PAPER Special Section on Parallel and Distributed Computing and Networking

An FPGA Implementation for a Flexible-Length-Arithmetic Processor Employing the FDFM Processor Core Approach*

Tatsuya KAWAMOTO[†], ***Nonmember***, **Xin ZHOU**[†], ***Student Member***, **Jacir L. BORDIM**[††], **Yasuaki ITO**[†a)], ***and*** **Koji NAKANO**[†], ***Members***

SUMMARY Algorithms requiring fast manipulation of multiple-length numbers are usually implemented in hardware. However, hardware implementation, using HDL (Hardware Description Language) for instance, is a laborious task and the quality of the solution relies heavily on the designer expertise. The main contribution of this work is to present a flexible-length-arithmetic processor based on FDFM (Few DSP slices and Few Memory blocks) approach that supports arithmetic operations on multiple-length numbers using FPGAs (Field Programmable Gate Array). The proposed processor has been implement on the Xilinx Virtex-6 FPGA. Arithmetic instructions of the proposed processor architecture include addition, subtraction, and multiplication of integer numbers exceeding 64-bits. To reduce the burden of implementing algorithm directly on the FPGA, applications requiring multiple-length arithmetic operations are written in a C-like language and translated into a machine program. The machine program is then transferred and executed on the proposed architecture. A 2048-bit RSA encryption/decryption implementation has been used to assess the goodness of the proposed approach. Experimental results shows that the computing time, using the proposed architecture, of a 2048-bit RSA encryption takes only 2.2 times longer than a direct FPGA implementation. Furthermore, by employing multiple FDFM cores for the same task, the computing time reduces considerably.

key words: *multiple-length-numbers, multiple-length-arithmetic, FPGA, RSA, montgomery modular multiplication*

1. Introduction

An FPGA (Field Programmable Gate Array) is a programmable logic device designed to be configured via HDL (Hardware Description Language) after manufacturing. Owing to its programmable features and affordable prices, FPGAs devices gained considerable attention in recent years [2]. FPGAs can implement hundreds of circuits that work in parallel and can be explored to accelerate useful computations. The most common FPGA architecture consists of an array of logic blocks, I/O pads, block RAMs and routing channels. Furthermore, embedding DSP (Digital Signal Processor) slices into an FPGA device is reported to provide higher performance and, consequently, providing a broader range of applications [3].

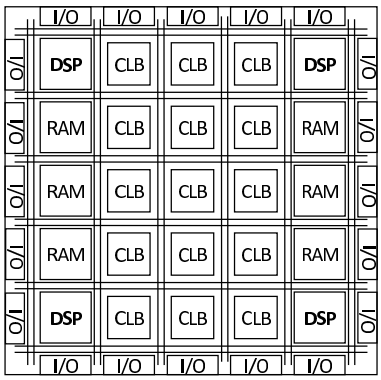
Manuscript received January 8, 2016.
Manuscript revised May 9, 2016.
Manuscript publicized August 24, 2016.

[†]The authors are with the Department of Information Engineering, Hiroshima University, Higashihiroshima-shi, 739–8527 Japan.
[††]The author is with the Department of Computer Science, University of Brasilia, Brasilia-DF, Brazil.

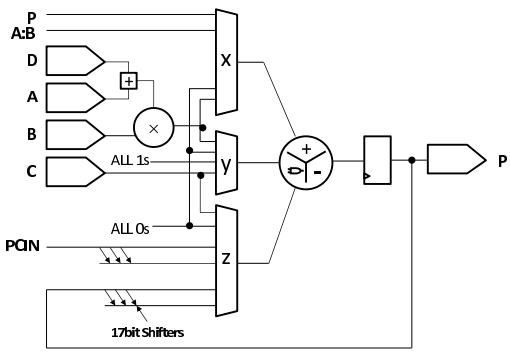
*A preliminary version of this paper has been presented at the Third International Symposium on Computing and Networking (CANDAR 2015) [1].

a) E-mail: yasuaki@cs.hiroshima-u.ac.jp

DOI: 10.1587/transinf.2016PAP0029



(a) Internal configuration of FPGA.



(b) Architecture of DSP48E1.

Fig. 1 FPGA (Field Programmable Gate Array).

Figure 1 (a) depicts the internal configuration of a common FPGA board [2]. The figure shows the CLB (Configurable Logic Blocks) used on a Virtex-6, which consists of 2 sub-logic blocks, called "slice". The use of LUTs (Look Up Tables) and flip-flops in the slices allows the implementation of a number of combinatorial and sequential circuits. The Virtex-6 FPGA incorporates the DSP48E1 slices, which are equipped with a multiplier, adders, logic operators, etc. As illustrated in Fig. 1 (b), 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 an 18-bit and 25-bit 2's complement numbers yielding a 48-bit 2's complement production. Programmable pipelining of input operands, intermediate products, and accumulator outputs enhances throughput and improves clock frequency. The DSP48E1 uses a pipeline registers between operators to reduce the delay. The block RAM in the Virtex-6 FPGA is an embedded mem-

ory supporting synchronized read and write operations [4]. The block RAM can be configured as a 36k-bits dual-port block RAMs, FIFOs, or two 18k-bits dual-port RAMs. In this work, the latter configuration is used (i.e., 2k × 18-bit dual-port RAM).

Integer numbers exceeding two of more computer words are called *multiple-length numbers* [5]. Similarly, arithmetic operations on multiple-length numbers is called *multiple-length arithmetic*. Thus, an application involving integer arithmetic operations on multiple-length numbers exceeding 64-bits cannot be performed directly by a conventional 64-bit CPUs as its instruction set support integers of at most 64-bits in length. When dealing with multiple-length arithmetic, the CPU needs to repeat arithmetic operations on fixed 64-bits. Such procedure, however, increases the execution time significantly.

An alternative to speed up computation of multiple-length numbers is to employ algorithms implemented in FPGAs. On the other hand, the implementation of an algorithm in hardware is a cumbersome procedure where the resulting speed up gain also depends on the designer expertise. That is, to implement an algorithm in hardware using HDL, such as Verilog, users should have sufficient knowledge of digital circuit design. Furthermore, low level, binary instructions make the task of coding and debugging a non-trivial one. Thus, the implementation of an algorithm using HDL is usually a difficult task for non-expert or beginners to accomplish. By contrast, higher-level languages, such as C programming language, allows the programmer to concentrate on the logic of the problem to be solved rather than the intricacies of the hardware architecture required by low-level languages such as assembly.

The main contribution of this paper is to present an intermediate approach of software and hardware using FPGAs to support arithmetic operations on multiple-length numbers where the developing and debugging tasks can be easily accomplished even by non-experts. More specifically, this paper proposes a flexible-length-arithmetic processor based on the FDFM (*Few DSP slices and Few Memory blocks*) approach that supports applications involving arithmetic operations on multiple-length numbers. To achieve this, a compiler consisting of lexical scanner *Flex* [6] and a context parser *Bison* [7] are used to convert a C-like language program into an assembly language program. Since the assembly program cannot be executed directly on the proposed processor, a translator is proposed for converting the assembly program into a machine-executable program. The above steps allows converting a program written in a C-like language into a machine program that is executed on the proposed processor. Hence, in the proposed approach, coding and debugging tasks are simplified, allowing non-experts to take the advantage of the FPGAs to perform arithmetic operations on multiple-length numbers. The contributions of this paper can be summarized as follows:

(*i*) A flexible-length arithmetic processor based on FDFM approach for computing multiple-length number exceeding 64-bits and even longer than 2048-bits using a single machine instruction;

(*ii*) An intermediate approach of software and hardware that simplifies the tasks of coding and debugging applications requiring multiple-length arithmetic operations;

(*iii*) The task of computing multiple-length integer numbers of 64-bits, 128-bits, even longer than 2048-bits can be accomplished without modification of hardware design of the processor;

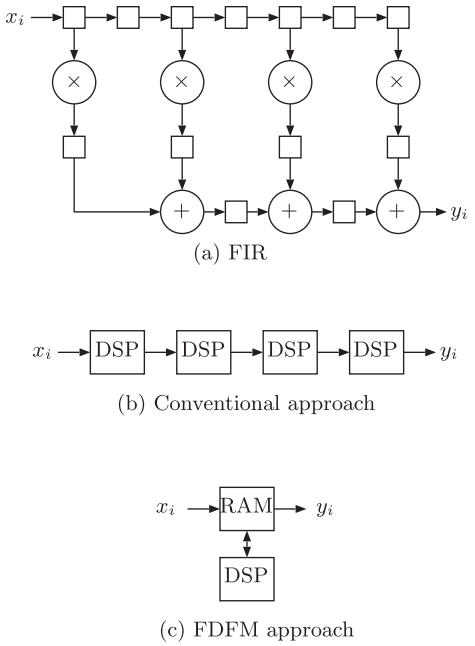
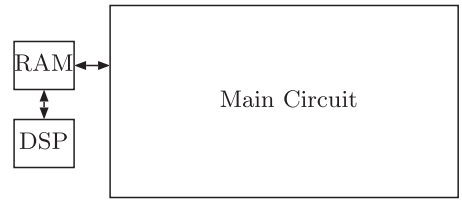


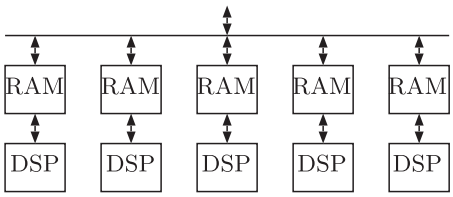
Fig. 2 FDFM approach over conventional one to compute the FIR

Next, a simple example of the FDFM approach is presented. Figure 2 (a) illustrates a hardware algorithm to compute the output of the 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 for implementing the FIR is to use four DSP slices as illustrated in Fig. 2 (b) [3]. In such case, the hardware implementation requires the number of DSP slices to match the number of multipliers. However, the FDFM approach uses one or few DSP slices and one or few block RAMs to implement the FIR. Figure 2 (c) depicts the FDFM approach using one DSP slice and one block RAM to implement the FIR where the coefficients $a_0, a_1, \ldots, a_n$ are stored in the block RAM. For further details on the FDFM approach, the interested reader may refer to [8]–[11] and the references within. The work in [12] presents a comparison of a soft-core processor implementation and a direct implementation on FPGAs employing the FDFM approach.

As mentioned above, the FDFM approach allows the designer to take advantage of the available hardware resources. As illustrated in Fig. 3 (a), the FDFM can be implemented using as few as a single DSP and a block RAM. This is particularly useful for the case where the main circuit uses most of the FPGA hardware resources. However, when



(a) Minimum implementation



(b) Parallel implementation

Fig. 3 Adapting the FDFM approach to the available hardware resources.

hardware resources are less constrained, multiple FDFM processor cores working in parallel can be implemented as shown in Fig. 3 (b). This work explores the aforementioned strategies. Indeed, the proposed flexible-length-arithmetic processor based on the FDFM approach can be implemented using a single DSP slice and two block RAMs. The proposed FDFM processor supports 38 assembly instructions. The experimental results on a Virtex-6 FPGA shows that the computing time of a 2048-bits RSA encryption takes only 2.2 times longer than a direct FPGA implementation.

Thus, there are two points of the advantage for the proposed processor; (i) our processor is compact and high-performance by the FDFM approach and (ii) users can implement high-performance software programs with flexible-length-arithmetic instructions on the proposed processor easily. The proposed processor is compactly designed and uses very few FPGA resources. We use only one DSP slice, two block RAMs, and a few CLBs to implement the processor. The processor directly supports flexible-length-arithmetic instructions by the DSP slice. Therefore, these instructions can be executed without any overheads caused by flexible-length-arithmetic. The performance of the proposed processor is a little lower than hardware implementations that are designed for a specific computation. However, considering that users can program by software, our processor has enough performance and high versatility. On the other hand, flexible-length-arithmetic computations are used in many applications such as data encryption/decryption, scientific computation, etc. It is generally difficult for non-experts to implement them not only in hardware circuits but also in software programs. However, on the proposed processor, users can utilize flexible-length-arithmetic instructions. Furthermore, a software program using C-like language can be easily written for a specific application requiring multiple-length-arithmetic operations with the compiler and assembler. As shown in the above, although the program is executed as software implementation, the performance is close to that of the specific hardware circuit.

There are several existing FPGA implementations proposed for performing arithmetic operations of multi-length numbers [13], [14]. Kalathungal [13] proposed an ALU (Arithmetic Logic Unit) that supports addition, subtraction, multiplication and division for multi-length integer numbers. Since the straightforward methods are used to perform these arithmetic operations and the proposed ALU does not use any DSP slice and block RAM, the implementation of the ALU for up to 96-bit integers in an Altera CYCLONE V FPGA, is clocked at only 20MHz. In other words, the proposed ALU is not suitable to perform arithmetic operations for large numbers. Pfander *et al.* [14] proposed a configurable block for performing only multi-length multiplication with a word-length that can be changed at runtime. The proposed configurable block consists of 1-bit full adders, and performs multiplication based on addition and shift operation. If the length of input number increases, the path of carry propagation and the usage of FPGA resources increase. Hence, the proposed block is also not suitable to perform multiplication for large numbers. There are also some existing soft processors proposed for overcoming the problem of designing hardware with FPGAs [15], [16]. LaForest [15] proposed a soft-processor that is clocked at a high frequency, and implemented it in an Altera Stratix IV FPGA EP4SE230F29C2. However, the proposed soft processor does not support the arithmetic operation for large numbers. McGettrick *et al.* [16] used FPGA as reconfigurable digital signal Processor accelerators for digital signal processing to obtain short reconfiguration time. However, the arithmetic operations for multi-length numbers are also not supported by this work.

The rest of this paper is organized as follows: Section 2 briefly describes the multiple-length-arithmetic operation. Section 3 describes the proposed processor architecture for multiple-length-arithmetic operations. Section 4 details the implementation of the RSA algorithm on the proposed architecture. In addition, the C-like language compilation and assembly process are discussed in this section. Section 5 shows the experimental results for a single DSP slice using the FDFM approach while Sect. 6 presents a multicore DSP slice system and evaluates its performance. Finally, Sect. 7 concludes this work.

2. Multiple-Length-Arithmetic Operation

The main purpose of this section is to describe the multiple-length-arithmetic operations. For this purpose, consider a multiple-length number represented as an array of r-bits block. In general, $r = 32$ or 64 in conventional CPUs. In the proposed processor r is fixed in 17-bits which matches the bit-length manipulated by the DSP slice. Let R denote the length of the multiple-length number and d be the number of r-bit blocks. Clearly, $d = \lceil \frac{R}{r} \rceil$ blocks. For example, a 1024-bit integer consists of 61 blocks.

An example of two multiple-length number are pre-

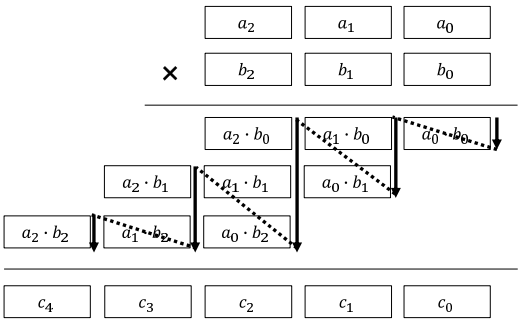


Fig. 4 Comba method for computing the product of two multiple-length numbers

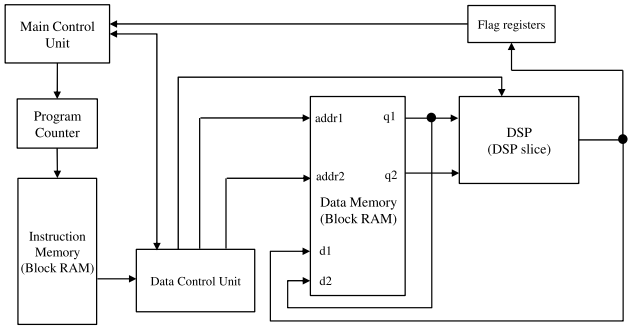


Fig. 5 Proposed processor architecture for handling multiple-length-arithmetic operations.

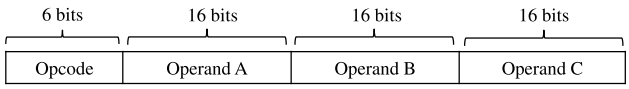


Fig. 6 Multiple-length instruction format.

sented next. Suppose that A and B represent two multi-length numbers. Also, let their product be stored in C, that is $C = A \cdot B$. In this work, the method proposed by Comba [17], hereafter denoted as "Comba method", is used to compute the product of multiple-length numbers. For simplicity, the multiplicand and the multiplier are assumed to have the same length. Algorithm 1 presents the details of the Comba method. In the algorithm, $\{x, y, z\}$ denotes the concatenation of x, y, and z. In the Comba method, each block of results of the multiplication is computed from lower blocks to upper blocks, one by one, as shown in Fig. 4. The multiplication using the Comba method can be computed by the function multiplication and accumulation of one DSP slice [3]. The result is stored in data memory, block by block.

Algorithm 1: Comba method
Input: $A = (a_{d-1}, \ldots, a_0)$, $B = (b_{d-1}, \ldots, b_0)$
Output: Product $C = A \cdot B = (c_{2d-1}, \ldots, c_0)$
s, t, u: r-bit integers
1. $s \leftarrow 0, t \leftarrow 0, u \leftarrow 0$
2. **for** $i = 0$ **to** $d - 1$ **do**
3. **for** $j = 0$ **to** i **do**
4. $\{s, t, u\} \leftarrow \{s, t, u\} + a_j \times b_{i-j}$
5. **end for**
6. $c_i \leftarrow u$
7. $u \leftarrow t, t \leftarrow s, s \leftarrow 0$
8. **end for**
9. **for** $i = d$ **to** $2d - 2$ **do**
10. **for** $j = i - d + 1$ **to** $d - 1$ **do**
11. $\{s, t, u\} \leftarrow \{s, t, u\} + a_j \times b_{i-j}$
12. **end for**
13. $c_i \leftarrow u$
14. $u \leftarrow t, t \leftarrow s, s \leftarrow 0$
15. **end for**
16. $p_{2d-1} \leftarrow u$

3. Proposed Processor Architecture

This section presents the proposed processor architecture for handling multiple-length-arithmetic operations. As shown in Fig. 5, the designed processor consists of a Program Counter, Instruction memory, Data memory, DSP, flag registers, and control unit. The processor is based on the Harvard architecture where the instruction and data memories are separated [18]. The proposed processor executes the instructions one by one in a pipeline fashion. More precisely, a single DSP slice is responsible for computing all arithmetic operations. The proposed processor supports 38 instructions, not only multiple-length arithmetic operations, but also single-block arithmetic operations. Table 1 shows the list of the main instructions of the proposed processor and the respective amount of clock cycles to compute them. For the reader's benefit, the main elements of the proposed processor, such as instruction memory, data memory, and DSP are described next.

Control units: The proposed architecture consists of two control units: (i) the Main control unit; and (ii) the Data control unit. The Main control unit controls the behavior of the whole processor, including the Program counter, Data control unit, and Instruction memory. Data control unit, on the other hand, is responsible for handing each operation. More specifically, conditional and unconditional jump instructions such as JMP, JZ, JNZ, JC, and JNC are executed by the Main control unit of the proposed processor, where the jump instructions are executed by the Main control unit and the JMP control unit in our previous work [11]. By merging the Main control unit and JMP control unit, we simplified the circuit and decreased the clock cycles compared with the previous work. On the other hand, the proposed processor executes the arithmetic instructions using the Main control unit and the Data control unit, where an ALU control unit is also used in the previous work.

Instruction memory: Array of memory composed of block RAMs which are used to store program instructions. In the proposed architecture, the instruction memory is used to store multiple-length arithmetic instructions, each of which consists of 54-bits as depicted in Fig. 6.

Table 1 The list of main instructions of the proposed processor and their respective clock cycles

Mnemonic	Operation	The size of operands [bits]					
		64	128	256	512	1024	2048
ADD A, B, C	$A \leftarrow B + C$	17	25	41	70	130	250
SUB A, B, C	$A \leftarrow B - C$	14	21	38	69	129	249
MUL A, B, C	$A \leftarrow B \cdot C$	32	88	296	1031	3851	14891
MULV A, B, C	$A_0 \leftarrow B_0 \cdot C_0, A_1 \leftarrow B_1 \cdot C_1, \ldots, B_{d-1} \cdot C_{d-1}$	22	38	70	130	250	490
INC A	$A \leftarrow A + 1$	15	19	27	42	72	132
DEC A	$A \leftarrow A - 1$	14	18	26	41	71	131
CMP A, B	$A - B$	14	18	26	41	71	131
SHL A, B, x	$A \leftarrow B << x$	15	19	27	42	72	132
SHR A, B, x	$A \leftarrow B >> x$	15	19	27	42	72	132
MOV A, B	$A \leftarrow B$	10	14	22	37	67	127
MOVP A, B_x, B_y	$A \leftarrow B_x, \ldots, B_y$	$y - x + 7$					
JMP A	$PC \leftarrow A$	3					
JC A	$PC \leftarrow A$ if carry	3					
JNC A	$PC \leftarrow A$ if not carry	3					
JZ A	$PC \leftarrow A$ if zero	3					
JNZ A	$PC \leftarrow A$ if not zero	3					

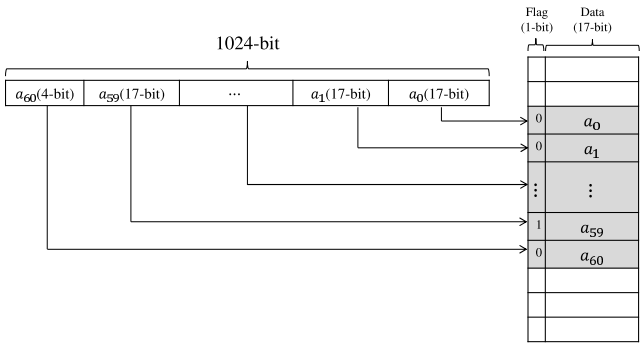


Fig. 7 Multiple-length numbers stored in data memory

Data memory: Array of memory composed of block RAMs where data, including multiple-length numbers, is stored. Figure 7 shows how a 1024-bits data is stored into the data memory. Every 17-bits block data, together with a 1-bit flag, represents a bit-block of 18-bits. The flag bit, stored in the most significant bit of each block, is used to find the second last 17-bits block data. If a 17-bits block is the second last block of a multiple-length number, the flag bit is set to 1, otherwise it is set to 0. As the multiple-length-arithmetic operations are performed in pipelined fashion, the flag bit is used to identify the end of a data block in advance, that is, before the last data block is read. In Fig. 7, a multiple-length number A consisting of 1024-bits is divided into $\lceil \frac{1024}{17} \rceil = 61$ blocks of 17-bits each, represented as a_0, a_1, …, a_{60}. In this example, the flag bit is set to 1 in block a_{59}. Using the above architecture, the designed processor allows for flexible-length arithmetic capable of computing multiple-length integer numbers, such as 64-bits, 128-bits, and even longer than 2048-bits without further modification.

DSP and flag registers: The DSP is an arithmetic and logical unit. Given two inputs from Data memory, arithmetic operations, such as addition and multiplication, can be performed with their result being stored to the Data memory. DSP corresponds to one DSP slice in the FPGA and it is controlled by Data control unit. The appropriate function of the DSP slice is selected according to the operation begin executed. These operations are performed in a pipeline fashion. Two flag registers, zero flag and carry flag, are used to hold the state of the operations. These registers are 1-bit registers and are used for conditional jump instructions, such as JNZ (jump if not zero) and JC (jump if carry). The result of the previous instruction determines the value of these registers. For instance, the zero flag holds 1 if the result of the previous instruction is zero (otherwise, it holds 0). Conversely, the carry flag holds 1 if the result of the previous instruction overflows or becomes negative (otherwise, it holds 0).

4. Coding and Porting Applications to the Proposed Architecture

This section presents the C-like language and the steps necessary to compile, translate and execute an application on an FPGA device using the approach proposed in this work. For this purpose, the well-known RSA algorithm, which uses multiple-length numbers has been selected [19]. By using the proposed architecture and tools presented in this section, the computation on multiple-length numbers can be performed effortlessly. Indeed, as in traditional high-level languages, the user centers on coding the algorithm in a C-like language and uses the provided tools to translate the code into a machine language. This process is straightforward and can be accomplished by users without solid knowledge on FPGA or circuit design. Next, a brief overview of the RSA algorithm is presented following by the necessary steps for porting and executing it on an FPGA device.

In the RSA, 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, respectively; (E, M) and (D, M) are, respectively, the encryption and decryption keys. Usually, the bit length in P, E, D, and M is 512 or longer. Also, the modular 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* [20] is used, which does not directly compute the modulo operation.

The proposed C-like language comprises a large set of instructions, including `if`, `if-else`, `while`, `do`, `goto`, etc. Also, it includes basic arithmetic operations such as addition (+), subtraction (−), multiplication (∗), negation (−), bit shifts (>>, <<), comparisons (==, ! =, >, >=, <, <=), where the operands can be configured as multiple-length numbers. Due to space limitation, in this paper we restrict our attention to those instructions necessary for coding the R-bit Montgomery modular multiplication $C = A \cdot B \cdot 2^{-R} \bmod M$. The C-like code for the Montgomery modular multiplication is shown in Code 1. In the C-like code, the multiple-length variables A, B, M, $invM$, C, $R1$, $R2$, and $R3$ are declared as `multi (d) var = value_var`. That is, the prefix `multi` denotes that this variable is a multiple-length number consisting of d 17-bits blocks `value_var`. The `value_var` denotes the initial value of the variable, which is configured by the programmer. When the initial value is not set, the variable is set to a default value of 0. In the C-like code of Montgomery modular multiplication, the multiple-length numbers A, B, M, and $invM$ are given as input and the result is stored to C. Although the code comprises of multiple-length-arithmetic operations, the coding task is straightforward. Note that the bit-length of the variables can hold 64-bits, 128-bits, even longer than 2048-bits without any modification to the code. For example, the data in register $R1$ is divided into d blocks of 17-bits each and these are stored in several block registers such as $R1_0$, $R1_1$, ... $R1_{d-1}$. For the case of other registers, $R2 = [R1_0, R1_{d-1}]$ denotes that $R1_0, R1_1, \ldots R1_{d-1}$ of $R1$ are copied to $R2$, block by block. The number of duplicated blocks can be specified by the programmer.

The next task is to translate the code written in the C-like language to a machine program that can be executed in the proposed architecture. A brief description of the compiler and assembler used to port the C-like code into a machine program is provided next.

Compiler: Converts a program written in C-like language into an assembler program using the lexical scanner *Flex* [6] and the context parser *Bison* [7]. *Flex* is a tool for generating a scanner to recognize lexical patterns in a text file while *Bison* is a tool for generating parsers to analyze input text based on rules defined by a context-free grammar. *Flex* and *Bison* are combined as a compiler for translating a program coded in C-like language into an assembly language program. The assembly code generated from Code 1 is shown in Code 2. Although the assembly code includes multiple-length-arithmetic operations, it consists of only 12 instructions.

Assembler: Converts an assembly language program into a machine program. This task is performed by a two-pass assembler. The assembly Code 2 is converted into the machine program shown in Code 3 using the two-pass assembly. Besides, the resulting machine program is written into the instruction memory of proposed processor into the FPGA device. Also, in the proposed architecture, the data is stored into the data memory and the instructions are stored into the instruction memory. Thus, the assembler also generates a list of data, which consists of the initialized values of the declared variables. This list is stored into the data memory, block by block, where the size of each block is 17-bits in length. All instructions of a machine program are read out from the instruction memory and executed, one by one, using the data stored into the data memory.

From the above, one can verify that the task of writing multiple-length arithmetic programs and porting them to an FPGA device can be easily accomplished. Indeed, in the proposed approach, the coding and debugging tasks are simplified, allowing non-experts to take the advantage of the

Code 1: C-like code for R-bit Montgomery modular multiplication $C = A \cdot B \cdot 2^{-R} \bmod M$

```
multi (d)       A = value_A          ; Definition of A
multi (d)       B = value_B          ; Definition of B
multi (d)       M = value_M          ; Definition of M
multi (d)       invM = value_invM    ; Definition of invM
multi (d)       C                    ; Definition of C
multi (2d)      R1                   ; Definition of R1
multi (d)       R2                   ; Definition of R2
multi (2d + 1)  R3                   ; Definition of R3

R1 = A * B                ; A · B is stored to R1
R2 = [R1_0, R1_{d-1}]     ; R1_0, ..., R1_{d-1} is copied to R2
R3 = R2 * invM            ; R2 · (−M^{-1}) is stored to R3
R2 = [R3_0, R3_{d-1}]     ; R3_0, ..., R3_{d-1} is copied to R2
R3 = R2 * M               ; R2 · M is stored to R3
R3 = R1 + R3              ; R1 + R3 is stored to R3
R2 = [R3_d, R3_{2d-1}]    ; R3_d, ..., R3_{2d-1} is copied to R2
if (R2 >= M) {              R2 is compared with M
    R2 = R2 − M           ; if (R2 ≥ M), R2 − M is stored to R2
}
C = R2                    ; R2 is copied to C
halt                      ; program is terminated
```

Code 2: Assembly code for R-bit Montgomery modular multiplication $C = A \cdot B \cdot 2^{-R} \bmod M$

```
A, d :        value_A          ; Definition of A
B, d :        value_B          ; Definition of B
M, d :        value_M          ; Definition of M
invM, d :     value_invM       ; Definition of invM
C, d :        0                ; Definition of C
R1, 2d :      0                ; Definition of R1
R2, d :       0                ; Definition of R2
R3, 2d + 1 :  0                ; Definition of R3

    MUL   R1, A, B          ; A · B is stored to R1
    MOVP  R2, R1_0, R1_{d-1}  ; R1_0, ..., R1_{d-1} is copied to R2
    MUL   R3, R2, invM      ; R2 · (−M^{-1}) is stored to R3
    MOVP  R2, R3_0, R3_{d-1}  ; R3_0, ..., R3_{d-1} is copied to R2
    MUL   R3, R2, M         ; R2 · M is stored to R3
    ADD   R3, R1, R3        ; R1 + R3 is stored to R3
    MOVP  R2, R3_d, R3_{2d-1} ; R3_d, ..., R3_{2d-1} is copied to R2
    CMP   R2, M             ; R2 is compared with M
    JC    _001F             ; if carry (R2 < M), jump to label _001F
    SUB   R2, R2, M         ; R2 − M is stored to R2
_001F:                      ; the label
    MOV   C, R2             ; R2 is copied to C
    HALT                    ; program is terminated
```

Table 2 Synthesized results and performance evaluation of the direct FPGA implementation and the proposed processor for a 2048-bits RSA encryption

	Direct implementation [9]	Single Core	Multi Cores (306 cores)
Device	XC6VLX240T-1	XC6VLX240T-3	
CLBs(Slices)	180	167	51535
Block RAMs	1	2	383
DSP48E1 slices	1	1	306
Clock frequency[MHz]	447.02	310.07	240.20
Multiplication	School method	Comba method	
Supported instructions	1 (RSA only)	38	
RSA implementation	Hardware (Verilog HDL)	Software (C-like language and assembly language)	
Computing time[ms]	277.26	613.71	792.23
Throughput [1/s]	3.61	1.63	386.25

Code 3: The Translated machine code for R-bit Montgomery modular multiplication $C = A \cdot B \cdot 2^{-R} \bmod M$

```
* * * VARIABLE LIST * * *
000:    A                   ; variable A
03D:    B                   ; variable B
07A:    M                   ; variable M
0B7:    invM                ; variable invM
0F4:    C                   ; variable C
131:    R1                  ; variable R1
1AB:    R2                  ; variable R2
1E8:    R3                  ; variable R3

* * * LABEL LIST * * *
_001F   00A

* * * MACHINE PROGRAM * * *
000: 0D003D00000131         MUL R1, A, B
001: 1B013101AB016D         MOVP R2, R1_0, R1_{d-1}
002: 0D00B701AB01E8         MUL R3, R2, invM
003: 1B01E801AB0224         MOVP R2, R3_0, R3_{d-1}
004: 0D007A01AB01E8         MUL R3, R2, M
005: 0501E8013101E8         ADD R3, R1, R3
006: 1B022501AB0261         MOVP R2, R3_d, R3_{2d-1}
007: 14007A01AB0000         CMP R2, M
008: 23000A00000000         JC _001F
009: 08007A01AB01AB         SUB R2, R2, M
                        _001F:
00A: 1A01AB00F40000         MOV C, R2
00B: 00000000000000         HALT
```

FPGAs to perform arithmetic operations on multiple-length numbers. Once the program is written in the C-like language, the compiler and assembler are used to translate the code into machine language. The assembler also assists in the task of transferring the machine program to be executed by the proposed processor in an FPGA.

The next section evaluates the performance of multiple-length arithmetic programs executed by the proposed processor against a direct FPGA implementation.

5. Evaluation of a Single-Core Processor System

The proposed flexible-length-arithmetic processor architecture is used to implement the modular multiplication algorithm presented in the previous section. The performance evaluation is carried on a Xilinx Virtex-6 XC6VLX240T-3 [2], programmed by software and synthesis with Xilinx ISE Foundation 14.7. To assess the performance of the proposed processor implementation, a direct hardware implementation is used. For this purpose, the hardware implementation proposed by Bo *et al.* [9], evaluated on Xilinx Virtex-6 FPGA XC6VLX240T-1 and programmed using Verilog HDL, is used. The circuit for the direct hardware implementation for computing the RSA encryption also employs the FDFM approach. The implementation proposed in [9] is reported to attain high-performance. On the other hand, it uses a specialized circuit, designed and implemented by an expert. The developing and debugging tasks as well as eventual modifications to this circuit are not easily achieved by an apprentice and, in certain cases, it is a difficult task even for an experienced user. The architecture proposed in this work, on the other hand, can be employed to a variety of applications requiring flexible-length-arithmetic operations.

Table 2 presents the synthesized results for a 2048-bits RSA computation using the direct hardware implementation and the proposed architecture. We note that the synthesized results, that do not include the result of the place and route, were obtained by the built-in synthesizer of Xilinx ISE Foundation 14.7. As can be observed, the proposed processor uses fewer CLBs slices than the direct implementation. As the proposed architecture uses only one DSP slice and two block RAMs, it does not require additional circuits consisting of CLBs, such as barrel shifter. Recall that the proposed processor supports 38 instructions in a compact architecture. In contract, the direct implementation supports a single instruction. Since the proposed processor uses very few resources of the FPGA, it can be applied to scenarios in which FPGA resources are scarce. As shown in Fig. 3, the proposed processor can be adapted to better use the available resources. The synthesized results show that the maximum clock frequency of the proposed processor is 310.07MHz. We use the built-in ISE Simulator of Xilinx ISE Foundation 14.7 to evaluate the computing time of the proposed processor. The simulated results show that the proposed processor takes 190293088 clock cycles to compute a 2048-bits RSA encryption. Hence, the proposed processor computes a 2048-bit RSA encryption in expected 613.71ms. The results show that the proposed processor takes only 2.2 times longer than the time required by the direct implementation to compute a 2048-bits RSA encryption.

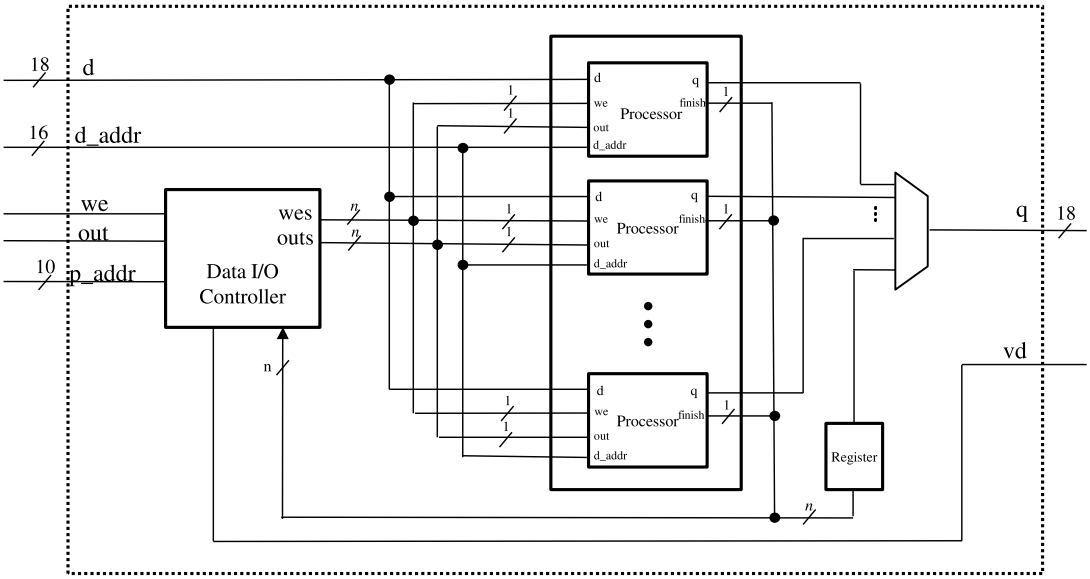


Fig. 8 Multicore processor system using proposed flexible-length-arithmetic processors

Table 3 Synthesized results and performance evaluation of implementation of *rsa_512* and the proposed processor for a 512-bits RSA encryption

	rsa_512 [21]	Proposed processor
Device	XC6VLX240T-1	XC6VLX240T-1
CLBs(Slices)	7319	167
Block RAMs	5	2
DSP48E1 slices	50	1
Clock frequency[MHz]	125.14	310.07
Clock cycles	195335	3780563
Computing time[ms]	1.56	12.19

We also compare the proposed processor with an existing implementation *rsa_512* [21] that computes 512-bit RSA encryption available in OpenCores [22]. We implemented the *rsa_512* on the FPGA Xilinx FPGA XC6VLX240T-1 that is the same FPGA used in our implementation. The synthesis results of the proposed processor and *rsa_512* are shown in Table 3. The execution time for a 512-bit RSA encryption of *rsa_512* is 1.56ms, that is 7.8 times faster than the proposed processor. However, *rsa_512* uses much more FPGA resources than the proposed processor. For example, as shown in Table 3, the implementation of *rsa_512* uses 7319 CLBs (Slices). It is 48.3 times more than that of our proposed processor. The implementation of *rsa_512* uses 50 DSP slices to perform *Montgomery modular multiplication*, where the proposed processor uses only one DSP slice to perform *Montgomery modular multiplication* taking more clock cycles than *rsa_512*. Since the number of DSP slices in the FPGA utilized in this paper is 768, at most 15 processors of *rsa_512* can be arranged in the FPGA. On the other hand, we can implement 306 proposed processors in the FPGA as shown in the next section, that is approximately 6.1 times more processors. Therefore, considering the balance between performance and size, both of them are almost the same. However, our processor can be used not only for 512-bit RSA encryption, but also for larger RSA encryption and any other computations without modification of the hardware. Therefore, our processor is more versatile than *rsa_512*.

6. Evaluation of a Multicore Processor System

This section presents and evaluates a multicore-processor system. As shown in Fig. 3 (b), the multicore system comprises of several processor cores of the FDFM approach that work in parallel. Figure 8 illustrates the architecture of the multicore-processor system. The "Data I/O controller", shown in the figure, allows to verify the computation status from outside of the multicore-processor. That is, the controller has access to data memory in each processor and indicates whether the computation has completed. The implemented multicore-processor uses 306 processor cores on a Virtex-6 family FPGA XC6VLX240T. The implementation uses 51735 CLBs, 383 block RAMs, and 306 DSP slices. The synthesized results obtained by the built-in synthesizer reported that the multicore implementation runs at 240.20MHz. As can be observed in Table 2, the clock frequency is approximately 30% lower than that of single processor due to increased circuit delay. Since this implementation consists of 306 processor cores and considering the total performance of the multicore-processor, the effect of the performance derived from the decrease of clock frequency is not large. Recall that single processor computes a 2048-bits RSA encryption in 190293088 clock cycles, thus, each processor of multicore processor system computes a 2048-bits RSA encryption in expected 190293088/240.20MHz = 792.23ms. Calculated simply, the computing time for a 2048-bits RSA encryption of the multicore-processor system is 792.23/306 = 2.59ms. In other words, the multicore-processor system presented in this section computes a 2048-bits RSA encryption at rate

of 386 times per second.

To compare the performance of our processor with the software implementation, we implemented a software that computes 2048-bit RSA encryption in Intel Core i7-4790 (3.6GHz) CPU using GNU Multi-Precision (GMP) library [23]. According to the performance evaluation, the execution time for a 2048-bit RSA encryption on the CPU is 5.66ms. On the other hand, the computing time for a 2048-bits RSA encryption of the multicore-processor system is 2.59ms. Thus, the total performance of our approach is 2.19 times higher than that of the implementation in CPU using GMP library.

7. Conclusions

This work presented an intermediate approach of software and hardware using FPGAs to support arithmetic operations on multiple-length numbers. More specifically, a flexible-length-arithmetic processor, based on the FDFM approach, is proposed. In the proposed approach, an application requiring multiple-length arithmetic operations is written in a C-like language and translated into a machine program. The machine program is then transferred and executed on the proposed processor on the FPGA. Hence, the coding and debugging tasks are simplified, allowing non-experts to take the advantage of FPGAs devices to perform arithmetic operations on multiple-length numbers. Experimental results showed that a single processor system, comprising of a single FDFM processor core, takes at most 2.2 times longer than the time required by a tailored-designed FPGA implementation to compute a 2048-bits RSA encryption. This paper also presents a multicore processor system that uses 306 FDFM processor cores. The multicore processor system computes a 2048-bits RSA encryption at a rate of 386 times per second. These results confirm that the proposed flexible-length-arithmetic processor attains a fair performance while furnishing the designer with the necessary tools for writing and porting applications that require multiple-length-arithmetic operations to be executed on FPGAs devices. That is, the proposed processor simplifies the coding, debugging and porting tasks without sacrificing performance.

References

[1] T. Kawamoto, Y. Ito, and K. Nakano, "A flexible-length-arithmetic processor based on FDFM approach in FPGAs," Proc. of International Symposium on Computing and Networking, pp.364–370, 2015.
[2] Xilinx Inc., Virtex-6 FPGA Configuration User Guide (v3.8), 2014.
[3] Xilinx Inc., Virtex-6 FPGA DSP48E1 Slice User Guide (v1.3), 2011.
[4] Xilinx Inc., Virtex-6 FPGA Memory Resources (v1.8), 2014.
[5] G. Novak, "Artificial intelligence," in Academic Press Dictionary of Science and Technology, p.160, Academic Press, San Diego, CA, 1992.
[6] G.T. Nicol, Flex: The Lexical Scanner Generator, Free Software Foundation, 1993.
[7] C. Donnelly and R. Stallman, Bison: The YACC-compatible Parser Generator, Free Software Foundation, 1995.
[8] Y. Ago, A. Inoue, K. Nakano, and Y. Ito, "The parallel FDFM processor core approach for neural networks," Proc. of International Conference on Networking and Computing, pp.113–119, 2011.
[9] 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, 2011.
[10] Y. Ito, K. Nakano, and S. Bo, "The parallel FDFM processor core approach for CRT-based RSA decryption," Int. J. Networking and Computing, vol.2, pp.56–78, 2012.
[11] M.N.I. Mondal, K. Sai, K. Nakano, and Y. Ito, "A flexible-length-arithmetic processor using embedded DSP slices and block RAMs in FPGAs," Proc. of International Symposium on Computing and Networking, pp.75–84, 2013.
[12] K. Sai, A design of a flexible-length-arithmetic system using embedded DSP slices in FPGAs, Master's thesis, Hiroshima University, 2012 (in Japanese).
[13] A. Kalathungal, An Arbitrary Precision Integer Arithmetic Library for FPGA s, Ph.D. Thesis, University of Cincinnati, 2013.
[14] O.A. Pfänder, R. Nopper, H.-J. Pfleiderer, S. Zhou, and A. Bermak, "Configurable blocks for multi-precision multiplication," IEEE International Symposium on Electronic Design, Test and Applications, pp.478–481, 2008.
[15] C.E. LaForest, High-Speed Soft-Processor Architecture for FPGA Overlays, Ph.D. Thesis, University of Toronto, 2015.
[16] S. McGettrick, K. Patel, and C. Bleakley, "High performance programmable FPGA overlay for digital signal processing," in Reconfigurable Computing: Architectures, Tools and Applications, vol.6578, pp.375–384, Springer, 2011.
[17] P.G. Comba, "Exponentiation cryptosystems on the IBM PC," IBM Systems Journal, vol.29, no.4, pp.526–538, 1990.
[18] J.L. Hennessy and D.A. Patterson, Computer Architecture, Fourth Edition: A Quantitative Approach, Morgan Kaufmann Publishers, 2006.
[19] R.L. Rivest, A. Shamir, and L.M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," Commun. ACM, vol.21, no.2, pp.120–126, 1978.
[20] P.L. Montgomery, "Modular multiplication without trial division," Math. of Comput., vol.44, no.170, pp.519–521, 1985.
[21] E.C. Villar and J.C. Villar, "High performance RSA 512 bit IPCore." http://opencores.org/project,rsa_512, 2012.
[22] "OpenCores." http://opencores.org/
[23] T. Granlund, "GNU MP: The GNU Multiple Precision arithmetic library." http://gmplib.org/

Tatsuya Kawamoto received the B.E. degree from Hiroshima University, Japan in 2014. He is currently working towards a M.E. degree at Hiroshima University.

Xin Zhou received the B.E. degree from Jiangxi Science & Technology Normal University (China), in 2006, and the M.E. degree from Hiroshima University (Japan), in 2013. He is currently working towards a D.E. degree at the Department of Information Engineering, Hiroshima University.

Jacir Luiz Bordim received B.Sc. and M.Sc. degrees in Computer Science in 1994 and 2000, respectively. Received the Ph.D. degree in Information Science from Japan Advanced Institute Of Science And Technology in 2003, with honors. He worked as a researcher at ATR-Japan from 2003 to 2005. Since 2005 he is an Associate Professor with the Department of Computer Science at University of Brasilia. Dr. Bordim has published and served in many international conferences and journal. His interest includes mobile computing, collaborative computing, trust computing, distributed systems, opportunistic spectrum allocation, MAC, routing protocols and reconfigurable computing.

Yasuaki Ito received the B.E. degree from Nagoya Institute of Technology (Japan) in 2001, the M.S. degree from Japan Advanced Institute of Science and Technology in 2003, and the D.E. degree from Hiroshima University (Japan), in 2010. He was a Research Associate in 2004–2007 and an Assistant Professor in 2007–2013 at Hiroshima University. Since 2013, Dr. Ito has been with the School of Engineering, at Hiroshima University, where he is working as an Associate Professor. His research interests include reconfigurable architectures, GPU computing, parallel computing, computational complexity and image processing.

Koji Nakano received the B.E., M.E. and Ph.D. degrees from Department of Computer Science, Osaka University, Japan in 1987, 1989, and 1992 respectively. In 1992-1995, he was a Research Scientist at Advanced Research Laboratory. Hitachi Ltd. In 1995, he joined Department of Electrical and Computer Engineering, Nagoya Institute of Technology. In 2001, he moved to School of Information Science, Japan Advanced Institute of Science and Technology, where he was an associate professor. He has been a full professor at School of Engineering, Hiroshima University from 2003. He has published extensively in journals, conference proceedings, and book chapters. He served on the editorial board of journals including IEEE Transactions on Parallel and Distributed Systems, IEICE Transactions on Information and Systems, and International Journal of Foundations on Computer Science. He has also guest-edited several special issues including IEEE TPDS Special issue on Wireless Networks and Mobile Computing, IJFCS special issue on Graph Algorithms and Applications, and IEICE Transactions special issue on Foundations of Computer Science. He has organized conferences and workshops including International Conference on Networking and Computing, International Conference on Parallel and Distributed Computing, Applications and Technologies, IPDPS Workshop on Advances in Parallel and Distributed Computational Models, and ICPP Workshop on Wireless Networks and Mobile Computing. His research interests includes image processing, hardware algorithms, GPU-based computing, FPGA-based reconfigurable computing, parallel computing, algorithms and architectures.