N}{ek} Add(k) + Mul(k)$$

for the parallel implementation proposed in section 1. The ratio of latencies for the two algorithms is $\frac{2}{13}\log_2 \frac{4N}{ek}$ .Table 2.2 shows the typical values for $\frac{2}{13}\log_2 \frac{4N}{ek}$ .

| $\frac{2}{13}\log_2 \frac{4N}{ek}$ | k=16 | k=32 | k=64 |
|---|---|---|---|
| N=256 | 0.89 | 0.78 | 0.68 |
| N=512 | 1.04 | 0.94 | 0.83 |
| N=1024 | 1.20 | 1.09 | 0.98 |
| N=2048 | 1.35 | 1.24 | 1.14 |

**Table 2.2 Typical values for** $\frac{2}{13}\log_2 \frac{4N}{ek}$

Table 2.2 shows that Algorithm 1 is slightly better than Algorithm with respect to how much parallelism there is. However, Algorithm 2 still has significant advantages over Algorithm 1 with respect to the energy consumption. Moreover, when $N$ gets larger and larger, (2) will scale faster than (5) and finally lose its advantage.

# 3. Modular Multiplication.

Suppose *a* is a *2N*-bit integer and *n* is a *N*-bit integer.

## 3.1 Direct Modular Algorithms

### 3.1.1 Restoring Division Algorithm and Complexity

Input: a, n
Output: R = a mod n
1. $R_0$ := a
2. n := $2^N$ n  $Shift(N, N)$
3. for i = 1 to N
4. $R_i := R_{i-1} - n$  $Add(2N,2N)$
5. if $R_i < 0$ then $R_i := R_{i-1}$  $condition$
6. n := n/2  $Shift(2N,1)$
7. return $R_k$

**Algorithm 3. The Restoring Division Algorithm 1.**

The cost for Algorithm 3 is

$$T(N) = shift(N,N) + N\{Add(2N,2N) + condition + shift(2N,1)\}$$

$$\approx N * 2Add(N) + N * condition \approx \frac{2N^2}{k} Add(k) + N * condition$$

Or, it can be implemented so that

$$Average(T(N)) = shift(N,N) + N\{\frac{1}{2} Add(2N) + \frac{3}{2} condition + shift(2N,1)\}$$

$$\approx N * Add(N) + \frac{3}{2} N * condition \approx \frac{N^2}{k} Add(k) + \frac{3}{2} N * condition$$

Input: a, n
Output: R = a mod n
**1.** $R_0$ := a
**2.** n := $2^N$ n  // $Shift(N, N)$
**3.** for i = 1 to N
**4.** if(the most significant bit of $R_{i-1}$ is 0) do  // $condition$
**5.** $R_i := R_{i-1}$
**6.** else do $R_i := R_{i-1} - n$  // $Add(2N,2N)$
**7.** if $R_i < 0$ then $R_i := R_{i-1}$  // $condition$
**8.** n := n/2  // $Shift(2N,1)$
**9.** return $R_k$

**Algorithm 3.1 The Restoring Division Algorithm 2**

### 3.1.2 Norestoring Division Algorithm and Complexity

Input: a, n
Output: R = a mod n
**1.** $R_0$ := a
**2.** n := $2^N$n // $Shift(N, N)$
**3.** for i = 1 to N
**4.** if $R_{i-1} > 0$ // $condition$
**5.** then $R_i := R_{i-1} - n$ // $Add(2N,2N)$
**6.** else $R_i := R_{i-1} + n$ // $Add(2N,2N)$
**7.** n := n/2 // $Shift(2N,1)$
**8.** if $R_k < 0$ then R := R + n // $condition$ , $Add(2N,2N)$
**9.** return $R_k$

**Algorithm 4. The Nonrestoring Division Algorithm**

The cost for Algorithm 4 is

$$Average(T(N)) \approx N * Add(N) + N * condition \approx \frac{N^2}{k} Add(k) + N * condition$$

### 3.1.3 $a*b$ mod $n$

Suppose the Karatsuba-Ofman algorithm is used for multiplication and the non-restoring division algorithm is applied for modular. If a, b and n are N-bit integers, the cost for $a*b$ mod $n$ is

$$T(N) \approx Mul(N) + \frac{N^2}{k} Add(k) + N * condition$$

$$\approx \left(\frac{N}{k}\right)^{\log 3} Mul(k,k) + 14\left(\frac{N}{k}\right)^{\log 3} Add(k) + \frac{N^2}{k} Add(k) + N * condition$$

$$\approx \left(\frac{N}{k}\right)^{\log 3} Mul(k,k) + \{14\left(\frac{N}{k}\right)^{\log 3} + \frac{N^2}{k}\} Add(k) + N * condition$$

### 3.1.4 Parallelism Analysis

The control-data flow graph for a latency-optimized implementation for the nonrestoring division algorithm line 4-7 is showed in Figure 3.1

If the algorithm is implemented in the way of figure 3.1,

$$Average(T(N)) \approx 2N * Add(N) + N * Mux \approx \frac{2N^2}{k} Add(k) + N * Mux$$

The latency will be

$$Average(T(N)) \approx N * Add(N) + N * Mux = \frac{N^2}{k} Add(k) + N * Mux .$$

The parallelism in the direct modular algorithms is very limited.

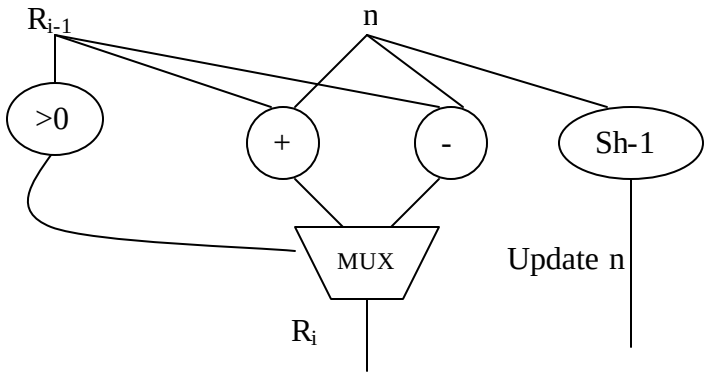



**Figure 3.1**

### 3.2 Blakley Algorithm

Blakley's method directly computes $a*b$ mod $n$ by interleaving the shift-add steps of the multiplication and the shift-subtract steps of the division.

Input: a, b, n
Output: R = a*b mod n
**1.** R : = 0
**2.** for i = 0 to N-1
**3.** R := 2R + $a_{N-1-i}$*b // $Shift(N,1) + Add(N) + condition$
**4.** R := R mod n // $2condition + 4Add(N)$
**5.** return R

**Algorithm 5. The Blakley Algorithm**

After line 3, $0 \le R \le 3n-3$. Therefore, line 4 is equal to

4.1 If $R \ge n$ do // $Add(N) + condition$
4.2 R:=R-n // $Add(N)$
4.3 If $R \ge n$ do // $Add(N) + condition$
4.4 R:=R-n // $Add(N)$

$$Average(T(N)) = N\{Shift(N,1) + 4Add(N) + 3condition\} = N * Shift(N,1) + 4\frac{N^2}{k}Add(k) + 3N * condition$$

If we ignore all the Shift and condition, we have for Blakley algorithm

$$Average(T(N)) \approx 4\frac{N^2}{k}Add(k)$$

For Karatsuba-Ofman Algorithm $T(N) \approx \left(\frac{N}{k}\right)^{\log 3} Mul(k,k) + \{14\left(\frac{N}{k}\right)^{\log 3} + \frac{N^2}{k}\}Add(k)$

Suppose the cost for *Mul(k)* is 3 times of the cost for *Add(k)*. Then we have the following tables for different $N$ and $k$.

| $\frac{T_{Blakley}(N)}{T_{KO}(N)}$ | N=256 | N=512 | N=1024 | N=2048 |
|---|---|---|---|---|
| k=16 | 2.99 | 3.19 | 3.36 | 3.50 |
| k=32 | 3.27 | 3.42 | 3.55 | 3.65 |
| k=64 | 3.48 | 3.60 | 3.69 | 3.76 |

**Table 3.1**

Table 3.1 shows that the Karatsuba-Ofman+Non-restoring division is much better than Blakley with regard to the cost computed as the number of k x k-bit add and multiplications.

### 3.2.1 Parallelism Analysis

Blakley Algorithm has little parallelism. In this respect, Blakley Algorithm is much worse than other choices.

## 3.3Montgomery Algorithm

*a*,*b* and *n* are *N*-bit integers. $r = 2^N$ and gcd(*r*,*n*)=1. Let *n'* be the integer so that $r * r^{-1} - n * n' = 1$.

### 3.3.1 The Algorithm and Complexity

**The Montgomery Product Algorithm**

$u = a*b*r^{-1} \bmod n$

function MonPro(a, b)

**1**. t := a*b // $Mul(N)$

**2.** m := t*n' mod r // $Mul(N)$

**3.** u := (t + m*n)/r // $Mul(N) + 2Shift(2N, N) + Add(N)$

**4.** if $u \geq n$ then return u-n // $Add(N) + condition + Add(N)$

else return u

**The Montgomery Multiplication Algorithm**

function ModMul(a, b, n) { n is an odd number}

**1.** Compute n' using the extended Euclidean algorithm.

**2.** a' := a*r mod n

// $2Shift(N, N) + N\{\frac{1}{2}Add(2N) + condition + Shift(2N, 1)\}$

**3.** x := MonPro(a', b)

// $3Mul(N) + \frac{3}{2}Add(N) + 2Shift(2N, N) + condition$

**4.** return x

**Algorithm 6. The Montgomery Multiplication Algorithm**

The average cost for Alorithm 6 is

$$Average(T(N)) = 3Mul(N) + \frac{3}{2}Add(N) + 2Shift(2N, N) + condition$$

$$+ 2Shift(N, N) + N\{\frac{1}{2}Add(2N) + condition + Shift(2N, 1)\}$$

$$\approx 3Mul(N) + N * Add(N) + N * condition$$

Compared with the Karatsuba-Ofman and Non-restoring division solution, it has no complexity advantage. However, we'll later show that Montgomery algorithm is much better when a sequence of multiplications is to be performed, which is the case in modular exponentiation.

### 3.3.2 Parallelism Analysis

Parallelism in the Montgomery modular multiplication algorithm is very little.

## 4. Modular Exponentiation

Suppose *e* is an M-bit integer. $x = a^e \bmod n$. *n* is an odd number. *n* and *a* are *N*-bit integers.

Karatsuba-Ofman Algorithm is chosen to do multiplication here. This algorithm requires much less computational resources than the standard algorithm, although the later is a little bit better with respect to parallel implementation.

From multiplication to exponentiation, the most popular way is the multiply-and-square way, which includes *m*-ary methods.

For binary multiply-and-square algorithm, averagely, we need *M NxN*-bit modular square and *M/2 NxN* modular multiplication. 4.1 and 4.2 detail the modular exponentiations built from different modular algorithms examined in section 3.

### 4.1 Karatsuba-Ofman + Nonrestoring division +Multiply-and-Square Algorithm

For Karatsuba-Ofman algorithm, square and multiplication are same in computation. Therefore, averagely, we need *3M/2* multiplication and *3M/2* non-restoring division. Therefore, according to section 3, we have

$$Average(T_{MAS}(N)) \approx \frac{3M}{2}\left[\left(\frac{N}{k}\right)^{\log 3} Mul(k,k) + \{14\left(\frac{N}{k}\right)^{\log 3} + \frac{N^2}{k}\}Add(k)\right]$$

### 4.2 Karatsuba-Ofman+Montgomery Modular +Multiply-and-Square Algorithm

**function ModExp(a, e, n) f n is an odd number g**
**1.** Compute n' using the extended Euclidean algorithm.
**2.** a' := a* r mod n

$$// 2Shift(N,N) + N\{\frac{1}{2}Add(2N) + condition + Shift(2N,1)\}$$

**3.** x' := r mod n
**4.** for i = M-1 down to 0 do
**5.** x := MonPro(x', x')

$$// 3Mul(N) + \frac{3}{2}Add(N) + 2Shift(2N,N) + condition$$

**6.** if $e_i$ = 1then x' := MonPro( a', x')

$$// 3Mul(N) + \frac{3}{2}Add(N) + 2Shift(2N,N) + 2*condition$$

**7.** x := MonPro(x',1)
**8.** return x

**Algorithm 7. Montgomery Modular Exponentiation Algorithm**

For Algorithm 7, we have

$$Average(T_{Mont}(N)) \approx \frac{3M}{2}\{3Mul(N) + \frac{3}{2}Add(N)\} + N * Add(N)$$

$$\approx \frac{3M}{2}\left[\left(\frac{N}{k}\right)^{\log 3} Mul(k,k) + \{14\left(\frac{N}{k}\right)^{\log} + \frac{3N}{2k}\}Add(k)\right] + \frac{N^2}{k}Add(k)$$

Obviously, Montgomery Algorithm is much better than the Nonrestoring division +Multiply-and-Square method..

If we assume the *Mul(k)* cost is 3 times of the *Add(k)* cost, we have the following table.

| $Average(T_{MAS}(N)/T_{Mont}(N))$ | N=1024,k=16 | N=1024,k=32 | N=1024,k=64 |
|---|---|---|---|
| M=1 | 0.96 | 1.08 | 1.18 |
| M=4 | 1.62 | 2.06 | 2.58 |
| M=16 | 1.95 | 2.67 | 3.67 |
| M=64 | 2.05 | 2.89 | 4.11 |
| M=256 | 2.08 | 2.95 | 4.23 |
| M=1024 | 2.09 | 2.96 | 4.26 |

**Table 4.1**

Table 4.1 clearly demonstrates Montgomery algorithm's advantage over other choices for doing a sequence of modular multiplication as in modular exponentiation. The reason is the cost of line 2 in **ModExp** is amortized by *M*. In section 3.3.1, it is not amortized in

**ModMul**, which is the reason why Montgomery algorithm is not good for doing modular multiplication.

# 5. Experiments

Different algorithms for multiplication, modular multiplication and modular exponentiation are implemented in C. We rely on the compiler to do codes optimization for parallelism and register assignments.

## 5.1 Software Power Estimation

People proposed methods to estimate the energy consumption for processor instructions, which is called the instruction-level power estimation. If the energy consumption for each instruction is known, switching energy consumption between different instructions, cache miss and memory accessing energy consumption are taken into consideration, the energy consumption for the software can be estimated with reasonable accuracy [Tiwari et al, 1994].

Researchers in MIT developed a web-based software profiling tool, ***JouleTrack***, that estimates the energy consumption of software on three standard processors. The user can upload the code, select the available processor operating conditions, memory maps, and compilation options and get a cycle accurate report of the program execution, along with energy statistics [Joule Track].

We use the StrongARM SA-1100 software profiler to estimate the energy consumption of our algorithms implemented in C if it is run on StrongARM SA-1100. (StrongARM processor is proposed for the portable computing devices for which the energy consumption is critical).

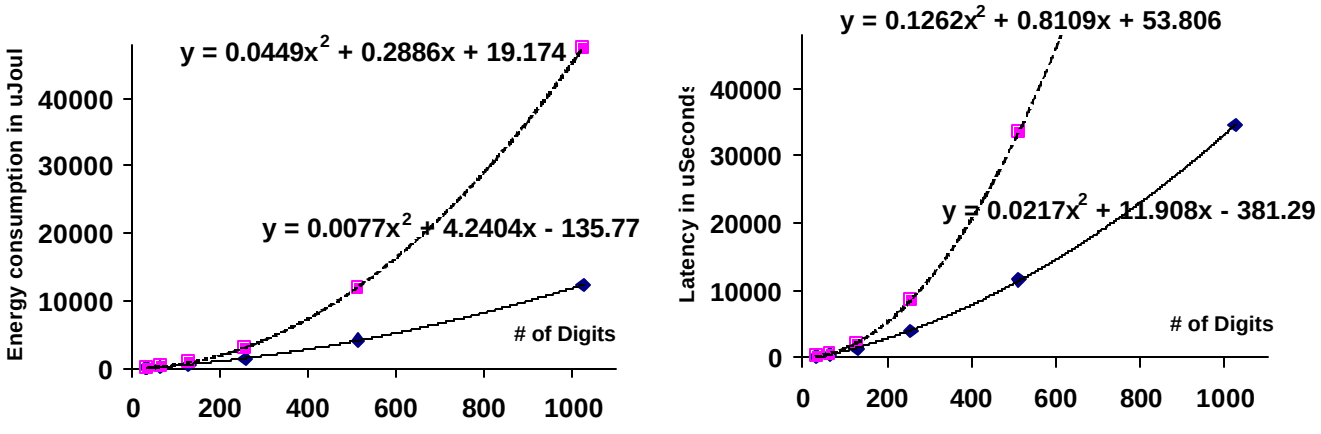
## 5.2 Multiplication

**Profiling Environment:**

StrongARM SA-1100 Energy Profiling Results

| | |
|---|---|
| Operating frequency | 206MHz |
| Operating voltage | 1.5 Volts |
| Simulation level | 0 |

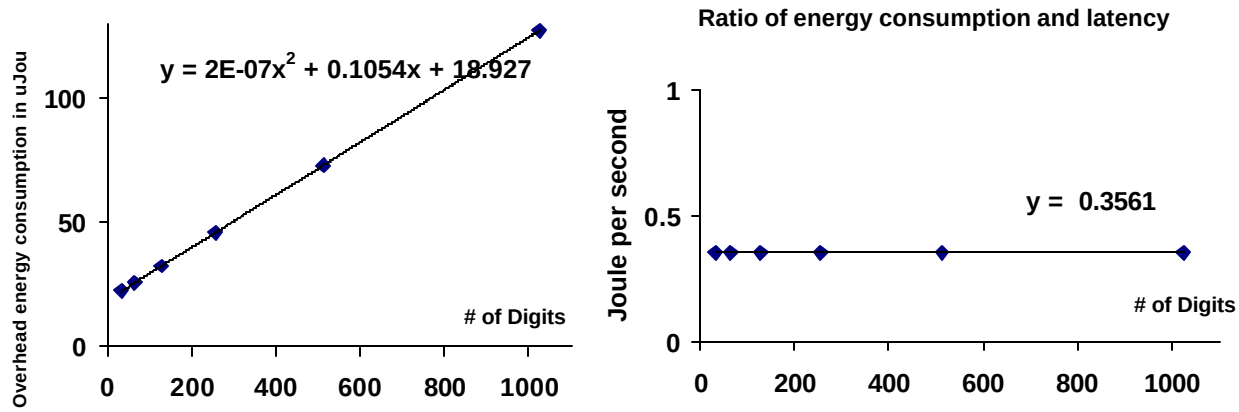
Figure 5.1a shows the energy consumption of Algorithm 1 and. Algorithm 2 for $N$ of various lengths. Here N denotes the number of digits instead bits. One digit is equal to $\log_2 10$ bits. The k for the Karatsuba-Ofman Algorithm (Algorithm 2) here is fixed as 4. Figure 5.1b shows the performance, in term of latency, of Algorithm 1 and Algorithm 2 for $N$ of the same values. It's obviously that Algorithm 2 is much better.




a. Energy Consumption b. Latency

**Figure 5.1**
The dashed line is for Algorithm 1 and the solid for Algorithm2

There are overhead codes like variable declaration and initialization. The energy consumption for the overhead is showed in Figure 5.2a. It is almost linear with respect to the number of digits.




a. Energy Consumption for the Overhead b. Latency/Energy consumption Correlation

**Figure 5.2**

### 5.3 Correlation between Latency and Energy Consumption

For StrongArm SA-1100, we found that the energy consumption is proportional to the performance in terms of latency. The longer the algorithm keeps the processor running, the more energy consumption. Taking Figure 5.1 for example, Figure 5.2b shows the ratio of energy consumption and latency of Algorithm 1 is almost constant for N of various lengths.

However, StrongArm SA-1100 doesn't utilize the parallelism of the software very much like VLIW processors do. As showed in section 1, if the processor can utilize the parallelism in the algorithm, the latency can be significantly reduced while the energy

consumption will stay unchanged. On the other hand, for a processor with little parallel processing capability like StrongArm SA-1100, the compiler automatically optimizes the energy consumption when it optimizes the latency. In this case, compilation for low power is the same as compilation for high performance.

### 5.4 Energy Consumption Breakdowns

***JouleTrack*** gives the energy consumption breakdowns with level 2 simulation complexity. For Algorithm 2 ( Karatsuba-Ofman algorithm), when N=1024 and k=4, breakdowns are showed in figure 5.3

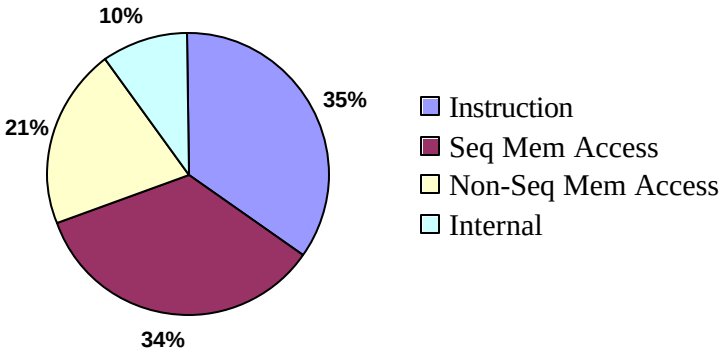



**Figure 5.3.**

Figure 5.3 clearly demonstrate the memory access (both sequential and nonsequential) consume more than half (55%) of the total energy.

## 6. Summary and Conclusion

### 6.1 Summary

The complexity for different modular exponentiation algorithm analyzed is summarized as follows.

**1.Karatsuba+Montgomery+ Multiply-and-Squaring**

$$Average(T_{Mont}(N)) \approx \frac{9M}{2}\left[\left(\frac{N}{k}\right)^{\log 3} Mul(k,k) + 14\left(\frac{N}{k}\right)^{\log 3} Add(k)\right] + \frac{N^2}{k} Add(k) + \frac{9}{4} M \frac{N}{k} Add(k)$$

**2.Karatsuba + Nonrestoring+Multiply-and-Squaring**

$$Average(T_{MAS}(N)) \approx \frac{3M}{2}\left[\left(\frac{N}{k}\right)^{\log 3} Mul(k,k) + \{14\left(\frac{N}{k}\right)^{\log 3} + \frac{N^2}{k}\} Add(k)\right]$$

**3. Standard + Non-restoring+Multiply-and-Squaring**

$$Average(T_{STD}(N)) \approx \frac{3}{2} M \{(4(\frac{N}{k})^2 + \frac{N^2}{k}) Add(k) + (\frac{N}{k})^2 Mul(k)\}$$

**4.Blakley+Multiply-and-Squaring**

$$Average(T_{Blakley}(N)) \approx 6M \frac{N^2}{k} Add(k)$$

For k=32, M=1024, suppose *1 Mul(k)= 3 Add(k)*, the complexity of 2-4 normalized by that of 1 is showed in Figure 6.1.

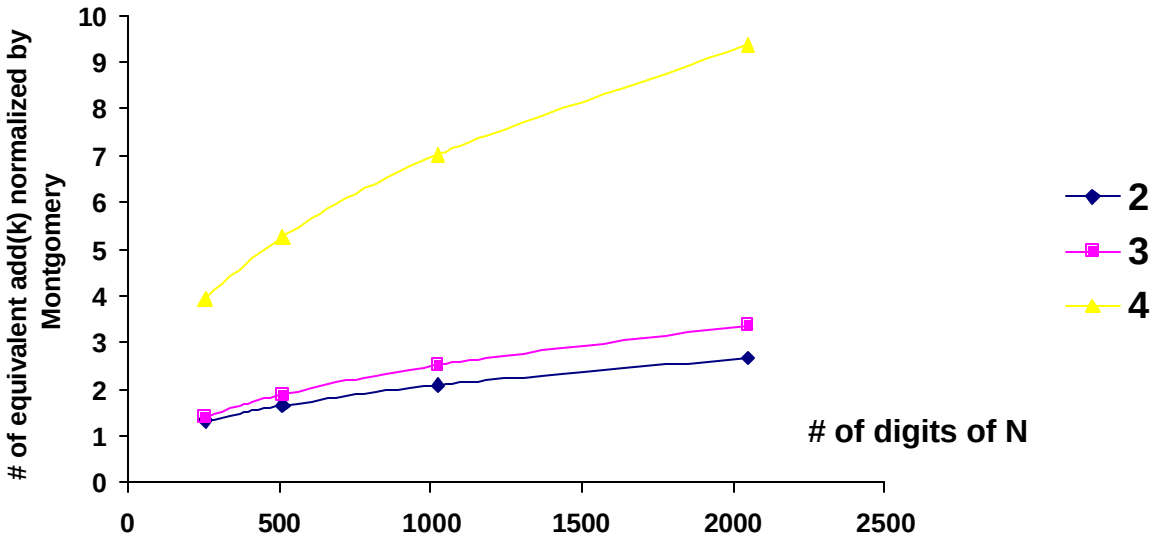



**Figure 6.1**

For Montgomery Algorithm (1), the complexity for different values of k is showed in Figure 6.2.

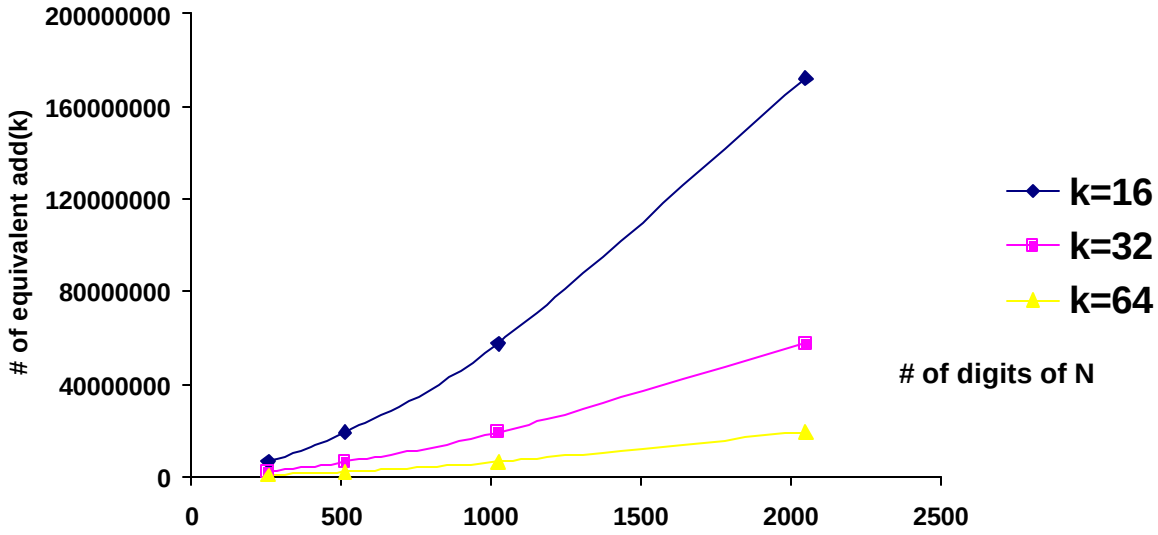



**Figure 6.2**

### 6.2 Importance of algorithm choices

Wise choosing the algorithm will save efforts most. From the analysis of previous sections, the best algorithm for modular exponentiation studied in this project is to use the Karatsuba-Ofman algorithm for integer multiplication, use the Montgomery algorithm for dealing with modular and use multi-and-square method for exponentiation.

### 6.3 Importance of efficient implementation for addition

All algorithm studied in this project are dominated by *k x k*-bit additions in both energy consumption and latency as demonstrated in the previous sections. Unlike k x k-bit multiplication, which can be done basically in parallel, these *k x k*-bit additions in most cases are data dependent on each other. Therefore, the efficient implementation of k x k-bit addition will be of extreme importance to balance latency and energy consumption. Compared to addition, if k x k-bit multiplication is negligible in the overall latency and

### 6.4 Problem of Carry-in

The problem of the carry-in is pervasive in the modular exponentiation. Carry-ins are treated as addition in most cases, which is not an efficient way. If the carry-in could be taken into consideration efficiently, the improvement in latency and energy consumption will be significant. In most cases, carry-in is 1 bit, that 0 or 1. A fast instruction, which does conditional increment, will help a lot. The instruction takes two source operands, one is 1-bit and the other k-bit, and increases the k-bit source by one if the 1-bit source is 1.

### 6.5 Parallelism.

There is a lot of parallelism in the multiplication of very long integers. The latency can be significantly reduced if all the parallelism is utilized. Compiler and the ISA are of extreme importance in this respect.

### 6.6 Cache/Memory Optimization

As Figure 5.3 demonstrates, cache/memory consumes more than half of the total energy. Therefore, cache/memory access should be the first place for computer architects to look at for reducing cryptographic algorithm energy consumption.

### 6.7 Problem of k

In this project, we assumed that the very long integer is of *N*-bit and all the arithmetic/logic operations are done in the *k*-bit fashion. For instance, In a 16-bit processor, we have to use 16-bit arithmetic operations to implement *N*-bit arithmetic operations. On the other hand, as nowadays processors are having 32-bit and 64-bit registers and data-bus, we have several choices for k. For example, for 64-bit processors, we can let $k$=16, let each register holds 4 operands, and use several 16 x 16-bit adders to do several additions in one instruction. What is the best value for *k*? It depends on both the microarchitecture and the ISA. In this project, we give both cost and latency of different ways to implement the multiplication of very long integers. As section 1 and 2 detailed, the cost and the latency are determined by the following several factors (Please refer to equation 1-5.).

1. The length of the operands, *N*,
2. The length of the basic addition and multiplication operands, *k*

3. The cost /latency of the basic addition, *Add(k)*
4. The cost/latency of the basic multiplication, *Mul(k)*
5. How much parallelism is utilized

Factor 1 is determined by the algorithm and cryptosystem requirements. Factor 3 and 4 are determined by the micro-architecture. Factor 5 is determined by both micro-architecture and ISA if the algorithm is fixed. To decide the value for $k$, we have to a lot of issues. Algorithms have to be compared under given cryptosystem requirement, because different algorithm scales differently when $k$ changes given $N$. The micro-architecture has to be evaluated with respect to how *Add(k)* and *Mul(k)* scale when $k$ changes. ISA has to be evaluated to see how much parallelism can be utilized.

## Reference

C. K. Koc (1993), High-speed RSA implementation. RSA Data Security Conference, Redwood City, California, January 14-15, 1993

V. Tiwari et al(1994), Power Analysis of Embedded Software: A First Step Towards Software Power Minimization, , IEEE Transactions on VLSI Systems, December 1994.

JouleTrack, http://dry-martini.mit.edu/JouleTrack/

Source codes for the Karatsuba-Ofman algorithm
http://www.users.csbsju.edu/~cburch/proj/karat/