International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 2, pages 420–435, July 2016

Efficient Implementation of FDFM Approach for Euclidean Algorithms on the FPGA

Xin Zhou, Koji Nakano, Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, Hiroshima, 739-8527 JAPAN

Received: April 6, 2016
Revised: May 16, 2016
Accepted: June 1, 2016
Communicated by Jacir L. Bordim

Abstract

The FDFM (Few DSP slices and Few block Memories) approach is an efficient approach which implements a processor core executing a particular algorithm using few DSP slices and few block RAMs in a single FPGA. Since a processor core based on the FDFM approach uses few hardware resources, hundreds of processor cores working in parallel can be implemented in an FPGA. The main contribution of this paper is to develop a processor core that executes Euclidean algorithm computing the GCD (Greatest Common Divisor) of two large numbers in an FPGA. This processor core that we call GCD processor core uses only one DSP slice and one block RAM, and 1280 GCD processors can be implemented in a Xilinx Virtex-7 family FPGA XC7VX485T-2. The experimental results show that the performance of this FPGA implementation using 1280 GCD processor cores is 0.0904μs per one GCD computation for two 1024-bit integers. Quite surprisingly, it is 3.8 times faster than the best GPU implementation and 316 times faster than a sequential implementation on the Intel Xeon CPU.

Keywords: GCD, FPGA, DSP slice, block RAM, Euclidean

1 Introduction

The GCD (Greatest Common Divisor) computation is widely used in computer systems for cryptography, data security and other important algorithms. Most of the time of these computer systems is consumed for computing the GCDs of very large integers. Therefore, it is an important task of accelerating the GCD computation. However, arithmetic operations on integers numbers exceeding 64 bits cannot be performed directly by a conventional 64-bit CPUs as its instruction set support integers of at most 64-bit in length. It is an efficient way to implement the arithmetic operations on large integers using hardware device such as FPGA, VLSI or GPU.

The FPGA (Field-Programmable Gate Array) is an integrated circuit designed to be configured by a designer after manufacturing. It contains an array of programmable logic blocks called CLB (Configurable Logic Block), and the reconfigurable interconnects allow the blocks to be inter-wired in different configurations. Recent FPGAs have embedded DSP48E1 slices and block RAMs. The Xilinx Virtex-7 series FPGAs have DSP slices equipped with a multiplier, adders, logic operators, etc [1]. More specifically, the DSP slice has a two-input multiplier followed by multiplexers and a three input adder/subtractor/accumulator. The DSP slice also has pipeline registers between operators to reduce the propagation time. The block RAM is an embedded memory supporting synchronized read and write operations, and can be configured as a 36Kbit or two 18Kbit dual port RAMs [2]. They are widely used in consumer and industrial products for accelerating processor intensive algorithms [3, 4, 5, 6, 7, 8, 9]. Since the continuing decline in the ratio of FPGA price to performance and its programmable features, FPGA is suitable for a hardware implementation of general purpose computing. The main contribution of this paper is to present an efficient processor core that executes the Euclidean algorithm computing the GCD of two large integers using an FPGA. The proposed processor core is designed based on the *FDFM (Few DSP slices and Few block Memories) approach* [10]. The key idea of the FDFM approach is to use few DSP slices and few block RAMs for constituting a processor core. We must note that the FDFM approach has some advantages. First, despite the main circuit occupies most of hardware resources of the FPGA, we can also implement the necessary hardware algorithm in the FPGA using remaining few resources as shown in Figure 1 (1). On the other hand, we can implement multiple FDFM processors working in parallel if enough hardware resources are available as illustrated in Figure 1 (2). In this paper, we also employ the FDFM approach to implement parallel GCD computation in the FPGA. For example, in this paper, we propose a processor core for GCD computation of 1024-bit, 2048-bit, 4096-bit, and 8192-bit integers, that uses only one DSP slice and one block RAM. We implement one processor core in the FPGA, and the frequency of the FPGA is over 380MHz, that is extremely high. If only one proposed GCD processor core is implemented in the FPGA for computing one GCD of 1024-bit, 2048-bit, 4096-bit, and 8192-bit integers, it takes 73.12μs, 253.35μs, 915.78μs, and 3614.91μs, respectively. In other words, single GCD processor core has competitive performance. Since the proposed GCD processor core uses very few resources of FPGA, we can implement more than one thousand identical processor cores in an FPGA, that all processor core work are paralleled to execute bulk GCD computation. The pairwise GCD computation that computes all pairs of integers in a set, can be used to evaluate the performance of the implementation of thousand processor cores.

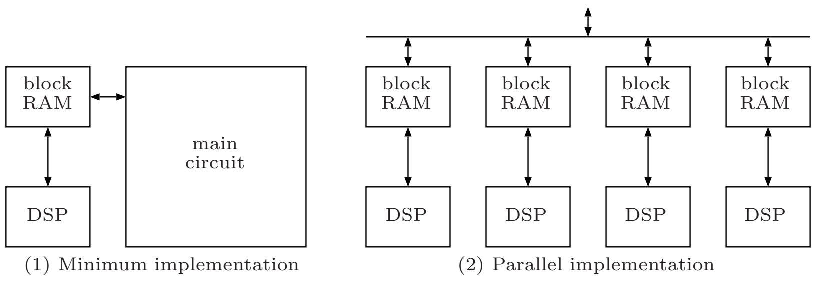

Figure 1: Advantages of our FDFM approach

One of the applications for benchmarking pairwise GCD computation is breaking weak RSA keys. RSA [11] is one of the most well-known public-key cryptosystems widely used for secure data transfer. RSA cryptosystem has an encryption key open to the public. An encryption key includes a modulus n called *an RSA modulus* such that $n = pq$ for two distinct large prime numbers p and q. If the values of p and q are available, the encrypted message can be easily converted to the original message. Thus, the safety of RSA cryptosystem relies on the difficulty of factoring RSA modulus n of two large prime numbers p and q. Suppose that we have a set of many RSA encryption keys collected from the Web. If some of RSA moduli in encryption keys are generated by inappropriate implementation of a random prime number generator, they may reuse the same prime number. We call the keys sharing a prime number as *weak RSA keys*. If two RSA moduli share a prime number, they can be decomposed by computing the GCD of these two moduli. It is well known that the GCD can be computed very easily by Euclidean algorithms [12]. Hence, we can compute the GCDs of all pairs of RSA moduli in the Web to find the RSA keys that shraing the same prime number. In this paper, pairwise GCD computation for RSA moduli is used to measure the performance of the proposed GCD processor core based on FDFM approach. We have succeeded in implementing 1792 GCD processor cores in a Xilinx Virtex-7 family FPGA XC7VX485T-2. However, when the circuit of 1792 GCD processor cores is operated on the FPGA device, this circuit becomes unstable because the number of used resources of FPGA is too close to the maximum available resourses. Finally, we implement 1280 GCD processor cores in the FPGA, that compute the GCDs of all pairs of RSA moduli that are stored in an off-chip DDR3 memory MT8JTF12864HZ-1G6G1. Our implementation of 1280 GCD processor cores computes one GCD of two 1024-bit RSA moduli in expected $0.0904\mu s$.

Several hardware implementations for computing the GCD on FPGAs have been presented [13, 14]. However, they just implemented Binary Euclidean algorithm to compute the GCD using programmable logic blocks as it is. Hence, they can support the GCD computation for numbers with very few bits. On the other hand, several previously published papers have presented GPU implementations of Binary Euclidean algorithm in CUDA-enabled GPUs. Fujimoto [15] has implemented Binary Euclidean algorithm using CUDA and evaluated the performance on GeForce GTX285 GPU. The experimental results show that the GCDs for 131072 pairs of 1024-bit numbers can be computed in 1.431932 seconds. Hence, his implementation runs $10.9\mu s$ per one 1024-bit GCD computation. Scharfglass *et al.* [16] have presented a GPU implementation of Binary Euclidean algorithm. It performs the GCD computation of all 199990000 pairs of 20000 RSA moduli with 1024 bits in 2005.09 seconds using GeForce GTX 480 GPU. Thus, their implementation performs each 1024-bit GCD computation in $10.02\mu s$. Later, White [17] has showed that the same computation can be performed in 63.0 seconds on Tesla K20Xm. It follows that it computes each 1024-bit GCD in $3.15\mu s$. Quite recently, Fujita *et al.* have presented new Euclidean algorithm called Approximate Euclidean algorithm and implemented it in the GPU [18]. Approximate Euclidean algorithm performs perform each 1024-bit GCD computation in $0.346\mu s$ on GeForce GTX 780Ti and $28.6\mu s$ on Intel Xeon X7460 (2.66GHz) CPU. Our implementation of 1280 GCD processor cores in Xilinx VC707 evaluation board [19] equipped with FPGA XC7VX485T-2 performs one 1024-bit GCD computation in $0.0904\mu s$ which is 3.8 times faster than the GPU and 316 times faster than the CPU.

This paper is organized as follows. We first review several Euclidean algorithms in Section 2. Then, we show the implementation of GCD processor core in Section 3. Section 4 shows the architecture of parallel GCD computation using multiple GCD processor cores, that compute the GCD of all pairs of RSA moduli stored in an off-chip DDR3 memory. We show the experimental results in Section 5. Section 6 concludes our work.

2 Euclidean Algorithms for computing GCD

This section review classical Euclidean algorithm and Fast Binary Euclidean algorithm for computing the GCD of two numbers X and Y. We then show Hardware Binary Euclidean algorithm by modifying Fast Binary Euclidean algorithm, that is implemented in an FPGA.

Let $\texttt{GCD}(X,Y)$ denote the GCD of X and Y. For any odd integer X and even integer Y, $\texttt{GCD}(X,Y) = \texttt{GCD}(X,\frac{Y}{2})$ holds. Also, for any even integers X and Y, $\texttt{GCD}(X,Y) = 2 \cdot \texttt{GCD}(\frac{X}{2},\frac{Y}{2})$ holds, and so we can obtain a factor of 2 in the GCD of X and Y very easily.

For simplicity, we assume that both inputs X and Y are odd and $X \geq Y$ holds. Based on the fact, it should have no difficulty to modify all GCD algorithms shown in this paper to handle even input numbers. Let $\texttt{swap}(X,Y)$ denote a function to exchange the values of X and Y. We can write a standard Euclidean algorithm for computing the GCD of X and Y as follows:

```
[Original Euclidean algorithm]
gcd($X$,$Y$){
    do {
        $X \leftarrow X$ mod $Y$; //$X < Y$ always holds
        swap($X,Y$); //$X > Y$ always holds
    } while($Y \neq 0$)
    return($X$);
}
```

Since $X \geq Y$ holds, modulo computation is performed and X will store the value of $X \bmod Y$, which is less than Y. After that, $\texttt{swap}(X,Y)$ is executed and $X > Y$ always holds. The same operation is repeated until $Y = 0$ and X stores the GCD of input integers X and Y. However, modulo computation used in Original Euclidean algorithm is costly. So, Binary Euclidean algorithm which does not execute it, is often used to compute the GCD efficiently:

```
[Binary Euclidean algorithm]
gcd($X$,$Y$){
    do {
        if($X$ is even) $X \leftarrow \frac{X}{2}$;
        else if($Y$ is even) $Y \leftarrow \frac{Y}{2}$;
        else $X \leftarrow \frac{X-Y}{2}$;
        if($X < Y$) $\texttt{swap}(X, Y)$;
    } while($Y \neq 0$)
    return($X$);
}
```

If X (or Y) is even, then X (or Y) is halved to remove the least significant bit of X (or Y) which is 0. If both X and Y are odd, X-Y is computed. Since result of subtraction of two odd numbers is even, $\frac{X-Y}{2}$ is performed to remove the least significant bit of $X - Y$, then X will store the value of $\frac{X-Y}{2}$. If $X < Y$ holds, $\texttt{swap}(X, Y)$ is performed, then $X \geq Y$ always holds. Note that the Binary Euclidean algorithm removes one 0 bit from the least significant bit of X (or Y) and $\frac{X-Y}{2}$ in each iteration of the do-while loop. We can reduce the number of iterations of the do-while loop by removing consecutive 0 bits. Let $\texttt{rshift}(X)$ be a function returning the number obtained by removing consecutive 0 bits from the least significant bit of X. For example, if $X = 11010100$ in binary system, then $\texttt{rshift}(X) = 110101$ in binary notation. Using swap and rshift functions, we can write the Fast Binary Euclidean algorithm as follows:

```
[Fast Binary Euclidean algorithm]
gcd($X$,$Y$){
    do {
        $X \leftarrow \texttt{rshift}(X - Y)$;
        if($X < Y$) $\texttt{swap}(X, Y)$;
    } while ($Y \neq 0$)
    return($X$);
}
```

In each iteration of the do-while loop, at least one 0 bit is removed from X (or Y). Hence, for any input numbers, the number of iteration of the do-while loop in Fast Binary Euclidean algorithm is no larger than that in the Binary Euclidean algorithm. However, we need to read all bits of X and Y to exchange them if we implement function `swap` as it is. Also, `rshift` function needs a large barrel shifter. Hence, we should avoid direct implementations of these functions in the FPGA. Instead of function $\texttt{rshift}(X)$, we implement function $\texttt{rshift}_k(X)$, which removes at most k consecutive 0 bits from the least significant bit of X. In other words, if X has at most k consecutive 0 bits from the least significant bit, all of them can be removed in one iteration of do-while loop by executing $\texttt{rshift}_k(X)$. If X has more than k consecutive 0 bits, then k 0 bits from the least significant bit are removed, and $\texttt{rshift}_k(X)$ is repeated until X is odd. For example, $\texttt{rshift}_2$(1101,1000)=11,0110 and $\texttt{rshift}_2$(1101,1010)=110,1101 hold. Using $\texttt{rshift}_k$, we can describe the Hardware Euclidean-based GCD algorithm as follows:

```
[Hardware Binary Euclidean algorithm]
gcd($X$,$Y$){
    do {
        if($X$ is even) $X \leftarrow \texttt{rshift}_k(X)$;
        else if ($Y$ is even) $Y \leftarrow \texttt{rshift}_k(Y)$;
        else if($X \geq Y$) $X \leftarrow \texttt{rshift}_k(X - Y)$;
        else $Y \leftarrow \texttt{rshift}_k(Y - X)$; // $X < Y$
    } while($X \neq 0$ and $Y \neq 0$)
    if($X \neq 0$) return($X$);
    else return($Y$);
}
```

Note that operation $\texttt{rshift}_k$ may return an even number. Hence, one of X or Y can be an even number. If this is the case, either $X \leftarrow \texttt{rshift}_k(X)$ or $Y \leftarrow \texttt{rshift}_k(Y)$ is executed until both of them are odd. Hence, both X and Y are odd, whenever $\texttt{rshift}_k(X-Y)$ is executed. Thus, the argument of $\texttt{rshift}_k$ is always even and the least significant bit is 0 when it is executed.

Table 1 shows the average number of iterations of the do-while loop 1024-bit RSA moduli for each values of k of $\texttt{rshift}_k$. Note that $k=\infty$ corresponds to Fast Binary Euclidean algorithm, which performs `rshift` function that removes all consecutive 0 bits. Clearly, the number of iterations is smaller for large k. In the preliminary version of this paper [20], we have implemented a barrel shifter using CLBs (Configurable Logic Blocks) to compute $\texttt{rshift}_k$. To balance the computing time and the used hardware resources, we have selected $k=4$. Since the CLBs are costly to compute $\texttt{rshift}_k$ for larger k, we use a multiplier embedded in DSP slice to compute $\texttt{rshift}_k$ for $k=17$. Hence, we can reduce the number of CLBs and implement more GCD processor cores in an FPGA. Since the subtraction of two very large numbers X and Y returning a result which has more than 17 consecutive 0 bits from the least significant bit is a very rare case, $\texttt{rshift}_{17}$ of our implementation and ideal $\texttt{rshift}_k(k=\infty)$ of the Fast Binary Euclidean algorithm has almost the same number of iterations as shown in the table. Also, the number of iterations of $k=17$ is less than that of $k=4$.

Table 1: The average number of iterations of the do-while loop for 1024-bit RSA moduli

	Hardware Binary Euclidean									Fast Binary
k	1	2	3	4	5	6	7	8	17	Euclidean
iterations	1445.8	964.3	827.0	772.0	747.1	735.3	729.7	726.8	724.0	723.9

3 A GCD processor core for large integers

This section shows a *GCD processor core*, which computes the GCD of two very large numbers based on the Hardware Binary Euclidean algorithm. Our GCD processor uses only one 18k-bit block RAM and one DSP slice in the FPGA. The 18k-bit block RAM is configured as a simple dual-port memory [2] with ports A and B of width 36 bits and 18 bits, respectively. Figure 2 illustrate the configuration of the 18k-bit block RAM used in our GCD processor core. Two large numbers X and Y of Hardware Binary Euclidean algorithm are stored as 18-bit words. If each of them has 1024 bits, it is stored in $\lceil \frac{1024}{18} \rceil = 57$ words. Let $X_{56}X_{55}\cdots X_0$ denote 57 words representing X such that $X=\sum_{i=0}^{56} X_i \cdot 2^{18i}$ holds. Similarly, let $Y_{56}Y_{55}\cdots Y_0$ denote the words representing Y. Since the operations $\texttt{rshift}_{17}(X-Y)$ and $\texttt{rshift}_{17}(Y-X)$ of Hardware Binary Euclidean algorithm are executed for computing the GCD of X and Y, we want to read X and Y simultaneously. Hence, port A of the block RAM is configured as read-only 36-bit mode. On the other hand, since the result of operation $\texttt{rshift}_{17}(X-Y)$ (or $\texttt{rshift}_{17}(Y-X)$) is overwritten to one of X or Y, the port B is configured as write-only 18-bit mode.

Reading: Since port A of the block RAM is configured as read-only 36-bit mode, the block RAM is a 512×36-bit memory for port A. We can read 36-bit data $X_iY_i (0 \leq i \leq 56)$ from address i using port A for performing the operation $\texttt{rshift}_{17}(X-Y)$ (or $\texttt{rshift}_{17}(Y-X)$).

Writing: Since port B is configured as write-only 18-bit mode, the block RAM is a 1024×18-bit memory for port B. We can write 18-bit data X_i in address $2i$ or and Y_i in address $2i+1$ $(0 \leq i \leq 56)$ using port B. In other words, the result of $\texttt{rshift}_{17}(X-Y)$ (or $\texttt{rshift}_{17}(Y-X)$) can be overwritten to X (or Y).

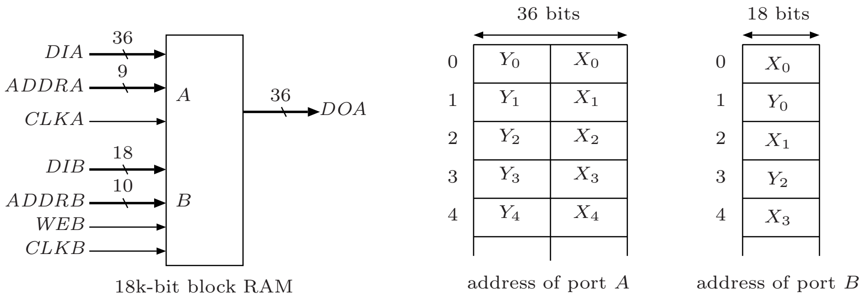

Figure 2: A 18k-bit block RAM and the memory configuration

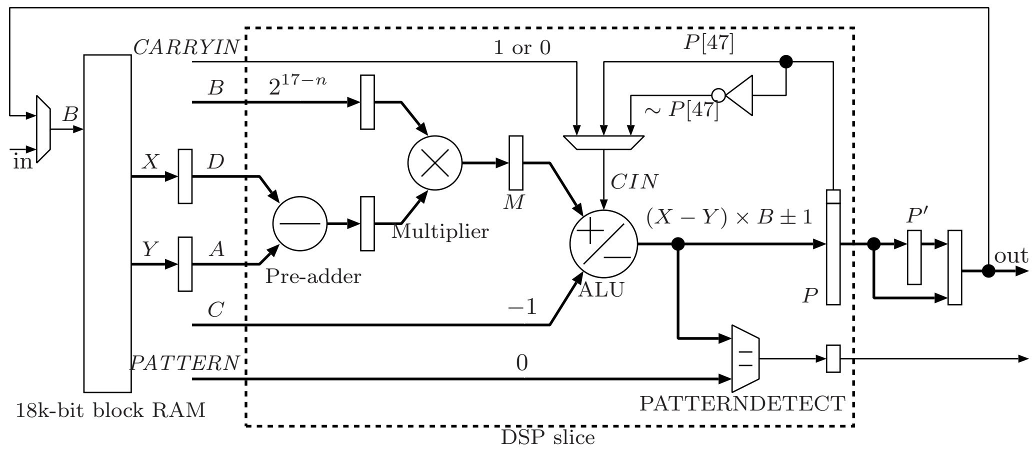

Figure 3: The architecture of a GCD processor

The DSP slice in our GCD processor core uses a pre-adder, a multiplier and a three input ALU (Arithmetic Logic Unit) as illustrated in Figure 3. Suppose that $X \geq Y$ holds. We briefly show how to use the DSP slice for executing the operation $\texttt{rshift}_{17}(X-Y)$ of Hardware Binary Euclidean algorithm. The 36-bit data $X_iY_i (0 \leq i \leq 56)$ is read from the block RAM one by one, and is connected to the pre-adder of DSP slice. The operation $X-Y$ needs to be executed from the least significant bit of large numbers X and Y. Thus, the pre-adder is used to compute $X_i - Y_i$ for each 36-bit data X_iY_i one by one from X_0Y_0. Since $X_i - Y_i$ is computed one by one, if $X_0 - Y_0 < 0$ holds, we need to borrow from the higher bit which is in the next word $X_1 - Y_1$. In other words, $X_1 - Y_1 - 1$ needs to be computed for 36-bit data X_1Y_1, and we call -1 *borrow*. Let b_0 denote the borrow from $X_0 - Y_0$, and let $b_i (1 \leq i \leq 55)$ denote the borrow from $X_i - Y_i - b_{i-1}$. We note that $X_0 - Y_0$ needs to be computed for X_0Y_0, and $X_i - Y_i - b_{i-1}$ needs to be computed for $X_iY_i (1 \leq i \leq 56)$. However, we can not compute the borrow using the pre-adder because it has only two input ports. Thus, we first perform the shift operation to remove the consecutive 0 bits from the least significant bit using the multiplier. The multiplier performs the operation $(X_i - Y_i) \times 2^{17-n}$ for each $X_i - Y_i (0 \leq i \leq 56)$ one by one, where $n (1 \leq n \leq 17)$ is the number of consecutive 0 bits from the least significant bit of $X - Y$. If $X - Y$ has more than 17 consecutive 0 bits from the least significant bit, n has the value 17. For example, if $X_0 - Y_0 = (11, 0010, 1\underline{000}, \underline{0000}, \underline{0000})$, that is, $X - Y$ has 11 consecutive 0 bits from the least significant bit. The multiplier computes $(X_0 - Y_0) \times 2^{17-11} = (1100, 101\underline{0}, \underline{0000}, \underline{0000}, \underline{00}00, 0000)$. We note that the 11 bits consecutive 0 bits are on the right of the 17-th bit of $(X_0 - Y_0) \times 2^6$. For other $n (1 \leq n \leq 17)$, the n bits consecutive 0 bits are also on the right of the 17-th bit of $(X_0 - Y_0) \times 2^{17-n}$. We use this feature to remove the consecutive 0 bits in the following. Next, since we suppose that $X \geq Y$ holds, ALU computes $(X_i - Y_i) \times 2^{17-n} - b_{i-1}$. Otherwise, if $X < Y$ holds, ALU can also compute $-(X_i - Y_i) \times 2^{17-n} - b_{i-1} = (Y_i - X_i) \times 2^{17-n} - b_{i-1}$. In other words, the computation of $(X - Y)$ and $(Y - X)$ can be switched dynamically by controlling the behavior of ALU, and the borrow is computed after the shift operation using the ALU. For example, suppose that $X \geq Y$ and $X_0 < Y_0$ hold, and $X_0 - Y_0 = (11, 0010, 1\underline{000}, \underline{0000}, \underline{0000})$. The multiplier computes $(X_0 - Y_0) \times 2^{17-11} = (1100, 101\underline{0}, \underline{0000}, \underline{0000}, \underline{00}00, 0000)$, where the 11 consecutive 0 bits are all on the right of the 17-th bit of $(X_0 - Y_0) \times 2^6$ as we shown above. Then, ALU outputs $(X_0 - Y_0) \times 2^6$ as it is. To remove 11 consecutive 0 bits from the least significant bit of $X - Y$, we retain the higher $18 - 11 = 7$ bits from the 17-th bit of $(X_0 - Y_0) \times 2^6$, which is (110,0101). On the other hand, suppose that $X_1 - Y_1 = (01, 1011, 0100, 1011, 0100)$, the multiplier also computes $(X_1 - Y_1) \times 2^6 = (0110, 110\underline{1}, \underline{0010}, \underline{1101}, \underline{00}00, 0000)$. Since there is a borrow from $X_0 - Y_0$, ALU computes $(X_1 - Y_1) \times 2^6 - 1 = (0110, 110\underline{1}, \underline{0010}, \underline{1100}, \underline{11}11, 1111)$. We note that the higher 18 bits of $(X_1 - Y_1) \times 2^6 - 1$ is equal to $X_1 - Y_1 - 1$. Since only 7 bits of $X_0 - Y_0$ are retained, we need to pick up 11 bits from the least significant bit of $X_1 - Y_1 - 1$ to restructure the first 18-bit word of $\texttt{rshift}_{17}(X - Y)$. Hence, we pick up 11 consecutive bits on the right of the 17-th bit of $(X_1 - Y_1) \times 2^6 - 1$, which is (100,1011,0011). Then, we concatenate 11 bits data (100,1011,0011) of $X_1 - Y_1 - 1$ with 7 bits data (110,0101) of $X_0 - Y_0$ to restructure the first word of $\texttt{rshift}_{17}(X - Y)$, which is (10,0101,1001,1110,0101). Also, the other words of $\texttt{rshift}_{17}(X - Y)$ can be obtained in the same way. The configuration of DSP slice is described as follows:

Pre-adder: The pre-adder of DSP slice has 25-bit port D and 30-bit port A. The 36-bit output of the block RAM are connected to the pre-adder via a pipeline register. X is given to port D, and Y is given to port A. The remaining bits of the ports are padded with 0. The pre-adder of DSP slice can compute $D - A$, A and D by controlling its behavior, in other words, the pre-adder outputs $X - Y$, Y or X optionally. For example, to perform the operation $X - Y$, the subtraction $X_i - Y_i$ is performed for each 36-bit data X_iY_i one by one, and the output of pre-adder is connected to the multiplier.

Multiplier: The embedded multiplier has two input ports, where one accepts an 18-bit two's complement operand from port B via a pipeline register, the other one accepts an 25-bit two's complement operand from the pre-adder via a pipeline register. We use the multiplier to perform the multiplication between the result of pre-adder and value of port B, where B has the value $2^k (0 \leq k \leq 17)$ in our implementation. Thus, the operations $(X_i - Y_i) \times B$, $X_i \times B$, and $Y_i \times B$ can be executed using multiplier. In other words, shift operation can be executed for $X - Y$, X, and Y. Hence, we do not need a barrel shifter which is used in the preliminary version of this paper [20]. The output of multiplier is connected to ALU(Arithmetic logic unit) via a pipeline register M as shown in the Figure 3.

ALU: The ALU (Arithmetic Logic Unit) has three input ports, that are connected to register M, input port C of DSP slice, and port CIN, respectively. The most significant bit of register P, the negation of the most significant bit of register P and port $CARRYIN$ are connected to port CIN of ALU. Port CIN can select one of the three values by controlling its behavior. The ALU can performs several operations such as $M + C + CIN$ and $-M - C - CIN - 1$. In our implementation, C is configured as the value -1. Since M is connected to the output of multiplier, we can control the behavior of the ALU dynamically for computing $(X_i - Y_i) \times B + CIN - 1$ if $X \geq Y$ holds, and computing $-(X_i - Y_i) \times B - CIN = (Y_i - X_i) \times B - CIN$ if $X < Y$ holds, where CIN is used as the borrow corresponding to the subtraction of previous 36-bit data $X_{i-1}Y_{i-1}$. In the preliminary verison of this paper [20], to dynamically compute $X - Y$ and $Y - X$, we exploit two multiplexers by configuring connecting both X and Y, where the multiplexers is implemented using logic resources of FPGA. Hence, the used FPGA resources of GCD processor core proposed in this paper has decreased. The value computed by ALU is then connected to register P.

Pattern detector: The pattern detector can determine that the value of register P matches a pattern or not, as qualified by a mask. The mask is used as enable signals for pattern detector. More specifically, if a certain bit of mask is set to "0", the corresponding bit of $PATTERN$ and P is compared. Otherwise, the comparison of the corresponding bits is not performed. The value of port $PATTERN$ is configured as 0.

Using the block RAM and the functionality of DSP slice, we can perform Hardware Binary Euclidean algorithm without fabric barrel shifter and multiplexers that are used in the preliminary verison of this paper. We show how each operation in Hardware Binary Euclidean algorithm can be performed. Let $x_{1023}x_{1022}\cdots x_0$ denote 1024 bits representing X such that $x_{17}x_{16}\cdots x_0$ represents X_0. Similarly, let $y_{1023}y_{1022}\cdots y_0$ denote 1024 bits representing Y such that $y_{17}y_{16}\cdots y_0$ represents Y_0.

X is even: The number X is write to the block RAM word by word. Thus, the condition can be determined by reading the least significant bit of X_0 when X is input into the block RAM.

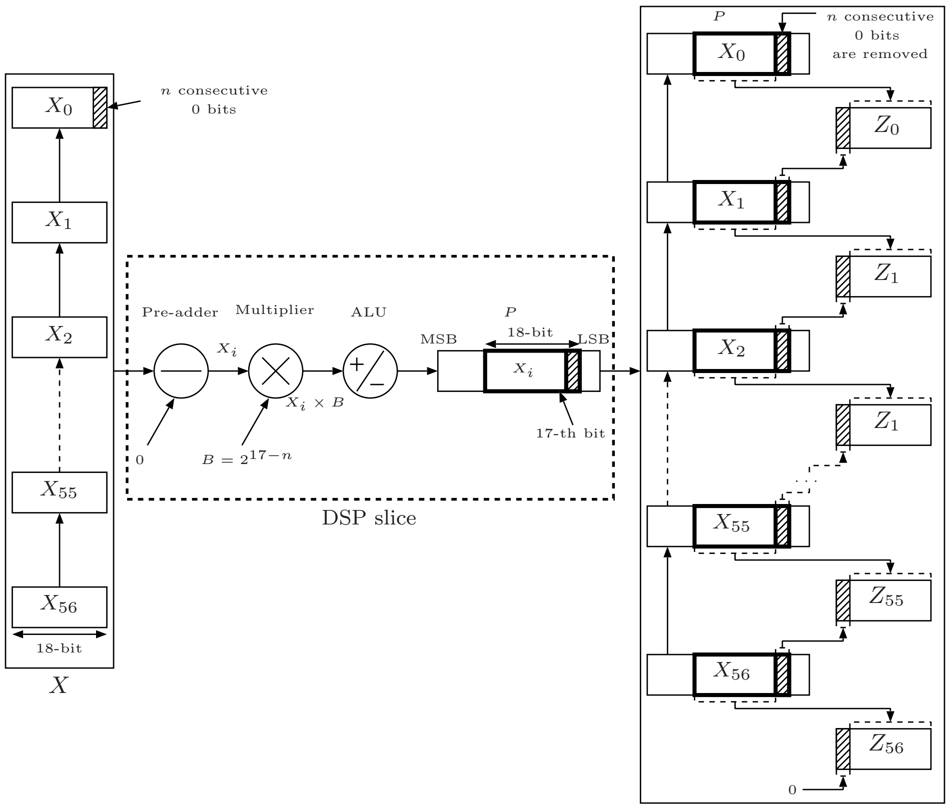

Figure 4: The outline of rshift$_{17}(X)$

$X \leftarrow$ `rshift`$_{17}(X)$: If X is even, function `rshift`$_{17}(X)$ is executed to remove the consecutive 0 bits from the least significant bit of X. Suppose that we need to compute Z=`rshift`$_{17}(X)$. Let $Z_{56}Z_{55}\cdots Z_0$ denote 57 words representing Z and show how `rshift`$_{17}(X)$ is computed as the flow shown in Figure 4. All words of X are sequentially read from the block RAM beginning with X_0 and then processed one by one in a pipelined order. $X_i (0 \leq i \leq 56)$ is given to the pre-adder of DSP slice. The pre-adder outputs X_i as it is. Also, we must obtain the number of consecutive 0 bits from the least significant bit of X_0 to execute shift operation using the multiplier. Let $\delta = \delta_{17}\delta_{16}\cdots\delta_0$ denote the result of logic prefix-or operation of X_0. The operation $\delta_i \leftarrow x_i \vee \delta_{i-1} (1 \leq i \leq 17)$ is performed, where $\delta_0 = x_0 = 0$ since X is even. For example, suppose that $X_0 = (11, 0010, 1011, 1011, 0000)$, where the number n of consecutive 0 bits of X is 4. We have $\delta = (11, 1111, 1111, 1111, 0000)$. Note that except the consecutive 0 bits from the least significant bit, the other bits all have the value 1. Let $\lambda = \lambda_{17}\lambda_{16}\cdots\lambda_0$ denote the result of exclusive-or operation of δ. The operation

$\lambda_i \leftarrow \delta_i \oplus \delta_{i-1} (1 \leq i \leq 17)$ is performed, where $\lambda_0 = \delta_0 = 0$ holds. For the δ shown above, $\lambda = (00, 0000, 0000, 0001, 0000)$ holds. The only one bit that has the value 1 indicates that there are 4 consecutive 0 bits from the least significant bit of X_0. Then, the inverse of λ which has the value $(00, 0010, 0000, 0000, 0000)$, is configured as the value of port B to perform shift operation using the multiplier of DSP slice. We note that if X has $n(0 < n \leq 17)$ consecutive 0 bits, the value of B will be 2^{17-n}. Otherwise, $B = 2^0$ holds. In the case of executing operation $X \leftarrow \texttt{rshift}_{17}(X)$, pre-adder directly outputs X_0 to the multiplier. The product of $X_0 \times 2^{17-n}$ is then computed by the multiplier. Similarly, for other words of X, $X_i \times 2^{17-n} (1 < i \leq 56)$ are also computed one by one in the same way. We note that the consecutive 0 bits of X_0 are always on the right of 17-th bit from the least significant bit of $X_0 \times 2^{17-n}$. In the example above, since $n = 4$ holds, $X_0 \times 2^{17-4} = (110, 0101, 0111, 011\underline{0}, \underline{000}0, 0000, 0000, 0000)$, where the 4 consecutive 0 bits from the least significant bit of X_0 are all on the right of 17-th bit of $X_0 \times 2^{13}$. The resulting value of $X_0 \times 2^{13}$ is then transferred to ALU via register M. The ALU outputs $M + C + CIN$, where port C is configured as a constant -1. CIN is used as borrow of subtraction of $X - Y$ which is not needed for executing $\texttt{rshift}_{17}(X)$, thus CIN is set to 1. Therefore, ALU outputs the resulting value $X_0 \times 2^{13}$ to register P. We then retain higher $18 - 4 = 14$ bits from 17-th bit of P, that is (11,0010,1011,1011). In other words, the 4 consecutive 0 bits from the least significant bit of X_0 are removed. Since 4 consecutive 0 bits are removed from X_0, we must pick n bits from its next word X_1 of X to restructure the new word Z_0 of $Z = \texttt{rshift}_{17}(X)$. Suppose that $X_1 = (01, 1101, 0010, 0011, 1011)$, the same operation is performed for X_1, and $X_1 \times 2^{17-4} = (011, 1010, 0100, 011\underline{1}, \underline{011}0, 0000, 0000, 0000)$ will be stored in register P in the next clock cycle since the architecture is pipelined. Similarly, 4 bits from the least significant bit of X_1 are also on the right of 17-th bit of P. Thus, we can easily pick 4 bits from the least significant bit of X_1 that are store on the right of 17-th bit of P, and then concatente with retained 14 bits of X_0 to restruct the new word $Z_0 = (10, 1111, 0010, 1011, 1011)$ of $Z = \texttt{rshift}_{17}(X)$. As shown in Figure 4, since $X_{56}X_{55} \cdots X_0$ are input one by one, $Z_{56}Z_{55} \cdots Z_0$ can be computed one by one and then transferred to the block RAM to overwrite the old X. We say that $X \leftarrow \texttt{rshift}_{17}(X)$ is executed such that n consecutive 0 bits from the least significant bit of X are removed. If input X has more than 17 consecutive 0 bits from the least significant bit, the function $\texttt{rshift}_{17}(X)$ is repeated until X is odd. Also, if input Y is even, the same operation is performed for Y.

$X \geq Y$: The condition $X \geq Y$ can be determined by comparing X and Y from the most significant bit. More specifically, X and Y are compared from the words X_{56} and Y_{56}. The words $X_{56}Y_{56}$ are read from the block RAM concurrently, then are connected to port D and A of DSP slice, respectively. We always assume that $X \geq Y$ holds, thus, the pre-adder computes $X_{56} - Y_{56}$, and the resulting value is input to multiplier. The port B is configured as 2^{17} in this case. Thus, multiplier computes $(X_{56} - Y_{56}) \times 2^{17}$. However, since B is 18-bit two's complement, the most significant bit of B is sign bit. Hence, if $B = 2^{17}$, the operation $(X_{56} - Y_{56}) \times (-2^{17})$ is computed by multiplier, and the resulting value is then transferred to ALU. The ALU outputs the value to register P as it is. Clearly, the value of $X_{56} - Y_{56}$ is left shifted by 17 bits, and is stored in register P from 34-th bit to 17-th bit. If $X_{56} > Y_{56}$ holds, the most significant bit of P have the value 1 since $(X_{56} - Y_{56}) \times (-2^{17})$ is computed by the multiplier. We determine the condition $X \geq Y$ if the most significant bit $P[47]$ of P has the value 0. However, the value $X_{56} - Y_{56}$ may be 0 if $X_{56} = Y_{56}$ holds. Thus, we use the pattern detector to determine that 18 bits in $P[34{:}17]$ of register P are all 0 or not. If $X_{56} = Y_{56}$, $P[34 : 17] = 0$ holds and the detector outputs the value 1. We need to compare the next words $X_{55}Y_{55}$ to determine the condition $X \geq Y$. It takes 3 clock cycles to determine the condition $X_{56} = Y_{56}$ from the words $X_{56}Y_{56}$ are input to the DSP slice, because three-stage pipeline registers are used as shown in Figure 3. And in most of cases, we can determine the condition $X \geq Y$ by comparing the words $X_{56}Y_{56}$. Hence, we start to execute the operation $\texttt{rshift}_{17}(X - Y)$ one clock cycle after the words $X_{56}Y_{56}$ are input to DSP slice. More specifically, we start the execution of $\texttt{rshift}_{17}(X - Y)$ from words X_0Y_0 without waiting the determination of the condition $X \geq Y$, which we will show in operation $X \leftarrow \texttt{rshift}_{17}(X - Y)$. If $X_{56} = Y_{56}$ is determined after 3 clock cycles, we terminate the execution of $\texttt{rshift}_{17}(X - Y)$, and restart to compare the next words $X_{55}Y_{55}$ to determine the condition $X \geq Y$.

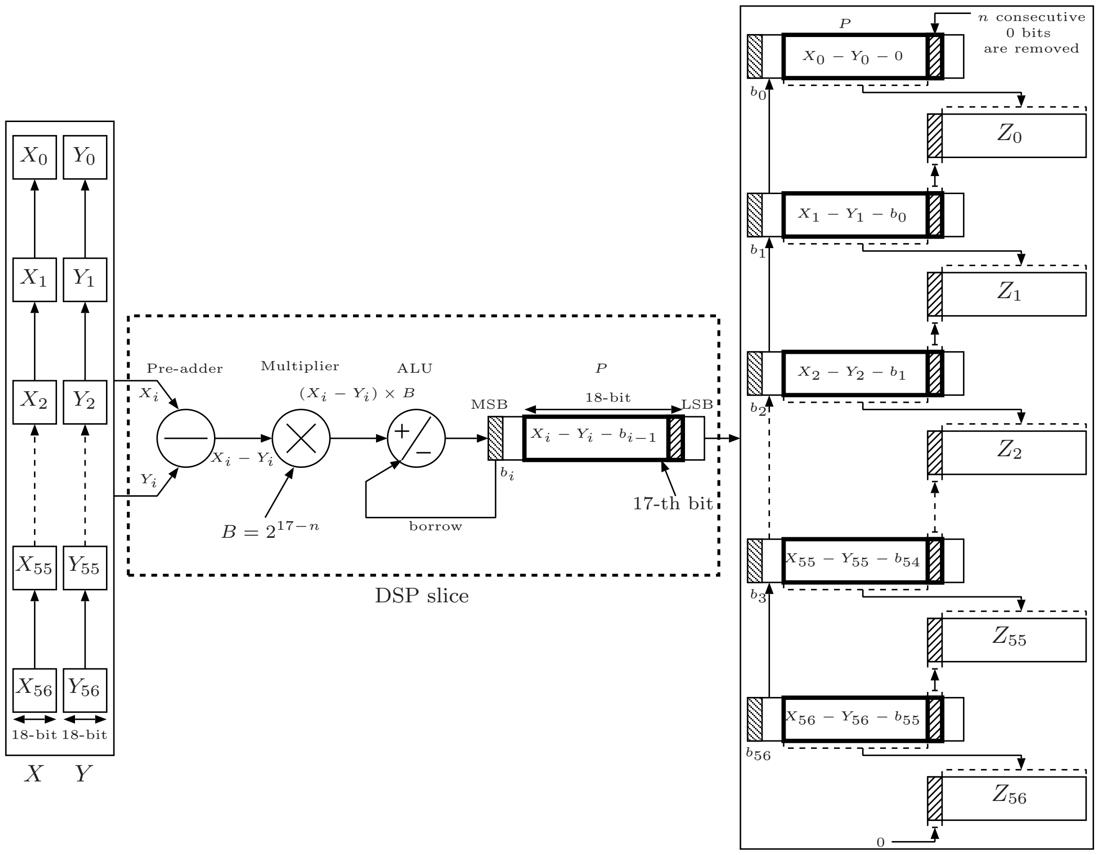

Figure 5: The outline of $\texttt{rshift}_{17}(X - Y)$

$X \leftarrow \texttt{rshift}_{17}(X - Y)$: Suppose that we need to compute $Z = \texttt{rshift}_{17}(X - Y)$. Let $Z_{56}Z_{55} \cdots Z_0$ denote 57 words representing Z. As mentioned above, if we execute the function $\texttt{rshift}_{17}(X - Y)$ after the condition $X \geq Y$ is determined which takes 3 clock cycles, that is, any operation can be performed in 3 clock cycles for each iteration of do-while loop of the Hardware Binary Euclidean algorithm. Fortunately, we do not need to wait for the determination of the condition $X \geq Y$. In our implementation, all words of X and Y are read from the block RAM one by one beginning with X_0Y_0, one clock cycle after $X_{56}Y_{56}$ are read from the block RAM to determine the condition of $X \geq Y$. Thus, $X_0 - Y_0$ is computed by pre-adder since we assume that $X \geq Y$ always holds. The resulting value of $X_0 - Y_0$ is input to the multiplier, then $(X_0 - Y_0) \times 2^{17-n}$ is computed by the multiplier, where n is the number of consecutive 0 bits from the least significant bit of $X - Y$. Since determination of the condition $X \geq Y$ is executed one clock cycle earlier than function $\texttt{rshift}_{17}(X - Y)$, we can dynamically control the behavior of the ALU depending on the determination of the condition $X \geq Y$. More specifically, the result of $X_{56} - Y_{56}$ is obtained one clock cycle earlier than $(X_0 - Y_0) \times 2^{17-n}$ is accepted by ALU. Hence, if $X_{56} > Y_{56}$ holds, we control the behavior of ALU to compute $(X_0 - Y_0) \times 2^{17-n} + CIN - 1$. If $X_{56} < Y_{56}$ is determined, $-(X_0 - Y_0) \times 2^{17-n} - CIN$ is computed by the ALU. The selection of CIN depends on the borrow of subtraction of words $X_iY_i (0 \leq i \leq 56)$, and we can also dynamically select the value of CIN by controlling its behavior. For example, if $X \geq Y$ holds, we select CIN as 1 to compute $(X_0 - Y_0) \times 2^{17-n} + 1 - 1 = (X_0 - Y_0) \times 2^{17-n}$. If $X < Y$ holds, we select CIN as 0 to compute $-(X_0 - Y_0) \times 2^{17-n}$. Then, the result of the ALU is stored to register P. Hence, by checking the most significant bit $P[47]$ of register P, we can obtain the borrow of the subtraction $X_0 - Y_0$. Suppose that $X \geq Y$ is determined. If $X_0 \geq Y_0$, $P[47] = 0$ holds, otherwise, $P[47] = 1$ holds.

In the same way, $(X_1 - Y_1) \times 2^{17-n} + CIN - 1$ is computed by ALU in the next clock cycle, We select the value of CIN as the negation of $P[47]$ as the borrow from $X_0 - Y_0$. Thus, if $X_0 \geq Y_0$, $(X_1 - Y_1) \times 2^{17-n} + 1 - 1 = (X_1 - Y_1) \times 2$ is computed. Otherwise, $(X_1 - Y_1) \times 2^{17-n} - 1$ is computed. Next, we briefly show how to obtain the word Z_0 of $Z = \mathtt{rshift}_{17}(X - Y)$ is computed as shown in Figure 5. Suppose that $X_0 \geq Y_0$ holds. Since the result of $X_0 - Y_0$ is shifted by $17 - n$ bits and stored in P, the n consecutive 0 bits from the least significant bit of $X_0 - Y_0$ are on the right of 17-th bit of P. Hence, we retain $18 - n$ bits on the left of 17-th bit of P to store in a register. In other words, the n consecutive 0 bits from the least significant bit of $X_0 - Y_0$ stored on the right of 17-th bit of P are removed. Also, $X_1 - Y_1$ is shifted by $17 - n$ bits and stored in P. Similarly, the n bits from the least significant bit of $X_1 - Y_1$ are stored on the right of 17-th bit of P. Then, we can easily pick up n bits from the least significant bit of $X_1 - Y_1$ to concatente with higher $18 - n$ bits of $X_0 - Y_0$ to restruct the new word Z_0 as shown in Figure 5. The same operation is executed for all words $X_iY_i (0 \leq i \leq 56)$ in a pipelined order. Hence, the words $Z_{56}Z_{55} \cdots Z_0$ can be obtained one by one and are then written back to the block RAM to overwrite the old X.

$X \neq 0$: We use a register to store the current number of bits of X. If operation $X \leftarrow \mathtt{rshift}_{17}(X)$ or $X \leftarrow \mathtt{rshift}_{17}(X - Y)$ is executed, we rewrite the value of this register. We determine the condition $X \neq 0$ if the number of bits of X is not 0.

Let us briefly confirm that the GCD processor core can execute Hardware Binary Euclidean algorithm. By controlling the behavior of pre-adder, multiplier and ALU of DSP slice, we can compute $\mathtt{rshift}_{17}(X - Y)$, $\mathtt{rshift}_{17}(Y - X)$, $\mathtt{rshift}_{17}(X)$ and $\mathtt{rshift}_{17}(Y)$ without multiplexers and barrel shifter that use resources of FPGA. The resulting value can be written to the block RAM to overwrite X or Y. The conditions "X is even" and "Y is even" can be determined when X_0 and Y_0 are written in the block RAM. The condition "$X \geq Y$" can be determined by checking X and Y from the MSB (Most Significant Bit). More specifically, if $X_{56} > Y_{56}$ holds, "$X \geq Y$" is determined. We execute the $\mathtt{rshift}_{17}(X - Y)$ without waiting the determination of the condition "$X \geq Y$", because the condition "$X \geq Y$" can be determined by comparing the words X_{56} and Y_{56} in most of the cases. However, if $X_{56} = Y_{56}$, we terminate the execution of $\mathtt{rshift}_{17}(X - Y)$, and then read and compare X_{55} with Y_{55}. During the computation of Hardware Binary Euclidean algorithm, the number of bits of X and Y is decreased. For example, if X_{56} and Y_{56} both decrease to 0, the next iteration of the do-while loop of Hardware Binary Euclidean algorithm is only performed for words $X_iY_i (0 \leq i < 56)$. We use registers to store the current numbers of bits of X and Y. If the number of bits is 0, we terminate the algorithm.

4 Implementation of Hierachical GCD cluster with DDR3 Memory

This section shows a hierarchical parallel architecture based on the hierarchical GCD cluster [20] using an off-chip DDR3 memory equipped in Xilinx VC707 evaluation board [19]. The proposed GCD processor core is compactly designed based on the FDFM approach. We use only one DSP slice, one block RAM and a few CLBs to implement the processor core. Therefore, single proposed FDFM GCD processor core is clocked at high frenquency and provides high performance that we show in the next section. On the other hand, by employing multiple proposed FDFM GCD processors, the computing time reduces considerably. Since the proposed GCD processor is designed based on the FDFM approach and uses very few FPGA resources, we have succeeded in implementing more than one thousand proposed GCD processor cores working in parallel in the FPGA, thus, it makes sense to use multiple servers. Each server controls more than one hundred GCD processor cores. The hierarchical GCD cluster consists of multiple GCD clusters, each of which involves multiple GCD processor cores as illustrated in Figure 6. A single central server controls local servers, each of which maintains GCD processor cores in the same GCD cluster.

We show how the hierarchical GCD cluster is used to execute pairwise GCD computation for RSA moduli. The DDR3 memory consists of 8 banks. Each bank has a memory array that can be used to store lots of moduli. Suppose that we have a lot of moduli collected from the Web and all moduli are divided into two sets. We store two sets of moduli to two different banks of DDR3 memory for simplifying the address/control circuit. Our goal is to compute all pairs of moduli using the hierarchical GCD cluster in an FPGA. For this purpose, we partition all moduli of each set into groups with m moduli each. FPGA picks one group from each set and sends them to the central server, respectively. Let $N = \{n_0, n_1, \ldots n_{m-1}\}$ and $N' = \{n'_0, n'_1, \ldots n'_{m-1}\}$ denote two groups of m moduli each that the central server in the FPGA has received. The hierarchical GCD cluster computes $\gcd(n_i, n'_j)$ for all pairs of i and j $(0 \leq i, j \leq m-1)$, and reports the GCDs larger than 1.

Next, we will show how the hierarchical GCD cluster computes the GCDs of N and N' using GCD clusters. Each group of m moduli is partitioned into b blocks of k moduli each, where $m = bk$. Let $N_i = \{n_{ik}, n_{ik+1}, \ldots, n_{(i+1)k-1}\}$ and $N'_i = \{n'_{ik}, n'_{ik+1}, \ldots, n'_{(i+1)k-1}\}$ $(0 \leq i \leq b-1)$ be two sets of k moduli in the i-th groups of sets N and N', respectively. Each cluster is assigned a task to compute the GCDs of all pairs X $(\in N_i)$ and Y $(\in N'_j)$ for a pair i and j $(0 \leq i, j \leq b-1)$. For this purpose, all moduli in N_i and in N'_j are copied from the block RAM in the central server to that in the local server of a GCD cluster. After the local server receives all moduli, the cluster starts computing the GCDs of all pairs X $(\in N_i)$ and Y $(\in N'_j)$. The local server then picks a pair X and Y and copies them to the block RAM of a GCD processor. Upon completion of the copy, the GCD processor starts computing the GCD of X and Y by the Hardware Binary Euclidean algorithm. This procedure is repeated for all GCD processors. If a GCD processor terminates the GCD computation, the local server sends a new pair to it. In this way, the GCDs of all pairs in N_i and N'_j are computed by a GCD cluster. When a GCD cluster completes the computation of all GCDs of a given pair of two groups, the central server picks a new pair i and j and sends all moduli in N_i and in N'_j to the local server. The same operation is repeated until the GCDs of all pairs N and N' are computed.

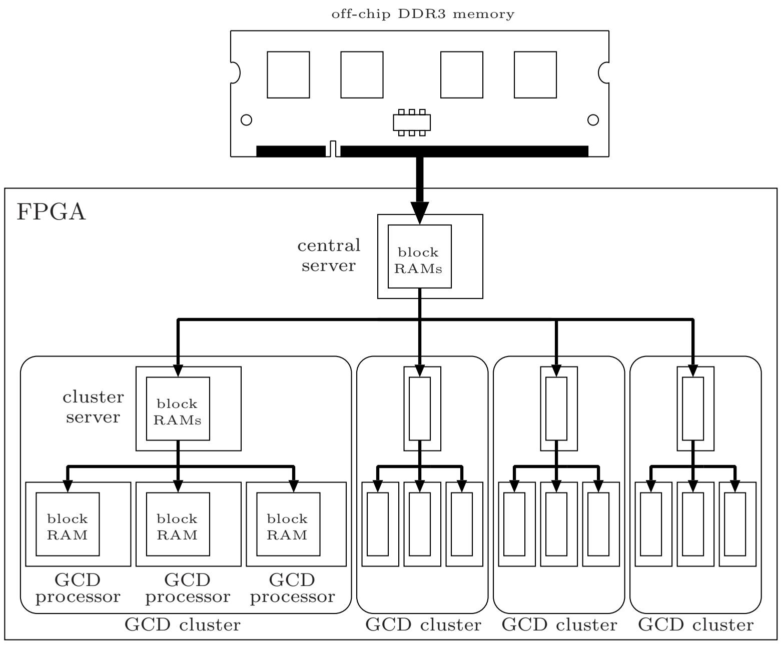

Figure 6: The architecture of the Hierarchical GCD cluster

5 Implementation results in the FPGA

We have implemented a GCD processor core for computing the GCD of 1024-bit, 2048-bit, 4096-bit, and 8192-bit integers in Xilinx Virtex-7 XC7VX485T-2. Table 2 shows the implementation results.

Slice Registers and Slice LUTs (Look-Up-Tables) are hardware resources in CLB (Configurable Logic Block) [21], which are used to implement sequential logics. The proposed GCD processor is compactly designed based on FDFM approach. More specifically, we use only one DSP slice to perform subtraction and shift operation for very large numbers and use one block RAM to store the computed result instead of using lots of CLBs. Therefore, the proposed FDFM GCD processor is clocked at over 380MHz and provides a high performance. Calculated simply, single proposed FDFM GCD processor core computes one GCD of two 1024-bit, 2048-bit, 4096-bit and 8192-bit moduli in expected 73.12μs, 253.35μs, 915.78μs and 3614.91μs.

Recall that we control the behavior of the embedded ALU of the DSP slice to perform $X - Y$ or $Y - X$ dynamically instead of two multiplexers used in our previous work [20]. Also, we use the embedded multiplier of the DSP slice to perform the shift operation instead of the barrel shifter used in our previous work, where the barrel shifter uses a lot of FPGA logic resources. Since these mechanisms simplify the circuit of the proposed processor, the frequency of the proposed FDFM processor is over 380MHz that is higher than that of the implementation of our previous work.

Table 2: Implementation results of one GCD processor for 1024-bit, 2048-bit, 4096-bit, and 8192-bit moduli

	Slice Registers	Slice LUTs	DSP slices	18k-bit block RAMs	Clock cycles for computing	Clock Frequency
Available	607200	303600	2800	2060	one GCD	(MHz)
1024-bit	179	163	1	1	28006.1	383.00
2048-bit	185	174	1	1	98198.5	387.60
4096-bit	191	178	1	1	359131.4	392.16
8192-bit	197	188	1	1	1381328.5	382.12

First, the simulation of pairwise GCD computation for 1024-bit RSA moduli without DDR3 memory is performed. In our implementation, a GCD cluster with a local cluster with eight 18k-bit block RAMs and 128 GCD processor cores are used. Since four 18k-bit block RAMs can store $\lfloor \frac{4 \cdot 1024}{57} \rfloor = 71$ moduli with 1024 bits, each GCD cluster computes the GCDs of $71 \times 71 = 5041$ pairs of blocks stored in block RAMs. Hence, each GCD processor computes the GCDs for expected $\frac{5041}{128} = 39.4$ pairs of 1024-bit moduli. Also we arranged 64 block RAMs to the central server. Since a block of moduli is stored in four block RAMs, we can think that the central server has $8 \times 8 = 64$ pairs of blocks. Thus, each cluster computes the GCDs for moduli in expected $\frac{64}{14} = 4.5$ pairs of blocks since we have succeeded in implementing 14 clusters in an FPGA. Table 3 shows the implementation results of clusters of our work. Since a cluster server uses eight 18k-bit block RAMs, each GCD cluster with 128 GCD processors involves $128 + 8 = 136$ block RAMs. In this paper, the implementation of the hierarchical GCD cluster with 14 GCD clusters and the central server, uses $14 \cdot 128 = 1792$ DSP slices and $14 \cdot 136 + 64 = 1968$ block RAMs. Due to the overhead for the connection between the central server and GCD clusters, the clock frequency is decreased to 207.04MHz. The used block RAMs of the implementation with 14 clusters are close to the available number. Since the proposed GCD processor core is compactly designed, the number of processor cores in our implementation is more than that of the preliminary verison of this paper [20].

Table 3: Implementation results of the GCD cluster and the hierarchical GCD cluster for 1024-bit moduli

	Slice Registers	Slice LUTs	DSP slices	18kb block RAMs	Clock Frequency
Available	607200	303600	2800	2060	(MHz)
one cluster	23414	20598	128	136	327.87
hierarchical clusters	325987	272127	1792	1968	207.04

We have evaluated the number of clock cycles to compute all GCDs of $71 \times 71 = 5041$ pairs of 1024-bit moduli by one GCD cluster. For this purpose, we have used RSA moduli generated by OpenSSL Toolkit. By performing the simulation, one cluster with 128 processors takes 1157789 clock cycles to compute the GCDs of 5041 pairs. If a GCD cluster is clocked at 207.04MHz as shown in Table 3, the expected computing time is 1157789/207.04MHz = 5.592ms. Also, it takes about $71 \times 2 \times 57 = 8094$ clock cycles to transfer a pair of two blocks involving 71 moduli each and this overhead is negligible. Since up to 14 clusters can be implemented theoretically, we can expect that the GCDs of $5041 \times 14 = 70574$ pairs can be computed in the same time. Therefore, we say that one GCD can be computed in expected 5.592ms/70574 = 0.0792μs.

Table 4: Implementation results of hierarchical GCD clusters for 1024-bit, 2048-bit, 4096-bit, and 8192-bit moduli

	Slice Registers	Slice LUTs	DSP slices	18kb block RAMs	Clock Frequency	Average Time	Number of
Available	607200	303600	2800	2060	(MHz)	(μs)	clusters
1024-bit	235486	206955	1280	1424	250.00	0.0904	10
2048-bit	220697	204460	1152	1424	250.00	0.3422	9
4096-bit	230636	213670	1152	1568	250.00	1.2537	9
8192-bit	244621	226521	1152	1568	250.00	4.7895	9

Next, for measuring the performance of GCD computation accurately, we implement the hierarchical GCD cluster to compute all pairs of moduli stored in an off-chip DDR3 memory MT8JTF12864HZ-1G6G1 [22] equipped in VC707 evaluation board [19]. Unfortunately, if the used resources of FPGA is close to the available number, the circuit of FPGA becomes unstable and can not compute the results correctly when it is actually operated in the evaluation board. According to the experimental results, 10 clusters can be implemented in the FPGA clocked at 250MHz for pairwise GCD computation of 1024-bit RSA moduli. In other words, 1280 GCD processor cores can be implemented in FPGA XC7VX485T-2 equipped in VC707 evaluation board, and works in parallel to compute GCDs of all pairs of 1024-bit RSA moduli stored in the off-chip DDR3 memory.

We use the built-in CORE Generator software of Xilinx Vivado design suite 2015.1 to generate a DDR3 memory interface core in the FPGA to control the write and read operations of the DDR3 memory. The DDR3 memory consists of 8 banks. Each bank has a $2^{14} \times 2^{10}$ memory array, of which each element has 64-bit. In other words, each bank of the DDR3 memory can store up to $(2^{14} \times 2^{10} \times 64\text{bits})/1024\text{bits} = 1048576$ 1024-bit RSA moduli. The DDR3 memory runs in 500MHz that is 2 times faster than the FPGA. Moreover, the DDR3 memory offers high-speed data transfers on the rising and falling edges of the clock of it. Hence, the DDR3 memory can perform $500\text{MHz}/250\text{MHz} \times 2 = 4$ times write or read operations in one clock cycle of the FPGA. Hence, we can read $4 \times 64\text{bits} = 256\text{bits}$ data from DDR3 memory in one clock cycle of the FPGA. Suppose that we have a lot of 1024-bit RSA moduli collected from the Web, we divide all moduli into two sets and store them to two different banks of the DDR3 memory. We partition all moduli of each set into groups with $71 \times 8 = 568$ moduli each. FPGA picks one group from each set and sends them to the central server, respectively. More specifically, we send read commands to the DDR3 memory interface core for reading a 1024-bit modulus. Then, the interface core performs the read operation of the DDR3 memory and the modulus is transferred to FPGA after a few clock cycles. The obtained 1024-bit modulus is then stored to the block RAMs of central server as 18-bit words in 57 clock cycles, and we read the next 1024-bit modulus at the same time. The same operation is repeated until two groups of 568 moduli are stored in the central server. Moreover, the interface core processes a refresh operation to maintain the data of the DDR3 memory in refresh interval, and other operations of DDR3 memory must wait for the refresh operation. By implementing the hierachical GCD cluster with 1280 processor cores in the FPGA, we have that it takes 7294417 clock cycles compute the GCDs of $568 \times 568 = 322624$ pairs, where 71646 clock cycles for transferring 568×2 1024-bit moduli from DDR3 memory to the central server of the FPGA is included. Comparing with the total clock cycles for computing the GCDs of 322624 pairs of 1024-bit moduli, the clock cycles for transferring moduli from DDR3 memory to central server is negligible. Moreover, after all moduli of central server are transferred to the clusters, we can read the next two groups with 568 moduli each from DDR3 memory while the GCD computation of the clusters is still being performed. In other words, the operation of transferring moduli from DDR3 memory to central server can be overlapped. Hence, we note that the transfer time is not significant. Since the hierachical GCD cluster runs in 250MHz, the computing time is 7294417/250MHz = 29.178ms. Therefore, we say that one GCD can be computed in 29.178ms/322624 = 0.0904μs. For performing pairwise GCD computation of 2048-bit, 4096-bit, and 8192-bit moduli, we have succeeded in implementing the hierachical GCD cluster that has 9 clusters in the FPGA, where the frequency of FPGA is also 250MHz. The implementation results of hierarchical clusters and computing time for one GCD of 1024-bit, 2048-bit, 4096-bit, and 8192-bit moduli is also shown in Table 4. The hierarchical GCD cluster is designed based on FDFM GCD processors that are compact and use very few FPGA resources. One of the advantage of the FDFM approach is that we can implement multiple FDFM processors working in parallel to reduce the computing time if enough hardware resources are available. Comparing with single FDFM GCD processor core, the computing time of the hierachical GCD cluster for one GCD reduces considerably by employing more than one thousand FDFM GCD processor cores.

According to the implementation results as shown in Table 4, the hierachical GCD cluster computes one GCD of two 8192-bit moduli in 4.7895μs, that is 52.98 times slower than the time for computing one GCD of two 1024-bit moduli. We show the reason of the large difference. Since the large input numbers are stored in the block RAM as 18-bit words and processed word by word, if the width of the input numbers increases, the number of iterations of the do-while loop of the Hardware Binary Euclidean algorithm will increase. Also, the clock cycles for performing each iteration of the do-while loop will increase. Hence, the proposed GCD processor takes more clock cycles for computing one GCD of larger numbers. For example, as shown in Table 2, single proposed processor takes 1381328.5 clock cycles for computing one GCD of two 8192-bit moduli, that is 49.32 times more than that for computing one GCD of two 1024-bit moduli. This is the main reason for the large difference of the computing time for 1024-bit and 8192-bit moduli. Recall that each 1024-bit and 8192-bit modulus is stored in the block RAM as 57 and 456 18-bit words, respectively. Hence, the central server and cluster server take more time for transferring the 8192-bit moduli than that for 1024-bit moduli. However, since the data transfer is overlapped with the GCD computation, the transfer time does not significantly affect the large difference of the computing time for 1024-bit and 8192-bit moduli. Moreover, the number of clusters for 8192-bit moduli is 9, that is less than that for 1024-bit moduli. Based on the reasons above, one GCD of two 8192-bit moduli is computed 52.98 times slower than one GCD of two 1024-bit moduli in the implementation of hierarchical GCD cluster.

6 Conclusions

We have presented an efficient processor core for computing GCDs of very large numbers. Since the processor is designed based on the FDFM approach, each processor core uses only one DSP slice and one 18k-bit block RAM. We implement the hierarchical GCD cluster with 1280 processor cores in Xilinx FPGA XC7VX485T-2. The implementation with 1280 processor cores executes pairwise GCD computation for 1024-bit RSA moduli stored in an off-chip DDR3 memory on Xilinx VC707 evaluation board. The experimental results shows that our implementation of 1280 GCD processor cores computes one GCD of two 1024-bit RSA moduli in 0.0904μs including the time of data transferring from off-chip DDR3 memory to FPGA. It is 3.8 times faster than the best GPU implementation and 316 times faster than a sequential implementation on the Intel Xeon CPU.

References

[1] Xilinx Inc., *7 Series DSP48E1 Slice User Guide*, Nov. 2014.

[2] ——, *7 Series FPGAs Memory Resources User Guide*, Nov. 2014.

[3] 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.

[4] 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.

[5] X. Zhou, Y. Ito, and K. Nakano, “An efficient implementation of the Hough transform using DSP slices and block RAMs on the FPGA,” in *Proc. of IEEE 7th International Symposium on Embedded Multicore Socs*, 2013, pp. 85–90.

[6] Y. Ago, K. Nakano, and Y. Ito, “A classification processor for a support vector machine with embedded DSP slices and block RAMs in the FPGA,” in *Proc. of IEEE 7th International Symposium on Embedded Multicore Socs (MCSoC)*, 2013, pp. 91–96.

[7] X. Zhou, Y. Ito, and K. Nakano, “An efficient implementation of the gradient-based Hough transform using DSP slices and block RAMs on the FPGA,” in *Proc. of IEEE International Parallel and Distributed Processing Symposium Workshops*, 2014, pp. 762–770.

[8] K. Hashimoto, Y. Ito, and K. Nakano, “Template matching using DSP slices on the FPGA,” in *Proc. of International Symposium on Computing and Networking*, 2013, pp. 338–344.

[9] X. Zhou, Y. Ito, and K. Nakano, “An efficient implementation of the one-dimensional Hough transform algorithm for circle detection on the FPGA,” in *Proc. of International Symposium on Computing and Networking*, 2014, pp. 447–452.

[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] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, pp. 120 – 126, 1978.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, 1997.

[13] R. Devi, J. Singh, and M. Singh, “VHDL implementation of GCD processor with built in self test feature,” *International Journal of Computer Applications*, vol. 25, no. 2, pp. 50–54, July 2013.

[14] S. D. Kohale and R. W. Jasutkar, “Power optimization of GCD processor using low power Spartan 6 FPGA family,” *International Journal of Conceptions on Electronics and Communication Engineering*, vol. 2, no. 1, pp. 1–6, June 2014.

[15] N. Fujimoto, “High throughput multiple-precision GCD on the CUDA architecture,” in *International Symposium on Signal Processing and Information Technology*, Dec 2009, pp. 507–512.

[16] K. Scharfglass, D. Weng, J. White, and C. Lupo, “Breaking weak 1024-bit RSA keys with CUDA,” in *Internatinal Conference of Breaking weak 1024-bit RSA keys with CUDA*, Dec 2012, pp. 207 – 212.

[17] J. R. White, “PARIS: A PARALLEL RSA-PRIME INSPECTION TOOL,” Ph.D. dissertation, California Polytechnic State University - San Luis Obispo, June 2013.

[18] T. Fujita, K. Nakano, and Y. Ito, “Bulk GCD computation using a GPU to break weak RSA keys,” in *International Parallel and Distributed Processing Symposium Workshops*, May 2015.

[19] Xilinx Inc., *VC707 Evaluation Board for the Virtex-7 FPGA*, Sept. 2015.

[20] X. Zhou, Y. Ito, and K. Nakano, “A parallel FDFM approach for breaking weak RSA keys using the FPGA,” in *Proc. of International Conference on Parallel Processing and Applied Mathematics*, Sept. 2015.

[21] Xilinx Inc., *7 Series FPGAs Configurable Logic Block User Guide*, Nov. 2014.

[22] Micron Inc., *1GB, 2GB, 4GB (x64, SR) 204-Pin DDR3 SODIMM*, May 2015.