2012 IEEE 14th International Conference on High Performance Computing and Communications

ASIP-based Design and Implementation of RSA for Embedded Systems

Zhongbo Wang, Zhiping Jia, Lei Ju, Renhai Chen
School of Computer Science and Technology, Shandong University,
Shandong Provincial Key Laboratory of Software Engineering, Jinan, Shandong, China
wzb@mail.sdu.edu.cn, jzp@sdu.edu.cn, julei@sdu.edu.cn

***Abstract*—The RSA public-key cryptosystem is widely used to provide security protocols and services in the network communication. However, design and implementation of the RSA cryptosystem to meet the real-time requirements of embedded applications are challenging issues, due to the computation intensive characteristics of the RSA arithmetic operations and the limited resources in the embedded systems. Various implementation and optimization methods have been proposed for RSA algorithm. However, software execution of RSA on general-purpose processors usually suffers from slow execution speed; while application-specific integrated circuit (ASIC) based approaches are lack of flexibility. In this work, we present a systematic design approach of application-specific instruction-set processor(ASIP) for the RSA cryptographic algorithm. We identify and optimize the custom instructions in the RSA algorithm, and extend the instruction set architecture (ISA) of a standard 32-bit RISC processor to accommodate them. We employ the Electronic System Level (ESL) methodology in the development of the proposed ASIP in the Xilinx Virtex5 LX110T FPGA platform. Compared to the original RISC ISA, our extended ASIP achieves approximate 2.69 times performance improvement with only 25.6% more resource required.**

***Keywords*-RSA; ASIP; ESL; FPGA;**

I. Introduction

Nowadays, cryptography systems have been taking an important role in the security of the current Internet and the Internet of Things. Without them, especially the public-key cryptosystems, we could not purchase goods so safely and conveniently over the Internet . The RSA cryptosystem has became one of the most widely used public-key cryptosystems, since it was firstly proposed by Rivest, Shamir and Adleman of MIT in 1977. It is considered to be an efficient, high-quality public-key cryptographic algorithm. Furthermore, it doesn't have the key distribution problem which is common in the symmetric-key cryptosystems. The RSA algorithm can also be used to generate the unforgeable signature that is very useful in electronic commerce, since it is not only a trap-door one way function, but also a trap-door one way permutation [18]. As we know, the symmetric-key cryptographic algorithm is more efficient in the data encryption than the public-key cryptographic algorithm, but should get the shared private key in advance. Therefore, the RSA public-key algorithm can be used to transmit the key of the symmetric-key cryptographic algorithm, such as Advanced Encryption Standard (AES) algorithm, as a bootstrap in a hybrid cryptosystem. RSA has been successfully integrated into many modern security protocols such as SSL/TLS, WTLS, IKE, etc.

The security of the RSA cryptosystem relies on the difficulty of factoring the product of two large primes. The product, called modulus in the RSA cryptosystem, is usually recommended to have a size of 1024 binary bits or more to guarantee the security of the system. As to be described in Section II, the arithmetic operations in the RSA cryptosystem is large integer modular arithmetic operations, including modular addition, modular substraction, modular multiplication and modular exponentiation. Among these arithmetic operations, modular exponentiation is the core operations, as well as the most time-consuming part in the algorithm, due to its computation-intensive nature. Modular exponentiation of such large integers in the RSA cryptosystem cannot be easily and efficiently implemented in the resource-limited embedded environment, such as smart cards, sensor nodes and other embedded devices with modest processing power. Therefore, efficient implementation of the RSA cryptosystem plays a significant role in providing a security embedded environment for the embedded network communication.

A considerable number of methods of the efficient software implementation of the RSA algorithm have been proposed ([12]). Methods that use the Chinese Remainder Theorem (CRT) to accelerate the decryption of the RSA algorithm were proposed in [10], [17], which are suitable for the disproportion computing environment, such as a powerful server with the low performance embedded device clients. Various efficient exponentiation algorithms such as binary squaring and multiplication method, m-ary method, sliding-window method, addition chain method etc, were studied in [7], [14]. Montgomery proposed an algorithm for modular multiplication avoiding the trial division [15], which is suitable for both software and hardware efficient implementation for modular multiplication in RSA. Several variants of Montgomery modular multiplication algorithms were studied in [8], [11], [14].

Application-specific integrated circuit (ASIC) based implementations of RSA with Montgomery algorithm and its variants were studied in [4], [9], [16], [21], [22]. To large extent, the forementioned methods have speeded up the RSA cryptographic algorithm. However, hardware-based

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.202

implementation of RSA is not flexible enough to support configurability and scalability. Furthermore, the hardware implementations gain high speedup in the cost of increasing the chip resources and areas, which is not always possible in the resource-restricted embedded system environment.

In the present paper, we propose an application-specific instruction-set processor (ASIP) based design and implementation of RSA algorithm, which combines the advantages of both the software implementations and the hardware implementations. A security ASIP design was presented in [13] for RSA/ECC algorithms. Compared with [13], we adopt a multiple-input and multiple-output (MIMO) approach in our custom instruction generation and exploration, which improves the density and performance of the generated code. The instruction set architecture (ISA) of a standard 32-bit RISC processor is extended to accommodate the selected custom instructions. We employ the Electronic System Level (ESL) methodology in the development of the proposed ASIP in the Xilinx Virtex5 LX110T FPGA platform. Compared to the original RISC ISA, our extended ASIP achieves approximate 2.69 times performance improvement with only 25.6% more resource required.

Section II describes the details of the RSA cryptosystem and its elaborate implementation in our work. Section III briefly describes the overview of design methodology and custom instructions selection algorithm. Section IV briefly describe the the architecture of our designed ASIP for RAS and its instruction set, and Section V describe the final selected custom instruction of our RSA ASIP in more detail. We describe the process of the selection of the final custom instruction set and the experimental result in Section VI. Finally, Section VII presents our conclusion.

II. The RSA Algorithm and Its Implementation

In this section, we briefly describe the RSA algorithm and its implementation. The overview of RSA cryptosystem is depicted as Figure 1.

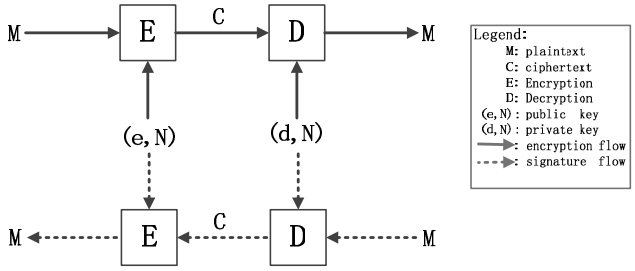

Figure 1. The RSA Cryptosystem

A. The details of RSA Algorithm

RSA can not only be used for data encryption like other public-key cryptographic algorithms, but also can be used for signature and authentication. This property is based on that the RSA algorithm is not only a trap-door one way function but also a trap-door one way permutation [18]. In the following subsections, we brief the details of the RSA Algorithm in data encryption and signature.

1) Data Encryption: RSA is a block cryptographic algorithm. Therefore, a lengthy plaintext message should be divided into segments of appropriate length, called blocks. Such a message block can be presented as an integer number. In order to obtain the ciphertext of a given message block valued M, the encryption is done as Equation (1).

$$C = M^e \pmod N \tag{1}$$

where N is the common RSA modulus, a large integer with a size of 1024 bit at least; e is the RSA key encryption exponent; M is the plaintext; C is the encrypted ciphertext; both C and M are integers in $[0,\ N-1]$.

In order to obtain the original plaintext message M, the encrypted message C only needs to be decrypted as Equation (2).

$$M = C^d \pmod N \tag{2}$$

where d is the RSA key decryption exponent. N, M and C are defined as above in Equation (1). As above defined in the Equation (1) and Equation (2), the public key is the pair $(e,\ N)$, and the private key is the pair $(d,\ N)$. Therefore, $(e,\ N)$ is made public and d is kept secret in the RSA cryptosystem. The common modulus N in the keys is the product of two large precomputed random primes, p and q. The security of the RSA cryptosystem relies on the difficulty of factoring N. Therefore, the binary bit length of N is recommended to 1024 at least (or more) nowadays to guarantee the validity of the cryptographic system.

The public key exponent e is a random generated number, and satisfies:

$$gcd(e,\ \phi(N)) = 1 \tag{3}$$

(gcd means great common divisor, $\phi(\ )$ is Euler's totient function and $\phi(N) = (p-1) * (q-1)$).

The private key exponent d can be calculated by:

$$d = e^{-1} \pmod{\phi(N)} \tag{4}$$

meaning d is the e's multiplicative inverse modulo $\phi(N)$. We can see e and d are coprime to $\phi(N)$, respectively.

2) Signature: Utilizing the RSA cryptosytem to make a signature for a message, we only need to encrypt the message using Equation (2) with the private key $(d,\ N)$. Then, any recipient can recover and verify the message using Equation (1) with the public key $(e,\ N)$.

Through the above description of the RSA algorithm, we can see the core arithmetic operations of RSA are large integer modular exponentiation operations. These large multiprecision integer modular exponentiation arithmetic operations are so compute-intensive that they are the main time-consuming part of the RSA algorithm and the focus

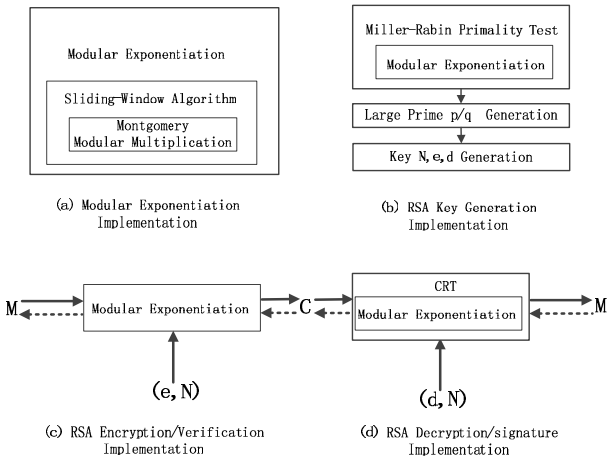

Figure 2. Implementation of RSA Algorithm

of effective implementation. In Subsection II-B, we give the details of the implementation of the modular exponentiation in our work.

B. The Implementation of RSA

The overview of the implementation of RSA Algorithm in our work is depicted in Figure 2. The initial step of the implementation is key generation. In this step, we employ Miller-Rabin primality test method to generate the large prime integer p and q as Figure 2 (b). The encryption and decryption are the core of the RSA cryptosystem and both of them are based on modular exponentiation as Figure 2 (c) and (d). Many effective implementation methods focus on the implementation of the large multiprecision integer modular exponentiation.

Algorithm 1 $MontMul(X, Y, N, N')$ — Montgomery modular multiplication Algorithm.

Require: input: integers $N = (n_{t-1} \cdots n_1 n_0)_b$, $X = (x_{t-1} \cdots x_1 x_0)_b$, $Y = (y_{t-1} \cdots y_1 y_0)_b$, with $0 \leq X, Y < N$, $R = b^t$ with $gcd(N, b) = 1$, and $N' = -N^{-1} \ mod \ b$.
Ensure: output: $X * Y * R^{-1} \ mod \ N$.
1: $A := 0$; (Notation: $A = (a_t a_{t-1} \cdots a_1 a_0)_b$)
2: **for** i from 0 to $t-1$ **do**
3: $z := (a_0 + x_i * y_0) * N' \ mod \ b$; (Notation: z is a digit of radix b)
4: $A := (A + x_i * Y + z * N)/b$;
5: **end for**
6: **if** $A \geq N$ **then** $A := A - N$;
7: **Return**(A);

Large integer modular exponentiation can be implemented using modular multiplication. The exponentiation arithmetic operations of large multiprecision integer can be implemented by using binary squaring and multiplication method, k-ary method, sliding-window method, etc. Montgomery modular multiplication can be used to calculate the modular multiplication efficiently. The high radix Montgomery modular multiplication algorithm are described in **Algorithm 1**. There are two ways to reduce the computing time required to calculate exponentiation. One way is to decrease the time of the multiplication of two integers; the other is to reduce the number of multiplications used in the exponentiation operations. If the number of binary bits of the exponent is smaller, then the modular exponentiation becomes possibly faster. Therefore, the Chinese Remainder Theorem can be utilized to speed up the modular exponentiation [17] in the scenario of decryption or signature. The k-ary method and the sliding-window method can reduce the number of modular multiplication operations to promote a certain improvement. These methods can be flexibly implemented in a software way. Besides, the modular multiplication can also be easily and efficiently implemented using Montgomery multiplication with our dedicated instruction set extension processor. The implementation of the modular exponentiation in our work using sliding-window method and Montgomery modular multiplication is described as **Algorithm 2**. In the next section, the methodology of the design and implementation of our dedicated processor will be described in detail.

Algorithm 2 $ModExp(M, e, N, N', wsize)$ — Sliding-Window Modular Exponentiation Algorithm.

Require: input: integers $M = (m_{t-1} \cdots m_1 m_0)_b$, $e = (e_q \cdots e_1 e_0)_2$ with $e_q = 1$, $N = (n_{t-1} \cdots n_1 n_0)_b$, with $0 \leq M, < N$, $0 < e < N$; $R = b^t$ with $gcd(N, R) = 1$ and $gcd(N, b) = 1$; $N' = -N^{-1} \ mod \ b$ and $RR = R^2 \ mod \ N$; and $wsize$ is the sliding window size.
Ensure: output: $M^e \ mod \ N$.
1: precomputation:{
2: $W_1 := MontMul(M, RR, N, N')$;
3: $j := 2^{wsize-1}$;
4: $W_j := W_1$;
5: **for** i from 1 to $wsize - 1$ **do**
6: $W_j := MontMul(W_j, W_j, N, N')$;
7: **end for**
8: **for** i from $j + 1$ to $2^{wsize} - 1$ **do**
9: $W_i := MontMul(W_{i-1}, W_1, N, N')$;
10: **end for**
11: }
12: $A := MontMul(RR, 1, N, N')$;
13: $i := q$;
14: **while** $i \geq 0$ **do**
15: **if** $e_i = 0$ **then** { $A := MontMul(A, A, N, N')$; $i := i - 1$;}
16: **else** $(e_i \neq 0)$ **do :**
17: { **if** $(i + 1) < wsize$ **then** break;
18: find the sliding windiow size bitstring $e_i e_{i-1} \cdots e_{i-wsize+1}$;
19: $p := (e_i e_{i-1} \cdots e_{i-wsize+1})_2$; $i := i - p$;
20: **for** k from 1 to $wsize$ **do**
21: $A := MontMul(A, A, N, N')$;
22: **end for**
23: $A := MontMul(A, W_p, N, N')$;
24: }
25: **end while**
26: **for** k from i to 0 **do**
27: $A := MontMul(A, A, N, N')$;
28: **if** $e_i \neq 0$ **then** { $A := MontMul(A, W_1, N, N')$;}
29: **end for**
30: $A := MontMul(A, 1, N, N')$;
31: **Return**(A);

III. METHODOLOGY

In this section, we describe the design methodology used in our work. An application-specific instruction-set processor (ASIP) is a processor optimized for a particular application. A series of researches have been studied in the design of ASIP and many valued methods are put forth in [1], [2],

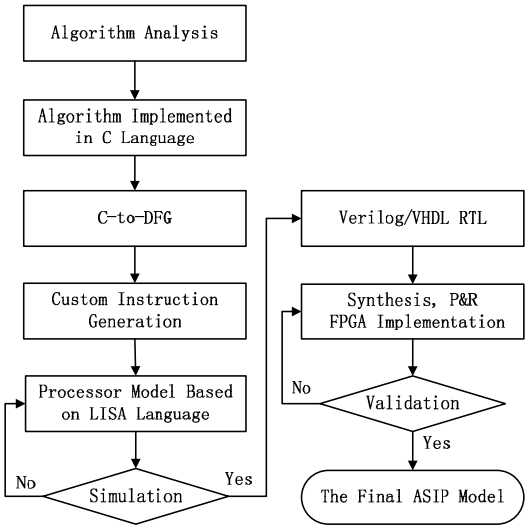

Figure 3. Design flow of ASIP based on ESL

[3], [5], [6], [19], [20]. In our work, the overall design flow of ASIP for RSA is illustrated in Figure 3. Firstly, we can obtain some helpful information through the analysis of the algorithm to guide the selections of the custom instructions, such as the timing-consuming parts and the repeat used parts. Secondly, the algorithm is implemented in the C language. Then, we can get the data flow graph (DFG) of the algorithm with its C implementation. Many selection approaches for custom instruction based on DFG have been proposed. In the process of the custom instruction generation, we employ a variant of the MM-LEVEL algorithm proposed in [6]. Such an example of discovering the specific muladd instruction is illustrated in Figure 4. The core idea of MM-LEVEL is identifying the MAXMISOs (maximal multiply-input-single-output block) from the original DFG of the algorithm, then collapsing DFG using the new MAXMISO node in place of the subgraph of DFG selected as MAXMISO and combining available MAXMISO nodes at the same level as MIMO. In order to satisfy the constrains, ultimately, this MM-LEVEL algorithm can be solved as an ILP problem. With the selected custom instructions, we can construct the processor model in the Language for Instruction-set Architectures (LISA). The preliminary simulation of the processor model should be done until all the requirements are satisfied. Then, the simulated processor model is translated into the Verilog/VHDL description in RTL for hardware synthesis. Ultimately, an application-specific processor is implemented in the FPGA platform through the process of simulation and validation.

The core part of our ASIP design method is custom instruction selection. A great many candidate custom instructions can be identified by the MM-LEVEL Algorithm. However, not all of these candidate instructions can be implemented in the final ASIP with limit resource, frequency requirement, etc. Through the high level analysis of the specific algorithm, the code module usage ratio can be obtained

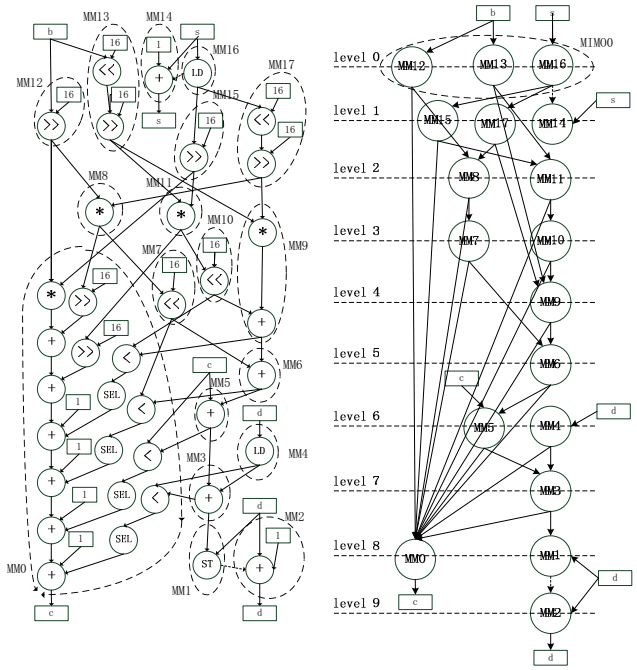

Figure 4. the MM-level of the muladd DFG

to guide the selection of the candidate custom instructions. Meantime, the rules of the custom instruction design also require the custom instructions designed to be compatible with the instruction in original instruction set. If both new custom instructions and the original instructions have the unified instruction format, the complexity of the architecture design can be simplified, to some degree. The execution time of the custom instruction should not exceed the original instruction cycle to accommodate pipeline. Considering the various kinds of factors, we draw the following rules to design the custom instruction in our ASIP design.

1) Unified instruction format: The new custom instruction and the original instruction should have the same instruction word length and should use the same bit sequences of the instruction word to encode the instruction opcode.

2) Keeping system frequency: The new custom instruction should be executed in the system pipeline as the original instruction without underclocking. The hardware implementation of the custom instruction execution unit should not be too complex. the execution time of the custom instruction should not exceed the original instruction execution cycles, otherwise, to underclock of the system to accommodate the new custom instruction without breaking the pipeline, but degrade the performance of the whole system significantly.

3) The limit of the number of custom instruction: One of the motivations of ASIP is to use less resources but better performance. The number of the custom instructions in the ISA of the ASIP should not be too many in order to reduce the hardware resource usage and the complexity of the whole

system.

the above rules can give some assistance in our initial custom instruction selection. We can get a lot of candidate custom instructions employing the forementioned methods. Utilizing these candidate custom instructions, we can construction many candidate processor models. In order to determine the final processor model, we need to explore all kinds of candidate processor models, and implement them to obtain all aspects of experimental parameters data of these processors, such as resource usage, system frequency, etc. Meantime, the specific algorithm implementation should be run in such specific processors to gain the the instruction cycles and execution time. Through quantifying these data of all the candidate processors, the final custom instruction set can be selected in the final processor model, with the objective of higher performance and less resource usage.

IV. Architecture of ASIP for RSA

In this section, we brief the architecture of our designed ASIP for RAS. The overall architecture of our RSA processor is described in Figure 5. It overall contains 32K

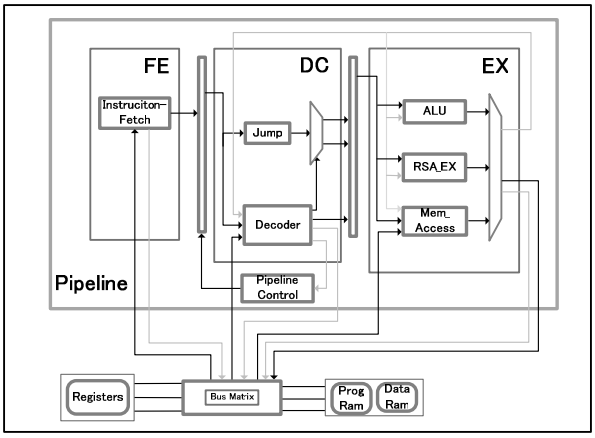

Figure 5. Architecture of ASIP for RSA

Data RAM, 32K Prog-RAM, Registers, a simple three stage pipeline and a bus matrix. Both the Data RAM and the Prog-RAM can be addressed from $0x0000$ to $0x7FFF$. The independent Data RAM and Prog-RAM allow the architecture to access the data and the instructions at the same time to gain Higher parallelism. Apart from 32 general purpose registers, one Fetch Program Register (FPR), one Stack Pointer Register (SPR) and one Linker Register (LR) are used in this dedicated processor, too. A simple three stage pipeline including FE, DC and EX stages, is employed in the execution part of our designed architecture. The FE and DC units are nearly same as the original core without increasing the resource almost, as our customer instructions are formed in the unified instruction form as the original instructions. The pipeline control unit is used to deal with the jump instruction. When the jump instruction is fetched in an executing program, FPR contains the targeted address. At the same time, pipeline control unit flushes the EX stage to avoid an error occurrence. In the EX stage of the pipeline, a dedicated execution unit for RSA specific instructions is provided. The Bus Matrix is used to connect all the units of the architecture and maintain their communication, especially useful in bus contention situation. It can largely improved the whole architecture performance when intensive data and instructions concurrent accesses lead to fierce bus contentions.

Table I
THE RSA PROCESSOR ISA

General Purpose Instructions	add, sub, mul, and, or, xor, sll, srl, sra
Jump Instructions	bne, beq, blt, ble, bgt, bal, j, jr
Memory Access Instructions	lw, lh, lhu, lb, lbu, sw, sh, sb
RSA Specific Instructions	getbit, shift_l, shift_r, adc, sbb, muladd, muladd2

In our final dedicated processor, the whole instruction set contains 32 instructions listed in Table I. 7 RSA specific instructions are identified from our implementation of the algorithm after rigorous selections, which can gain higher performance but use less resource , relatively.

V. Micro-Architecture of RSA Specific Instructions

In this section, we briefly describe our identified custom instructions for RSA. Firstly, we give the format of these dedicated instructions in Table II. These specific instructions with unified instruction format are completely compatible with the instruction set of the original processor model. Therefore, we don't need special decoder for our identified custom instructions in the pipeline design of our processor model. As Section III mentioned, we employ an ESL based method in the process of our dedicated processor design. Then we discuss the details of the micro-architecture of each specific instruction in the following.

A. adc instruction and sbb instruction

In modular exponentiation operation, modular addition and modular subtraction are two kinds of the most common arithmetic operations. In the original processor model without the addition with carry (adc) instruction and the subtraction with borrow (sbb) instruction, when processing the addition (subtraction) chain, the carry (borrow) bit must be carefully considered. Utilizing our specific adc and sbb instructions, we can easily cope with addition (substraction) chain using a dedicated register to save the carry (borrow) bit as common addition (substraction). Their original pseudo C code is illustrated in Figure 6.

Table II
THE SPECIFIC INSTRUCTIONS FOR RSA

Instruction	Format					
adc	opcode	src1	src2	dest	xxxxxxxxxx	
	31 26	25 21	20 16	15 11	10	0
sbb	opcode	src1	src2	dest	xxxxxxxxxx	
	31 26	25 21	20 16	15 11	10	0
getbit	opcode	src1	src2	dest	xxxxxxxxxx	
	31 26	25 21	20 16	15 11	10	0
shift_l	opcode	src1	src2	dest	xxxxxxxxxx	
	31 26	25 21	20 16	15 11	10	0
shift_r	opcode	src1	src2	dest	xxxxxxxxxx	
	31 26	25 21	20 16	15 11	10	0
muladd	opcode	src1	src2	dest	xxxxxxxxxx	
	31 26	25 21	20 16	15 11	10	0
muladd2	opcode	src1	src2	src3	dest	xxxxx
	31 26	25 21	20 16	15 11	10 6	5 0

```
u32 R0, R1, R2, R3 = 0;

R0 += R1;
R3 += (R0 < R1) ? 1 : 0;

R0 += R2;
R3 += (R0 < R2) ? 1 : 0;

R2 = R3;
```

(a) adc pseudo C code

```
u32 R0, R1, R2, R3 = 0;

R3 += (R0 < R1) ? 1 : 0;
R0 -= R1;

R3 += (R0 < R2) ? 1 : 0;
R0 -= R2;

R2 = R3;
```

(b) sbb pseudo C code

Figure 6. pseudo C code of adc and sbb

However, using our custom instructions adc and sbb to implement such operations, we just need one instruction. The addition with carry instruction,
$adc\ R0, R1, R2 \quad ; (R2, R0) = R0 + R1 + R2$
has the same function as Figure 6 (a). Similarly, the substraction with borrow instruction,
$sbb\ R0, R1, R2 \quad ; (R2, R0) = R0 - R1 - R2$
implement the function as Figure 6 (b).

After synthesis process, the micro-architectures of adc and sbb are shown in Figure 7 and Figure 8, respectively.

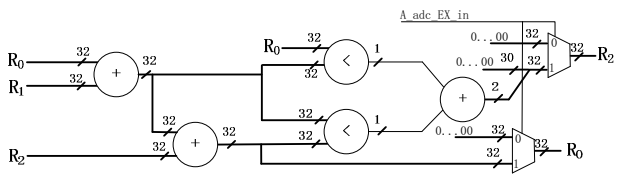

Figure 7. Micro-Architecture of adc instruction

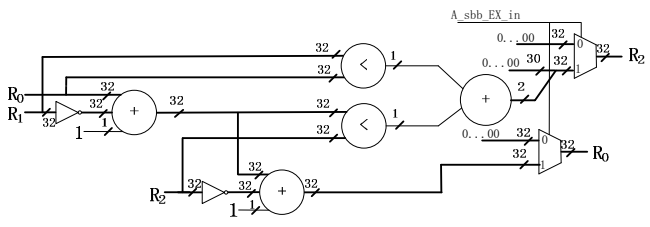

Figure 8. Micro-Architecture of sbb instruction

B. getbit instruction

In modular exponentiation operations, the bit operations of the exponent are frequently used as in Section II Algorithm 2. Furthermore, our method also identify such a specific instruction $getbit$. Its pseudo C code is
$R2 = (R0 >> R1)\ and\ 0x1$.
In our original processor model, its assembly code is 3 instructions,
$add\ \ R2,\ R0,\ 0$
$srl\ \ R2,\ R2,\ R1$
$and\ \ R2,\ R2,\ 0x1$.
We can implement the same function in one instruction $getbit$ of our extended instruction set. With the specific instruction $getbit$, its equivalent assembly code is
$getbit\ \ R0,\ R1,\ R2$.
After being taken into the design flow, its micro-architecture is given in Figure 9.

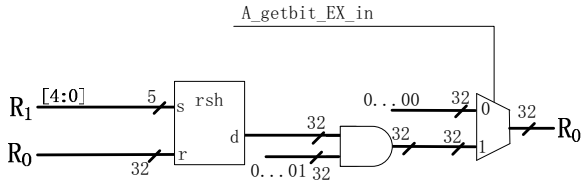

Figure 9. Micro-Architecture of getbit instruction

C. shift_l instruction and shift_r instruction

Our method also identifies two specific instructions (shift_l, shift_r) for bit shift operations. The original bit shift operations only simply discard the overflow bits, however, these bits are very useful in our multiple precision integer arithmetic operations. Their pseudo C code is depicted as Figure 10.

```
u32 R0, R1, R2, R3 = 0;

R3 = R0 >> (32-R1);
R0= (R0 << R1) | R2
R2 = R3;
```

(a) the original shift_l pseudo C code

```
u32 R0, R1, R2, R3 = 0;

R3 = R0 << (32-R1);
R0 = (R0 >> R1) | R2
R2 = R3;
```

(b) the original shit_r pseudo C code

Figure 10. The pseudo C code of shift_l and shift_r

With our custom instruction set, only use one instruction to implement such functions, respectively. Instruction
$shift\_l\ \ R0,\ R1,\ R2$
has the equivalent function as Figure 10 (a). Similarly, the function of Figure 10 (b) can be implemented by one instruction
$shift\_r\ \ R0,\ R1,\ R2$.

After synthesis, Their micro-architectures are illustrated as Figure 11 and Figure 12, respectively.

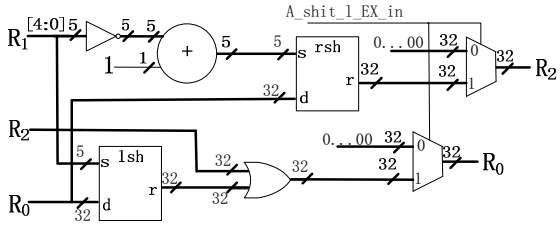

Figure 11. Micro-Architecture of shift_l instruction

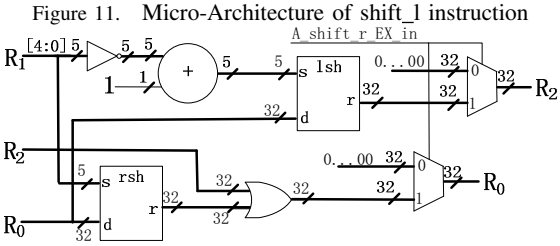

Figure 12. Micro-Architecture of shift_r instruction

D. muladd instruction and muladd2 instruction

As forementioned, the core operations of RSA are modular exponentiation and the main operations of the modular exponentiation are modular multiplication. In the modular multiplication implementation, single precision multiplication and addition are the main arithmetic operations. We give the C code implementation of the $muladd$ and $muladd2$ operations in Figure 13.

```
u32 R0, R1, R2;

part0 = R0 & 0xFFFF;
part1 = R0 >> 16;
part2 = R1 & 0xFFFF;
Part3 = R1 >> 16;

part4 = R2;

mult0 = part0 * part2;
mult1 = part0 * part3;
mult2 = part1 * part2;
mult3 = part1 * part3;

half0 = mult1 >> 16;
half1 = mult2 >> 16;

mult3 +=  half0;
mult3 += half1;

half3 = mult1 << 16;
half4 = mult2 << 16;

mult0 += half3;
mult3 +=  mult0 < half3 ? 1 : 0;

mult0 += half4;
mult3 +=  mult0 < half4 ? 1: 0;

mult0 +=  part4;
mult3 +=  mult0 < part4 ? 1 : 0;


R0 = mult0;
R2 = mult3;
```

(a) the muladd C code

```
u32 R0, R1, R2, R3;

part0 = R0 & 0xFFFF;
part1 = R0 >> 16;
part2 = R1 & 0xFFFF;
Part3 = R1 >> 16;
part4 = R2;
part5 = R3;

mult0 = part0 * part2;
mult1 = part0 * part3;
mult2 = part1 * part2;
mult3 = part1 * part3;

half0 = mult1 >> 16;
half1 = mult2 >> 16;

mult3 +=  half0;
mult3 += half1;

half3 = mult1 << 16;
half4 = mult2 << 16;

mult0 += half3;
mult3 +=  mult0 < half3 ? 1 : 0;
mult0 += half4;
mult3 +=  mult0 < half4 ? 1: 0;
mult0 +=  part4;
mult3 +=  mult0 < part4 ? 1 : 0;
mult0 += part5;
mult3 +=  mult0 < part5 ? 1: 0;


R0 = mult0;
R2 = mult3;
```

(b) the muladd2 C code

Figure 13. the C code of muladd and muladd2

Instead of that, we can use one instruction to implement each function in our extended instruction set.

$$muladd \;\; R0, R1, R2 \qquad ;(R2, R0) = R0 * R1 + R2$$
$$muladd2 \;\; R0, R1, R2, R3 \;\; ;(R3, R0) = R0 * R1 + R2 + R3$$

After the synthesis, the micro-architectures of $muladd$ and $muladd2$ are illustrated in Figure 14 and Figure 15, respectively.

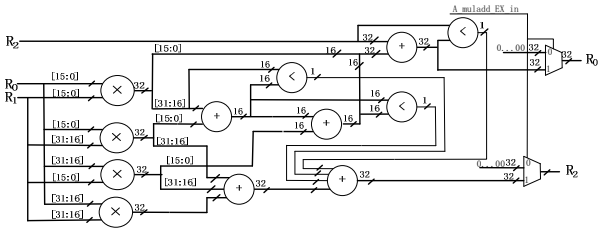

Figure 14. Micro-Architecture of muladd instruction

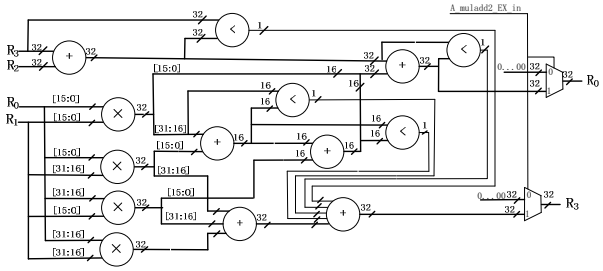

Figure 15. Micro-Architecture of muladd2 instruction

VI. Result and Discussion

In this section, We brief the method to determine the final processor model with the experiment data. Using the custom instruction selected method mentioned in III, many candidate custom instructions are selected and these instructions can be used to implemented a great many candidate processors. These candidate processors are implemented using FPGA technology, targeted to Virtex5 LX110T at frequency 200MHz. Then we run the RSA algorithm on every candidate processor and get the experimental parameter data of such specfic processor. Comparing with experimental result of the original processor, we form two performance indicators: $speedup$ and $efficiency$.

$$speedup(p_i) = T_0/T_i \tag{5}$$

(T_i means the execution time of the RSA algorithm on the processor i, p_0 and p_i are the original processor and the candidate processor i, respectively).

$$efficiency(p_i) = speedup(p_i)/(R_i/R_0) \tag{6}$$

(R_0 and R_i are resource usage of the original processor and the candidate processor i, respectively).

with the experimental data, the number of these candidate processors can be firstly reduced by $speedup$, and then among them, the processor with highest $efficiency$ will be our final processor model. In our work, the final processor with 7 RSA specific instructions mentioned in Section IV is selected. Its LUT count increases from 18234 to 22918 which represents only 25.6% more resource utilization. Its experimental result compared with that of the original processor is illustrated in Table III. It shows that our RSA processor can achieve average 2.69 times improvement. In the decryption, we use CRT to accelerate the process and obtain more improvement.

Table III
THE COMPARISON OF THE EXPERIMENTAL RESULTS

RSA	**execution time**		**cycles count**		**speedup**
Encryption	**original**	**extended**	**original**	**extended**	
1024	$19.01ms$	$6.79ms$	3801696	1356384	2.80
2048	$141.76ms$	$52.51ms$	28350336	10501240	2.70
RSA	**execution time**		**cycles count**		**speedup**
Decryption	**original**	**extended**	**original**	**extended**	
1024	$5.26ms$	$1.97ms$	1053621	383329	2.67
2048	$38.02ms$	$14.67ms$	7603625	2933891	2.59

VII. CONCLUSION AND REMARKS

In this paper, we propose a systematical method to design and implement an application specific instruction-set processor (ASIP) for RSA. Our implemented RSA processor not only achieves higher speed, but also has the flexibility of the software implementation and less resource requirement than the hardware implementation. Besides, other public-key encryption algorithms can also benefit from our dedicated processor. In the future, a high parallel architecture for RSA implementation will be explored.

VIII. ACKNOWLEDGEMENTS

This work is partly supported by the Natural Science Foundation of China (NSFC) under grant No.61070022 and 60903031, Shandong Provincial Natural Science Foundation No. ZR2010FM015, ZR2011FQ036 and Innovation Foundation of Shandong University No.2011TB018. .

REFERENCES

[1] K. Atasu, R.G. Dimond, O. Mencer, W. Luk, C. Ozturan, and G. Diindar. Optimizing instruction-set extensible processors under data bandwidth constraints. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2007.

[2] K. Atasu, C. Ozturan, G. Dundar, O. Mencer, and W. Luk. Chips: Custom hardware instruction processor synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):528–541, 2008.

[3] N. Cheung, S. Parameswaran, and J. Henkel. Inside: Instruction selection/identification & design exploration for extensible processors. In *IEEE/ACM international conference on Computer-aided design*, page 291, 2003.

[4] AP Fournaris and O. Koufopavlou. A new rsa encryption architecture and hardware implementation based on optimized montgomery multiplication. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 4645–4648, 2005.

[5] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(2):18, 2011.

[6] C. Galuzzi, E.M. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis. Automatic selection of application-specific instruction-set extensions. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 160–165, 2006.

[7] D.M. Gordon. A survey of fast exponentiation methods. *Journal of algorithms*, 27(1):129–146, 1998.

[8] J. Großschädl, R. Avanzi, E. Savaş, and S. Tillich. Energy-efficient software implementation of long integer modular arithmetic. *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 75–90, 2005.

[9] D. Harris, R. Krishnamurthy, M. Anders, S. Mathew, and S. Hsu. An improved unified scalable radix-2 montgomery multiplier. In *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pages 172–178, 2005.

[10] R.J. Hwang, F.F. Su, Y.S. Yeh, and C.Y. Chen. An efficient decryption method for rsa cryptosystem. In *Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on*, volume 1, pages 585–590, 2005.

[11] C. Kaya Koc, T. Acar, and B.S. Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *Micro, IEEE*, 16(3):26–33, 1996.

[12] C.K. Koc. High-speed rsa implementation. Technical report, Technical Report, RSA Laboratories, 1994.

[13] R. Lu, X. Zeng, J. Han, Y. Gu, and L. Mai. Design and vlsi implementation of a security asip. In *IEEE International Conference on ASIC*, pages 866–869, 2007.

[14] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of applied cryptography*. 1997.

[15] P.L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[16] S.B. Ors, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of a montgomery modular multiplier in a systolic array. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003.

[17] J.J. Quisquater and C. Couvreur. Fast decipherment algorithm for rsa public-key cryptosystem. *Electronics letters*, 18(21):905–907, 1982.

[18] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[19] H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, K. Karuri, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr. Asip architecture exploration for efficient ipsec encryption: A case study. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(2):12, 2007.

[20] F. Sun, S. Ravi, A. Raghunathan, and N.K. Jha. Custom-instruction synthesis for extensible-processor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):216–228, 2004.

[21] A. Tenca, G. Todorov, and Ç. Koç. High-radix design of a scalable modular multiplier. In *Cryptographic Hardware and Embedded SystemsłCHES 2001*, pages 185–201, 2001.

[22] A.F. Tenca and Ç.K. Koç. A scalable architecture for modular multiplication based on montgomery's algorithm. *Computers, IEEE Transactions on*, 52(9):1215–1221, 2003.