I J C T A, 9(4), 2016, pp. 1921-1931
© International Science Press

Implementation of Montgomery Modular Multiplication Algorithm for Multi-Core Systems

Venkata Siva Prasad Ch.[1], **S. Ravi**[2], **M. Anand**[2]

ABSTRACT

In this paper Design the Montgomery Modular Multiplication algorithm for Efficient Multi-core systems using in Large Data in Communication and Industry fields. As HW/SW co-design technique is used to find the efficient system architecture and the instruction scheduling method VLIW prototype of general algorithms are developed with Linux kernel for Maintaining the Area Of the design and Time Response of the Multi-Core System. Scheduling in a multi-core systems while simulation process that can reduce the number of Iteration in the data transfers between different cores in these cores are Implemented in the depend on the Threads. The proposed processor is implemented under Multi-core Gizmo Processor technology. Compared to the implementations on a single-core system, the performance can be improved on Modified Multiplication Algorithm. Moreover, we also study the area reduction techniques for proposed multi-core processor from the perspectives of algorithm, architecture, and circuit.

Keywords: Montgomery Modular Multiplication, Multi-core, Parallel Computation, VLIW.

1. INTRODUCTION

The Explosive grown in telecommunication network& Large Scale Industry and Internet popularity for Transfer large data from one Location(Server) To other Location(client) Across The World for that time care for sending big data through network protocol in information Timing issues is also increasing & mainly constrains on the Time and Power consumption and Area. The algorithm has involved in the use of modular exponentiation of large numbers for encryption of data Developed in the Very large Instruction Word(VLIW) method is considered secure since factorization becomes intractable for very large numbers. To overcome designed that Montgomery Modular Reduction Algorithm was introduced to Developed the implementation space efficient, high speed architectures, When performing parallel computation, task scheduling is highly dependent on the hardware architecture. Here using a Very Long Instruction Word (VLIW) processor as a prototype, This processor can be configured to have 1, 2, 4, 8 or even more cores. Each core can work separately to be general, only the very basic instructions are supported. The rest of the paper is organized as follow. Section 2 briefly reviews work on the Montgomery algorithm. In section 3, we describe the multi-core architecture with VLIW of our platform. in section 4, Implementations of parallel MMM algorithm scheduling methods are proposed. Finally, we show results in section 5 and conclude the paper including future work in section 6.

2. MONTGOMERY MULTIPLICATION ALGORITHM

The proposed Montgomery algorithm allows modular arithmetic to be accomplished efficiently when the modulus is large (2048 bits or more).

[1] Research Scholar, ECE Department, Dr. M.G.R. Educational and Research Institute University, Chennai, India, *Email: siva6677@gmail.com*

[2] Professor & Head, ECE Department, Dr. M.G.R. Educational and Research Institute, Chennai, India, *Email: ravi_mls@yahoo.com*

- Faster way to do modular exponentiation
- Operate on Montgomery residues
- Division becomes a simple shift
- The Montgomery algorithm consists of two approaches: multiplication and reduction.

Montgomery multiplication is a method for computing *x. y mod m* for positive integers *x*, *y*, and *m*. It moderates execution time on a computer when there are large numbers of multiplications to be done with the same modulus *n*, and with a small number of multipliers. In precise, it is useful to compute *x e mod m* for a large value of *m*.

In common, Montgomery multiplication algorithm computes the Montgomery product.

$$Mon\ Mul(x', y') = x'.y'.r - 1(\text{mod}\ \ m)$$

Where the multipliers *x* and *y* are less than the modulus *m*. it is needed to declare another integer *r* which must be greater than *m*, as the gcd (*r*, *m*) = 1. The method, really, changes the reduction modulo *m* to *r*. usually *r* is chosen to be an integral power of 2. Therefore, the reduction modulo *r* is simply a masking operation.

As held from the *MonMul ( )* function above, x and y is numbers that represent the m-residues, which can be calculated as follows

$$x' = x\,.r\ mod\ n$$
$$y' = y\,.r\ mod\ n$$

The two integer's *r-1* and *n'* are calculated, by using the Extended Euclidean algorithm, such that:

$$r\ r - 1 - m\ m' = 1$$

The final result of the Montgomery multiplication will be in the *n*-residue.

$$z' = x'\ .y'\ .r - 1\ mod\ m$$

Finally, a conversion step has to be performed to transform the result back from the m-residue representation to normal residue representation.

$$z = MonMul\ (z', 1)$$

The computation of *MonMul (x, y)* is given in Algorithm 1.

Algorithm 1: Montgomery Modular Multiplication.

Step 1: Input an odd modulus *n* and a radix $r = 2\lceil \log 2\ n \rceil$, such that GCD (*m*, *r*) = 1, an auxiliary value $m' = -m - 1$ mod *r*, 2 m-residue integers *x'* and *y'*.

Step 2: function: MonMul (*x'*, *y'*).

Step 3: Calculate $Z: = x'\ .y'$.

Step 4: Calculate $z: = (t + [t.\ m'\ mod\ r].\ m)/r$.

Step 5: If $z \geq m$ then return (*u-m*) else return *z*.

Step 6: Output $x'\ .y'\ .r\text{–}1$ (mod *m*).

The Montgomery modular multiplication algorithm was designed to avoid division in modular multiplications. Given two n-bit inputs, *X* and *Y*, this algorithm gives $Z = X \cdot Y \cdot R\text{–}1$ mod *M*, where *R* equals to 2*n* and *M* is the n-bit modulo. Algorithm 1 shows the Montgomery modular multiplication of conditional by choosing a suitable the operands *X*, *Y* and *M* are divided into *w*-bit words. In the beginning of each iteration, $X0 \cdot Y\text{i}$ is calculated to generate *T*. After the generation of *T*, the multiplication of $X \cdot Yi$ and

reduction of C are performed together by doing $Z = Z + X \cdot Yi + M \cdot T$. After that, $Z0$ always becomes 0. After s iterations and one conditional $Z = X \cdot Y \cdot R–1 \bmod M$ is obtained.

Illustration of Montgomery Modular Multiplication

With two cores

- $X = 7 = 0111$
- $Y = 5 = 0101$
- $M = 11 = 1011$
- Z initially 0
  - $Z = (0 + 5 + 11) / 2 = 8$
  - $Z = (8 + 5 + 11) / 2 = 12$
  - $Z = (12 + 5 + 11) / 2 = 14$
  - $Z = (14 + 0) / 2 = 7$ (final result).

3. SCHEDULING ARCHITECTURE FOR MULTI-CORE PROCESSOR

The hardware architecture and explore the best software algorithm for the fixed hardware configuration. the main focus of this paper, needs an environment to get a quick and correct evaluation of cost and performance for which allows us to estimate immediate system performance in a cycle-accurate manner before synthesizing the entire design.

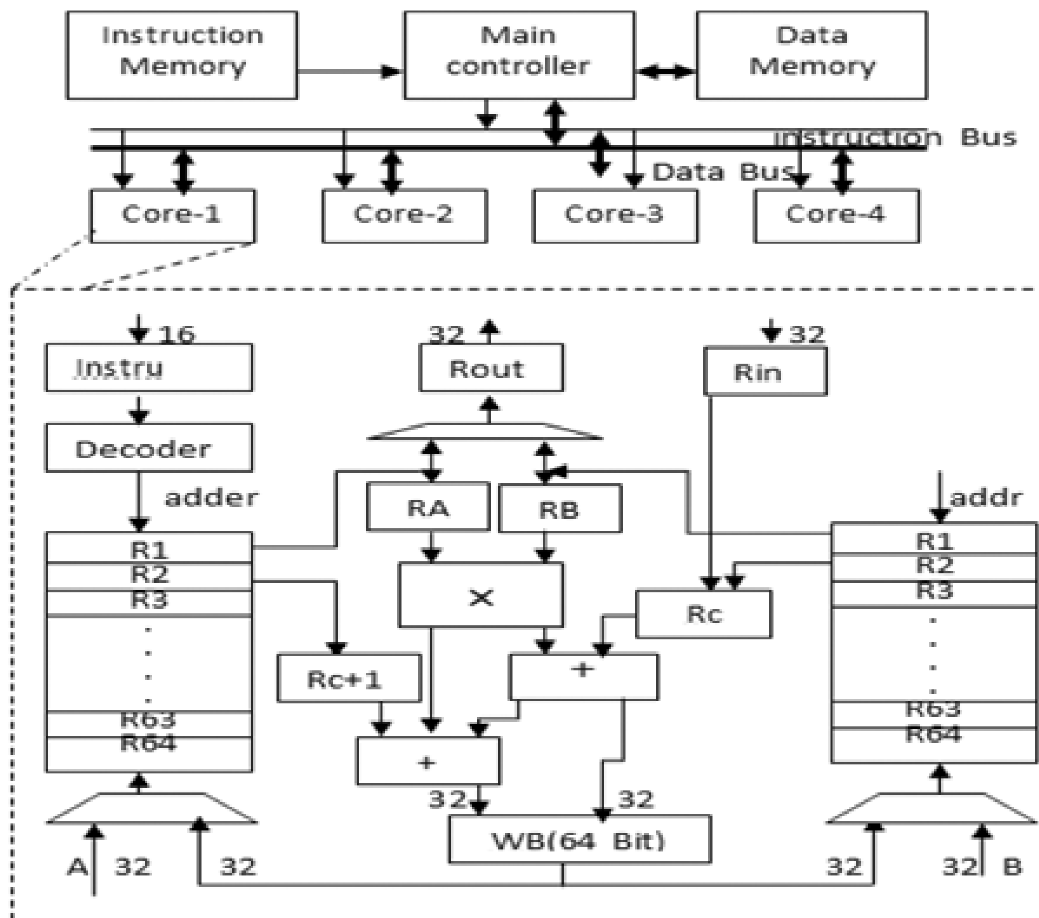


Figure 1: 64 Bit Scheduling Element

This platform consists of a main controller, a data memory, an instruction memory and several cores. Only the main controller can access the instruction memory and the data memory. The main controller fetches instructions from the instruction memory and dispatches them to all cores in parallel via the instruction bus. W denotes the operation size of w-bit core, A 64-bit (w = 64) core is also shown in Figure 1. It is a highly simplified Load/Store CPU. It has a instruction decoder, a register file with 64 general 64-bit registers and one status register. The Arithmetic Logic Unit(ALU) includes one 64-bit multiplier and one 64-bit adder. It also has an output register to store the data that will be written to the data memory, and an input register to buffer the data from the data memory. When data needs to be moved from one core to another, it is first stored to the data memory, then loaded by the destination core. The purpose of this prototype processor is to explore different algorithms on multi-core systems.

3.1. Very long instruction word (VLIW)

The New Concept of VLIW refers to processor architectures designed to take advantage of instruction level parallelism (ILP). Whereas conventional processors mostly allow programs only to specify instructions that will be executed in sequence, a VLIW processor allows programs to explicitly specify instructions that will be executed at the same time (that is, in *parallel*). This type of processor architecture is intended to allow higher performance without the inherent complexity of some other approaches. VLIW code is ordered for the processor at compile time, this is all done before the code is ever actually executed. As a VLIW compiler sorts through the code, it examines it to determine which instructions will be able to be executed simultaneously. This is often done via a process called trace scheduling will be in figure-2 has explained operation of VLIW of operation with 4 cores running in the process with execution units.

The ability to integrate coding into larger words and fewer lines of instructions, VLIW has become extremely useful in its applications such as like Dolby Audio, video players and video games. The VLIW being what it is, offers special embedded software features for many of these multimedia types. However, followed by its simple in architecture is its extremely complicated software compilers having to advance check the instructions to allow such parallelism.

4. IMPLEMENTATION OF PARALLEL MONTGOMERY MODULAR MULTIPLICATION

The proposed an iteration-based scheduling method of each Processing Element (PE) performs one iteration of the loop in Algorithm 1. It’s is attractive because carries are only used in the local PE. Note that this method

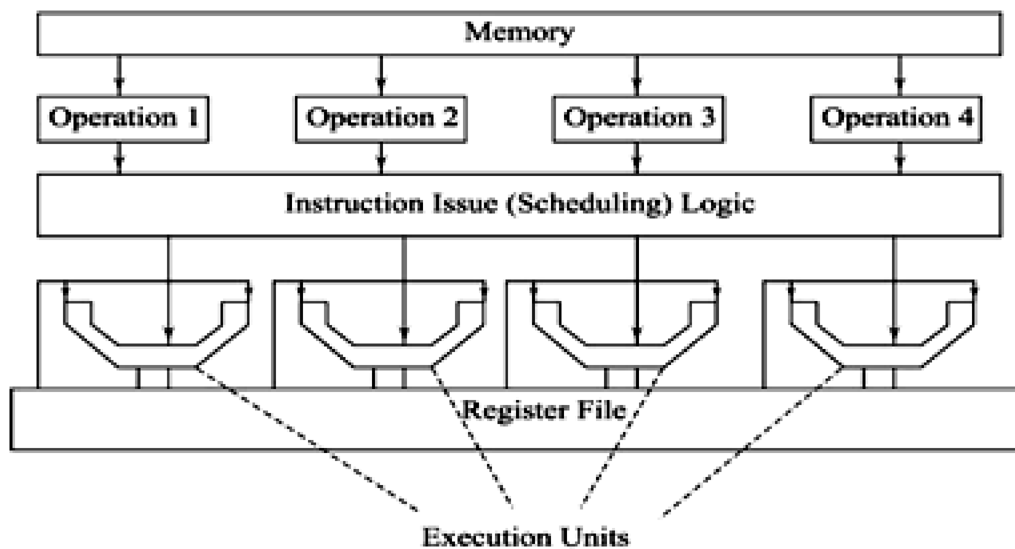


Figure 2: VLIW Processor With 4 Core Operations

was originally designed for a hardware implementation. the inherent parallelism of Algorithm 2 to the original Montgomery multiplication of The resulting parallel Montgomery multiplication given with Algorithm 2 below. On a multi-core processor, one can also parallelize the additions given on lines 4 to 6 of Algorithm 1. When the number of cores available is a power of 2, this partial product accumulation can be achieved in a binary tree fashion, as shown below, with at most $\lceil \log 2\ s \rceil$ steps where s is the number of cores available.

For $i = 1$ to log2 s

For $j = 0$ to s 2 $i - 1$ $tj \leftarrow tj + tj + s\ 2i$

end for end for In the above setting, all the cores are exploited as evenly as possible with the maximal utilization which would result in the minimal latency. However, this optimal chain of additions would not always be possible. In the rest of this section, we provide some addition chains for efficient implementations of Algorithm 2 on processors with 2, 4 and 6 cores as examples

Two dimensions of parallelism:

- Width of processing element *w*
- Number of pipelined PEs *p*
  - Multiply takes *k = n/p kernel cycles*

Montgomery proposed a new algorithm where division is avoided. An integer Z is represented as $Z \cdot R$ mod M, where M is the modulo and $R = 2r$ is a radix that is co primes to M. As shown in Figure-3shows the

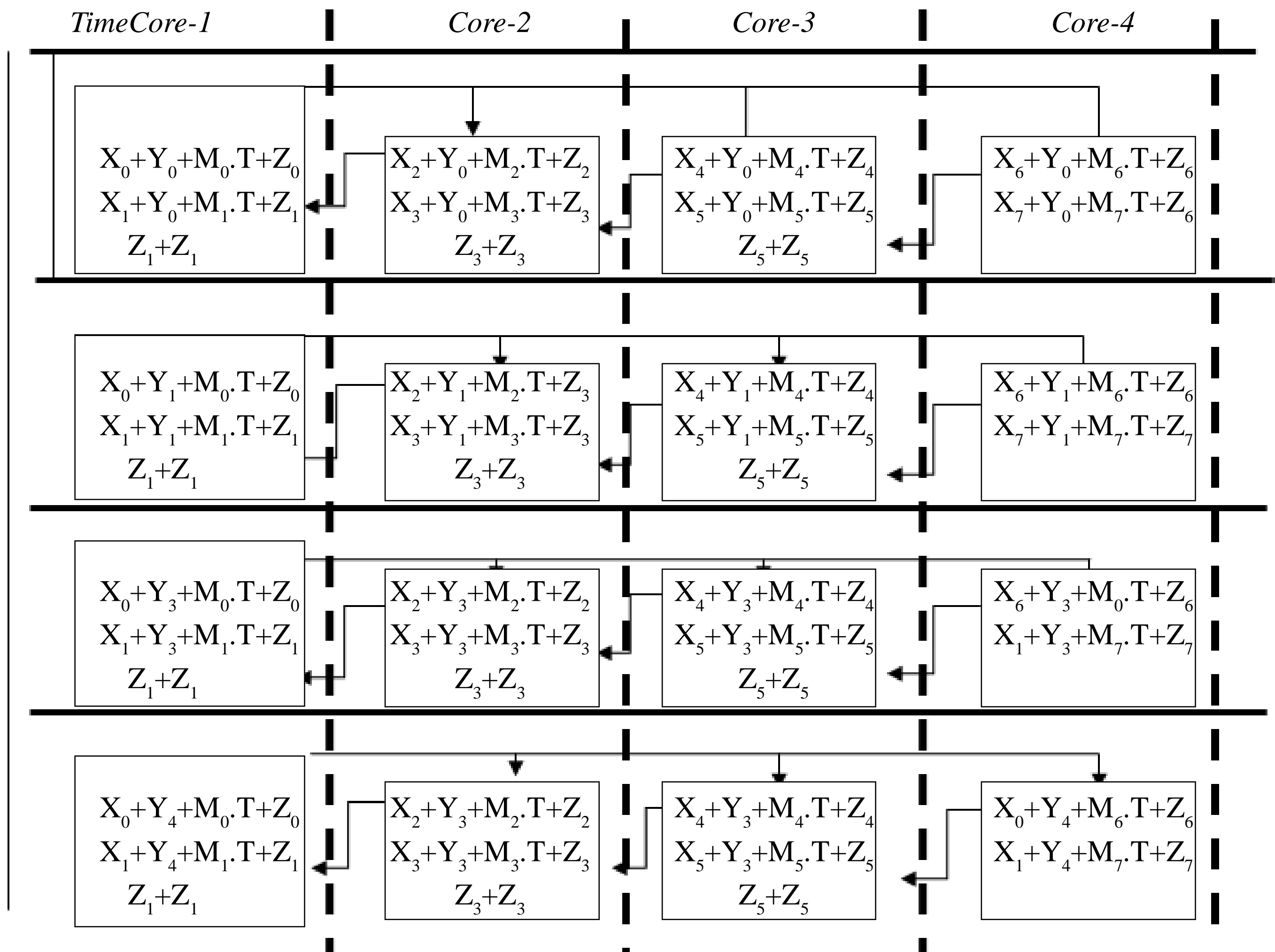


Figure 3: Multi-core Process Scheduling

scheduling method, denoted as Algorithm-2, for 256-bit Montgomery multiplication for a 4-core system. As $n = 256$ and $w = 64$, each Core has 64 iterations are needed. Core-1 performs the first iteration and generates Z0 to Z63 one by one. Each word is transferred to core-2 as soon as it is generated. Core-2 then performs the second iteration and then transfers $Z64$ to $Z127$ to core-3. After 4 iterations $Z = (Z251, ..., Z0)$ is transferred back to core-1 from core-4 and the 5th iteration begins. As in total 64 iterations are required, each core needs to perform 4 iterations. After 8 iterations and a conditional subtraction, $Z = X \cdot Y \cdot R–1$ mod M is generated and stored separately in four cores. Z can be written to the data memory or can be used by another modular multiplication. As a result, the number of load and store operation are reduced. When using one core to perform 256-bit Montgomery modular multiplication, 16 clock cycles are required. The implementation result is summarized in Table-2 and Table-4. According to the table, the bottleneck of this implementation is addition operations. Many hardware implementations were proposed to improve the performance. The increasing use of multi-core systems have opened another window for improving the performance of software implementations. Even for embedded systems several multi-core processors are now available.

Algorithm 2. Parallel Montgomery Residue multiplication

Input: $X;Y\Box\in Zn$ where n is an odd integer and M

Then Z = $–M$

–1 mod 2m where $m = \lceil \log2\ n \rceil$.

Output: $X\Box \bullet Y \bullet 02–m$ mod n.

1: $Z \leftarrow$ Parallel Multiply (X; Y) {Algorithm 1}

2: $Zi \leftarrow$ Parallel Multiply (Z; M) mod 2m {Algorithm 1}

3: $Zi \leftarrow$ ParallelMultiply(Zi; M) {Algorithm 1}

4: $Zi \leftarrow (Zi + Z) = 2m$

5: if $Zi \geq M$ then

6: Return ($Zi – M$)

7: else

8: Return (Zi)

9: end if

Illustration of the Montgomery Residues

- Let the modulus M be an odd n-bit integer $2^{n-1} < M < 2^n$
- Define $r = 2^n$ $\overline{a} = ar$ mod M
- Define the M-residue of an integer $a<M$ as
- There is a one-to-one correspondence between integers and M-residues for

  $0 < a < M\text{-}1$

  Example

- $M = 11$, $r = 16$

$$\overline{0} = 0*16 \mod 11 = 0$$
$$\overline{1} = 1*16 \mod 11 = 5$$
$$\overline{2} = 2*16 \mod 11 = 10$$
$$\overline{3} = 3*16 \mod 11 = 4$$
$$\overline{4} = 4*16 \mod 11 = 9$$
$$\overline{5} = 5*16 \mod 11 = 3$$
$$\overline{6} = 6*16 \mod 11 = 8$$
$$\overline{7} = 7*16 \mod 11 = 2$$
$$\overline{8} = 8*16 \mod 11 = 7$$
$$\overline{9} = 9*16 \mod 11 = 1$$
$$\overline{10} = 10*16 \mod 11 = 6$$

5. RESULT & DISCUSSION

The multi-core platform implemented with Open Mp Multi-Kernel. The operands, *X*, *Y* and *M*, are stored in the data memory. comparison with implementation of both Algorithm-1 and Algorithm-2 on the platform with various hardware configurations. The results are presented in below tables. We implemented our algorithm for the operand sizes of 1024, 2048, 4096, 8192, 16384 and 32768 bits on general-purpose multi-core processors using OpenMP and obtained the timings. As shown in Table-1 the 12 operand with instructions set 6 of results in Table-2 with 8.33% reduction and with table-3 algorithm-2 has given the 16.33% instruction lines saved using Montgomery Algorithm.

With the MMM algorithm -1 the lines saved per instruction set in scheduling of with Multi-core

The modulation has on with algorithm 2 of multi-core set of instructions.

The Multi-core processor of the modular has on the instruction of the VLIW of a Montgomery Modular Instructions are

Table 1
2048 instructions with 6 lines VLIW

Max Instructions/Line 6 Ops	Register 1	Register 2	Register 3
ADD	R10	R4	R6
MULT	R6	R4	R10
MULT	R1	R6	R5
ADD	R5	R1	R6
ADD	R1	R1	R1
MULT	R1	R1	R1
ADD	R1	R1	R1
SUB	R1	R3	R5
ADD	R1	R1	R1
SUB	R5	R3	R1
ADD	R1	R4	R9
ADD	R4	R9	R1

Table 2
Algorithm-1 with Multi-core

VLIW INTEGRATED OUTPUT

ADD R10 R4 R6	
MULT R6 R4 R10	MULT R1 R6 R5
ADD R5 R1 R6	
ADD R1 R1 R1	
SUB R1 R3 R5	
ADD R1 R1 R1	
SUB R5 R3 R1	
ADD R1 R4 R9	
ADD R4 R9 R1	

VLIW Line %

Total % of Lines Saved = 16.666666666666664%

Table 3
Algorithm -2 Saved Lines

ADD R10 R4 R6	
MULT R6 R4 R10	MULT R1 R6 R5
ADD R5 R1 R6	
ADD R1 R6 R8	
MULT R8 R1 R6	
AD R1 R1 R1	
DIV R1 R3 R5	
ADD R1 R1 R1	
SUB R5 R1 R3	ADD R1 R4 R9
ADD R4 R9 R1	

VLIW Line %

Total % of Lines Saved = 16.666666666666664%

Table 4
-2048 instructions with 16 lines VLIW

16 Ops	Register 1	Register 2	Register 3
MULT	R1	R4	R6
ADD	R3	R8	R3
MULT	R2	R3	R4
MULT	R1	R3	R7
ADD	R1	R1	R1
OR	R6	R1	R4
SUB	R7	R5	R8
ADD	R7	R1	R1
MULT	R7	R4	R6
SUB	R4	R6	R9
ADD	R1	R2	R4

(*contd...*)

(*Table 4 contd...*)

Max Instructions/Line

16

Ops	Register 1	Register 2	Register 3
DIV	R3	R6	R9
MULT	R2	R3	R1
ADD	R1	R1	R4
SUB	R3	R1	R2
ADD	R6	R3	R3
DIV	R6	R5	R5
ADD	R1	R4	R1
ADD	R2	R2	R4
MULT	R10	R3	R6
XOR	R6	R8	R2
OR	R2	R6	R9
MULT	R4	R2	R9
ADD	R1	R3	R6
SUB	R9	R10	R4
NOR	R4	R4	R2
MULT	R3	R4	R1
XNOR	R5	R4	R7
XNOR	R9	R1	R9
NOR	R5	R6	R10
ADD	R2	R5	R7
MULT	R4	R2	R8

Table 5
Algorithm-1 saves lines

VLIW INTEGRATED OUTPUT

ADD R3 R4 R5		
MULT R5 R3 R4	ADD R1 R1 R1	MULT R10 R5 R7
ADD R1 R1 R1	DIV R5 R7 R10	MULT R2 R7 R4
ADD R1 R1 R1		
ADD R1 R1 R1	XNOR R7 R4 R2	
ADD R1 R1 R1		
ADD R1 R1 R1	OR R2 R7 R4	
MULT R4 R2 R7	ADD R1 R1 R1	
NOR R1 R7 R9		
ADD R1 R1 R1		
ADD R1 R1 R1		
MULT R1 R7 R7		
ADD R1 R1 R1		
SUB R9 R1 R7		
ADD R1 R1 R1		

VLIW Line %

Total % of Lines Saved = 29.666666666666668%

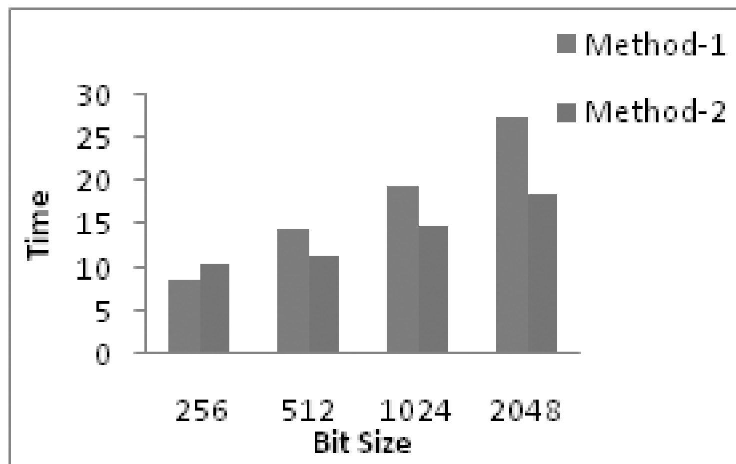
Table 6
Algorithm-2 saves lines

VLIW INTEGRATED OUTPUT

MULT R1 R4 R6	ADD R3 R8 R3		
Mult R2 R3 R	Add R2 R1 R1		
Sub R2 R2 R2	Or R6 R1 R4	Sub R7 R5 R8	
Add R7 R1 R1	Mult R7 R4 R6		
Sub R4 R6 R9			
Add R1 R2 R4	Div R3 R1 R9		
Mult R1 R3 R2	Add R1 R1 R4	Or R3 R5 R8	Mult R7 R5 R6
Div R6 R5 R5	Add R1 R1 R1	Add R2 R2 R4	
Mult R10 R3 R6	Sub R2 R2 R2		
Or R2 R6 R9			
Mult R4 R2 R9	Add R1 R3 R6		
Add R1 R1 R1	Nor R4 R4 R2		
Mult R3 R4 R1	Xnor R5 R4 R7		
Or R9 R6 R7	Nor R5 R6 R10		
Add R2 R5 R7	Mult R4 R2 R8		

VLIW Line %

Total % of Lines Saved = 53.125%



6. CONCLUSION

This paper introduced an efficient software implementation of the Montgomery multiplication algorithm on a multi-core system. Multi-core processor that is devoted to computing scheduling algorithms scheduling method could reduce the number of data transfers between different cores. the performance of 256-bit Montgomery multiplication was improved by using 4-core systems, respectively with the algorithm-2. Overcome of the low-latency and high-throughput VLIW of Montgomery Modular Multiplication computation. High performance of Very long-Instruction word of MMs is achieved by using efficient modular multipliers and employing fast inter-core communication. Experimental results show that our design outperforms previous works based on varied platforms in performance, for instance, it can complete 1024-bit VLIW in saved data lines 0.087 ms at 960 MHz. the future work includes a hardware implementation based on our proposed parallel-processing algorithm with a special data-path that can perform multiple operations Increasing the speed of the multi-core system.

REFERENCE

[1] Nicolau; J. A. Fisher "Measuring the Parallelism Available for Very Long Instruction Word Architectures" IEEE Transactions on Computers , Volume: C-33, Issue: 11, 1984, pp. 968–976.

[2] T. Fryza; R. Marsalek; F. Adamec, "Effective programming of very long instruction word digital signal processors", Radioelektronika, 2012 22nd International Conference, 2012, pp. 1–4.

[3] X. Yang; Y. Zhang; D. Liu; D. Guo; H. He, "Single instruction multiple data code auto generation for a very long instruction words digital signal processor in sensor-based systems" IET Wireless Sensor Systems, Volume: 3, Issue: 2, 2013, pp. 119–125.

[4] R. Jordans; R. Corvino; L. Józwiak, "Algorithm Parallelism Estimation for Constraining Instruction-Set Synthesis for VLIW Processors" Digital System Design (DSD), 15th Euromicro Conference, 2012, pp. 152–155.

[5] S. Rajaraman; P. Sirpotdar; A. Wavare; A. B. Patki, "Multithreading implementation in a single core TMS320C6713 DSP" Advances in Communication and Computing Technologies (ICACACT), International Conference, 2014, pp. 1–5.

[6] D. Sabena; M. S. Reorda; L. Sterpone "On the Automatic Generation of Optimized Software-Based Self-Test Programs for VLIW Processors" IEEE Transactions on Very Large Scale Integration (VLSI) Systems,Volume: 22, Issue: 4, 2014, pp. 813–823.

[7] M. Milward; D. Stevens; V. Chouliaras, "Embedded UML design flow to the configurable LE1 Multi-Core VLIW processor" Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop, 2012, pp. 1–8.

[8] T. J. Lin; C. N. Liu; S. Y. Tseng; Y. H. Chu; A. Y. Wu "Overview of ITRI PAC project - from VLIW DSP processor to multicore computing platform" VLSI Design, Automation and Test, 2008. VLSI-DAT 2008. IEEE International Symposium, 2008, pp. 188–191.

[9] Xu Yang [1] ; Yanjun Zhang [1] ; Dake Liu [1] ; Deyuan Guo [2] ; Hu He [2] "Single instruction multiple data code auto generation for a very long instruction words digital signal processor in sensor-based systems" Volume 3, Issue 2, June 2013, pp. 119–125.

[10] Z. Chen; P. Schaumont, "A Parallel Implementation of Montgomery Multiplication on MulticoreSystems: Algorithm, Analysis, and Prototype", IEEE Transactions on Computers, Volume: 60, Issue: 12, 2011, pp. 1692–1703.

[11] R. Dou; J. Han; Y. Bo; Z. Yu; X. Zeng, "An Efficient Implementation of Montgomery Multiplication on Multicore Platform With Optimized Algorithm, Task Partitioning, and Network Architecture" IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume: 22, Issue: 11 2014, pp. 2245–2255.