Hardware-Software Co-design of a Cryptographic Application

Abstract

This paper describes and analyzes different approaches to the design and implementation of a primality test for long numbers. Probabilistic primes have great significance in the RSA encryption scheme, extendedly used nowadays on telecommunication systems. Nevertheless, the high computational complexity of this kind of applications makes their software implementations too slow, even when running on fast processors. Therefore, the use of a specific hardware co-processor is the only reasonable solution to obtain good performance. A deep analysis of hardware and software implementations of the example has demonstrated how the co-design of a complex system can produce the best results in terms of cost and performance. Besides, the gap between a real design and co-design flow can be reduced.

1. Introduction

Hardware-software co-design addresses the development of complex heterogeneous systems looking for the best trade-offs among the different solutions. Several problems can be considered under the co-design paradigm [8]: ASIPs synthesis, execution acceleration by means of *custom-computing machines*, design of *System-on-a-Chip (SoC)* or *Embedded-systems* design. However, current methodologies lack contact with real examples from which important knowledge can be extracted. In this paper, we present an in-depth analysis of a co-design case study. We have chosen a cryptographic application: a *Primality Test.*

Security has become a key issue in the world of electronic communications. The security of a strong system mainly dwells in the secrecy of the key rather than in the supposed secrecy of the algorithm. Nevertheless, the time overhead due to data encryption and decryption should not impose a bottleneck in the communication process. Consequently, most cryptographic applications have two antagonistic goals which are very appealing for a co-design approach:

1. To work safe, cryptographic applications need enormous numbers which guarantee the security of the system. Due to the high mathematical complexity of the operations involved in this kind of applications, hardware accelerators are required to work at real time speed.

2. Internet applications are so widespread that the implementation of cryptographic boards should be done as cheap as possible. Thus, the architecture that implements these boards should be mainly based on standard processors.

Nowadays important efforts are made to automate as much as possible the whole co-design cycle. We are currently working in this field, and we found that it is extremely important to have a deep knowledge of the whole co-design cycle, paying special attention to the practical issues a designer takes into account while dealing with a given design problem. Thus, we have looked for an example complex enough and with hard timing constraints to need a co-design methodology. The primality test completely satisfies these premises.

Many co-design examples can be found in the literature. Most of them deal with a particular co-design problem (as it can be memory access [6]) or apply a co-design methodology to a particular example (POLIS [1], COSMOS [2], etc.). In this paper we have completely developed two antagonistic implementations of a data-driven case study. The main objective of this work is to extract as much information as possible from the design experience to incorporate in a given co-design methodology. Additionally, this example will contribute to a set of fully characterized benchmarks for co-design.

The paper is structured as follows. First, the chosen case study will be described in depth. Next sections introduce the two approaches followed to implement the primality test: the software and the hardware implementations. After that, an analysis of the solutions found will be presented, outlining their major features. The paper ends with a summary of the results and some conclusions.

2. Description of the problem

Public key cryptography (RSA [11]) as well as other ways of ciphering and digital authentification based on modular arithmetic is widely used in the world of secret security transmission. But these systems have an important drawback: the slowness of their basic operation, the modular exponentiation with very long integers. Since software implementations are too slow, even running on fast processors, the use of specific hardware seems to be the only reasonable solution to obtain good performance. Moreover, the robustness of these systems is strongly based on the use of enormous prime numbers, because their factorization is almost impossible. But the performance of the system is consequently reduced. Therefore, a key issue that must be considered when designing a cryptosystem is to guarantee that a given number is prime, with a reasonably high probability.

Primality testing may use either deterministic or probabilistic methods. For long numbers a deterministic test would be impossible in terms of computation time. Thus, we will concentrate on probabilistic test, which give a probability of primality for a given number, but their computation is not extremely long. One of the most common tests used nowadays is the Rabin-Miller Test [10]. The test is based on the repetition of a set of operations on the number under test. The number of repetitions controls the probability of failure of the test. The operations that are repeated are actually the *Miller* algorithm, the main part of the test, that is described below.

```
miller(number, base) {
   k=1;
   compute_m_r(); // n = 1+2^k*m
   if (mod_exp(base, m, number)==1 ||
        mod_exp(base, m, number)==-1)
       return 1;
   else
     while (k!=r) {
       if (mod_exp(base,(2^k*m),number)==-1)
         return 1;
       k++;
       if (k >= r)
         return 0;
     }
   return 0;
}
```

Figure 1: Pseudo-code of the Miller Test.

Miller Test Let n be an odd integer, which can be expressed as $n = 1 + 2^r m$, with m an odd number and $r \geq 1$. Let b such that $\gcd(b, n) = 1$ (n is not divided by b). If either $b^m \equiv \pm 1 (\text{mod } n)$ or $b^{2^k} \equiv -1 (\text{mod } n)$ for some $k \leq r$ then we can say that n passes Miller's Test to base b.

The pseudo-code of Miller's Test is shown in figure 1. As can be seen in the pseudo-code, the Miller's test is strongly based on the use of a special operation: the modular exponentiation. If a number does not pass the test, we can guarantee that the number is composite. However, if the number passes the test we cannot be complete confident that the number is prime. There is still a probability of failure. This probability of no success can be diminished using the Rabin-Miller test.

Rabin-Miller Test Let n be an odd positive integer and let b_i for $i = 1, \ldots, k$ be such that $1 < b_i < n - 1$, and $\gcd(b_i, n) = 1$. If n passes Miller's Test for all bases b_i, then the probability that n is prime is at least $1 - 1/4^k$. Thus, changing the cardinality of the set of bases we can tune the probability of a number to be prime. The pseudo-code of the Rabin-Miller's test is written in figure 2.

```
rabin(number) {
  for (i=0; i < repetitions; i++) {
      base = bases[i];
      prime = miller(number,base);
      if (!prime) return 0;
    }
  return 1;
}
```

Figure 2: Pseudo-code of the Rabin Test.

2.1. Design methodology

The main goal of the implementation of this case study is to clearly identify both the advantages and drawbacks of pure hardware and software development cycles. Once this is accomplished, mixed implementations can be better evaluated. Moreover, the conclusions that can be drawn from this design experience can be translated to the co-design methodologies, which should come closer to the current design on practice.

First of all, we have decided to completely describe both implementations with classical software and hardware specification languages (C and VHDL respectively), instead of using a unified language. This decision was made to obtain the best implementation in both cases, because those languages were designed to completely exploit all the advantages of their respective implementation mediums.

The verification of the functionality has been carried out by profiling and simulation of a given set of benchmarks. The benchmarks include the 50,000 first odd numbers and well-known prime numbers, as they are the *Mersenne Numbers* (numbers of the form $M(n) = 2^n - 1$, with n a prime integer, some of them composites) or prime numbers under 10^6, which are tabularized.

Profiling, simulation and synthesis results will characterize the main blocks of the descriptions, as it might be required by a hardware-software partitioning tool. In our case, instead of having estimates we will handle real values. Thus, a secondary goal of this work is to identify the basic objects that could be used in a mixed implementation and fully characterize them.

3. Software Implementation

The software implementation of the Rabin-Miller primality test is strongly based on the use of a library for arbitrary precision arithmetic, because of the use of very long integers. In our case we have used the GNU Multiple Precision Library, *GMP* [4] for several reasons:

- The source code, written in C, is freely available.
- The library is designed to operate very fast.
- All the complex operations required by the test are implemented in the library, including the modular exponentiation.

We have chosen the C programming language as description language for several reasons. First, C was specifically created to allow the programmer access to almost all of the internals of the machine: registers, input/output slots, memory, etc. Since we are interested on the co-design of the example, this language provides us the basic support to deal with the hardware parts of a SoC design. At the same time, C allows data hiding and modularisation, what is needed to allow very complex programs to be created in an organized and well-timed way. Finally, the resulting code executes faster than most other high level languages.

The algorithm has been coded using the formulation of Knuth [5]. The test has been preceded by a filter function that divides the number under test by the first prime numbers. The use of this function is based on the lower computation time required by a division compared to the modular exponentiation.

The implementation must be extensively checked to characterize the memory space requirements and the performance that the program provides. We are specially interested on the performance bottlenecks and the memory problems that can arise when running the program in a standard processor. We have used GPROF profiler to analyze the results and refine the software implementation.

3.1. Analysis of the Software Implementation

The whole software development process has been carried out on a Pentium II PC (350 MHz, 128 MB RAM) under Linux (Debian GNU/Linux 2.1). GNU gcc compiler, debugger and profiler have been used for their widely known quality and efficiency.

Function	# Calls	Time (s)	Mem. (bytes)
mpz_cmp	35810	0	B0
mpz_sub_ui	24526	0	200
mpz_powm	71692	6763.14	14D0
mpz_gcd	36717	0.41	418
mpz_add_ui	34929	0	1F0
mpz_tdiv_q	12263	0.19	570
mpz_mul	11356	0.04	530
congruent	69972	0.94	12C8

Table 1: Profiling of the Miller function

Software estimation remains difficult because of problems in modeling the environment. We have thus used profiling to characterize the software implementation. The analysis performed includes the generation of the execution flow graph and the call tree, the computation of the execution time of all the functions the code includes and the measurement of the memory space required by the different fragments of code.

Table 1 summarizes the results obtained when running the set of benchmarks described in section 2.1.

4. Hardware Implementation

The hardware implementation has been done using VHDL [7] as description language because of its reusability, among many other well-known advantages. The whole circuit has been coded with synthesizable VHDL and parameterized with the bit-width of the number under test. Following this methodology we can easily synthesize a circuit with different sizes.

The resulting architecture comes from the immediate mapping of the test into hardware blocks. Following the pseudo-code of the algorithm we can identify the following basic blocks:

- Modular Exponentiation, which is the main block of the circuit. The Rabin-Miller's test can be actually seen as a sequence of modular exponentiations.
- ROM, which stores the set of bases, b_i, used on the test.
- Comparator, whose output tells us if the result of the exponentiation has been '1', '-1' or none of these.
- Shift Registers, that implement the divisions and multiplications that have to be carried out with the exponent. Additionally these registers load data from the outside.

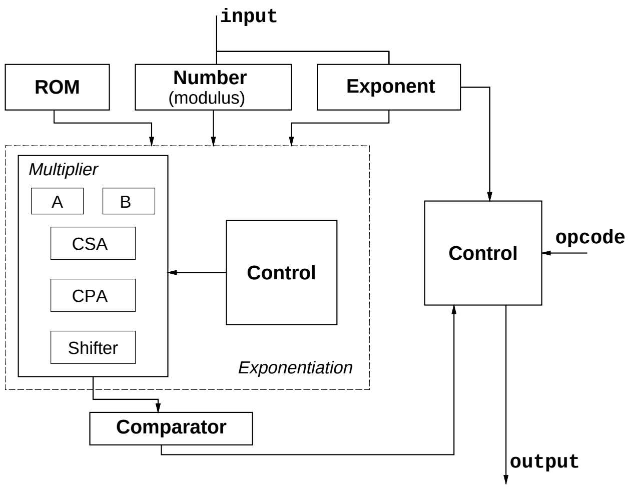


Figure 3: Block diagram

```
$R = 0;$
for (i=0; i<n; i++){
        $Q_i = ((R_0 + A_i B_0)(r - M_0)^{-1})\ mod\ r;$
        $R = (R + A_i B + Q_i M)\ div\ r;$
    }
if $(R > M)\ R = R - M;$
```

Figure 4: Montgomery's Algorithm.

- Finite State Machines, which take care of the control signals used to coordinate the operation of the different blocks.

The block diagram of figure 3 describes the interaction among the elements previously described.

4.1. Modular multiplication

The key point of the hardware implementation of the primality test is the modular exponentiation, which we have implemented following the method *square and multiply* [5]. The main element of this implementation is the modular multiplication (MM in the following); the faster the MM is performed, the faster the test is run. Among the many algorithms that have been designed to solve MM we have chosen the Montgomery algorithm [9], in particular the hardware implementation proposed by Eldridge and Colin D. Walter [3], because of the good results it produces. The Montgomery algorithm provides high performance while requiring lower area. Another advantage of this method is that it requires simpler and faster selection logic.

The pseudo-code of the algorithm is shown in figure 4, where R is both an iteration register and the place where the final operation result is stored. M is the modulus, A and B are the numbers under multiplication, r is the radix used in the algorithm, n is the number of digits and Q the quotient of the operation.

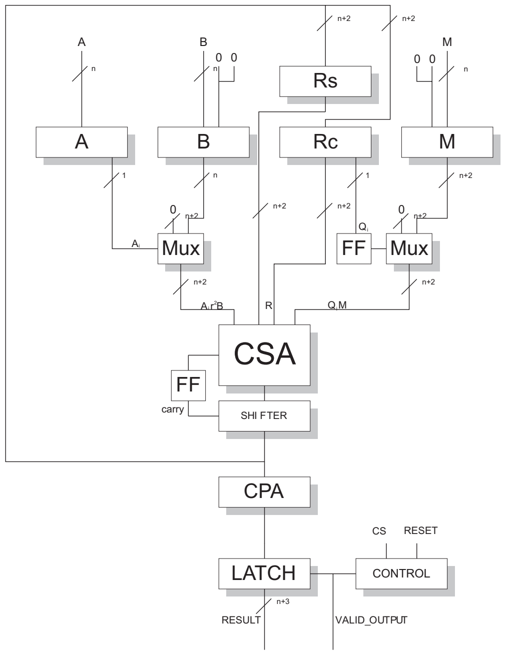


Figure 5: Montgomery multiplier

The choice of the radix value is very important to obtain good performance while keeping an acceptable area. For high radix values the algorithm runs less iterations. But this does not necessary mean that the computation is faster, since the clock frequency decreases due to the larger multiplier size. Taking all that into consideration, we have implemented the Montgomery algorithm with the minimum radix value, $r = 2$. This value drastically simplifies the hardware, because since M is prime $M_0 = 1 \Rightarrow (r - M_0) = 1$.

Paying attention to the code, the hardware needed to implement the MM is:

- R, M, A, B: Registers of n+2 bit width to store the result, modulus and numbers to multiply. Register A allows shifting.
- Q_{i+1}*: D Flip-flop to save the bit* Q_{i+1} needed on the next iteration.
- A counter with $log_2(n+2)$ bit width which controls the number of iterations the algorithm requires.
- Two adders: A three input *carry save adder* (CSA) and a *carry propagate adder* (CPA) to reorganize the result.
- A shifter: since $r = 2$ is the radix, the shifter implements the division by the radix.

Block	Area (eq. gates)	Max. delay (ns)	Max. fre (MHz)
CSA	3650	0.65	
CPA	2592	2.42	
Shifter	535	0.44	
ROM	142.6	0.63	
Comparator	1236	2.5	
Shift Reg.	1688		666
Shift Reg. 2	1295		375
Multiplier	14057	4.01	192
Exponentiation	30644		130
Rabin-Miller	39277		130

Table 2: Summary of hardware synthesis

The way these elements are interconnected is shown in figure 5. Obviously, those elements are the data-path of the modular multiplication module. A FSM is needed to generate and perform the control of the algorithm.

It is important to remark that resource sharing has been a key issue considered during the whole design process. This is due to the large size of the numbers under test, that demand enormous amounts of silicon just for storage.

4.2. Analysis of the Hardware Implementation

The hardware description of the Rabin test has been synthesized with the library MTC45000 of 0.35 μm. Due to the long computation time the synthesis tool requires (*Synopsys*), we have synthesized a primality test for numbers of 128 bits. It is very straight-fordward (but very time consuming) to synthesize a circuit for numbers of 1024 bit-width. As it is well-known, hardware estimation is still hard today, because it depends on *fast synthesis*. Synthesis results are summarized in table 2, where the main blocks of the circuit are characterized by means of their area and timing parameters (delay for combinational blocks and working frequency for sequential modules).

It can be seen in table 2 that the speed of this implementation is clearly bounded by two modules. First, the combinational block CPA, with a delay of 2.42 ns, limits the speed of the multiplier. Second, the exponenciator limits the fequency of the whole circuit. This is mainly due to an additional delay introduced by the control FSM.

Regarding the area, the largest module is the exponentiator, with around 80 % of the total area if we add the control and the datapath, as it is clearly depicted on figure 6. Since the main element of this datapath is modular multipier, we have also represente on the right side of figure 6 the distribution of area within this module.

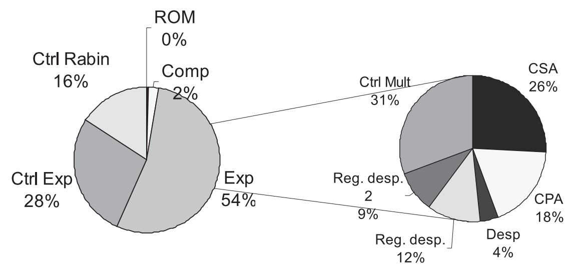
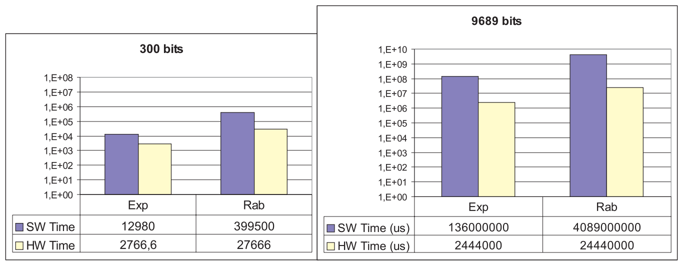


Figure 6: Hardware area of the different blocks



	Exp	Rab
▨ SW Time	12980	399500
□ HW Time	2766,6	27666

	Exp	Rab
▨ SW Time (us)	136000000	4089000000
□ HW Time (us)	2444000	24440000

Figure 7: Time improvement of the hardware implementation for two different numbers

5. Comparative study and lessons learned

As it is well-known, hardware development cycle is extremely long compared with software. In our case, the hardware design time has been 80% of the whole case study design. For instance, extensive simulation has been carried out to functionally validate the specification and simulation of a single test takes longer than profiling of a set of numbers. Current *time-to-market* requirements preclude this expensive design style. The use of co-design methodologies and IP-based designs seems actually to be a necessity.

The performance provided by the software implementation is clearly influenced by the extremely large computation time of the modular exponentiation. This can be seen without doubt in the plots shown in figure 7. Those plots represent the execution time of both hardware and software implementations of the exponentiation and of the whole Rabin test. For very large numbers the time improvement provided by the hardware implementation is even greater. The other modules of the design do not present such a big difference in timing terms.

Taking all the previous information into consideration we can study mixed hardware-software implementations. These implementations whould be implemented as an embedded system, with a standard core processor and dedicated hardware. Once we have identified

	Impl. 1 Hw. Exponentiation.	**Impl. 2.** Hw Multiplier
HW Area (Eq. gates)	30,600	14,057
Freq. (MHz)	130	192
Transfer	$4n$/exp.	$4n^2$/exp

Table 3: Hw-Sw implementations

the performance bottlenecks of the software, two mixed implementations have been considered, which are described in table 3. A first one (Impl. 1 in table 3) would use a hardware co-processor implementing the modular exponentiation. A second one (Impl. 2) would only implement the modular multiplication in hardware, since this is the main block of the hardware design.

If we compare the area required by implementation 1 with the area of the all-hardware solution (table 2) we see that this mixed implementation only saves 25 % of the total area. Nevertheless, this can be an appealing solution, because the global cost of the implementation would be smaller. Besides, we can keep the software filter function that divides the number under test by a set of prime numbers, because this function can eliminate many composite numbers, with the corresponding computation time savings.

Implementation 2 can save even more area (around 50 %), and presents a faster clock speed, being thus a good candidate to be the best implementation. Nevertheless, this implementation presents a serious drawback: the large communication interchange demanded by the modular multipier. As can be seen in table 2, for this case the number of transfers required by exponentiation is $4n^2$, while impl. 1 only needs $4n$. Due to the fact that the numbers under test will be very long (1024 bits are needed for military constraints), the time spended transferring data from the standard processor to the co-processor would be extraordinarily large. Consequently, the impl. 1 should be the best solution to implement a primality test in a hardware-software architecture.

6. Conclusions

In this paper we have fully specified and designed two antagonistic implementations of a cryptographic application: the Rabin-Miller Primality Test. On the one hand, a software implementation has been done in a very short time, mainly due to the availability of libraries that work with very long numbers. The performance obtained with this implementation has not been very good due to the large computation time that some arithmetic operations (in particular the modular exponentiation) need. On the other hand, a hardware implementation has been done, presenting very good performance figures. Nevertheless, the cost of this implementation is quite high, due to the large area that is required.

Mixed hardware-software architectures have been evaluated to implement the primality test taking advantage of both design experiences. These implementations make profit of the flexibility and low cost of the software and the better performance of the hardware.

Our experience has demonstrated that co-design methodologies and system-level tools are needed to shorten the design time of complex systems. Additionaly, attractive implementations in terms of costs and performance can be provided by these approaches.

References

[1] T. Cuatto, C. Passerone, L. Lavagno, A. Jurecska, A. Damiano, C. Sansoe, and A. Sangiovanni-Vincentelli. A case study in embedded system design: an engine control unit. In *Proceedings of the Design Automation Conference*, June 1998.

[2] J. Daveau, G. Marchioro, and A. Jerraya. Hardware/software codesign of an atm network interface card: A case study. In *Proc. CODES/CASHE'98*, pages 15–18, March 1998.

[3] S. E. Eldridge and C. D. Walter. Hardware implementation of montgomery's modular multiplication algorithm. *IEEE, Trans. Computers*, 42(6):693–699, Jun 1993.

[4] T. Granlund. *The GNU Multiple Precision Arithmetic Library*. Free Software Foundation, Inc., Aug 2000. http://www.swox.com/gmp/.

[5] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., 1981.

[6] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, and A. Sangiovanni-Vincentelli. A case study on modeling shared memory access effects during performance analysis o hw/sw systems. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1998.

[7] R. Lipsett, C. F. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 1989.

[8] G. D. Micheli. Guest editor's introduction: Hardware-Software Codesign. *IEEE Micro*, pages 8–9, August 1994.

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

[10] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.

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