A Flexible-Length-Arithmetic Processor Using Embedded DSP Slices and Block RAMs in FPGAs

Md. Nazrul Islam Mondal, Kohan Sai, Koji Nakano, and Yasuaki Ito
Department of Information Engineering, Hiroshima University
1-4-1 Kagamiyama, Higashi-Hiroshima, 739-8527, Japan

***Abstract*—Some applications such as RSA encryption/decryption needs integer arithmetic operations with many bits. However, such operations cannot be performed directly by conventional CPUs, because their instruction supports integers with fixed bits, say, 64 bits. Since the CPUs need to repeat arithmetic operations to numbers with fixed bits, they have considerably overhead to execute applications involving integer arithmetic with many bits. On the other hand, we can implement hardware algorithms for such applications in the FPGAs for further acceleration. However, the implementation of hardware algorithm is usually very complicated and debugging of hardware is too hard. The main contribution of this paper is to present an intermediate approach of software and hardware using FPGAs. More specifically, we present a processor based on FDFM (Few DSP slices and Few Memory blocks) approach that supports arithmetic operations with flexibly many bits, and implement it in the FPGA. Arithmetic instructions of our processor architecture include addition, subtraction, and multiplication for numbers with variable size longer than 64 bits. To show the potentiality of our processor, we have implemented 2048-bit RSA encryption/decryption by software written by machine instructions. The resulting processor uses only one DSP48E1 slices and four Block RAMs (BRAMs), and RSA encryption software on it runs in 635.65ms. It has been shown that the direct hardware implementation of RSA encryption runs in 277.26ms. Although our intermediate approach is slower, it has several advantages. Since the algorithm is written by software, the development and the debugging are easy. Also, it is more flexible and scalable.**

***Index Terms*—Multiple-length-arithmetic, Montgomery Modular Multiplication, FPGA, DSP Slices, Block RAMs.**

I. Introduction

An FPGA is a programmable logic device designed to be configured by the customer or designer by HDL (Hardware Description Language) after manufacturing. An FPGA chip maintains relative lower price and programmable features [1], [2], [3], hence, it is widely used recently. We refer the readers to see some circuit implementations in FPGAs [4], [5], [6], [7], [8], [9], [10], [11], [12] to accelerate computation. In particular, since, FPGAs can implement hundreds of circuits that work in parallel, they are used to accelerate useful computations.

Applications require arithmetic operations on integer numbers which exceed the range of processing by a CPU directly is called Multiple Double Length Numbers or Multiple Precision Numbers and hence, computation of these numbers is called Multiple-Length-Arithmetic. More specifically, application involving integer arithmetic operations for multiple-length numbers with size longer than 64 bits cannot be performed directly by conventional 64-bit CPUs, because their instruction supports integers with fixed 64 bits. To execute such application, CPUs need to repeat arithmetic operations for those numbers with fixed 64 bits which increase the execution overhead. Alternatively, hardware algorithms for such applications can be implemented in FPGAs to speed up computations. However, the implementation of hardware algorithm is usually very complicated and debugging of hardware is too hard.

Since, low level of instructions, represented by 0's and 1's is an almost impossible to understand even by an expert, the debugging of an algorithm at this level is very hard. Moreover, to implement hardware algorithm, written by hardware language such as Verilog HDL, users should have sufficient knowledge of hardware such as registers which makes it complicated to the non-expert or to the beginners. The instructions in assembly language are written by alphanumeric symbols instead of 0's and 1's in low level that is an almost similar to the high level language, written by English which makes the instructions as well as algorithms easy to read, modify and debugging by the non-expert or by the beginners.

The main contribution of this paper is to present an intermediate approach of software and hardware using FPGAs (Field Programmable Gate Arrays) to support arithmetic operations for numbers with flexibly many bits such that the development and debugging of it become easier. More specifically, we propose a flexible-length-arithmetic processor based on FDFM approach that supports applications involving arithmetic operations for numbers with variable size longer than 64 bits and these applications, written by software become easier for debugging and further development.

For the reader's benefit, this paper precisely describes our main contributions as follows:

- We propose a flexible-length arithmetic processor based on FDFM approach for computing of integer numbers with flexibly many bits, even longer than 2048-bit by a single machine instruction.
- We present an intermediate approach of software and hardware to write the algorithm which makes the debugging and further development easy.
- Our designed processor provides flexibility so that it can be used for computing of integer numbers with flexibly many bit such as 64-bit, 128-bit, even longer than 2048-bit without further modification.

Since, our designed processor based on FDFM approach, the key idea of the FDFM approach is to use few DSP slices

and few block RAMs to perform routine computations which can be treated alternatively as a resource efficient approach. Let us explain briefly the FDFM approach using a simple example. Figure 1 (1) illustrates a hardware algorithm to compute the output of FIR (Finite Impulse Response) $y_i = a_0 \cdot x_i + a_1 \cdot x_{i-1} + a_2 \cdot x_{i-2} + a_3 \cdot x_{i-3}$. A conventional approach implementing the FIR is to use four DSP slices as illustrated in Figure 1 (2)[13]. In this conventional approach, the number of DPS blocks must be the same as that of multipliers in the hardware algorithm. However, FDFM approach uses one or few DSP slices and one or few block RAMs to implement the FIR. The Figure 1 (3) shows the FDFM approach using one DSP slice and one block RAM to implement the same mentioned above. Note that, the coefficients $a_0, a_1, \ldots$ are stored in the block RAM.

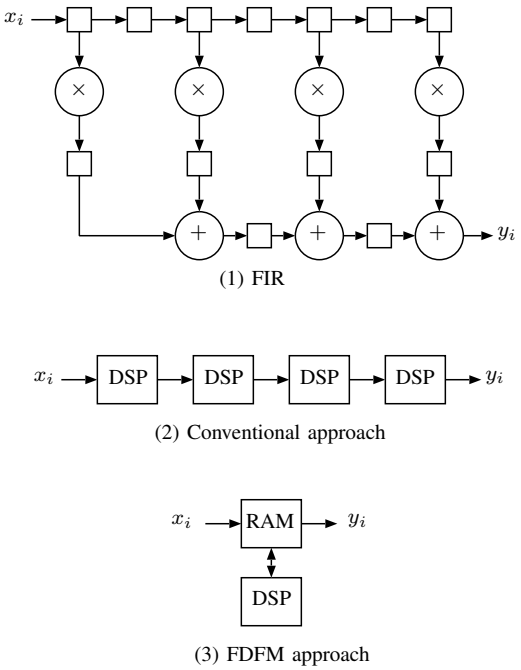

Fig. 1. FDFM approach over conventional one for FIR

For readers, we also refer to the papers [14], [15], [16], [17] in which they can find details about FDFM approach and conventional approach. Let us describe the two important advantages of the FDFM approach are as follows:

1. Even if the large main circuit occupies the most of hardware resources in the FPGA, we can implement a necessary hardware algorithm in the FPGA using remaining few hardware resources as illustrated in Figure 2 (1).

2. Also, if enough hardware resources are available, we can implement multiple FDFM processor cores that work in parallel (Figure 2 (2)). The resulting hardware implementation has maximum throughput by parallel computation.

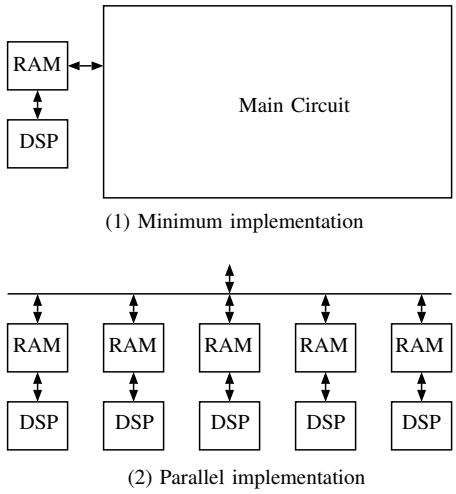

Fig. 2. Two advantages of FDFM approach

Because of the above mentioned advantages of the FDFM approach, we design flexible-length-arithmetic processor based on FDFM approach. More precisely, our proposed processor supports minimum implementation in Figure 2 (1). However, if enough hardware resources are available in FPGA, it can be used to work in parallel (Figure 2 (2)) to maximize throughput. Note that, our designed flexible-length-arithmetic processor using FPGA based on FDFM approach can perform arithmetic operations for numbers with variable size longer than 64 bits by a single command or instruction whereas today's PCs of 64 bits must require complicated arithmetic algorithm with many commands to compute these.

The most common FPGA architecture consists of an array of logic blocks, I/O pads, Block RAMs and routing channels. Furthermore, embedded DSP blocks which is integrated into an FPGA that makes a higher performance and a broader application. Figure 3 illustrates the Virtex-6 FPGA developed by Xilinx. The CLB (Configurable Logic Blocks) in Virtex-6 consists of 2 sub-logic blocks called slice. Using LUTs (Look Up Tables) and Flip-Flops in the slices, various combinatorial circuits and sequential circuits can be implemented. The Virtex-6 FPGAs also has DSP48E1 slices equipped with a multiplier, adders, and logic operators, etc. More specifically, as illustrated in Figure 4, the DSP48E1 slice has a two input multiplier followed by multiplexers and a three-input adder/subtractor/accumulator. The DSP48E1 multiplier can perform multiplication of the 18-bit and the 25-bit 2's complement numbers and produces one 48-bit 2's complement production. Programmable pipelining of input operands, intermediate products, and accumulator outputs enhances throughput and improves the frequency. The DSP48E1 also has pipeline registers between operators to reduce the delay. The block RAM in the Virtex-6 FPGA is an embedded memory supporting synchronized read and write operations.

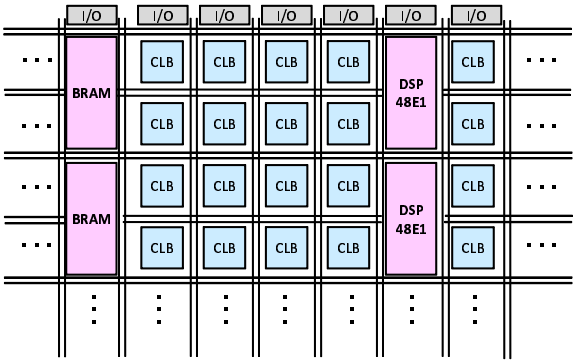

Fig. 3. Internal Configuration of Virtex-6 FPGA

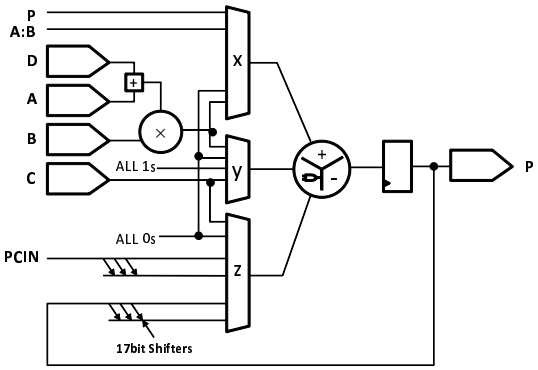

Fig. 4. Architecture of DSP48E1

In Virtex-6 FPGA, it can be configured as a 36k-bit dual-port block RAMs, FIFOs, or two 18k-bit dual-port RAMs. In our architecture, it is used as a 2k x 18-bit dual-port RAM.

We are mainly thinking the following scenario for designing a flexible-length-arithmetic processor:

1) Our aim is mainly to emphasis the beginners or non-expert users. Since our processor is designed to be implemented for computing numbers even longer than 2048-bit by a single machine instruction, not by hardware language, they can understand or change or modify it easily. More specifically, since the application algorithm is written by software, the development and debugging are easy to them.
2) Our designed processor can be used for integer arithmetic operations on numbers with variable size longer than 64 bits without further modifications.
3) We exploit the feature of embedded DSP (DSP48E1) block in FPGA for processing flexible-length numbers. Because of this feature, we process each 17-bit block of these numbers rather than single bit to speed up computations.

To the best of our knowledge, there is no related work so far. However, we have shown the potentiality of our designed flexible-length processor. For this purpose, we have implemented 2048-bit RSA encryption/decryption by software written by machine instructions and compare it with direct hardware implementation of 2048-bit RSA encryption/decryption [15], maximized making use of DSP blocks in FPGA. Direct hardware implementation of 2048-bit RSA encryption/decryption module runs in 447.027MHz using 123940864 clock cycles, that is, in 277.26ms. A multiplier in the DSP block works in more than 90 percent over all the clock cycles. Thus, it has minimum overhead. Also, it uses one DSP48E1, one BRAM and few (170) logic blocks (slices). Therefore, this is an optimal implementation.

On the other hand, our intermediate approach using our designed processor for implementing 2048-bit RSA encryption/decryption by software uses one DSP48E1, four BRAMs and few (170) logic blocks (slices) that runs in 635.65ms. Although, our intermediate approach is slower, it has several advantages. Since application algorithm is written by software, the development and debugging even by a non-expert are easy. Also, it is more flexible and scalable.

We summarize several significant points of our results as follows:

- A flexible-length-arithmetic processor is proposed for the applications which require arithmetic operations for numbers longer than 64 bits. Even, numbers longer than 2048 bits or higher can be computed by our designed processor without any modification.
- We say that our proposed processor is flexible in a sense that it can support arithmetic operations for numbers with flexibly many bits or numbers with variable size longer than 64 bits.
- We propose an intermediate approach of software and hardware to implement above mentioned applications which makes it easy for debugging and further development by the non-experts or by the beginners.
- We have shown the potentiality of our designed processor through the implementation of 2048-bit RSA encryption/decryption by software. The resulting processor uses one DSP48E1, four BRAMs and few (170) logic blocks (slices) and RSA encryption software runs in 635.65ms. We compare our results with the results of 2048-bit RSA encryption/decryption optimal implementation [15] by HDL. This optimal implementation requires one DSP48E1, one BRAM and few (180) logic blocks (slices) and runs it in 277.265ms. Although our approach is slower, however, it has several advantages. Since, the application algorithm is written by software, the development and debugging are easy. Also, it is an almost scalable based on experimental results.
- Further, we compare the results of the execution time for 64-bit, 128-bit, 256-bit, 512-bit and 1024-bit RSA encryption/decryption as illustrated in Table IV. Experimental results show that the execution time ratio of our work over optimal one is decreasing with increasing bit length which indicates that proposed processor architecture will be more efficient for a bit length higher than 2048-bit.

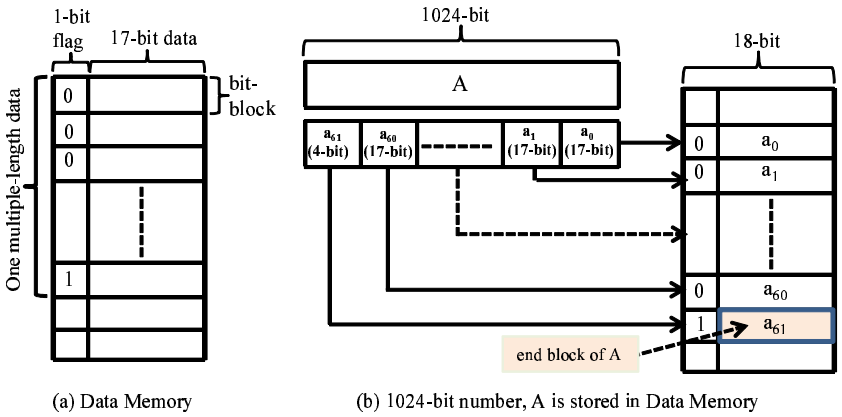

Fig. 5. Data of 1024-bit Length is Stored in Memory (BRAM)

The rest of this paper is organized as follows: Section II briefly describes the Multiple-length-arithmetic operation. In Section III, we describe our proposed architecture. The RSA cryptography as an application is described briefly in Section IV. Section V describes experimental results and discussions. Finally Section VI concludes this work.

II. Multiple-Length-Arithmetic Operation

The main purpose of this section is to describe Multiple-Length-Arithmetic operations. Suppose that A and B are two multiple-length numbers of 1024 bits each. We partition these numbers into several blocks of 17 bits. First, we see that how a multiple-length number of 1024 bits is stored in data memory. Figure 5(a) shows a data memory (BRAM). Every 17-bit block data together with 1-bit flag represents a bit-block of 18 bits and MSB (Most Significant Bit) of each bit-block is known as flag which set to 1 indicates the end of each stored multiple-length data into the data memory as shown in Figure 5(b). In this figure, multiple-length data A of 1024 bits is divided into 61 numbers of 17 bits block such as a_0, a_1, ..., a_{60}, a_{61}. Every 17 bits block of multiple-length data, A together with 1-bit flag is stored in different memory location of the data memory (BRAM).

Now, let us see the instruction memory as well as instruction format of multiple-length or multi-double long data as illustrated in Figure 6. Figure 6 (a) represents an instruction memory in which 53 bits instruction together with 1-bit flag can be stored at any address of the instruction memory addresses. In this case, 1-bit flag is set to 1 indicates the last instruction for execution. Note that, addresses of the instruction memory are handled by the Program Counter (PC) which will be described later.

Let us give an example of a multiplication of two multiple-length or two multi-double long data. However, we can also perform other arithmetic operations such as addition, subtraction, division, comparison of multi-double long data. Suppose u and v represent two multi-double long data. We are multiplying u by v and the result is stored in w, that is

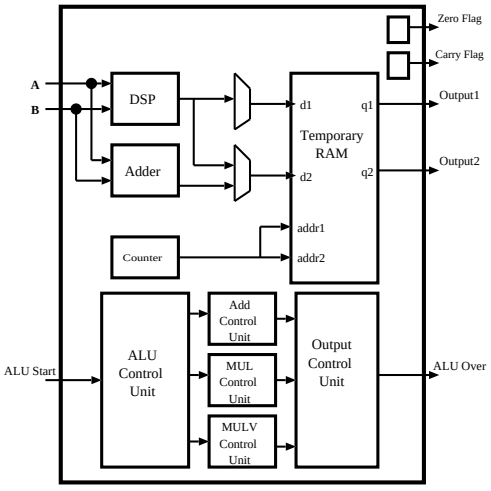

Fig. 9. ALU Architecture

$w = u \cdot v$. An assembly instruction for this computing of multi-double long data is as follows:

MUL A, B, C

In the above instruction, A, B and C are known as operands of 16-bit each which can be used to indicate 2^{16} different addresses 0, 1, ..., $2^{16} - 1$ of the data memory (BRAM) and MUL is known as OPCODE of 5-bit which determines the operation of operands (in this case multiplication) as illustrated in Figure 6(b). Let us see Algorithm 1 for multiplication of two multi-double long data u and v.

- Algorithm 1: Multi-Double Long Multiplication -

B: number of digits in radix-2^{17} operands
n: last number of digit of radix-2^{17} numbers in u
m: last number of digit of radix-2^{17} numbers in v
Input: $u = \sum_{i=0}^{n-1} u_i \cdot B^i$, $v = \sum_{i=0}^{m-1} v_i \cdot B^i$
Output: $w = u \cdot v$
1. **for** $j = 0$ **to** $m - 1$ **do**
2. $c \leftarrow 0$
3. $w_0 \leftarrow 0$

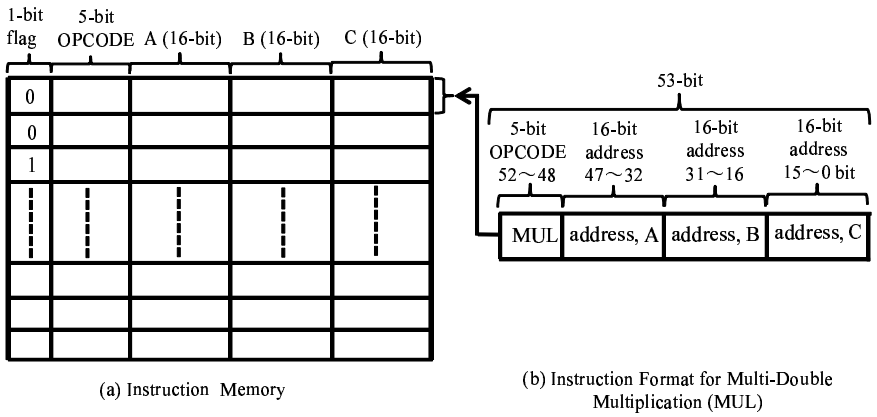

Fig. 6. An Instruction Memory and an Instruction Format for Multi-Double Data

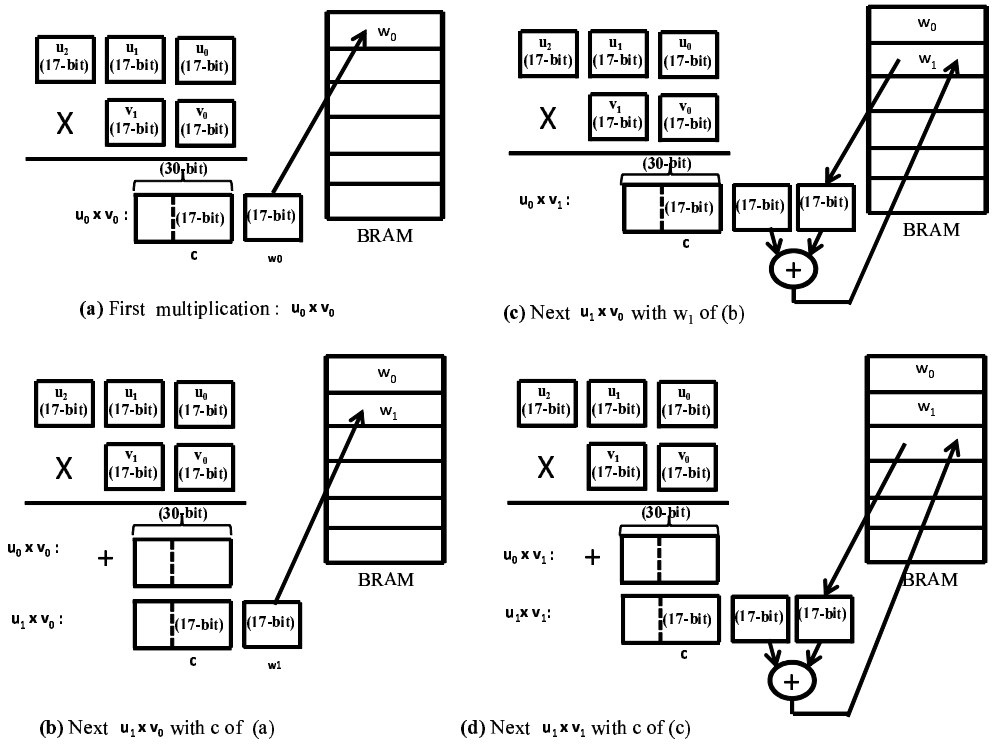

Fig. 7. Example of Multi-Double Long Multiplication

4. **for** $i = 0$ **to** $n - 1$ **do**
5. $\{c, w_{i+j}\} \leftarrow w_{i+j} + u_i \cdot v_j + c$
6. **end for**
7. $w_{n+j} \leftarrow c$
8. **end for**
9. return $\sum_{i=0}^{n+m-1} w_i \cdot B^i$

For the reader's benefit, we will show a simple example of the above algorithm as illustrated in Figure 7. In Figure 7 (a), u_0 of 17-bit block is multiplied by v_0 and the result of 34 bits is extended to 47 bits, because of the embedded multiplier in our target device. Then, 47 bits result is partitioned into higher 30 bits, c and lower 17 bits, w_0 and finally result, w_0 is stored in data memory (BRAM). Similarly, in Figure 7 (b), u_1 multiplied by v_0 and the result of 34 bits is extended to 47 bits, Then, 47 bits result is partitioned into higher 30 bits, c and lower 17 bits, w_1 and result, w_1 is stored in data memory (BRAM) and 30 bits, c is added with the c in Figure 7 (a). For simplicity, we ignore the multiplication operation with u_2. In Figure 7 (c), u_0 multiplied by v_1 and the result of 34 bits is extended to 47 bits, Then, 47 bits result is partitioned into higher 30 bits, c and lower 17 bits, w_1 which is again added with w_1 as illustrated in Figure 7 (b) and then finally result, w_1 is stored in data memory (BRAM) and so on.

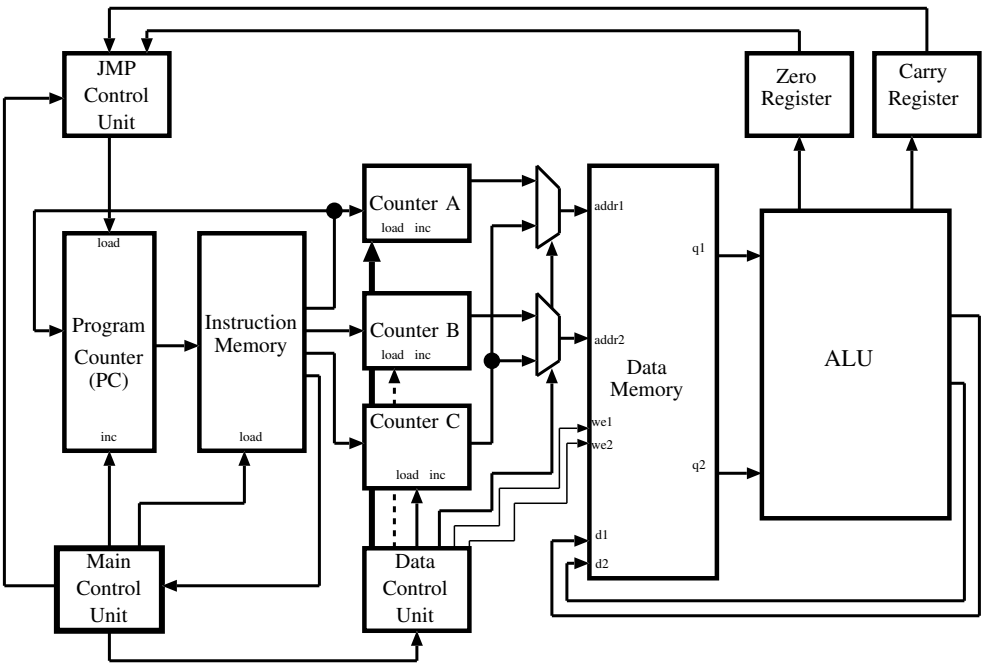

Fig. 8. Our Proposed Processor Architecture

Finally, multiplication result, w of two multi-length numbers is computed by the line 9 of an Algorithm 1.

III. Our Proposed Processor Architecture

Let us briefly describe our proposed processor architecture for multiple-length-arithmetic operations. Our designed processor consists of program counter, instruction memory, address counters, data memory, ALU, registers, control units as illustrated in Figure 8. For the reader's benefit, these elements are briefly described as follows:

- **program counter (PC):** PC is a counter which can hold or point the addresses of the instruction memory. Generally, it holds starting address of an instruction memory in which instruction is stored and after that it can be incremented to point the next address of the next instruction.
- **instruction memory:** It is an array of memory in which instructions of a program can be stored. In our case, this memory is used to store multiple-length arithmetic instructions, each of 54 bits which is shown in Figure 6.
- **address counter:** This is simply a counter which can count number as an address. For example, a 16-bit counter can count numbers from 0, 1, ..., $2^{16}-1$. In our case, we use 16-bit address counter to handle the addresses of the data memory.
- **data memory:** It is also an array of memory where information can be stored. More specifically, a data memory has b-bit data input, e-bit address input and c-bit data output, it can store 2^e words such as $M[0], M[1], \ldots, M[2^e-1]$ with c bits each, where $M[0]$, ..., $M[2^e-1]$ represent memory contents at 0, ..., (2^e-1) respectively. In our case, address is 16 bits, data at every address is 17 bits together with 1-bit flag, that is 18 bits. We use this memory to store multiple-length data.
- **ALU:** It is an arithmetic and logical unit which can perform arithmetic and logical operations for the given inputs. We have shown an architecture of the ALU in Figure 9. In this figure, two inputs A and B are given. Then arithmetic operations such as addition, multiplication etc. can be performed by adder and DSP and finally result is stored in temporary memory (BRAM) for further operations as it is included in Figure 8. This figure also have some control units to control ALU operations such as addition, multiplication etc. Flags indicate the status of the arithmetic operations.
- **register:** A register is a memory element which can be used for storing data or holds a state by flip-flops. For example, b-bit register which can store b-bit data. In our architecture in Figure 8, we have shown flag registers to hold state of operations. For example, Zero register and carry register can hold states for conditional jump instructions such as JNZ (jump if not zero), JC (jump if carry). Since these are executed based on the result of the previous instruction, they are known as conditional instructions.
- **control unit:** A control unit is used to control the operations. We have three control units such as data control unit, unconditional jump (JMP) control unit and main control unit. Data control unit is used to store data into

the data memory through data port d1 and d2 which are specified by the addresses generated by address counters A, B and C through port addr1 and addr2 respectively. This data stored operation is handled by write enable 1 (we1) and write enable 2 (we2) and these are controlled by the data control unit as illustrated in Figure 8. Main control unit mainly controls the operations of the PC, data control unit, instruction memory and JMP control unit such that operations run properly. A JMP control unit is used to handle unconditional jump instruction. In fact, it controls the program counter (PC), whether it increases or not through load signal.

We will describe a multiplication of two multi-double long data as it is seen in Section II using our processor architecture as illustrated in Figure 8. We assume that two multi-double long data are stored in data memory as shown in Figure 5. Temporary memory in ALU is used to store the result of operations temporarily. We also assume that the instructions of 54-bit each are stored in instruction memory. Let us recall the assembly instruction for multiplication of two multi-double long data as it is seen in Section II. Using our processor architecture, multiplication of two multi-double long data can be described is as follows:

- First, PC holds the address of this instruction (MUL A, B, C), that is already in instruction memory. Note that A, B and C, each of them indicates 16-bit data that is used to specify the address of data memory through port addr1 and addr2. More specifically, A, B or C can handle 2^{16} distinct addresses 0, 1, ..., $2^{16}-1$ of the data memory.
- After executing the above instruction, Counter A, Counter B and Counter C hold the first address value 0 as inputs initially that are controlled by load signal in each counter. Next, address value 0 from address counter A and B is used to specify first addresses for the first 17-bit blocks of the two multi-double long data in data memory. After that, ALU takes these data of 17-bit blocks as inputs and performs multiplication operation on these. Then lower 17-bit is stored in temporary memory in ALU and higher 17-bit is extended to 30-bit by DSP48E1 as illustrated in Figure 4 and then it is stored in register in DSP48E1. Sometimes, lower 17-bit result which is stored in temporary memory needs to write into data memory for further arithmetic operation. However, for the first multiplication, it is not needed. This data written operation from temporary to data memory is done through data port d1 or d2 which is controlled by write enable 1(we1) or write enable 2(we2) and address for it is specified by address counter C through address port addr1 or addr2. These operations are clearly illustrated in Figure 7.
- similarly, multiplication executes until there is no more 17-bit block data for multiplication. Finally multiplication result can be computed by the line 9 of an Algorithm 1 using different lower 17-bit blocks stored in temporary memory.

Since each address counter generates 16-bit address, we can also handle multiplication of numbers longer than 2048-bit using our processor architecture. In others, 16-bit address of the counter can be addressed up to 64 number of 1K x 18-bit data memory (BRAM). Note that, for storing 2048-bit operand, 7-bit address is enough. Because, 2048-bit is equal to the 120 multiplied by 17-bit block plus 8, that is, it requires 121 number of 17-bit block. Hence, 1K x 18-bit data memory (BRAM) can be even used for more than 2048-bit multiplication.

In the following section, we implement RSA cryptography using our proposed architecture and it is programmed by assembly language. The assembly instructions of number, 117, each of 54-bit are needed to implement modular exponentiation algorithm. Due to page limitation, we only show an assembly code for Montgomery Multiplication Algorithm (that corresponds to the Algorithm 4 in Section IV) as illustrated below which is used to implement the modular exponentiation algorithm. We refer the readers to Section IV for details. In below assembly code, the registers $R1$ and $R2$ are used to take inputs X and Y, each of 64-bit. Also 64-bit M is given. The registers $R3$, $R4$, $R5$ is used to hold the intermediate results and final results of the Montgomery Multiplication is stored either in $R6$ or C. Note that, 64-bit data in register $R1$ is divided into several blocks of 17-bit each and these are stored in several block registers such as $R1_0$, $R1_1$, $R1_2$, $R1_3$ (lower block to higher block). For the case of other registers, we can explain in similar way.

[Assembly Code for Montgomery Multiplication]

R1=X, R2=Y;
R3=0, R4=0, R5=0, R6=0;
C=0;

01:	MUL,	$R1_0$, $R2_0$,	$R3_0$;	X ($R1$)×Y($R2$) store in $R3$
02:	MOVI,	$R3_0$, $R4_0$,	$R3_3$;	copy blocks of $R3$ into $R4$
03:	MASK,	$R4_3$, $R4_3$,	1FFF;	make last 4-bit of $R4_3$ as 0's
04:	MUL,	$R4_0$, $-M_0^{-1}$	$R5_0$;	R4× $-M^{-1}$ store in $R5$
05:	MOVI,	$R5_0$, $R4_0$,	$R5_3$;	copy blocks of $R5$ into $R4$
06:	MASK,	$R4_3$, $R4_3$,	1FFF;	make last 4-bit of $R4_3$ as 0's
07:	MUL,	$R4_0$, M_0,	$R5_0$;	R4×M store in $R4$
08:	ADD,	$R3_0$, $R5_0$,	$R6_0$;	store $R3+R5$ in $R6$
09:	SHR,	$R6_3$, $R6_0$,	13;	64-bit shift right of $R6$
0A:	CMP,	$R6_0$, M_0;		$R6$ compare with M
0B:	JC,	0D;		if $R6 > M$, go to OD
0C:	SUB,	$R6_0$, M_0,	$R6_0$;	R6-M is the results in $R6$
0D:	MOV,	$R6_0$, C_0;		results in $R6$ move to C

For the benefit of readers, we also show how much cycles are required for each multiple-length-arithmetic operation of different bit lengths of R as illustrated in Table I. Note that, we use 117 number of assembly instructions with 16 different OPCODEs such as ADD, SUB, MUL etc. as shown in Table I to implement modular exponentiation algorithm.

TABLE I
NUMBER OF CYCLES FOR MULTIPLE-LENGTH-ARITHMETIC OPERATIONS

Bit length R	128	256	512	1024	2048
ADD(addition)	23	39	69	129	249
SUB(subtraction)	23	39	69	129	249
MUL(multiplication)	95	311	1061	3911	15011
MULV(vector multiplication)	23	39	69	129	249
INC(increment)	11	11	11	11	11
DEC(decrement)	11	11	11	11	11
CMP(compare)	19	27	42	72	132
JMP(jump)	5	5	5	5	5
JZ(jump if zero)	5	5	5	5	5
JNZ(jump if not zero)	5	5	5	5	5
JC(jump if carry)	5	5	5	5	5
JNC(jump if not carry)	5	5	5	5	5
MOV(Move all)	14	23	37	67	127
MOVI(Move specified block)	14	23	37	67	127
MASK(mask)	11	11	11	11	11
SHR(shift right)	19	27	42	72	132

IV. AN APPLICATION OF RSA CRYPTOGRAPHY USING OUR PROCESSOR

This section briefly reviews the RSA Cryptography which is described details in paper [15]. Using our processor, we implement the same algorithm by software, instead of HDL as illustrated in paper [15] to make it easy for modifications or changes by a non-expert or by a beginner.

In RSA [18], the modular exponentiation $C = P^E \bmod M$ or $P = C^D \bmod M$ are computed, where P and C are plain and cypher text, and (E, M) and (D, M) are encryption and decryption keys. Usually, the bit length in P, E, D, and M is 512 or larger. Also, the modulo exponentiation is repeatedly computed for fixed E, D, and M, and various P and C. Since modulo operation is very costly in terms of the computing time and hardware resources, *Montgomery modular multiplication* [19], [20], [21], [22] is used, which does not directly compute modulo operation.

Montgomery multiplication [19], [20], [21], [22] is an optimal method to calculate modular exponentiation. Three R-bit numbers X, Y, and M are given, and $(X \cdot Y + q \cdot M) \cdot 2^{-R} \bmod M$ is computed, where an integer q is selected such that the least significant R bits of $X \cdot Y + q \cdot M$ are zero. The value of q can be computed as follows. Let $(-M^{-1})$ denote the minimum non-negative number such that $(-M^{-1}) \cdot M \equiv -1(\text{ or } 2^R - 1) \pmod{2^R}$. Since M is odd, then $(-M^{-1}) < 2^R$ always holds. We can select q such that $q = ((X \cdot Y) \cdot (-M^{-1}))[r-1, 0]$. For such q, $(X \cdot Y + q \cdot M)[r-1, 0]$ are zero. We refer readers to paper [23] for an example.

Radix-2^r Montgomery multiplication is shown in Algorithm 2. In Algorithm 2, $d = \lceil R/r \rceil$ presents the number of digits in radix-2^r operands. The multiplier Y is partitioned by each r-bit and Y_i represents the i-th digit of Y. Therefore, Y could be given by $Y = \sum_{i=0}^{d-1} 2^{ir} \cdot Y_i$. After d loops, R-bit Montgomery multiplication can be computed. As far as now, Montgomery multiplication could be computed by multiplication, addition and shift operations without modulo operations.

- Algorithm 2: radix-2^r Montgomery Multiplication -
radix-2^r, $d = \lceil R/r \rceil$, $X, Y, M \in \{0, 1, ..., 2^R - 1\}$,
$Y = \sum_{i=0}^{d-1} 2^{ir} \cdot Y_i$, $Y_i \in \{0, 1, ..., 2^r - 1\}$
$(-M^{-1}) \cdot M \equiv -1 \bmod 2^r$, $-M^{-1} \in \{0, 1, ..., 2^r - 1\}$
Input: $X, Y, M, -M^{-1}$
Output: $S_d = X \cdot Y \cdot 2^{-dr} \bmod M$
1. $S_0 \leftarrow 0$
2. **for** $i = 0$ **to** $d - 1$ **do**
3. $\quad q_i \leftarrow ((S_i + X \cdot Y_i) \cdot (-M^{-1})) \bmod 2^r$
4. $\quad S_{i+1} \leftarrow (X \cdot Y_i + q_i \cdot M + S_i)$ / 2^r
5. **end for**
6. **if** $(M \le S_d)$ **then** $S_d \leftarrow S_d - M$

Since $X \cdot Y + q \cdot M \equiv X \cdot Y \pmod M$, we write $(X \cdot Y + q \cdot M) \cdot 2^{-R} \bmod M = X \cdot Y \cdot 2^{-R} \bmod M$. Let us see how Montgomery modular multiplication is used to compute $C = P^E \bmod M$. Suppose we need to compute $C = P^E \bmod M$. For simplicity, we assume that E is a power of two. Since R and M are fixed, we can assume that $2^{2R} \bmod M$ is computed beforehand. We first compute $P \cdot (2^{2R} \bmod M) \cdot 2^R \bmod M = P \cdot 2^R \bmod M$ using the Montgomery modular multiplication. We then compute the square $(P \cdot 2^R \bmod M) \cdot (P \cdot 2^R \bmod M) \cdot 2^{-R} \bmod M = P^2 \cdot 2^R \bmod M$. It should be clear that, by repeating the square computation using the Montgomery modular multiplication, we have $P^E \cdot 2^R \bmod M$. After that, we multiply 1, that is $(P^E \cdot 2^R \bmod M) \cdot 1 \cdot 2^{-R} \bmod M = P^E \bmod M$ is computed. In this way, cypher text C could be obtained.

Algorithm 3 shows the modular exponentiation using Montgomery multiplication of Algorithm 2. In Algorithm 3, E_b represents the size of E. Inputs $2^{2dr} \bmod M$ and $-M^{-1}$ are given. To use Montgomery modular multiplication, C and P are converted from 1 and P in the 1st line and the 2nd line, respectively. The line 1, 2, 4, 5 and 7 in Algorithm 3 can be computed using Montgomery multiplication of Algorithm 2.

- Algorithm 3: Modular Exponentiation -
$0 \le E \le 2^{E_b} - 1$, $E = \sum_{i=0}^{E_b - 1} 2^i \cdot E_i, E_i \in \{0, 1\}$
Input: $P, E, M, -M^{-1}, 2^{2dr} \bmod M$
Output: $C = P^E \bmod M$
1. $C \leftarrow \underline{(2^{2dr} \bmod M) \cdot 1 \cdot 2^{-dr} \bmod M}$;
2. $P \leftarrow \underline{(2^{2dr} \bmod M) \cdot P \cdot 2^{-dr} \bmod M}$;
3. **for** $i = E_b - 1$ **downto** 0 **do**
4. $\quad C \leftarrow \underline{C \cdot C \cdot 2^{-dr} \bmod M}$;
5. $\quad$ if $E_i = 1$ then $C \leftarrow \underline{C \cdot P \cdot 2^{-dr} \bmod M}$;
6 **end for**
7. $C \leftarrow \underline{C \cdot 1 \cdot 2^{-dr} \bmod M}$;

Now we will describe our algorithm in Algorithm 4. Let $\{A : B\}$ denote a concatenation of A and B. For example, if $A = (FF)_{16}$ and $B = (EC)_{16}$, $\{A : B\} = (FFEC)_{16}$. Algorithm 4 is an improved algorithm from Algorithm 2. Considering the features of our target Virtex 6 FPGA, radix-2^{17} is selected. Let R denotes the size of Montgomery multiplier operands X, Y, and M. Also, $d = \lceil R/17 \rceil$ is the number of digits of the operands on radix-2^{17}. In the algorithm, we introduce the condition $17d \ge R + 3$ to ignore the subtraction

shown in the 6th line of Algorithm 2. If the condition is satisfied, we can guarantee that at least 3-bit 0 is padded to the most significant bits of the most significant digit as the redundancy. Due to the stringent page limitation, the proof is omitted. However, we can say that $M \geq C$ is always satisfied in the modular exponentiation shown in Algorithm 3 . Further, in practical RSA encryption, the size of operands is radix-2 numbers such as 512-bit, 1024-bit, 2048-bit, and 4096-bit. For radix-2^{17} system, the condition $17d \geq R+3$ is satisfied. If the condition is not satisfied, we just need to append one redundant digit at the most significant digit.

- Algorithm 4: Our Montgomery Algorithm -

radix-2^{17}, $d = \lceil R/17 \rceil, 17d \geq R+3$,
$X, Y, M, S_i \in \{0, 1, ..., 2^R - 1\}$,
$-M^{-1}, \alpha, \beta, \gamma, C_\alpha, C_\beta \in \{0, 1, ..., 2^{17} - 1\}$, $C_\gamma, C_S \in \{0, 1\}$,
$X = \sum_{i=0}^{d-1} 2^{17i} \cdot X_i, X_i \in \{0, 1, ..., 2^{17} - 1\}, X_d = 0$
$Y = \sum_{i=0}^{d-1} 2^{17i} \cdot Y_i, Y_i \in \{0, 1, ..., 2^{17} - 1\}$
$M = \sum_{i=0}^{d-1} 2^{17i} \cdot M_i, M_i \in \{0, 1, ..., 2^{17} - 1\}, M_d = 0$
$S_i = \sum_{j=0}^{d-1} 2^{17j} \cdot S_{(i,j)}, S_{(i,j)} \in \{0, 1, ..., 2^{17} - 1\}, S_d = 0$
Input: $X, Y, M, -M^{-1}$
Output: $S_d = X \cdot Y \cdot 2^{-17d} \mod M$
1. $S_0 \leftarrow 0$
2. **for** $i = 0$ **to** $d - 1$ **do**
3. $\quad q \leftarrow ((X_0 \cdot Y_i + S_{(i,0)}) \cdot (-M^{-1})) \bmod 2^{17}$
4. $\quad C_\alpha, C_\beta, C_\gamma, C_S \leftarrow 0$
5. $\quad$ **for** $j = 0$ **to** d **do**
6. $\quad\quad \{C_\alpha : \alpha\} \leftarrow X_j \cdot Y_i + C_\alpha$
7. $\quad\quad \{C_\beta : \beta\} \leftarrow q \cdot M_j + C_\beta$
8. $\quad\quad \{C_\gamma : \gamma\} \leftarrow \alpha + \beta + C_\gamma$
9. $\quad\quad \{C_S : S_{(i+1,j-1)}\} \leftarrow \gamma + S_{(i,j)} + C_S$
10. $\quad$ **end for**
11. **end for**

Algorithm 4 is a radix-2^{17} digit serial Montgomery algorithm from Algorithm 2. In other words, each 17-bit, as 1 digit, is processed every clock cycle. For this reason, the operands X, Y, M, and S_i are split into 17-bit digits X_j, Y_j, M_j, and $S_{(i,j)}$, respectively. The loop from the 2nd to 11th lines of Algorithm 4 corresponds to the 2nd to 5th lines of Algorithm 2. Comparing the two algorithms, $S_{i+1} \leftarrow (X \cdot Y_i + q_i \cdot M + S_i)$ / 2^r of the 4th line of Algorithm 2 corresponds to the digit serial processing by 4th to 10th lines of Algorithm 4 . In Algorithm 4, C_α, C_β, C_γ, and C_S are carries and they are added at the next loop. In the algorithm, C_α, C_β are 17-bit carries for 17-bit MACC, and C_γ, C_S are 1-bit carries for 17-bit addition. For example, at the 6th line X_j, Y_i are timed and added to 17-bit carry C_α, the result is 34-bit. a product of X_j and Y_i, and an addition of the product and C_α are computed. The resulting upper 17-bit denotes a carry C_α which can be added at next loop. The lower 17-bit of result is α which is used at the 8th and 9th lines. These carries in our algorithm appear in both the 17-bit MACC and the 17-bit adder to prevent a long carry chain that causes circuit delay.

V. Experimental Results and Discussions

The proposed flexible-length-arithmetic processor architecture is used to implement modular exponentiation algorithm and evaluate on Xilinx Vertex-6 XC6VLX240T-3FF1156, programmed by software and synthesis with Xilinx ISE Foundation 13.4. Table II shows the synthesized result for our work.

TABLE II
Experimental result of our modular exponentiation using Virtex-6 FPGA

	Virtex-6
Number of occupied Slices	170/301440
Number of 18k-bit BRAMs	4/416
Number of DSP48E1s	1/768
Maximum Frequency[MHz]	299.89

An optimal implementation [15], which is evaluated on Xilinx Virtex-6 FPGA XC6VLX240T-1, programmed by hardware description language Verilog HDL and synthesized by Xilinx ISE Foundation 11.4. Note that, the optimal one, programmed by HDL is specialized design by an expert so that it is difficult to debug or change by a non-expert or sometimes even by an expert.

Table III shows the synthesized results of Virtex-6 for comparing both implementations. We have used less number of logic blocks. Four BRAMs are used instead of one in an optimal implementation [15]. However one DSP (DSP48E1) is used for the both implementations. Performance of our implementation in terms of frequency and execution time is slightly less than an optimal one [15]. We also compare the execution time ratio of our implementation over optimal one which shows that this ratio increases when the bit length, R increases as shown in Table IV. It means that proposed architecture will be more efficient for implementing higher than 2048-bit modular exponentiator algorithm.

Based on results in Table III and Table IV, our implementation is near to the optimal one. Hence, we say that our implementation is an almost scalable. However, the optimal one [15] is designed to be implemented by hardware language, HDL which is difficult for modifications or changes by non-expert, because this is specially designed by an expert.

On the other hand, our implementation of RSA encryption/decryption using proposed processor architecture is designed to be implemented by software, hence it is easy for modifications or changes by a non-expert or by a beginner which makes it flexible. Even though, it has ability to support higher bit (more than 2048-bit) of RSA encryption/decryption.

VI. Conclusions

In this paper, we have presented an intermediate approach of software and hardware using DSP Slices and Block RAMs in FPGAs. More specifically, a flexible-length-arithmetic processor based on FDFM approach is presented that supports arithmetic operations for numbers with flexibly many bits, even longer than 2048 bits. The potentiality of our processor is shown through the implementation of modular exponentiator algorithm by software and compare it with the results of an

TABLE III
COMPARISON WITH PREVIOUS 2048-BIT MODULAR EXPONENTIATOR ALGORITHM

	Optimal Implementation [15]	This Work
device	Xilinx XC6VLX240T-1	Xilinx XC6VLX240T-3FF1156
logic block	180 Slices	170 Slices
memory block	1 BRAM	4 BRAMs
DSP block	1 DSP48E1	1 DSP48E1
frequency[MHz]	447.02	299.89
execution time[ms]	277.26 (worst case)	635.65 (worst case)
scalable	yes	yes (almost)

TABLE IV
WORST-CASE EXECUTION TIME COMPARISON OF MODULAR EXPONENTIATOR USING VIRTEX-6 FPGA

bit length R	64	128	256	512	1024	2048
(A) Our work: execution time[ms]	0.11	0.42	2.12	12.48	85.69	635.65
(B) Optimal work: execution time[ms]	0.02	0.12	0.74	4.99	36.37	277.26
Ratio A/B:	5.5	3.5	2.9	2.5	2.4	2.3

optimal implementation [15] by hardware language. Results in Table III and Table IV show that our work is an almost near to the optimal one. Hence, it is an almost scalable. However, optimal one is designed to be implemented by an expert with hardware language. Hence, this is difficult for debugging and further development by the non-experts or by the beginners. On the contrary, our work using proposed processor architecture is designed to be implemented by software which is easy for debugging and development even by the non-experts or by the beginners. Hence, our work is more flexible. Undoubtedly, our designed processor can be used for those applications such as AES which requires integer arithmetic operations with many bits. In future, we have a plan to improve our instruction set for flexible-length-arithmetic operations.

REFERENCES

[1] *VIRTEX-6 FPGA Memory Resources(V1.5)*, Xilinx Inc., 2010.

[2] *VIRTEX 6 ML605 Hardware USER GUIDE (V1.2.1)*, Xilinx Inc., 2010.

[3] *VIRTEX-6 FPGA DSP48E1 SLICE USER GUIDE (V1.3)*, Xilinx Inc., 2011.

[4] J. Bordim, Y. Ito, and K. Nakano, "Accelerating the CKY parsing using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 803–810, 2003.

[5] J. L. Bordim, Y. Ito, and K. Nakano, "Instance-specific solutions to accelerate the CKY parsing for large context-free grammars," *International Journal on Foundations of Computer Science*, vol. 15, no. 2, pp. 403–416, 2004.

[6] Y. Ito and K. Nakano, "A hardware-software cooperative approach for the exhaustive verification of the Collatz conjecture," in *Proc. of International Symposium on Parallel and Distributed Processing with Applications*, 2009, pp. 63–70.

[7] K. Nakano and Y. Yamagishi, "Hardware n choose k counters with applications to the partial exhaustive search," *IEICE Transactions on Information and Systems*, vol. E88-D, no. 7, 2005.

[8] Y. Ito and K. Nakano, "Efficient exhaustive verification of the Collatz conjecture using DSP blocks of Xilinx FPGAs," *International Journal of Networking and Computing*, vol. 1, no. 1, pp. 19–62, 2011.

[9] K. Nakano and E. Takamichi, "An image retrieval system using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 811–818, May 2003.

[10] Y. Ago, Y. Ito, and K. Nakano, "An FPGA implementation for neural networks with the FDFM processor core approach," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 28, no. 4, pp. 308–320, 2013.

[11] Y. Ito and K. Nakano, "Low-latency connected component labeling using an FPGA," *International Journal of Foundations of Computer Science*, vol. 21, no. 03, pp. 405–425, 2010.

[12] X. Zhou, N. Tomagou, Y. Ito, and K. Nakano, "Efficient Hough transform on the FPGA using DSP slices and block RAMs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2013, pp. 771–778.

[13] *VIRTEX-6 FPGA DSP48E1 SLICE USER GUIDE (V1.2)*, Xilinx Inc., 2009.

[14] Y. Ago, A. Inoue, K. Nakano, and Y. Ito, "The parallel FDFM processor core approach for neural networks," in *Proc. of International Conference on Networking and Computing*, December 2011, pp. 113–119.

[15] S. Bo, K. Kawakami, K. Nakano, and Y. Ito, "An RSA encryption hardware algorithm using a single DSP Block and single Block RAM on the FPGA," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 277–289, July 2011.

[16] Y. Ito, K. Nakano, and S. Bo, "The parallel FDFM processor core approach for CRT-based RSA decryption," *International Journal of Networking and Computing*, vol. 2, pp. 56–78, 2012.

[17] K. Nakano, K. Kawakami, and K. Shigemoto, "RSA encryption and decryption using the redundant number system on the FPGA," in *In Proc. IEEE International Symposium on Parallel and Distributed Processing*, May 2009, pp. 1–8.

[18] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[19] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," in *Proc. of the 14th IEEE Symposium on Computer Arithmetic*, 1999, pp. 70–77.

[20] ——, "High-radix montgomery modular exponentiation on reconfigurable hardware," *IEEE Trans. on Computers*, vol. 50, no. 7, pp. 759–764, 2001.

[21] P. L. Montgomery, "Modular multiplication without trial division," *Math. of Comput.*, vol. 44, pp. 519–521, 1985.

[22] A. F. Tenca and C. K. Koç, "A scalable architecture for Montgomery multiplication," in *Proc. of the First International Workshop on Cryptographic Hardware and Embedded Systems*, 1999, pp. 94–108.

[23] M. Niimura and Y. Fuwa, "Improvement of radix-2^k signed-digit number for high speed circuit," *Formalized Mathematics*, vol. 11, no. 2, pp. 133–137, January 2003.