A Flexible-length-arithmetic Processor based on FDFM Approach in FPGAs

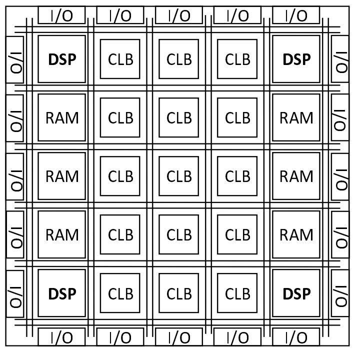
Tatsuya Kawamoto, Yasuaki Ito, Koji Nakano
Department of Information Engineering,
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 Japan
Email: {kawamoto, yasuaki, nakano}@cs.hiroshima-u.ac.jp

***Abstract*—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 Xilinx Virtex-6 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 assembly program. The resulting processor uses only one DSP48E1 slice and two block RAMs, and RSA encryption software on it runs in 613.71ms. 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 programs for the proposed processor can be written by software, the development and the debugging are easy. We have also succeeded in implementing 306 processor cores in one Xilinx Virtex-6 FPGA which work in parallel to improve the throughput greatly.**

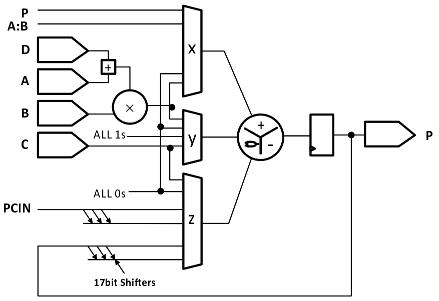
***Keywords*-Multiple-length-arithmetic, FPGA, DSP slices, Block RAMs, RSA, Montgomery modular multiplication**

I. Introduction

An FPGA (Field Programmable Gate Array) 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], hence, it is widely used recently. In particular, since FPGAs can implement hundreds of circuits that work in parallel, they are used 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, embedded DSP slices which are integrated into an FPGA makes a higher performance and a broader application [2]. Figure 1(a) roughly illustrates the internal configuration. 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 1(b), the DSP48E1 slice has a two input multiplier followed by multiplexers and a three-input adder/subtractor/accumulator. The DSP48E1



(a) Internal configuration of FPGA



(b) Architecture of DSP48E1

Figure 1. FPGA (Field Programmable Gate Array)

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 [3]. The block RAM 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$\times$18-bit dual-port RAM.

Applications require arithmetic operations on integer numbers which exceed the range of processing by a CPU directly is called *multiple-length 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 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 HDL 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 almost similar to the high level language, written in English which makes the instructions as well as algorithms easy to read, modify and debug by the non-expert or beginners.

The main contribution of this paper is to present an intermediate approach of software and hardware using FPGAs 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 (Few DSP slices and Few Memory blocks) 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:

(i) 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.

(ii) We present an intermediate approach of software and hardware to write the algorithm which makes the debugging and further development easy.

(iii) Our designed processor provides flexibility so that it can be used for computing of integer numbers with flexibly many bits such as 64-bit, 128-bit, even longer than 2048-bit without further modification.

Let us explain briefly the FDFM approach using a simple example. Figure 2(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

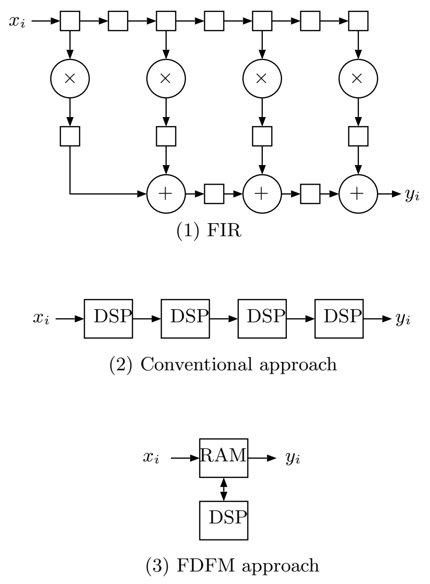


Figure 2. FDFM approach over conventional one for FIR

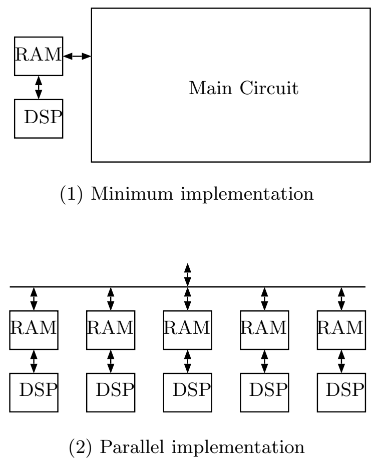


Figure 3. Advantages of FDFM approach

illustrated in Figure 2(2) [2]. In this conventional approach, the number of DSP slices 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 2(3) shows the FDFM approach using one DSP slice and one block RAM to implement the same mentioned above. The coefficients $a_0, a_1, \ldots$ are stored in the block RAM.

For readers, we also refer to the papers [4], [5], [6], [7] in which they can find details about FDFM approach and conventional approach. Let us describe the two important advantages of the FDFM approach, as follows. 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 3(1). Also, if enough hardware resources are available, we can implement multiple FDFM processor cores that work in parallel (Figure 3(2)). The resulting hardware implementation has maximum throughput by parallel computation.

Because of the above mentioned advantages of the FDFM approach, we design a flexible-length-arithmetic processor based on FDFM approach. 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 them. We are mainly thinking the following scenarios for designing a flexible-length-arithmetic processor:

1) Our aim is mainly to emphasize 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 HDL, 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) slice 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.

In our previous work, we proposed a preliminary version of a processor of this work [7]. This processor is also based on the FDFM approach. The processor consists of one DSP slice and four block RAMs. The main differences from the previous one are as follows. By improving the processor architecture, all the arithmetic operations are performed by only one DSP slices without additional circuits and memories. As a result, although the number of the supported instructions is increased from 17 to 38, the number of block RAMs is reduced to two while the size of the circuit is almost the same.

We have also implemented the multicore-processor system which contains 306 processor cores on the Virtex-6 FPGA. The implementation uses 51735 CLBs, 383 block RAMs, and 306 DSP slices. The timing analysis shows that our implementation runs in 240.20MHz. The clock frequency is 1.29 times lower than that of single processor since the circuit delay is increased. However, since the implementation consists 308 processor cores, considering the total performance of the multicore-processor, the effect of the performance derived from the decrease of clock frequency is not large.

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. Section VI presents multicore-processor system with 306 processor cores and its performance. Finally Section VII concludes this work.

II. Multiple-Length-Arithmetic Operation

The main purpose of this section is to describe multiple-length-arithmetic operations. In the following, we will represent multiple-length numbers as arrays of r-bit blocks. In general, $r = 32$ or 64 for conventional CPUs. On the other hand, in our case, $r = 17$ since the bit-length manipulated in the DSP slice is 17 bits. Let R denote the bit-length of numbers and d be the number of r-bit blocks. Therefore, $d = \lceil \frac{R}{r} \rceil$. For example, a 1024-bit integer consists of 61 blocks.

Now, let us give an example of a multiplication of two multiple-length data. Due to the page limitation, other arithmetic operations supported in our processor such as addition, subtraction, comparison, etc. are omitted, but the readers can understand how they are computed easily. Suppose A and B represent two multi-length numbers. We are multiplying A by B and the result is stored in C, that is $C = A \cdot B$. To compute this multiplication, *School method* is often used. The algorithm of School method is shown in Algorithm 1. For simplicity, in the algorithm, the sizes of the multiplicand and the multiplier are the same and $\{x, y\}$ denotes a concatenation of x and y. School method multiplies the multiplicand by each block of the multiplier and then adds up all the properly shifted results illustrated in Figure 4. In School method, it is necessary to store temporary data that are partial products during the computation. Therefore, our preliminary processor which adopts School method, uses additional storage consisting of block RAMs [7].

```
Algorithm 1: School method
Input: $A = (a_{d-1}, \ldots, a_0)$, $B = (b_{d-1}, \ldots, b_0)$
Output: Product $C = A \cdot B = (c_{2d-1}, \ldots, c_0)$
$t$: $r$-bit integer
1. for $j = 0$ to $d-1$ do
2.   $t \leftarrow 0, c_0 \leftarrow 0$
3.   for $i = 0$ to $d-1$ do
4.     $\{t, c_{i+j}\} \leftarrow c_{i+j} + a_i \cdot b_j + t$
5.   end for
6.   $c_{d+j} \leftarrow t$
7. end for
```

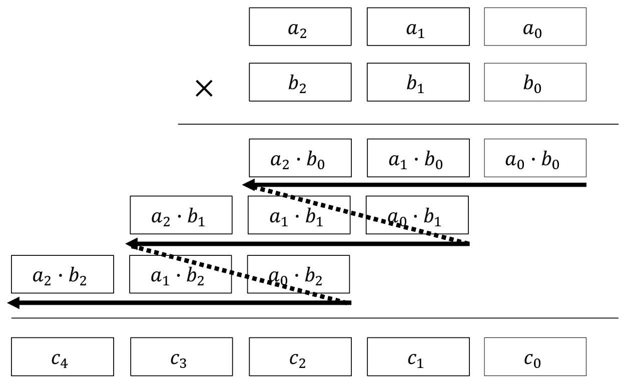


Figure 4. School method for multiple-length numbers $C = A \cdot B$

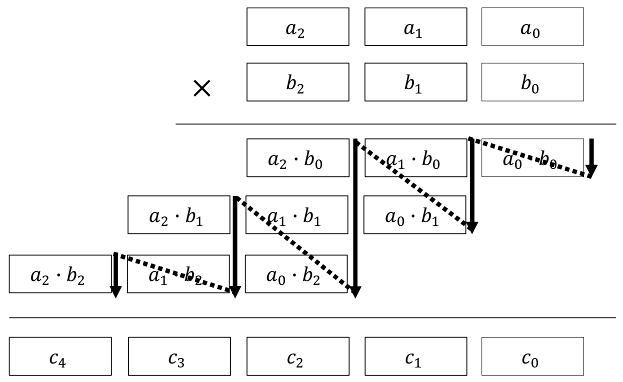


Figure 5. Comba method for multiple-length numbers $C = A \cdot B$

To avoid storing the temporary data, in the proposed processor, we use *Comba method* [8] for the multiple-length multiplication. The algorithm of Comba method is shown in Algorithm 2. According to the algorithm, the readers may think that it is more complicated than School method. However, the difference is only the order of multiplications of blocks and the number of multiplication of blocks is the same as illustrated in Figure 5. In Comba method, each block of results of the multiplication is computed from lower blocks to upper blocks one by one. The multiplication using Comba method can be computed by the function multiplication and accumulation of DSP slice [2] without any additional adders or memories.

```
Algorithm 2: 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. $p_{2d-1} \leftarrow u$
```

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, Data memory, DSP, flag registers, and control units as illustrated in Figure 6. Our processor is based on the Harvard architecture of which instruction and data memories are separated [9]. The proposed processor performs instructions one by one and each instruction is basically executed in pipeline fashion. Especially, all the arithmetic operations are computed only by one using DSP slice. Our processor supports 38 instructions, not only multiple-length arithmetic operations, but also single-block arithmetic operations. Table I shows the list of main instructions of the processor and clock cycles to perform them. For the reader's benefit, main elements of the processor, such as instruction memory, data memory, and DSP are briefly described, as follows.

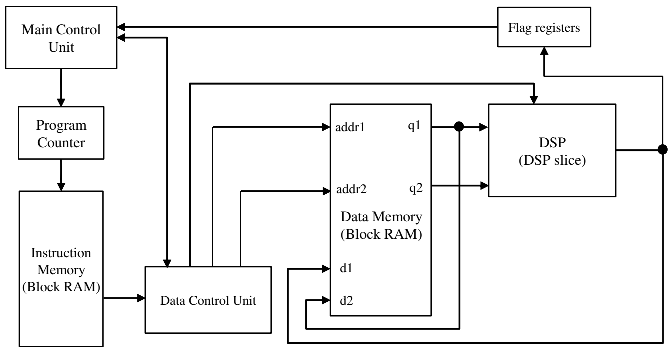


Figure 6. Our proposed processor architecture

Control units: We have two control units, Main control unit and Data control unit. Main control unit mainly controls the behavior of the whole processor such as Program counter, Data control unit, and Instruction memory. On the other hand, Data control unit is used to perform each operation.
Instruction memory: Instruction memory is an array of memory in which instructions of a program can be stored and it is composed of block RAMs. In our architecture, this memory is used to store multiple-length arithmetic instructions, each of 54 bits, and the format is shown in Figure 7.

Table I
THE LIST OF MAIN INSTRUCTIONS OF THE PROPOSED PROCESSOR AND THEIR 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	$A_0 \leftarrow B_0 \cdot C_0, A_1 \leftarrow B_1 \cdot C_1, ..., B_{d-1} \cdot C_{d-1}$	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					

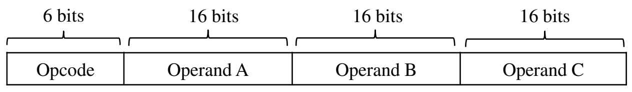


Figure 7. Data format of instructions

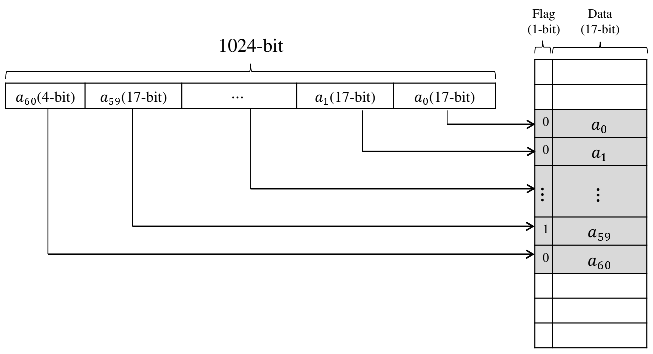


Figure 8. Multiple-length numbers stored in data memory

Data memory: Data memory is also an array of memory where various data including multiple-length numbers can be stored and it is composed of block RAMs. Figure 8 shows how to store a 1024-bit data to the memory. Every 17-bit 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-bit block data. If a 17-bit block is the second last block of a multiple-length number, the flag bit is set to 1, otherwise 0. The reason that the flag bit is used not for the last block but for the penultimate block is to know the end of data block in advance before the last data block is read since the multiple-length-arithmetic operations are performed in pipelined fashion. In this figure, multiple-length data A of 1024 bits is divided into $\lceil \frac{1024}{17} \rceil = 61$ numbers of 17 bits block such as a_0, a_1, …, a_{60}. The flag bit of a_{59}, in this case, is set to 1. Using the above architecture, 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.

DSP and flag registers: DSP is an arithmetic and logical unit which can perform arithmetic and logical operations for given inputs. Given two inputs from Data memory, arithmetic operations such as addition, multiplication etc. can be performed and the result of the operation is stored to Data memory. DSP corresponds to one DSP slice in the FPGA and it is controlled by Data control unit. According to the operation, by selecting the function of the DSP slice, the operation is performed. The operation by DSP is performed in pipelined fashion. Also, we have two flag registers, zero flag and carry flag to hold the state of operations. These flag registers are 1-bit registers and usually used for conditional jump instructions such as JNZ (jump if not zero) and JC (jump if carry). The value of them is determined by the result of the previous instruction. Zero flag holds 1 if the result of the previous instruction is zero. Otherwise it holds 0. On the other hand, carry flag holds 1 if the result of the previous instruction is in the range of numbers, that is the result becomes overflowed or negative. Otherwise it holds 0.

IV. AN APPLICATION OF RSA CRYPTOGRAPHY

In this section, we introduce an application of RSA cryptography [10] using the proposed processor. In 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, 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 modular exponentiation is repeatedly computed for fixed E, D, and M, and various

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

```
01: MUL   R1, A, B                ; A · B is stored to R1
02: MOVP  R2, R1_0, R_{d-1}       ; R1_0, ..., R1_{d-1} is copied to R2
03: MUL   R3, R2, -M^{-1}         ; R2 · (-M^{-1}) is stored to R3
04: MOVP  R2, R3_{,0}, R3_{d-1}   ; R3_0, ..., R3_{d-1} is copied to R2
05: MUL   R3, R2, M               ; R2 · M is stored to R3
06: ADD   R3, R1, R3              ; R1 + R3 is stored to R3
07: MOVP  R2, R3_d, R3_{2d-1}     ; R3_d, ..., R3_{2d-1} is copied to R2
08: CMP   R2, M                   ; R2 is compared with M
09: JC    0B                      ; if carry (R2 < M), jump to address 0B
0A: SUB   R2, R2, M               ; R2 - M is stored to R2
0B: MOV   C, R2                   ; R2 is copied to C
```

P and C. Since modulo operation is very costly in terms of the computing time and hardware resources, *Montgomery modular multiplication* [11] is used, which does not directly compute modulo operation.

We implement the RSA cryptography using our proposed architecture and it is programmed by the assembly language. The assembly code consists of 74 instructions. Due to the page limitation, we only show an assembly code for R-bit Montgomery modular multiplication $C = A \cdot B \cdot 2^{-R} \bmod M$ in Code 1. In the assembly code, the multiple-length numbers A and B, and modulus M are input and the result is stored to C. Although the code includes multiple-length-arithmetic operations, it consists of only 11 instructions. Also, each size of bits can be flexibly many bits such as 64 bits, 128 bits, even longer than 2048 bits without any modification of the codes except the constant value of the size of bits. Note that, data in register $R1$ is divided into several blocks of 17 bits each and these are stored in several block registers such as $R1_0$, $R1_1, \ldots$ (lower block to higher block). For the case of other registers, we can explain in similar way.

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-3 [1], programmed by software and synthesis with Xilinx ISE Foundation 14.7. To evaluate the proposed processor, we compare two FPGA implementations. One is a direct hardware implementation [5], which is evaluated on Xilinx Virtex-6 FPGA XC6VLX240T-1, programmed by Verilog HDL. The other is a preliminary version of processor of this work [7]. The circuit of direct hardware implementation can compute RSA encryption and it is based on FDFM approach. The implementation shows high-performance, but it is a specialized design by an expert so that it is difficult to develop, debug, and change circuits by a non-expert or sometimes even by an expert.

Table II shows comparisons about synthesized results and 2048-bit RSA computation between the direct hardware implementation, the processor of the previous work, and this work. As regards CLBs, the proposed processor uses the least of them since the proposed one uses only one DSP slice and does not use additional circuits consisting of CLBs such as an adder and barrel shifter that are used in the others. Additionally, the proposed processor supports 38 instructions, the circuit of the processor can be more compact than previous one that supports 17 instructions. Also, the number of block RAMs in the proposed processor is reduced from four to two. Therefore, the advantages of FDFM approach shown in Figure 3 can be obtained further more. On the other hand, the computing time of 2048-bit RSA encryption for the proposed processor is 2.2 times longer than that for the direct implementation. Considering that the RSA encryption in the proposed processor is programmed by software, however, the proposed processor can be utilized for various applications.

VI. Multicore-processor system

According to the above, we have designed a multicore-processor system that contains many processor cores of the FDFM approach that work in parallel as shown in Figure 3 (2). Figure 9 illustrates the architecture of the multicore-processor system. The data I/O controller in the figure is a controller that the data memory in each processor is accessed and the status whether the computation is finished or not is known from outside of the multicore-processor. We have implemented the multicore-processor system which contains 306 processor cores on a Virtex-6 family FPGA XC6VLX240T. The implementation uses 51735 CLBs, 383 block RAMs, and 306 DSP slices. The timing analysis reported that our implementation runs in 240.20MHz. The clock frequency is 1.29 times lower than that of single processor shown in Table II since the circuit delay is increased. However, since the implementation consists of 306 processor cores, considering the total performance of the multicore-processor, the effect of the performance derived from the decrease of clock frequency is not large. Calculated simply, this multicore-processor system can compute 2048-bit RSA encryption 388 times in one second, though the single core processor can compute it 1.6 times in one second.

VII. Conclusions

In this paper, we have presented an intermediate approach of software and hardware using one DSP slices and two 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. We have also succeeded in implementing 308 processor cores in one Xilinx Virtex-6 FPGA which work in parallel to improve the throughput greatly.

Acknowledgment

The authors would like to thank to Dr. Md. Nazrul Islam Mondal and Mr. Kohan Sai who have developed a preliminary version of a processor of this work [7].

Table II
SYNTHESIZED RESULTS AND PERFORMANCE EVALUATION OF THE DIRECT-HARDWARE PROCESSOR, OUR PREVIOUS PROCESSOR, AND OUR PROPOSED PROCESSOR FOR 2048-BIT RSA ENCRYPTION

	Direct implementation [5]	Previous work [7]	This work
Device	XC6VLX240T-1	XC6VLX240T-3	XC6VLX240T-3
CLBs(Slices)	180	170	167
Block RAMs	1	4	2
DSP48E1 slices	1	1	1
Clock frequency[MHz]	447.02	299.89	310.07
Multiplication	School method	School method	Comba method
Supported instructions	1 (RSA only)	17	38
RSA implementation	Hardware (Verilog HDL)	Software (Assembly language)	Software (Assembly language)
Computing time[ms]	277.26	635.65	613.71

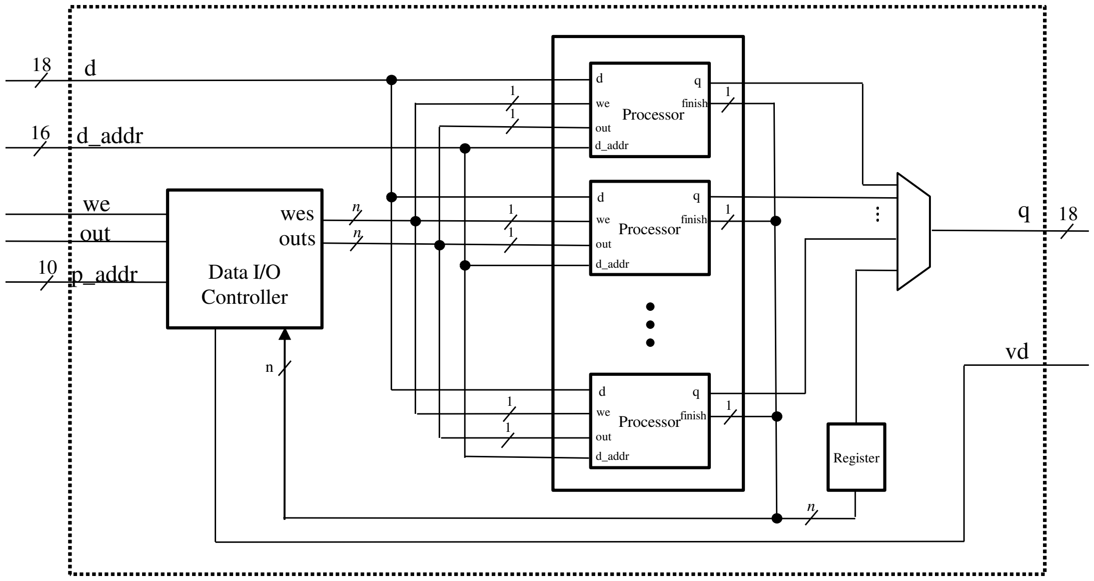


Figure 9. Multicore processor system using proposed flexible-length-arithmetic processors

REFERENCES

[1] Xilinx Inc., *Virtex-6 FPGA Configuration User Guide (v3.8)*, 2014.

[2] Xilinx Inc., *Virtex-6 FPGA DSP48E1 Slice User Guide (v1.3)*, 2011.

[3] Xilinx Inc., *Virtex-6 FPGA Memory Resources (v1.8)*, 2014.

[4] 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*, pp. 113–119, 2011.

[5] 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.

[6] 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.

[7] 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," in *Proc. of International Symposium on Computing and Networking*, pp. 75–84, 2013.

[8] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, 1990.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2006.

[10] 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.

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