Parametrized Hardware Architectures for the Lucas Primality Test

Adrien Le Masle, Wayne Luk
Department of Computing
Imperial College London, UK
{al1108,wl}@doc.ic.ac.uk

Csaba Andras Moritz
BlueRISC, Inc
Amherst, MA, USA
andras@bluerisc.com

***Abstract*—We present our parametric hardware architecture of the NIST approved Lucas probabilistic primality test. To our knowledge, our work is the first hardware architecture for the Lucas test. Our main contributions are a hardware architecture for calculating the Jacobi symbol based on the binary Jacobi algorithm, a pipelined modular add-shift module for calculating the Lucas sequences, a dependencies analysis and a scheduling of the Lucas sequences computation. Our architecture implemented on a Virtex-5 FPGA is 30% slower but 3 times more energy efficient than the software version running on a Intel Xeon W3505. Our fastest 45 nm ASIC implementation is 3.6 times faster and 400 times more energy efficient than the optimised software implementation in comparable technology. The performance scaling of our architecture is much better than linear in area. Different speed/area/energy trade-offs are available through parametrization. The cell count and the power consumption of our ASIC implementations make them suitable for integration into an embedded system whereas our FPGA implementation would more likely benefit server applications.**

I. Introduction

Many public-key cryptographic algorithms require large prime numbers in order to generate a key pair. This is for example the case of the Diffie-Hellman key exchange protocol or the most popular Rivest, Shamir and Adleman (RSA) encryption scheme. The common method for finding a large prime number consists in testing randomly generated numbers until a prime is determined. The AKS primality test [1] determines if a number is prime in polynomial time. This test is deterministic. However, due to the complexity of implementing this algorithm in hardware as well as in software, probabilistic tests are still in use.

A common primality tester first tries to divide the number under test by a few first primes, then performs a number of Miller-Rabin probabilistic tests [2]. The error probability of such a primality test depends on the bitwidth of the number under test and on the number of Miller-Rabin tests performed. The American National Institute of Standards and Technology (NIST) gives recommended values for different algorithms and different key sizes [3]. However, even with well-chosen parameters there is still a small probability that a composite number passes the primality test. Ultimately, a Lucas probabilistic test can be performed to ensure that the number is prime. In fact, as of today no composite number passing an appropriate number of Rabin-Miller tests plus a Lucas test is known.

FPGAs and ASICs are relevant platforms for cryptographic algorithm implementations. First, the structure of FPGAs makes them particularly fit for pipelined applications, which is the case for most of the basic cryptographic operations. Second, FPGAs and ASICs can be used to embed security into low power environments keeping very good performance. Finally, a pure hardware implementation of a cryptographic algorithm is inherently less vulnerable than its software counter-parts which are usually run in a multi-tasking operating system. For instance, software-based attacks such as cache-attacks [4] do not apply to hardware.

This paper presents the first hardware architecture for the NIST approved Lucas probabilistic primality test. Our main contributions include:

- A hardware architecture for calculating the Jacobi symbol based on the binary Jacobi algorithm
- A pipelined modular add-shift module for calculating the Lucas sequences
- A dependencies analysis and a scheduling of the Lucas sequences computation
- An implementation of the proposed design on a Xilinx Virtex-5 FPGA and in TSMC 65 nm and 45 nm ASIC processes
- A comparison of the performance of our hardware implementations with an optimised software implementation in terms of speed and energy efficiency

Our architecture implemented on a Virtex-5 FPGA is 30% slower but 3 times more energy efficient than the software version running on a Intel Xeon W3505. Our design is scalable and the performance scaling of our architecture is much better than linear in area. Hence we expect the energy efficiency of our FPGA designs would improve with advances in technology and area available. Our 65 nm ASIC implementation is more than 2 times faster and 40 times more energy efficient than the FPGA implementation. Our fastest 45 nm ASIC implementation is 3.6 times faster and 400 times more energy efficient than the optimised software implementation.

The rest of the paper is organised as follows. Section II explains the background relevant to our work. In section

The support of BlueRISC, Alpha Data, Xilinx, UK EPSRC and the HiPEAC NoE is gratefully acknowledged. The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement number 248976 and 257906.

III, we present the challenges we face when designing the Lucas primality tester and how we solve them. In section IV, we compare our software, FPGA and ASIC implementations. Finally, section V concludes the paper.

II. Background

The Lucas primality test is a probabilistic test based on the properties of the Lucas sequences. We first introduce the Lucas sequences and the Lucas theorem. Then we describe the Lucas primality test algorithm.

A. Lucas Sequences

The Lucas sequences $U(a,b)$ and $V(a,b)$ of the pair (a,b) are the sequences:

$$U(a,b) = (U_0(a,b), U_1(a,b), U_2(a,b), ...) \quad (1)$$

$$V(a,b) = (V_0(a,b), V_1(a,b), V_2(a,b), ...) \quad (2)$$

such that for each $k \geq 0$:

$$U_k(a,b) = \frac{\alpha^k - \beta^k}{\alpha - \beta} \quad (3)$$

$$V_k(a,b) = \alpha^k + \beta^k \quad (4)$$

where α and β are the two roots of the quadratic equation $x^2 - ax + b = 0$, with a, b chosen such that $a, b \neq 0$ and the discriminant $D = a^2 - 4b \neq 0$ [5].

Let us define the Jacobi symbol $\left(\frac{a}{n}\right)$. For a integer and p prime, we have:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } a = 0 \bmod p \\ 1 & \text{if } a \neq 0 \bmod p \text{ and } x^2 = a \bmod p \\ & \text{is soluble for some integer } x \\ -1 & \text{if } a \neq 0 \bmod p \text{ and } x^2 = a \bmod p \\ & \text{is not soluble} \end{cases}$$

For an integer n with prime decomposition $n = p_1^{\alpha_1} p_2^{\alpha_2} ... p_k^{\alpha_k}$, we have:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} ... \left(\frac{a}{p_k}\right)^{\alpha_k}$$

The Lucas theorem is defined as follows [5].

Lucas Theorem: Let a, b, D, and U_k be as above. If n is prime, $\gcd(b,n) = 1$ and $\left(\frac{D}{n}\right) = -1$, then n divides U_{n+1}.

The Lucas test is based on the contrapositive of the Lucas theorem stating that if, under the previous assumptions, an odd positive integer n does not divide U_{n+1}, then n is composite.

B. Lucas Test Algorithms

A simple algorithm performing a Lucas Test is given in Alg. 1. Note that a Lucas test alone is not enough to determine whether a number is prime with a low probability of error. Many composite numbers passing the Lucas test are known indeed. That is why the Lucas test has to be combined with another probabilistic primality test such as the Miller-Rabin test.

Algorithm 1: Simple Lucas test

Input: n odd integer
Output: *composite* if n is composite, *probably prime* if n is probably prime

1 Choose $a, b \neq 0$ such that $D = a^2 - 4b \neq 0$, $\gcd(b,n) = 1$ and $\left(\frac{D}{n}\right) = -1$
2 Compute $U_{n+1}(a,b)$
3 **if** $U_{n+1}(a,b)$ *mod* $n = 0$ **then**
4 **return** *probably prime*
5 **else**
6 **return** *composite*

Consider line 1 of Alg. 1. Different methods to choose a, b and D are given in [6], [7]. In particular, it is shown that D should not be a square modulo n. In that case, the Lucas test would be an ordinary primality test. That is in fact ensured by the condition $\left(\frac{D}{n}\right) = -1$. A method proposed by Selfridge [7] consists in choosing D as the first element in the sequence $\{5, -7, 9, -11, 13, ...\}$ such that $\left(\frac{D}{n}\right) = -1$, $a = 1$ and $b = (1 - D)/4$. If n is square, it can be shown that $\left(\frac{D}{n}\right) > -1$ for any D. Hence, so that the algorithm terminates, we have to check if n is a perfect square either before looking for a correct D or after a few trials for D.

Consider line 2. Several methods can be used to compute U_{n+1}. A simple method is to use equation 3 directly. However, this method is not efficient as it requires two exponentiations and possibly one division. Instead we can use a recurrence relation. For example, we can easily show that:

$$U_k(a,b) = aU_{k-1} - bU_{k-2} \quad (5)$$

$$V_k(a,b) = aV_{k-1} - bV_{k-2} \quad (6)$$

In [8], a more complex recurrence relation is used to come up with an efficient method to compute the Lucas sequences:

$$U_{i+j}(a,b) = U_i V_j - b^j U_{i-j} \quad (7)$$

$$V_{i+j}(a,b) = V_i V_j - b^j V_{i-j} \quad (8)$$

The American National Institute of Standards and Technology (NIST) approved Lucas probabilistic primality test is given in Alg. 2. The NIST algorithm first checks if n is a perfect square. If not, it uses the algorithm described by Selfridge to find a correct value for D. This test is sometimes called the Lucas-Selfridge probabilistic primality test. Lines 4-5 rely on the fact that if $\left(\frac{D}{n}\right) = 0$, a factor of n exists [6] and we can therefore stop the test. Lines 6 to 18 computes $U_{n+1}(a,b)$, with $a = 1$ and $b = (1 - D)/4$, using a modified version of the method presented in [8]. After r iterations, $U = U_{n+1}(a,b)$.

In [3], the NIST gives an algorithm to determine whether a number is a perfect square. This algorithm is complex for

Algorithm 2: Lucas probabilistic primality test (from [3])

Input: n odd integer
Output: *composite* if n is composite, *probably prime* if n is probably prime

1 **if** n *is a perfect square* **then**
2 **return** *composite*
3 Find the first D in the sequence $\{5, -7, 9, -11, 13, -15, 17, ...\}$ for which the Jacobi symbol $\left(\frac{D}{n}\right) = -1$.
4 **if** $\left(\frac{D}{n}\right) = 0$ *for any* D *in this sequence* **then**
5 **return** *composite*
6 $k = n + 1$
7 Let $k_r k_{r-1} ... k_0$ be the binary expansion of k, with $k_r = 1$
8 $U = 1$, $V = 1$
9 **for** $i = r - 1$ **to** 0 **do**
10 $U_{temp} = UV \bmod n$
11 $V_{temp} = (V^2 + DU^2)/2 \bmod n$
12 **if** $k_i = 1$ **then**
13 $U = (U_{temp} + V_{temp})/2 \bmod n$
14 $V = (V_{temp} + DU_{temp})/2 \bmod n$
15 **else**
16 $U = U_{temp}$
17 $V = V_{temp}$
18 **end**
19 **if** $U = 0$ **then**
20 **return** *probably prime*
21 **else**
22 **return** *composite*

hardware implementation as it requires divisions and squaring. A much simpler binary algorithm is presented in [9]. This algorithm returns the integer square root of an n-bit number together with the remainder using only shifts, additions and subtractions. The number tested is a perfect square if the remainder is equal to zero. We use this algorithm in our implementation as described in the next section.

Line 3 of Alg. 2 requires the ability to compute the Jacobi symbol $\left(\frac{D}{n}\right)$. As before, the algorithm given by the NIST in [3] is not fit for fast hardware implementation as it requires several modular reductions. Some more relevant binary Jacobi algorithms using only shifts, additions/subtractions and comparisons are presented in [10], [11]. The changes we make to the Jacobi symbol calculator and the perfect square test would not affect compliance to the NIST standard overall.

C. Parametric Montgomery Multiplier

Montgomery multiplication is commonly used to perform modular multiplication in hardware as it is fast and area-efficient. The Montgomery product of two n-bit integers A and B modulo an odd n-bit integer N is:

$$P = A.B.2^{-n} \bmod N \tag{9}$$

Montgomery multiplication introduces an extra factor 2^{-n} which is usually dealt with by converting the numbers in N-residue, that is Montgomery multiplying them by $2^{2n} \bmod N$ before performing modular multiplications. Then the result is converted back to normal representation by Montgomery multiplying it by 1.

The architecture of the Montgomery multiplier used in this paper is presented in Fig. 1. Short vertical and horizontal lines represent registers. Each Montgomery cell operates on sub-bits of the input A in a pipeline fashion. We call N_p the number of pipeline stages. Inside each cell, a number of additions are performed by carry-save adders (CSA). Each addition depends on the previous one with a loop-carried dependency of one. Hence the computation cannot be parallelised. However, the carry-save adders can be replicated in series so that several additions are performed in one clock cycle. This is equivalent to unrolling the loop. The number of replications is called N_r. A more precise description of the CSA-based parametric Montgomery multiplier architecture and the algorithm used are given in [12].

III. Lucas Primality Tester Design

Our Lucas Primality tester is based on Alg. 2 recommended by the NIST. We divide our Lucas hardware into three sub-modules:

1) Perfect square test module (lines 1-2)
2) Jacobi symbol calculator module (lines 3-5)
3) Lucas sequence (or $U_{n+1}(a, b)$) calculator module (lines 6-21)

Module 1 is an implementation of the algorithm presented in [9]. This algorithm is straightforward and only performs shifts, additions and subtractions.

In the following, we focus on module 2 and 3 which are the most interesting ones. Three main challenges have to be addressed when designing these modules:

Challenge 1. The Jacobi symbol calculator module has to work with negative integers and must not contain any complex modular reduction that would greatly increase the area taken by the design.

Challenge 2. We need to design parametric and efficient modules for the two main operations of the Lucas calculator: (a) modular multiplication and (b) modular add-shift.

Challenge 3. These two operations have to be rescheduled to optimise the speed of the design, making the most of the pipeline capabilities of the modules.

A. Challenge 1: Jacobi Symbol Calculator

Classical Jacobi algorithms require a full modular reduction at each iteration. This operation can only be implemented with a divider which is area consuming. This is not the case of the binary Jacobi algorithm presented in [10]. We modify this algorithm to compute $\left(\frac{a}{b}\right)$ for negative a. Our modified binary Jacobi algorithm is presented in Alg. 3. It is based on the general identities of the Jacobi symbol described in [10].

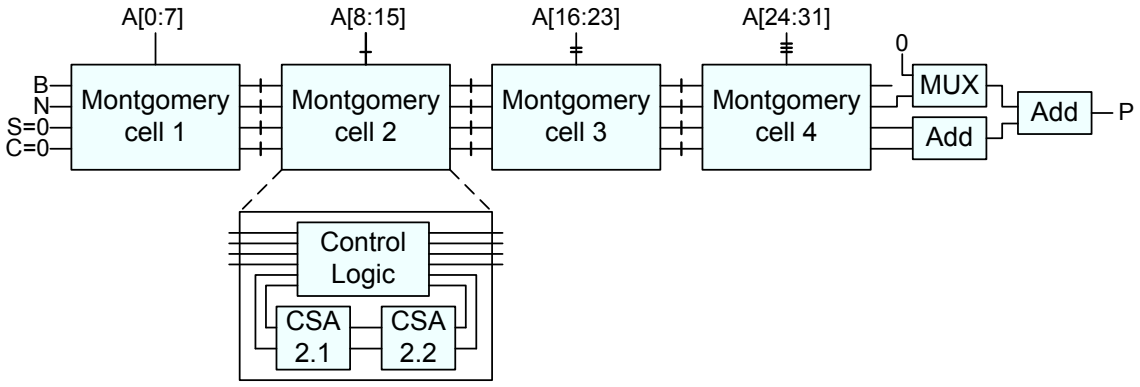

Fig. 1. Montgomery multiplier for $N_p = 4$ and $N_r = 2$

Algorithm 3: Binary Jacobi algorithm

Input: a integer, b odd positive integer
Output: $\left(\frac{a}{b}\right)$

1 $t = 1$
2 **if** $a < 0$ **then**
3 $a = -a$
4 **if** $b \ mod\ 4 = 3$ **then** $t = -t$
5 **end**
6 **while** $a \neq 0$ **do**
  // SHIFT
7 **while** $a \ mod\ 2 = 0$ **do**
8 $a = a/2$
9 **if** $(b \ mod\ 8 = 3)$ *or* $(b \ mod\ 8 = 5)$ **then** $t = -t$
10 **end**
  // SWAP
11 **if** $a < b$ **then**
12 $interchange(a, b)$
13 **if** $(a \ mod\ 4 = 3)$ *and* $(b \ mod\ 4 = 3)$ **then** $t = -t$
14 **end**
  // NEWA
15 $a = (a - b)/2$
16 **if** $(b \ mod\ 8 = 3)$ *or* $(b \ mod\ 8 = 5)$ **then** $t = -t$
17 **end**
18 **if** $b = 1$ **then**
19 **return** t
20 **else**
21 **return** *0*

All the modular reductions use a fixed modulo which is a power of two. In that case, $x \bmod y = (x \ \& \ (y-1))$. These operations are therefore very simple to implement in hardware. To reduce the critical path and therefore increase the maximum clock frequency, the comparators and the subtractors are pipelined.

B. Challenge 2: Lucas sequence calculator

Our Lucas sequence calculator performs the operations of lines 6 to 21 of Alg. 2. This part of the algorithm consists of several modular multiplications and modular add-shift operations. The operations performed in the for loop can be decomposed as shown in Alg. 4.

Algorithm 4: Decomposition of the operations performed in the for loop of the Lucas primality test algorithm

1 $U_{temp} = U.V \bmod n$
2 $V_{temp} = V^2 \bmod n$
3 $temp1 = U^2 \bmod n$
4 $temp1 = D.temp1 \bmod n$
5 $V_{temp} = (V_{temp} + temp1)/2 \bmod n$
6 **if** $k_i = 1$ **then**
7 $U = (U_{temp} + V_{temp})/2 \bmod n$
8 $temp2 = D.U_{temp} \bmod n$
9 $V = (V_{temp} + temp2)/2 \bmod n$
10 **else**
11 $U = U_{temp}$
12 $V = V_{temp}$

If k is an (r+1)-bit number with $k_r = 1$, the algorithm performs between $4r$ and $5r$ modular multiplications. We reuse the parametric Montgomery multiplier presented in [12] to perform these modular multiplications and solve challenge 2.a. All the operations are performed in n-residue representation, with n the number under test. Our multiplier has two main parameters: the number of pipeline stages N_p and the number of replications N_r.

Similarly, the algorithm performs between r and $3r$ modular add-shift operations. Alg. 5 shows how the modular addition with right shift is implemented. Note that the values of the two operands of this addition are always less than the modulo. Hence the modular reduction can be performed by a simple subtraction.

We develop a pipelined hardware design of the modular add-shift module. All the operations are computed by homogeneous blocks, the number of blocks being a parameter of the design. Our architecture is shown in Fig. 2. Short horizontal and vertical lines represent registers. The first row of adders computes $A+B$. Its output is given to the second row of adders which computes $A + B + N$ and to a shifter which computes

Algorithm 5: Modular addition with right shift

Input: N odd integer, $A < N$ and $B < N$ positive integers
Output: $S = (A + B)/2 \bmod N$

```
if (A + B) mod 2 = 1 then
    S = (A + B + N)/2
    if S ≥ N then S = S − N
else
    S = (A + B)/2
```

$(A+B)/2$. The output of the second row of adders is shifted right and given to a row of subtractors. These subtractors compute $(A+B+N)/2-N$. They are also used to determine if $(A+B+N)/2 \geq N$. The `MUX Select` generates the select signal of the multiplexer according to the borrow out of the row of subtractors and the value of the LSB of $A + B$. If l is the number of adders/subtractors in each row, after $l + 2$ cycles of latency our module generates one result per clock cycle. This solves challenge 2.b.

Our Lucas calculator has four main parameters: the bit-width of the number under test, the number of pipeline stages of the Montgomery multiplier, the number of replicated carry-save adders in each multiplier's pipeline block and the pipeline depth of the modular add-shift module. By combining Montgomery multiplication, often regarded as the best technique for modular multiplication, and our pipelined modular add-shift architecture, we get rid of the need for direct modular reduction. A proper scheduler needs to be designed to make the most of our parametric modules.

C. Challenge 3: Scheduling

To increase the speed of Alg. 4 we reschedule the modular multiplications and the modular additions. In particular we want to use the pipeline of the Montgomery multiplier in a relevant way. To keep the design simple we decide to execute the calculations of $(U_{temp} + V_{temp})/2 \bmod n$ and $(V_{temp} + temp2)/2 \bmod n$ even if $k_i = 0$. In that case, the result is disregarded. Hence $temp2$ always needs to be calculated and we always perform 5 Montgomery multiplications and 3 modular additions at each iteration. The true dependencies between the different operations are shown in the dependency graph of Fig. 3. Output and anti-dependencies have already been removed by register renaming.

The latency and the throughput of the Montgomery multiplier depend on the bit-width of the inputs, the number of pipeline stages and the number of replications. The latency of the modular add-shift module depends on the number of adders/subtractors blocks used in each row. However using a pipeline depth of more than 3 for the multiplier is under-efficient given the dependencies shown in the dependency graph of Fig. 3. Sticking to the instruction numbering introduced in the table of Fig. 3, we come up with the following schedule: Inst.3, Inst.1, Inst.2, Inst.4, Inst.6, Inst.5, Inst.7, Inst.8. Fig. III-C shows this schedule for a pipeline depth $N_p = 3$. To keep the example simple, we assume that each multiplier's pipeline stage takes 2 clock cycles to complete its operations. We also assume that the latency of the modular adder is 2 clock cycles. A1 and A2 represent the 2 pipeline stages of our modular adder. M1 to M3 represent the 3 pipeline stages of the multiplier. In the actual implementations, the latency of a multiplier's pipeline stage is higher than the latency of the modular adder (42 and 10 respectively for our fastest implementation presented in section IV) and the number of pipeline stages of the module adder is chosen to optimise the speed of the Lucas test. However, this simplification does not change the interpretation of this table. From clock 1 to clock 10, the multiplier performs 5 multiplications without stalling. At clock 11, no more multiplication can be given to the multiplier until Inst.7 terminates. Hence the multiplier has to stall. This is due to the data dependency between Inst.3 (the next multiplication to be performed) and Inst.7 which itself depends on Inst.5. This situation shows that the multiplier's pipeline cannot be run full all the time. This leads to some unavoidable performance loss.

Our particular schedule may not be optimal for all the values of the parameters but we found out that it is a relevant compromise under the assumption that the addition time is shorter than the multiplication time.

The main components of our Lucas calculator module are a Montgomery multiplier, a modular add-shift module and several registers to store the values of U_{temp}, V_{temp}, $temp1$, $temp2$, D, U, V and k. We also need an adder to compute $k = n + 1$ and a subtractor to negate D mod n when it is negative. A leading zeros detector is used to identify the most significant bit in the binary representation of k. Multiplexers select the inputs to the Montgomery multiplier and the modular add-shift module.

FIFOs and dependency tables record the state of the Montgomery multiplier's pipeline and of the modular add-shift module. This enables the scheduler to prevent read-after-write dependencies and to feed the Montgomery multiplier and the modular add-shift pipelines correctly.

D. Lucas prime tester

The square test module, the Jacobi symbol calculator module and the Lucas calculator are integrated into our Lucas prime tester. The square test and the search for the value of D are performed in parallel. If the number tested is a perfect square or if a D such that $\left(\frac{D}{n}\right) = 0$ is found, the test finishes and returns 0 (the number is declared composite). Otherwise, the Lucas sequence calculator is started as soon as a correct value for D is found. Then a comparator tests if $U_{n+1}(a, b) = 0$. The test finishes and returns 1 if $U_{n+1}(a, b) = 0$ (the number is declared probably prime) or 0 if $U_{n+1}(a, b) \neq 0$ (the number is declared composite).

IV. Results

A. FPGA results

We implement our design on a Virtex-5 XC5VLX330T-1 FPGA which is part of an Alpha Data ADM-XRC-5T2

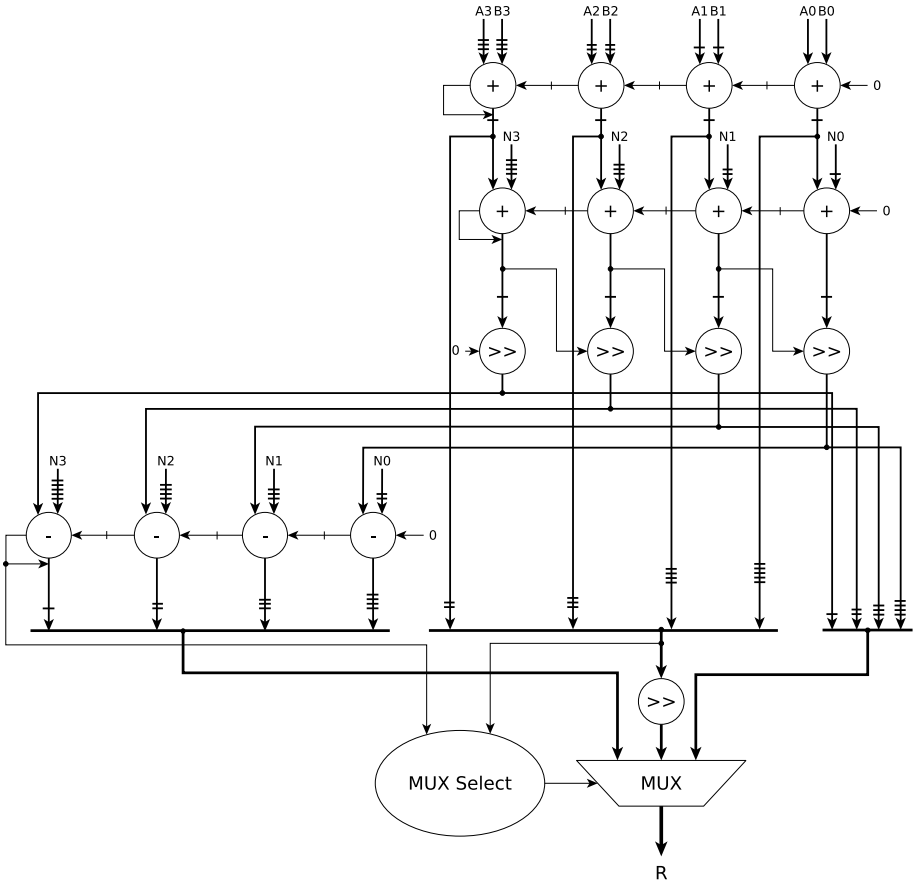

Fig. 2. Hardware design for modular add-shift

Instructions
1. $U_{temp} = U.V \mod n$
2. $V_{temp} = V^2 \mod n$
3. $temp1 = U^2 \mod n$
4. $temp1 = D.temp1 \mod n$
5. $V_{temp} = (V_{temp} + temp1)/2 \mod n$
6. $temp2 = D.U_{temp} \mod n$
7. $U = \begin{cases} U_{temp} & \text{if } k_i = 0 \\ (U_{temp} + V_{temp})/2 \mod n & \text{if } k_i = 1 \end{cases}$
8. $V = \begin{cases} V_{temp} & \text{if } k_i = 0 \\ (V_{temp} + temp2)/2 \mod n & \text{if } k_i = 1 \end{cases}$

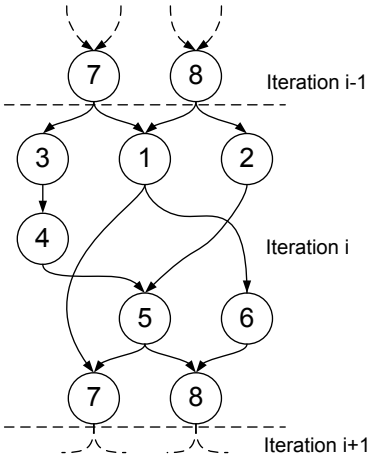

Fig. 3. Analysis of the true dependencies in the Lucas sequence calculation algorithm

Clock	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Inst.3	M1	M1	M2	M2	M3	M3											M1	M1	M2	...
Inst.1			M1	M1	M2	M2	M3	M3											M1	...
Inst.2					M1	M1	M2	M2	M3	M3										...
Inst.4							M1	M1	M2	M2	M3	M3								...
Inst.6									M1	M1	M2	M2	M3	M3						...
Inst.5													A1	A2						...
Inst.7															A1	A2				...
Inst.8																A1	A2			...

Fig. 4. Lucas sequence calculator schedule for a 3-stage multiplier's pipeline

board. Our hardware design is compared with an optimised software implementation of Alg. 2 using dedicated functions of the GMP library 5.0.1 for all the operations and compiled with the Intel Compiler 12.0. The targeted architecture for our software implementation is an Intel Xeon W3505 @ 2.53 GHz with 3 GB of main memory. Only one core of the Xeon processor is considered. We report the average execution time, the dynamic power consumption and the average energy for both the hardware and the software implementations. A random set of 10 000 1024-bit values is used as input. For the software version, the dynamic power is measured with a watt-metre at the output of the system. The dynamic power of the FPGA is estimated using the XPower Estimator spreadsheet v. 12.3 after mapping the design for the targeted board. The activity rate is kept to its default value of 12.5%. The synthesis operations reporting a clock of around 200 MHz for all the variations of our designs, we set the clock of the FPGA to a realistic value of 150 MHz for this estimation.

The results are reported in Tab. I. The parameters N_p and N_r represent respectively the number of pipeline stages of our Montgomery multiplier and the number of replicated carry-save adders in each pipeline block.

TABLE I
FPGA IMPLEMENTATION RESULTS OF 1024 BIT LUCAS PRIMALITY TESTERS

Type	Area (LUTs)	Average Ex. Time (ms)	Dynamic Power (W)	Average Energy (mJ/op)
Hardware ($N_p = 1$, $N_r = 1$)	56 329	24.34	5.3	129.00
Hardware ($N_p = 1$, $N_r = 2$)	58 739	12.66	5.4	68.37
Hardware ($N_p = 2$, $N_r = 1$)	60 431	13.16	5.4	72.38
Hardware ($N_p = 2$, $N_r = 2$)	70 651	7.33	5.8	42.51
Hardware ($N_p = 3$, $N_r = 1$)	68 386	11.27	5.8	65.37
Hardware ($N_p = 3$, $N_r = 2$)	82 792	6.61	6.0	39.66
Hardware ($N_p = 3$, $N_r = 4$)	106 158	4.29	6.1	26.17
Hardware ($N_p = 3$, $N_r = 8$)	142 647	3.11	6.2	19.28
Software	N/A	2.40	26	62.40

We see that the execution time decreases almost linearly with the number of replications N_r of our multiplier. The execution time is less impacted by an increase in the number of pipeline stages N_p. This is due to the fact that the pipeline of the multiplier is not always run full. Recall that $N_p = 3$ is a theoretical maximum for the number of pipeline stages due to the date dependencies in the Lucas algorithm. On the contrary, the only factor limiting N_r is the area available. The power consumption of our design slightly increases with N_r and N_p as the area and clock fanout increase. However this increase in power consumption is negligible compared with the decrease in execution time. In fact, the performance scaling of our architecture is much better than linear in area. Hence the average energy consumed by our design is still significantly reduced when increasing N_r and N_p.

Our most energy efficient design is 30% slower but 3 times more energy efficient than the software version. This design takes 70% of the total area available in the Virtex-5 LX330T. Most of the time is spent in Montgomery multiplications. The particular structure of our Montgomery multiplier makes it much faster than a software implementation if inputs can be provided at a high throughput without stalling the pipeline. In this application, the data dependencies in the algorithm limit the maximum value for the number of pipeline stages N_p and therefore the achievable speed of our multiplier. However, the results could be improved further by increasing the number of replications N_r if a bigger FPGA is available. In fact, the scalability of our design in terms of replications is only limited by the area available. Upcoming FPGAs such as Virtex 7 will be several times bigger than this Virtex-5 allowing much better performance while still having unused area. Hence we expect the energy efficiency of our FPGA designs would improve with advances in FPGA technology. It should also be noted that the Virtex-5 FPGA (65 nm) is one generation behind the Xeon W3505 CPU (45 nm).

Server applications would benefit from augmenting the CPU with one of more FPGAs configured with this design. In these applications, power consumption can turn out to be as important as performance. As a matter of fact, improving the energy efficiency of the system can lead to huge savings in fixed costs (cooling equipment) and variable costs (energy bill).

B. ASIC results

We synthesize our design for the TSMC 65 nm *tcbn65gplustc* library using Synopsys Design Compiler version D-2010.03-SP5-1. Our fastest design in 65 nm is also synthesized for the TSMC 45 nm *tcbn45gsbwptc* library to allow a fair comparison with the Xeon W3505 CPU which also adopts 45 nm technology. We suppose that all the inputs of our Lucas module are driven by a D flip-flop. We also assume that all the outputs of our module have the capacitive load of the same D flip-flop. We estimate the power of each synthesized architecture using PrimeTime-PX. For our estimation, a single 1024-bit input is considered. This input goes through all the steps of Alg. 2, requiring a full calculation of the Lucas sequence. Hence, we get worst-case results for the execution time and the dynamic power consumption. In order to obtain a more accurate estimation of the power consumption, a virtual clock network is created for each architecture. Our results are reported in Tab. II. The area is given in 2-input NAND gates. One NAND gate corresponds to 4 transistors.

We see that for our 65 nm architectures running at 1 GHz, the variation of the execution time with N_r and N_p follows the same trend as the FPGA implementation. Namely the execution time decreases almost linearly with N_r whereas it is less impacted by an increase in N_p. However, for N_r greater than 4, the replicated carry-save adders of our Montgomery multiplier are on the critical path of the Lucas module. Hence

TABLE II
ASIC IMPLEMENTATION RESULTS OF 1024 BIT LUCAS PRIMALITY TESTERS

Type	Area (kGates)	Freq. (GHz)	Exec. Time (ms)	Dynamic Power (mW)	Average Energy (mJ/op)
65 nm ASIC ($N_p = 1$, $N_r = 1$)	400	1.00	5.43	206.7	1.12
65 nm ASIC ($N_p = 1$, $N_r = 2$)	416	1.00	2.82	215.7	0.61
65 nm ASIC ($N_p = 2$, $N_r = 1$)	464	1.00	2.93	249.2	0.73
65 nm ASIC ($N_p = 2$, $N_r = 2$)	498	1.00	1.62	264.2	0.43
65 nm ASIC ($N_p = 3$, $N_r = 1$)	533	1.00	2.51	281.3	0.71
65 nm ASIC ($N_p = 3$, $N_r = 2$)	595	1.00	1.46	297.1	0.43
65 nm ASIC ($N_p = 3$, $N_r = 4$)	709	0.75	1.69	223.9	0.38
65 nm ASIC ($N_p = 3$, $N_r = 8$)	838	0.50	2.01	149.4	0.30
45 nm ASIC ($N_p = 3$, $N_r = 2$)	618	1.66	0.90	220.4	0.20
Software	N/A	2.53	3.30	28 000	92.40

the Lucas prime tester's maximum frequency decreases, which results in a significant performance lose. This threshold in the maximum value of N_r is not observable on our FPGA implementations as they run at a much lower frequency.

Our fastest 65 nm ASIC implementation is more than 2 times faster than the FPGA implementation while only consuming 300 mW. It is at least 40 times more energy efficient. Our fastest 45 nm ASIC implementation is 3.6 times faster and 400 times more energy efficient than the optimised software implementation. The cell count and the power consumption of our implementations make them suitable for integration into an embedded system, different speed/area/energy trade-offs being available through parametrization. Using our architecture, the Lucas test could therefore be integrated into a customisable core.

C. Security

Another primordial aspect of crypto-systems, often considered more important than speed and energy consumption, is security. If we suppose that an attacker can obtain physical access to the crypto-system, it seems clear that a hardware implementation of the Lucas test is more secured than its software counter-part. In a software environment, vulnerabilities of the operating system and both software-based [4] and hardware-based [13] attacks can be exploited to compromise the crypto-system. A hardware-based crypto-system is more robust in the sense that it is only vulnerable to hardware attacks. Moreover a hardware crypto-system can be effectively designed to prevent these attacks whereas a software will always be dependant on the security of both the underlying operating system and the underlying hardware.

V. CONCLUSION AND FUTURE WORK

This paper presents our parametric hardware architecture of the NIST approved Lucas probabilistic primality test. Primality testing is needed by the key generation process of many public-key crypto-systems. The use of a Lucas test as the last step of a probabilistic prime tester can greatly reduce the probability of error of the overall test. To our knowledge, our work is the first hardware implementation of this test in the literature. Our architecture implemented on a Virtex-5 FPGA is 30% slower but 3 times more energy efficient than the software version running on a Intel Xeon W3505. A 65 nm ASIC implementation is more than 2 times faster and 40 times more energy efficient than the FPGA implementation. A 45 nm ASIC implementation is 3.6 times faster and 400 times more energy efficient than the optimised software implementation in comparable technology. Our design is scalable and the performance scaling of our architecture is much better than linear in area. The cell count and the power consumption of our ASIC implementations make them suitable for integration into an embedded system whereas our FPGA implementation would more likely benefit server application. Moreover if we suppose that an attacker can obtain physical access to the crypto-system, implementing such a test purely in hardware greatly reduces the vulnerability of a crypto-system when compared to a software implementation.

Current and future work includes pipelining our Jacobi symbol calculator module and investigating a systolic array solution, making the instruction scheduling of our Lucas primality tester parametric, developing models and tools to find the optimum instructions scheduling given the bit-width and the pipeline depth of the different sub-modules, and developing protections against side-channel attacks.

REFERENCES

[1] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P," *Ann. of Math*, vol. 2, pp. 781–793, 2002.

[2] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128 – 138, 1980.

[3] Information Technology Laboratory (NIST), "Digital Signature Standard," pp. 68–76, FIPS PUB 186-3.

[4] D. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology, CT-RSA 2006*, vol. 3860, 2006, pp. 1–20.

[5] Z. S. McGregor-Dorsey, "Methods of primality testing," *MIT Undergrad. Journal of Math.*, vol. 1, 2000.

[6] R. Baillie and S. S. Wagstaff, Jr., "Lucas pseudoprimes," *Math. Comp.*, vol. 35, pp. 1391–1417, 1980.

[7] C. Pomerance, J. L. Selfridge, and S. S. W. Jr, "The pseudoprimes to 25e9," *Math. Comp.*, vol. 35, pp. 1003–1026, 1980.

[8] M. Joye and J.-J. Quisquater, "Efficient computation of full Lucas sequences," *Electronics Letters*, vol. 32, no. 6, pp. 537–538, Mar 1996.

[9] J. W. Crenshaw, "Integer square roots," http://www.embedded.com/98/9802fe2.htm.

[10] J. Shallit and J. Sorenson, "A binary algorithm for the Jacobi symbol," *SIGSAM Bull.*, vol. 27, no. 1, pp. 4–11, 1993.

[11] G. Purdy, C. Purdy, and K. Vedantam, "Two binary algorithms for calculating the Jacobi symbol and a fast systolic implementation in hardware," in *49th IEEE Inter. Midwest Symp. on Circuits and Systems*, vol. 1, Aug. 2006, pp. 428 –432.

[12] A. Le Masle, W. Luk, J. Eldredge, and K. Carver, "Parametric encryption hardware design," in *6th Inter. Symp. on Reconf. Computing*, 2010, pp. 68–79.

[13] A. Pellegrini, V. Bertacco, and T. Austin, "Fault-based attack of RSA authentication," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*, 2010, pp. 855–860.