Hardware-Software Codesign of RSA for Optimal Performance vs. Flexibility Trade-off

Malik Umar Sharif,
Rabia Shahid and Kris Gaj
ECE Department
George Mason University
Fairfax, VA 20151
Email: {msharif2, rshahid, kgaj}@gmu.edu

Marcin Rogawski
Cadence Design Systems
San Jose, CA 95134
Email: mrogawsk@gmu.edu

***Abstract*—Public-key cryptosystems such as RSA have been widely used to secure digital data in many commercial systems. Modular arithmetic on large operands used during modular exponentiation makes RSA computationally challenging. Traditionally, software implementations of these algorithms provided the highest flexibility but lacked performance. On the contrary, custom hardware accelerators provided the highest performance but lacked flexibility and adaptability to changing algorithms, parameters, and key sizes. In this paper, we present a hardware/software codesign of RSA cryptosystem that improves performance, while retaining flexibility. We adopted Xilinx Zynq-7000 SoC platform, which integrates a dual-core ARM Cortex-A9 processing system along with Xilinx programmable logic. The software part of our implementation is based on RELIC library (Efficient Library for Cryptography). The performance vs. flexibility trade-off is investigated, and the speed-up of our codesign implementation vs. the purely software implementation of RSA on the same platform is reported. Our results show a speedup of up to 57 times when compared with the software implementation for 2048-bit operand size. We also propose a generic model for HW/SW codesign focused on flexibility with comparable performance to existing HW/SW implementations.**

I. Introduction and Motivation

Hardware/software codesign allows the designer to partition the design into hardware and software to aim for the best of both worlds. The flexibility and short development time of software is combined with performance and low-power/low energy consumption of hardware. RSA implementations based on software platforms offer flexibility to adapt to algorithm changes but are also slower as compared to hardware designs. To achieve speed, implementing the whole algorithm in hardware is often considered as a viable choice but at the expense of flexibility in the design.

Proper partitioning to find an optimum boundary between software and hardware requires detailed know-how of the design and some level of expertise. Xilinx Zynq-7000 SoC platform allows high performance, latest Artix-7 based programmable logic and high performance interfaces between the processing system (PS) and programmable logic (PL).

In this paper, we present a study of RSA implemented through hardware/software codesign using Xilinx Zynq-7000 SoC platform. The originality of our work lies in exploring the best trade-off between achieving maximum flexibility from software, with an improvement in performance from hardware by balancing the partitioning between hardware and software components of the design.

II. Related Work

We highlight some of the attempts made to optimize RSA cryptosystem through HW/SW codesign. In [2], based on Right-to-Left algorithm for modular exponentiation, two implementations were proposed for HW/SW codesign. San et al. in [6] present their implementation of RSA using Zynq SoC. However, their design was targeted towards achieving maximum speedup in hardware by deploying the entire exponentiation in their accelerator, which again restricts flexibility through software.

To summarize from the stand point of scalability vs flexibility tradeoff, only designs from [6] and [7] support change in operand sizes from software. Majority of the designs just use the binary method to perform modular exponentiation. More efficient schemes like sliding window method are not exploited. The implementations with the highest level operation being Montgomery multiplier (MM) offer higher flexibility but lower performance as compared to the other approach, i.e., to implement the entire modular exponentiation (ME) in hardware.

Tenca et al. [10] proposed the very first scalable architecture for Montgomery Multiplication [3]. Suzuki in [9] combined the Multiple Word Radix-2 Montgomery Multiplication (MWR2MM) together with the quotient pipelining technique and proposed an architecture which can be mapped onto a modern high-performance DSP-oriented FPGA structure. We use a Montgomery multiplication algorithm based on quotient pipelining technique developed by Orup in 1995 [4].

III. Design Methodology and Environment

A. Hardware/Software Partitioning

We used the built in profiler provided as a part of Vivado Design Suite 2015.4, which showed aproximately 82% contribution of Montgomery Multiplication. In Figure 1, Scheme 1 offers full flexibility but low performance. In scheme 2, modular arithmetic is offloaded to the coprocessor for improved performance. In scheme 3, the entire ME is implemented in the coprocessor for maximum performance gain. However, the design is less balanced between software and hardware. Finally

TABLE I: Implemented hardware functions

Function Call	Description
SET_OP_SIZE()	To set operand size in PL using AXI-LITE interface
LW_M()	Loads modulus from PS through AXI-STREAM interface to the multiplier unit in PL
LW(Reg_Num, A)	Loads data from PS through AXI-STREAM interface to *Reg_Num* in local memory of PL
SW(A, Reg_Num)	Stores data from *Reg_Num* in local memory of PL through AXI-STREAM interface to PS
hw_mul_monty_orup4(dst, src1, src2)	Sends a control word through AXI-LITE interface to load operands from addresses *src1* and *src2* of the local memory to perform multiplication. The result is stored in address specified by *dst*

scheme 4 aims at gaining maximum performance with very limited flexibility.

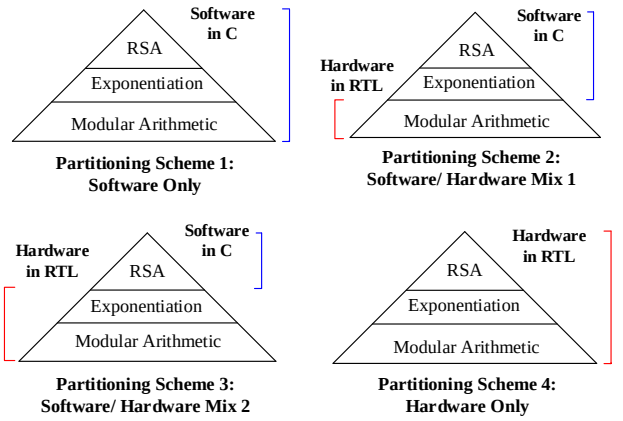

Fig. 1: Hierarchy of Operations and Possible Partitioning Schemes in RSA

Implementing the entire ME in hardware leans more towards performance optimization as it is quite close to having an entire RTL-based design. However, we implement partitioning scheme 2 as it leaves more room to achieve the balance between flexibility through software and performance from the coprocessor.

B. Operation of Processing System (PS)

Zynq SoC platform provides ARM Cortex-A9 microprocessor core. In order to achieve the flexibility, exponentiation is performed in software. Three exponentiation schemes, i.e., Left-to-Right (L2R), Right-to-left (R2L) or Sliding window method were implemented based on Montgomery multiplication using Orup's algorithm in RELIC [1]. However, we present sliding window exponentiation scheme with function calls to the hardware in algorithm 1. All schemes have three stages of operation, i.e., pre-processing, processing and post-processing. The pre-processing (only involved in sliding window exponentiation) includes the RELIC based function bn_rec_slw().Post-processing includes only one modular reduction. In the processing phase, we use the hardware API's to transfer data and control words to the hardware coprocessor while performing modular multiplications.

In Algorithm 1 line 3, the function bn_rec_slw() is a function in RELIC used to compute windows win[0], win[1], ..., win[l-1]. Each window, win[i], is either a zero or a non-zero window. A non-zero window is a subset of up to w exponent bits that starts with 1 and ends with 1. A zero window is a subset of exponent bits consisting of up to w zeros. For non-zero windows, val[win[i]] is the index of the leftmost 1 in the binary representation of win[i]. Once this function is called during pre-processing, l becomes the number of windows rather than the number of bits in the exponent. Also, the conversion of operands to/from Montgomery domain can be realized using Orup's Montgomery Product (OMP) in the coprocessor.

Algorithm 1 Modified Modular exponentiation with hardware functions using Sliding Window method

1: **Input**: a, b, m, w = 6 (maximum window size), $val[win[i]]$ = Index of the leftmost 1 in the binary representation of win[i], l = no. of bits in the exponent b, $R2 = 2^{(2*k*n)}$, where $k = 17$, d = 1 (delay parameter), $2 < m < 2^h$ ($h \in \{512, 1024, 1536, 2048\}$), $h' = h + k(d+1) + 1$, $n = \lceil h'/k \rceil$, $Z = 1$
2: **Output**: $c = a^b \mod m$
3: bn_rec_slw(win, &l, b, w) ——Pre-processing (Line 3)
4: LW(0, R2)
5: LW(1, a)
6: hw_mul_monty_orup4(1, 1, 0)
7: LW(4, Z)
8: hw_mul_monty_orup4(2, 4, 0)
9: hw_mul_monty_orup4(0, 1, 1)
10: **for** $i \leftarrow 1, 2^{(w-1)} - 1$ **do**
11: hw_mul_monty_orup4(2*i+1, 2*i-1, 0)
12: **end for** — Processing
13: **for** $i \leftarrow 0, l-1$ **do** — Lines(4-24)
14: **if** (win[i] = 0)
15: hw_mul_monty_orup4(2, 2, 2)
16: **else**
17: **for** $j \leq 0, val[win[i]]$ **do**
18: hw_mul_monty_orup4(2, 2, 2)
19: **end for**
20: **end if**
21: hw_mul_monty_orup4(2, 2, win[i])
22: **end for**
23: hw_mul_monty_orup4(2, 4, 2)
24: SW(c, 2)
25: $c \leftarrow c \mod m$ ——Post-processing (Line 25)
26: **return** c.

C. Choice of Communication Interface

Our design utilizes AMBA Advanced Extensible Interface 4 (AXI4), targeted at high clock frequency systems. We implemented designs based on both Accelerator Coherency Port (ACP) and high performance port (HP0) interfaces. Our

design connects the processing system (PS) to the hardware accelerator through DMA engine to stream data in burst mode. Once modular multiplication is completed, the design waits for the next control word through AXI-Lite interface. When the entire exponentiation is performed, the final result is sent back to PS through AXI-Stream interface.

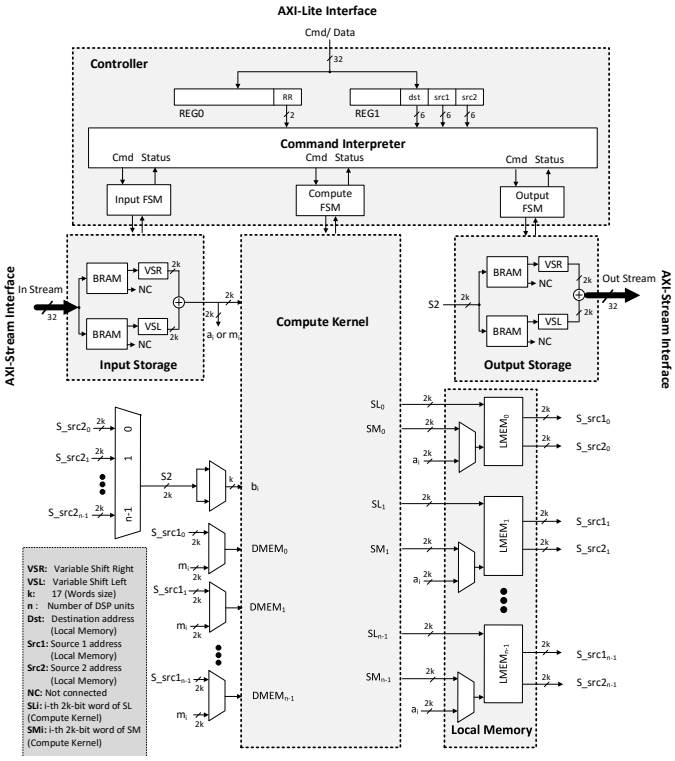

Fig. 2: Detailed Hardware Coprocessor Design

D. Implementing Programmable Logic (PL) - Our Hardware Accelerator

As shown in Figure 2, the compute kernel of our hardware coprocessor performs Montgomery modular multiplication. We extended the Orup's implementation [4] by Suzuki from [9] with an interface for input and output. The modulus M, used in Montgomery multiplication for the reduction part, is replaced by M (called $Mwave$ in all subsequent sections). $Mwave$ is computed only once before the computations, it remains constant for a given modulus M and radix. The datapath is scalable for multiple operand sizes, i.e., 512, 1024, 1536 and 2048 without any increase in the area. The operands are loaded into the local memory of the coprocessor at the start of the operation through AXI-Stream interface. The operands are stored locally into dual-port RAMs to reuse them and to minimize the overhead associated with data transfer from PS to PL. DSP48E macros are used to multiply operands in radix 2^{17}. Latency of these DSP units is adjusted to 3 to operate them at the maximum operating frequency to achieve better performance.

The controller consists of the command interpreter, input, output and compute FSMs. Command interpreter acts as a bridge between software and hardware. It receives command words from PS through AXI-Lite interface and distributes the control signals to all FSMs accordingly. I/O interface is implemented using dual port block memories to support width conversion necessary for the compute kernel to process data. Local memory provides access to intermediate results which requires only few clock cycles.

IV. Results

We present results for three exponentiation schemes, i.e., L2R, R2L and Sliding Window method which are generated using Vivado Design Suite 2015.4. The Zynq device used is xc7z020clg484-1, and the prototyping board is Zedboard Zynq Evaluation and Development Kit. The design is functionally verified using Vivado simulator and co-debugged using Integrated Logic Analyzer (ILA) core. It operates on two clocks, running at 100 and 200 MHz respectively. The arithmetic operations by DSP units utilize the faster clock, whereas the interface and rest of the design operates at 100 MHz.

Table II shows that the HW/SW codesign based approach yields better performance than the software-only implementation showing speedups of 57.67, 57.65 and 46.31 times for L2R, R2L and Sliding window scheme respectively. The sliding window method gives consistently better results than L2R and R2L for all operand sizes in term of the processing time. All clock cycles refer to the clock cycles of the 100 MHz system clock, measured using AXI Timer.

Table III compares our work with existing implementations available in literature. [6] follow more hardware oriented approach by offloading the entire exponentiation onto the coprocessor leaving limited room for flexibility through software. We provide a more balanced partitioning scheme to exploit maximum benefit from both hardware and software. In [2], a fixed 1024-bit RSA design is implemented with one exponentiation scheme, i.e., R2L. This scheme cannot be easily generalized to show gain for other exponentiation schemes. Also, their design is 3.5 times slower than our implementation. In [11], 8051 microcontroller is used. Although they implement several exponentiation schemes, their design is not scalable for any operand sizes other than 1024-bit. Our computation time is 4.6 times faster than their design based on L2R scheme.

HW-only designs are more geared towards optimizing for execution time and are less suited for applications that require flexibility. In codesign-based approach, the designer has to deal with additional overheads related to communication interfaces. The capability of controlling designs from software can also help to identify the best combination of parameters and exponentiation schemes for an embedded application. Implementing various exponentiation algorithms can also provide additional resistance against side-channel attacks.

V. Conclusions

We presented a novel HW/SW codesign approach to support algorithmic and implementation level flexibility. The generic approach can be applied to other public-key cryptosystems as well. In the current design, we achieved up to 57 times more speed as compared to our software implementation with comparable performance in hardware. The result shows that balanced partitioning of a design between hardware and software may seem challenging but it can support promising flexibility vs performance tradeoff.

TABLE II: Comparison of our HW/SW Implementation with software implementation based on RELIC for four operand sizes and three exponentiation schemes. Note: CC_{pre} - Clock cycles for preprocessing, CC_{post} - Clock cycles for postprocessing, CC_{proc} - Clock cycles for processing, CC_{sw} - Clock cycles for software, $CC_{hw/sw}$ - Clock cycles for hw/sw codesign

Op Size	**CC_{sw}**	**$CC_{hw/sw}$**				**Speedup**
(bits)		CC_{pre}	CC_{proc}	CC_{post}	CC_{total}	
L2R						
512	4,628,086	N/A	160,248 (98.71%)	2,098 (1.29%)	162,346	28.51
1024	29,081,675	N/A	636,626 (99.55%)	2,878 (0.45%)	639,504	45.48
1536	91,506,750	N/A	1,703,111 (99.80%)	3,478 (0.20%)	1,706,589	53.62
2048	210,613,761	N/A	3,647,842 (99.88%)	4,401 (0.12%)	3,652,243	57.67
R2L						
512	4,641,915	N/A	160,900 (98.49%)	2,460 (1.51%)	163,360	28.42
1024	29,119,311	N/A	637,572 (99.55%)	2,898 (0.45%)	640,470	45.47
1536	91,596,255	N/A	1,704,883 (99.76%)	4,111 (0.24%)	1,708,994	53.60
2048	210,780,530	N/A	3,651,888 (99.88%)	4,478 (0.12%)	3,656,366	57.65
Sliding Window						
512	3,731,606	4,070 (2.45%)	159,586 (96.09%)	2,416 (1.45%)	167,072	22.47
1024	22,725,405	7,745 (1.24%)	614,977 (98.30%)	2,887 (0.46%)	625,609	36.33
1536	70,955,868	11,654 (0.70%)	1,637,734 (99.04%)	4,215 (0.25%)	1,653,603	42.91
2048	162,227,749	15,223 (0.43%)	3,487,745 (99.44%)	4,356 (0.12%)	3,507,324	46.25

TABLE III: Comparison of our work with existing designs of modular exponentiation ME from literature. Note: * - the execution time was determined for the ME scheme and operand size marked by this symbol, SLID - Sliding Window Method, MPL - Montgomery Powering Ladder, BFL - Blinded Fault Resistant Exponentiation

Reference	ME Scheme (Flexibility)/ Operand Size (Scalability)	Coprocessor performs	Devices	Freq MHz	Area (LUTs/LEs, RAMs, DSP48)	Time* ms
HW/SW Codesign-based Implementations						
This work	L2R, R2L, SLID*/ 512, 1024*, 1536, 2048-bit	MM	Zynq SOC	100/200	10385, 26, 17	6.25
San et al. [6]	R2L /512, 1024*, 2048-bit	ME	Zynq SOC	100	6224, 0, 62	3.04
Issad et al. [2]	R2L/1024-bit	MM	Virtex-5	62.5	1848, 11, 22	22.25
Uhsadel et al. [11]	R2L, L2R*, MPL, BFR/1024-bit	ME	Virtex-4	111	27467, 0, 0	29.37
HW-only Implementations						
Suzuki et al. [9]	SLID/512, 1024*, 1536, 2048-bit	ME	Virtex-4	200/400	4190,7,17	1.71
Song et al. [8]	R2L/1024-bit	ME	Virtex-5	447	180,1,1	36.37
Wang et al. [12]	L2R/1024-bit	ME	Virtex-5	200	5730, 0, 0	679

REFERENCES

[1] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. http://code.google.com/p/relic-toolkit/.

[2] M. Issad, B. Boudraa, M. Anane, and N. Anane. Software/hardware co-design of modular exponentiation for efficient RSA cryptosystem. *Journal of Circuits, Systems, and Computers*, 23(3), 2014.

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

[4] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *Proceedings of the 12th Symposium on Computer Arithmetic*, pages 193–199, Jul 1995.

[5] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. HW/SW co-design for accelerating public-key cryptosystems over GF(p) on the 8051 micro-controller. *In World Automation Congress (WAC), IEEE*, pages 1–6, 2006.

[6] I. San and N. At. Improving the computational efficiency of modular operations for embedded systems. *Journal of Systems Architecture - Embedded Systems Design*, 60(5):440–451, 2014.

[7] M. Simka and V. Fischer. Hardware-software codesign in embedded asymmetric cryptography application — a case study. In *Proc. Field-Programmable Logic and Applications*, pages 1075–1078, 2003.

[8] B. Song, K. Kawakami, K. Nakano, and Y. Ito. An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the FPGA. *IJNC*, 1(2):277–289, 2011.

[9] D. Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. In *Workshop on Cryptographic Hardware and Embedded Systems—CHES 2007*, Berlin, 2007. Springer-Verlag.

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

[11] L. Uhsadel, M. Ullrich, I. Verbauwhede, and B. Preneel. Interface design for mapping a variety of RSA exponentiation algorithms on a hw/sw co-design platform. In *Proceedings of the 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, ASAP '12, pages 109–116, Washington, DC, USA, 2012. IEEE Computer Society.

[12] Z. Wang, Z. Jia, L.Ju, and R. Chen. ASIP-based design and implementation of RSA for embedded systems. In *HPCC-ICESS*, pages 1375–1382. IEEE Computer Society, 2012.