Montgomery Modular Multiplication Algorithm for Multi-Core Systems

Junfeng Fan, Kazuo Sakiyama and Ingrid Verbauwhede

Katholieke Universiteit Leuven,ESAT/SCD-COSIC,
Kasteelpark Arenberg 10
B-3001 Leuven-Heverlee, Belgium

June 12, 2007

1. What is Montgomery Modular Multiplication (MMM) Algorithm?

The Montgomery Multiplication Algorithm

Given n-bit modulo M, integer $x, y \in \mathbb{Z}_M$, $R = 2^n$

$Mont(x, y) = x \cdot y \cdot R^{-1} \bmod M$

1. What is Montgomery Modular Multiplication (MMM) Algorithm?

The Montgomery Multiplication Algorithm

Given n-bit modulo M, integer $x, y \in \mathbb{Z}_M$, $R = 2^n$
$Mont(x, y) = x \cdot y \cdot R^{-1} \bmod M$

2. Why Use Montgomery Modular Multiplication Algorithm?

Use Normal Multiplication

$Z = A \cdot B \bmod M$

1. $C = A \cdot B$
2. $Z = C - \lfloor \frac{C}{M} \rfloor \cdot M$

Use MMM

$Z = A \cdot B \bmod M$

1. $A' = Mont(A, R^2) = A \cdot R \bmod M$
2. $Z = Mont(A', B) = A \cdot B \bmod M$

1. What is Montgomery Modular Multiplication (MMM) Algorithm?

The Montgomery Multiplication Algorithm

Given n-bit modulo M, integer $x, y \in \mathbb{Z}_M$, $R = 2^n$

$Mont(x, y) = x \cdot y \cdot R^{-1} \bmod M$

2. Why Use Montgomery Modular Multiplication Algorithm?

Use Normal Multiplication

$Z = A \cdot B \bmod M$

1. $C = A \cdot B$
2. $Z = C - \lfloor \frac{C}{M} \rfloor \cdot M$

Use MMM

$Z = A \cdot B \bmod M$

1. $A' = Mont(A, R^2) = A \cdot R \bmod M$
2. $Z = Mont(A', B) = A \cdot B \bmod M$

3. Widely used in RSA, ECC, Diffie-Hellman...

Radix-2^w Montgomery Modular Multiplication Algorithm

Input: integers $M = (M_{s-1}, .., M_0)_r$, $X = (X_{s-1}, .., X_0)_r$, $Y = (Y_{s-1}, .., Y_0)_r$, where $0 \leq X, Y < M$, $r = 2^w$, $s = \lceil \frac{n}{w} \rceil$, $R = r^s$ with $gcd(M, r) = 1$ and $M' = -M^{-1} \text{mod } r$.

Output: $X \cdot Y \cdot R^{-1} \bmod M$

$Z = (Z_{s-1}, ..., Z_0)_r \leftarrow 0$
for $i = 0$ to $s - 1$ **do**
  $T \leftarrow (Z_0 + X_0 \cdot Y_i) \cdot M' \text{ mod } r$
  $Z \leftarrow (Z + X \cdot Y_i + M \cdot T)/r$
end for
if $Z > M$ **then**
  $Z \leftarrow Z - M$
end if
return Z

Hardware Implementations

1. Fast, Power efficient
   1. special data-path
   2. multiple processing elements (PE)
2. expensive, fixed functions
   1. Cost extra hardware
   2. Hard to update

Software Implementations

1. Cheap, flexible
   1. Sharing CPU with other applications
   2. Easy to modify
2. Slow
   1. General purpose data-path
   2. Normally single core

Hardware Implementations

1. Fast, Power efficient
    1. special data-path
    2. multiple processing elements (PE)
2. expensive, fixed functions
    1. Cost extra hardware
    2. Hard to update

Software Implementations

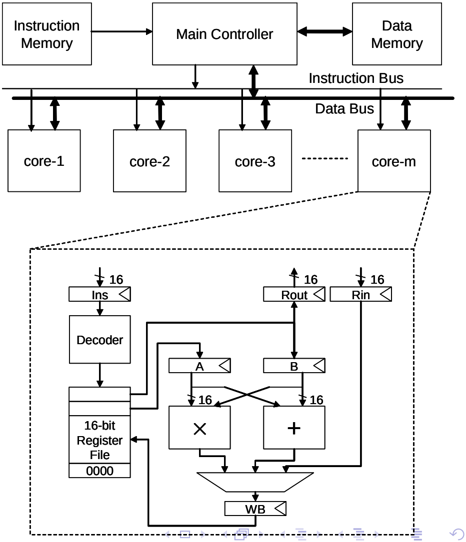
1. Cheap, flexible
    1. Sharing CPU with other applications
    2. Easy to modify
2. Slow
    1. General purpose data-path
    2. Normally single core

The question is:

How about using multi-core systems?

In the real world, a multi-core system can be

1. A processor with multiple cores: shared cache
2. A system with multiple processors: shared memory



Our prototype processor

1. Very Long Instruction Set (VLIW)
2. Shared single-port data memory

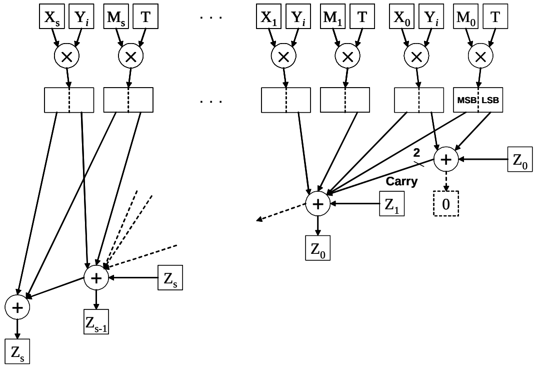
Opecode 4-bit	**Operand 1** 4-bit	**Operand 2** 4-bit	**Operand 3** 4-bit	**Description**
Nop				No operation
Load	**Ri**	**#Addr**		Load the data from location **Addr** of the data memory into register **Ri**
Store	**Ri**	**#Addr**		Store the data of register **Ri** to location **Addr** or the data memory
Mul	**Ri**	**Rj**	**Rk**	**R(i+1),Ri** = Rj· **Rk**
Add	**Ri**	**Rj**	**Rk**	**Ca,Ri** = **Rj** + **Rk**, **Ca** is the carry out and is stored in the status register
Adc	**Ri**	**Rj**	**Rk**	**Ca,Ri** = **Rj** + **Rk** + **Ca**
Sub	**Ri**	**Rj**	**Rk**	**Ri** = **Rj** - **Rk**

The question is:

How to map the Montgomery Modular Multiplication to this platform?

Data dependency in one loop

for $i = 0$ to $s - 1$ **do**
$T \leftarrow (Z_0 + X_0 \cdot Y_i) \cdot M' \mod r$
$Z \leftarrow (Z + X \cdot Y_i + M \cdot T)/r$
end for

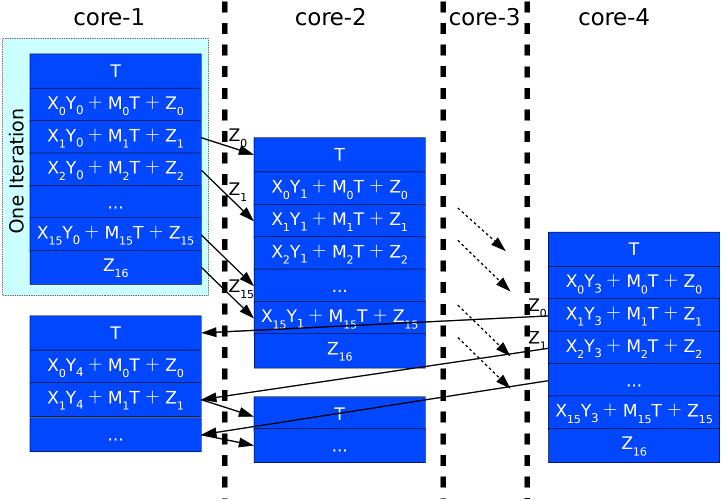


Basic considerations

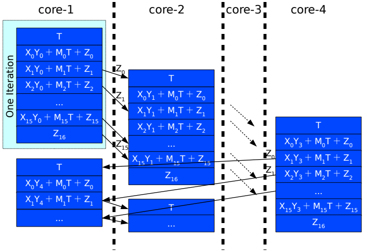
1. Number of **Mul** and **Add** are almost constant
2. Data transfers are expensive
3. Carry should be used in the local core

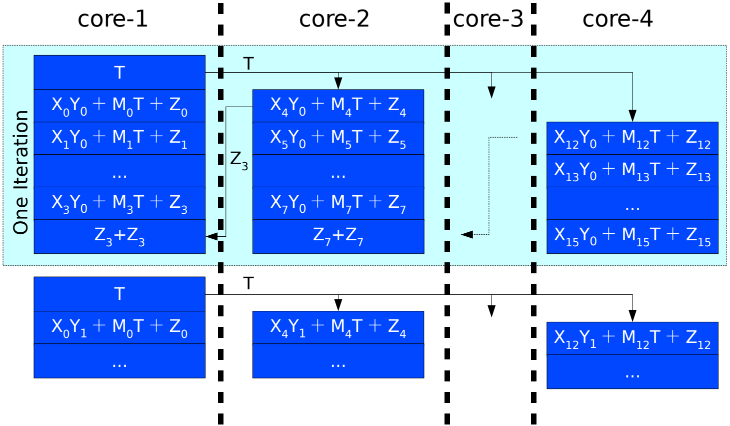
We propose

1. Instruction scheduling method-I: Each core performs one iteration
2. Instruction scheduling method-II: Multiple cores perform one iteration

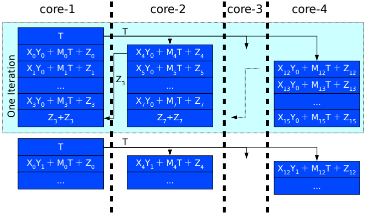

core-1
core-2
core-3
core-4
One Iteration
T
X0Y0 + M0T + Z0
X1Y0 + M1T + Z1
X2Y0 + M2T + Z2
...
X15Y0 + M15T + Z15
Z16
T
X0Y4 + M0T + Z0
X1Y4 + M1T + Z1
...
Z0
Z1
Z15
T
X0Y1 + M0T + Z0
X1Y1 + M1T + Z1
X2Y1 + M2T + Z2
...
X15Y1 + M15T + Z15
Z16
T
...
T
X0Y3 + M0T + Z0
X1Y3 + M1T + Z1
X2Y3 + M2T + Z2
...
X15Y3 + M15T + Z15
Z16
Z0
Z1

1. Carry is always used in the local core
2. Data transfers cause a heavy overhead
   1. Suppose Z has s words, one multiplication requires $s(s-1)$ data transfers
   2. For example, when performing 256-bit MMM, **240** data transfers are needed
3. $X_{s-1}, .., X_0$ and $M_{s-1}, .., M_0$ are loaded to each core in each iteration




core-1
core-2
core-3
core-4
One Iteration
T
$X_0Y_0 + M_0T + Z_0$
$X_1Y_0 + M_1T + Z_1$
...
$X_3Y_0 + M_3T + Z_3$
Z_3+Z_3
Z_3
$X_4Y_0 + M_4T + Z_4$
$X_5Y_0 + M_5T + Z_5$
...
$X_7Y_0 + M_7T + Z_7$
Z_7+Z_7
$X_{12}Y_0 + M_{12}T + Z_{12}$
$X_{13}Y_0 + M_{13}T + Z_{13}$
...
$X_{15}Y_0 + M_{15}T + Z_{15}$
T
$X_0Y_1 + M_0T + Z_0$
...
$X_4Y_1 + M_4T + Z_4$
...
$X_{12}Y_1 + M_{12}T + Z_{12}$
...

1. Carry is always used in the local core
2. Less data transfers are required
    1. Suppose Z has s words and a p-core system is used, one multiplication requires $3ps - 2s$ data transfers
    2. For example, when performing 256-bit MMM on a 4-core system, **96** data transfers are needed
3. Only $\lceil \frac{s}{p} \rceil$ words of $X_{s-1}, .., X_0$ and $M_{s-1}, .., M_0$ are loaded to each core in each iteration



Compared to the method-I, the method-II has two major advantages.

1. Operands and intermediate data are distributed in the register file of each core, thus less registers are required in each core.
2. Less data transfers reduce memory accesses, as a result, a single-port data memory can support more cores before becoming the bottleneck.

Table: Number of memory accesses required for one Montgomery multiplication for various Register File size (S_{rf}).

Processor type	S_{rf}	$N_{load-opr}$	$N_{load-tr}$	$N_{store-tr}$	N_{total}
Single-core	$S_{rf} > 3s$	$3s$	0	0	$3s$
	$2s < S_{rf} \leq 3s$	$s^2 + 2s$	0	0	$s^2 + 2s$
	$s < S_{rf} \leq 2s$	$2s^2 + s$	0	0	$2s^2 + s$
	$S_{rf} \leq s$	$2s^2 + s$	$s(s-1)$ *	s^2 *	$4s^2$
Multi-core Method-I	$S_{rf} > 2s$	$2ps + s$	$s(s-1)$	s^2	$2s^2 + 2ps$
	$s < S_{rf} \leq 2s$	$s^2 + ps + s$	$s(s-1)$	s^2	$3s^2 + ps$
	$S_{rf} \leq s$	$2s^2 + s$	$s(s-1)$	s^2	$4s^2$
Multi-core Method-II	$S_{rf} > \frac{3s}{p}$	$2s + ps$	$2(p-1)s$	ps	$5ps$
	$\frac{2s}{p} < S_{rf} \leq \frac{3s}{p}$	$s^2 + ps + s$	$2(p-1)s$	ps	$s^2 + 4ps - s$
	$\frac{s}{p} < S_{rf} \leq \frac{2s}{p}$	$2s^2 + ps$	$2(p-1)s$	ps	$2s^2 + 4ps - 2s$
	$S_{rf} \leq \frac{s}{p}$	$2s^2 + s$	$s^2 + (2p-3)s$ *	$s^2 + s$ *	$4s^2 + 2ps - s$

*Including store and load operations caused by calculating intermediate data.

Figure: Number of data memory accesses for various operand bit-length. ($w = 16$, $S_{rf} = 16$).

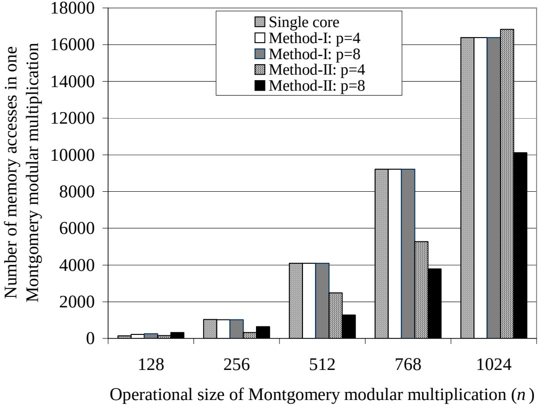
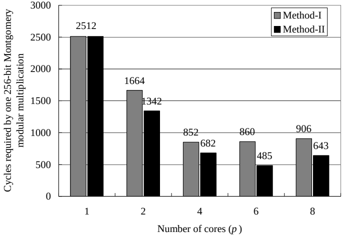


Figure: Performance of 256-bit Montgomery modular multiplication on a multi-core system. ($n = 256$, $w = 16$, $S_{rf} = 16$).



The performance of 256-bit MMM can be improved by a factor of **1.87** and **3.68** when using 2-core and 4-core systems, respectively.[Method-II]

Table: Performance comparison of modular multiplication.

Reference	Description	Platform	Area (Slices)	Freq. (MHz)	256-bit time(μs)	1024-bit time(μs)
This work (method-I)	4-cores/4 16x16 mults 4-cores/4 32x32 mults	Xilinx XC2VP30	2029 3173	125 93	6.8 2.6	131.0 44.0
This work (method-II)	4-cores/4 16x16 mults 4-cores/4 32x32 mults	Xilinx XC2VP30	2029 3173	125 93	5.5 2.2	134.7 33.0
Tenca *et al*.	Software	ARM	-	80	43	570
Itoh *et al*.	Software	DSP(TMS320C6201)	-	200	$2.68^{\ddagger}$	—
Brown *et al*.	Software	Pentium II	-	400	$1.57^{\S}$	—
Kelley *et al*.	4-PEs/8 16x16 mults	XC2V2000	360^{*}	135	0.68	8.3
Mentens	130 16x16 mults	XC2VP30	7244	64	0.31	1.07

* Author's estimation from the original paper.
‡ 239-bit Montgomery modular multiplication.
§ Using fixed modulo for fast reduction.

1. Hardware implementations
   1. Use specific data-path
   2. Use specific Register Files
2. Software implementations
   1. VLIW DSP
   2. Intel quad-core processors

Thanks!