FPGA Implementation of High Throughput Circuit for Trial Division by Small Primes

Gabriel Southern, Chris Mason, Lalitha Chikkam,
Patrick Baier, and Kris Gaj

George Mason University
{gsouther, cmason4, lchikkam}@gmu.edu
districtline@gmx.net
kgaj@gmu.edu

Abstract

Trial division is the most straightforward way to determine the prime factors of a number, but the execution time is exponentially dependent on the size of the number. We have developed a novel hardware architecture which performs trial division of large dividends by small prime divisors at a much higher throughput than previously reported architectures. Our design is implemented in FPGA devices and provides a speed-up of between one and two orders of magnitude over an optimized software implementation of the same algorithm. These results can be employed to speed up factoring algorithms like the quadratic sieve or number field sieve when implemented in reconfigurable computers.

1 Introduction

The hardness of factoring large integers is the basis of the security of the RSA asymmetric encryption algorithm. The Number Field Sieve (NFS) is the most efficient known method for factoring large integers. Several special purpose hardware devices such as TWINKLE [1], TWIRL [2], and SHARK [3] have been proposed which provide an estimate of the cost of building a machine capable of factoring a number in the range of 1024 bits. However, these machines are designed to be implemented using ASICs (Application Specific Integrated Circuits), which are only practical when produced in large volumes.

An alternative approach involves the use of FPGAs in a reconfigurable computer configuration. An advantage of this approach is that the costs of developing the FPGA components can be amortized across a wide range of applications. The COPACOBANA machine is a reconfigurable computer built using Spartan-3 FPGAs that is able to break DES encryption in less than 9 days [4]. While COPACOBANA is not yet capable of breaking RSA encryption,

its success against the once popular DES encryption algorithm demonstrates the effectiveness of reconfigurable computing in performing cryptanalysis.

The NFS process can be divided into four steps: polynomial selection, relation collection, linear algebra, and square root. Of these four, the relation collection step and the linear algebra step are the two most computationally intensive steps. The relation collection step is typically divided into two parts, sieving and cofactoring. The cofactoring stage can be divided into a brute force trial division step to identify small primes (less than 100,000) followed by more complex probabilistic methods such as the rho method, p-1 method, or elliptic curve method (ECM) [5]. Efficient hardware implementation of these methods is a subject of extensive research [5, 6, 7, 8, 9, 10, 11]. Less well studied are techniques to develop an efficient hardware implementation to perform the trial division by small primes. We have developed a new hardware architecture that performs the trial division step used to identify the small primes efficiently and implemented it in the Xilinx family of FPGAs.

In Section 2 we describe the problem that our circuit solves and its interface. In Section 3 we describe the circuit design in detail as well as the software implementation that we developed to provide test vectors and performance comparison results. In Section 4 we present our results and provide an analysis of the performance speedup obtained by the hardware implementation over software. Finally, in Section 5 we summarize our results, and provide some comments on the use of reconfigurable computing to perform cryptanalysis.

2 Circuit Interface

Our circuit is designed to solve the following problem: Given Inputs:

- Variables: Integers N_1, N_2, N_3, ... each of the size of k bits
- Constants: set of all primes smaller than a certain bound B $\{p_1 = 2, p_2 = 3, p_3 = 5, ..., p_t \leq B\}$

Outputs: For each integer N_i: A list of primes from the factor base that divide N_i, and the number of times each prime divides N_i. For example, if $N_i = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3} \cdot M_i$, where M_i is not divisible by any prime below B, then the output is: $(p_1, e_1), (p_2, e_2), (p_3, e_3), (M_i)$.

In NFS, sieving and cofactoring are performed for two sets of numbers, obtained by evaluating values of two different polynomials $F1(a, b)$ and $F2(a, b)$, for multiple pairs of integers (a, b). The respective computations are referred to as a rational side and an algebraic side of the relation collection step of NFS, respectively. For parameters of NFS proposed in [2], the sizes of numbers obtained after sieving have been estimated to be 216 bits for the rational side and 350 bits for the algebraic side [6].

Our circuit was designed to use $B = 100,000$, which resulted in a set of 9592 primes with which we performed trial division. The circuit was designed and tested for k equal to 512 bits; however, we also synthesized our circuit for

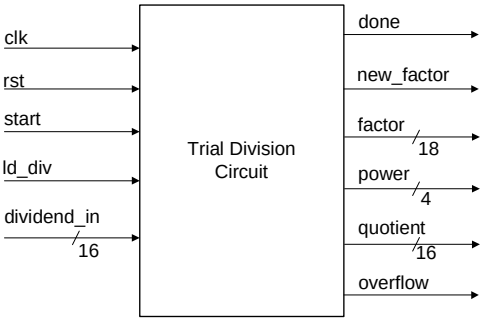


Figure 1. Interface of trial division circuit

Table 1. Description of trial division circuit interface

Signal	Purpose
clk	Clock signal for circuit
rst	Reset signal for circuit
ld_div	Indicates a new number is being loaded
start	Start processing
dividend_in	Number that will be processed
done	Processing is complete
new_factor	A new factor has been discovered
factor	Value of the new factor
power	Number of times the factor divided the dividend
quotient	Remaining quotient after processing
overflow	Indicates only prime factors identified

values of k equal to 216 and 350 bits in order to provide performance estimates for inputs of these sizes. Figure 1 is a diagram of the interface and Table 1 describes the purpose of the signals.

3 Design

3.1 Circuit Design

Our circuit was optimized to achieve maximum throughput of numbers fully processed per unit of time. The operations required to perform division are sequential, and typically are not well suited for parallel operations that can lead to increased throughput. However, we were able to take advantage of the fact that we were performing trial division on a fixed dividend with 9592 different divisors. This allowed us to develop a highly pipelined architecture that was relatively cost efficient. We also took advantage of the fact that the divisor

could be represented in fewer bits than the dividend. We stored negated values of prime numbers used as divisors in two's complement form. This allowed us to represent all values of the divisors in 18 bits. After the pipeline was full our circuit was able to determine the divisibility of a 512-bit number by each next small prime in one clock cycle, i.e., in the time required to add two 18-bit numbers using a ripple carry adder.

We targeted our design for FPGAs and specifically focused on the Xilinx Spartan-3 family of chips, although the VHDL code was written using a generic RTL style that should be applicable for any FPGA family. However, two features that we specifically targeted in the Xilinx FPGA family were the block RAM that we used as a ROM to store the prime divisors and the fast ripple carry logic.

While the majority of the division operations for a specified input value were performed with a fixed dividend, when a factor of the dividend was discovered we performed additional operations on the resulting quotient. In order to accomplish this, we partitioned our circuit into a large pipelined array divider and a smaller multi-cycle sequential divider.

The circuit was divided into the following major components: a highly pipelined array divider, a multi-cycle sequential divider, and a ROM to store the prime divisors, along with a control unit and registers to control dataflow, as shown in Figure 2. The circuit was designed to accept a large input value, typically 512 bits, so we used a shift register to store the number that was being processed as the input. This allowed the next number to be loaded while the current number was being processed, so that the use of a multi-cycle input process did not reduce the overall throughput of the circuit.

The number being processed was stored in a register during the entire processing time, and it provided the dividend input to the array divider. Each clock cycle, a new divisor was selected from the set of possible prime factors. These 9592 primes were stored in a block RAM portion of the FPGA device. The array divider was designed to determine divisibility of a number, and rather than storing the quotient, it produced a remainder and the associated divisor as output. Any divisor that produced a remainder of zero was a prime factor of the dividend, and was stored in a FIFO queue for further processing by the sequential divider.

Algorithm 1 shows the steps that the circuit follows to process a new input number. After initialization the sequential divider waits until a value is entered into the FIFO queue. At that point it processes the first prime factor that the array divider identified. The dividend register associated with the sequential divider is initially loaded with a new value at the same time as the dividend register used with the array divider during circuit initialization. However, the sequential divider has a different mode of operation. Instead of only producing a remainder result, the sequential divider also produces a quotient value. If the remainder from the division operation is equal to zero, then the power count

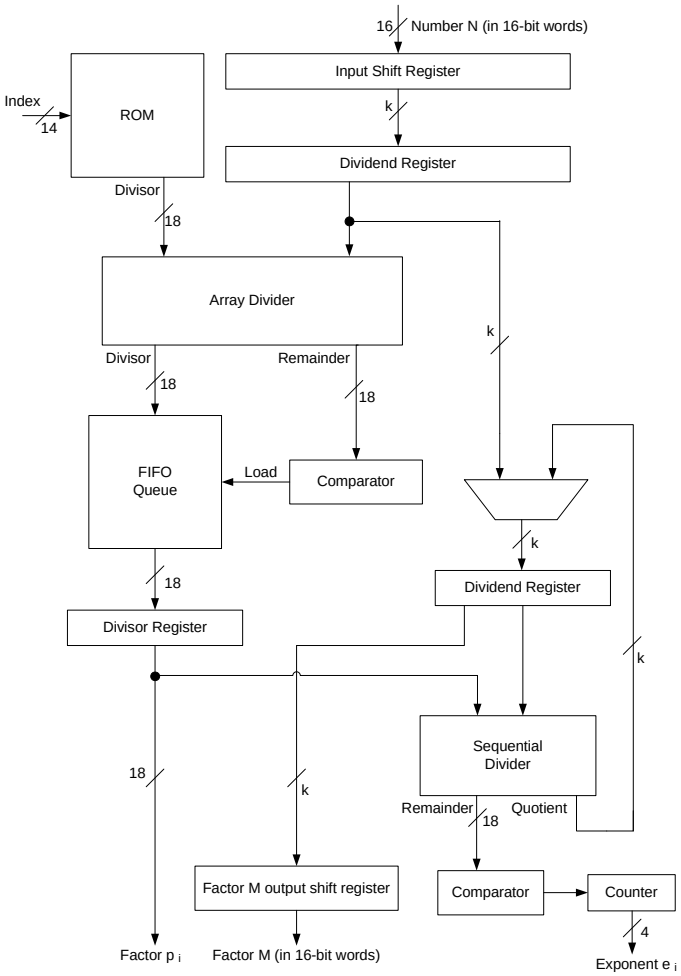


Figure 2. Trial division circuit components

```
Data: Dividend input: N = p_1^{e_1} · p_2^{e_2} ··· p_{c-1}^{e_{c-1}} · p_c^{e_c} · M
Result: Prime factors p_i, corresponding exponents e_i, and factor M
foreach prime p in prime_set do
    if p divides N then
        validPrimeList.Add(p);
    end
end
while validPrimeList.HasElements do
    prime p = validPrimeList.GetNextPrime();
    count = 1;
    while p divides N do
        N = N / p;
        count = count + 1;
    end
    resultList.Add(p, count);
end
```

Algorithm 1: Trial division circuit algorithm

for that factor is incremented by one, the resultant quotient is loaded into the dividend register, and the division operation is repeated. If the remainder is not zero, then the factor is provided as output from the circuit and the current count value is provided as the power output signal. The resultant quotient is not loaded into the sequential divider's dividend register; instead, the circuit waits for the next value in the FIFO queue to process.

When the pipeline is full, the array divider completes division with one divisor, and produces an associated remainder output every clock cycle. In contrast, the sequential divider requires k-1 clock cycles to perform division and produce a remainder and quotient output. Our analysis indicated that we could expect relatively few small factors to be identified, and thus the sequential divider would not be expected to reduce the overall throughput of the circuit. If the array divider did identify factors at a faster rate than the sequential divider could process them, the circuit would either have to stall or overflow. We chose to implement the overflow method. When this occurs, the circuit asserts the overflow line and only performs trial division using the array divider. The result is that in overflow mode the circuit identifies the prime factors, but not the powers of primes.

The number of division operations that the sequential divider could perform without overflow depended on the size of the dividend, because the throughput of the sequential divider was linearly dependent on the size of the dividend while the throughput of the array divider was nearly constant. For the dividend sizes that we tested, the results were 18 division operations for a 512-bit dividend, 27 division operations for a 350-bit operand, and 44 division operations for a 216-bit operand. The worst-case scenario would correspond to the case where each prime factor identified was raised to a single power. In this case, each

factor identified would require two division operations from the sequential divider which would result in the circuit entering overflow mode after identifying 9 factors.

3.2 Expected Number of Primes

In choosing how to handle the case of overflow, we performed an analysis using a software implementation of our algorithm to determine the likelihood of overflow occurring. The program was written to perform the procedure described in Algorithm 1 to determine the prime factors and powers of primes for a series of random numbers. Our experimental results indicated that only about 0.012% of the numbers sampled had more than 9 small prime factors, and thus our circuit would be able to operate normally in most cases. Having determined experimentally that the numbers we performed trial division on would have relatively few small prime factors, we developed the following theoretical explanation which verified the results we observed experimentally:

Let $r \in \mathbb{N}$ and let $p_1 = 2, p_2 = 3, \ldots, p_r$ be the r smallest primes; for $r = 9592$, this is the set of primes below $B = 100,000$. To determine the asymptotic probability q_c that a random integer has c distinct prime divisors below B, we let $Q(n, B)$ be the number of integers $1 \leq x \leq n$ such that x has exactly c distinct prime divisors below B, and we calculate

$$q_c = \lim_{n \to \infty} \frac{Q(n, B)}{n}.$$

The probability of a random integer being divisible by p_i is $\frac{1}{p_i}$, or equivalently it is not divisible by p_i with probability $(1 - \frac{1}{p_i})$. The probability that a random integer has no prime divisor below B is the product of these probabilities:

$$q_0 = \prod_{i=1}^{r} \left(1 - \frac{1}{p_i}\right).$$

While this can be computed exactly for small enough B, direct computation is not feasible for larger values. However, good approximations can be computed.

To calculate the probability that a random integer is divisible by some fixed p_j, but no other prime (below B) we replace the factor $(1 - \frac{1}{p_j})$ in the above product by the complementary probability $\frac{1}{p_j}$. Equivalently, this can be written as

$$q_0 \left(\frac{\frac{1}{p_j}}{1 - \frac{1}{p_j}} \right) = \frac{q_0}{p_j - 1}.$$

To obtain q_1 (the probability that a random integer has exactly one prime divisor below B) we sum over j and get

$$q_1 = q_0 \left(\sum_{j=1}^{r} \frac{1}{p_j - 1} \right).$$

We write $m_1 = \sum_{j=1}^{r} \frac{1}{p_j - 1}$ so that $q_1 = q_0 m_1$ and aim to find m_c for $c > 1$ so that $q_c = q_0 m_c$ for all $c \leq r$. Setting $m_0 = 1$, for consistency in the case $c = 0$, we obtain

$$m_c = \sum_{j_1 < \cdots < j_c} \prod_{l=1}^{c} \frac{1}{p_{j_l} - 1}.$$

That is, we sum the products over all distinct c-tuples of primes. This can be calculated more efficiently as follows. For $c > 0$ let

$$b_c = \sum_{i=1}^{r} \left(\frac{1}{p_i - 1} \right)^c.$$

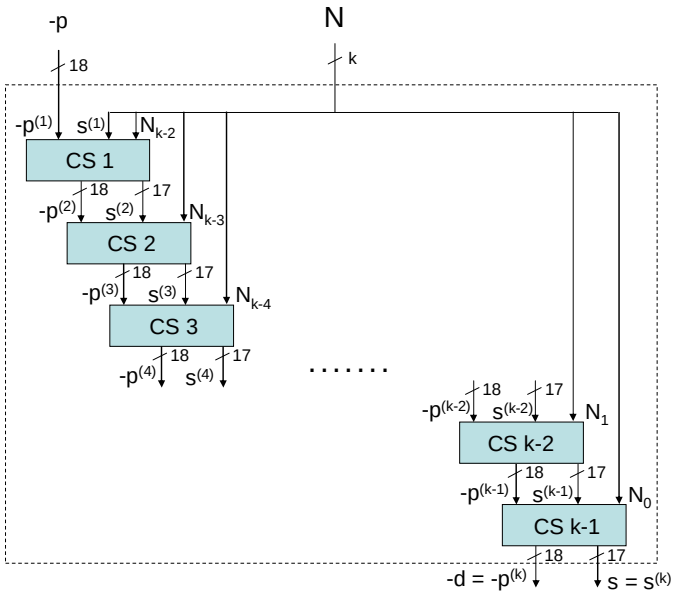
Then we can calculate inductively

$$\begin{aligned} m_{c+1} &= \frac{1}{c+1} \left(m_c b_1 - m_{c-1} b_2 + \cdots \pm m_0 b_{c+1} \right) \\ &= \frac{1}{c+1} \sum_{j=0}^{c} (-1)^j m_{c-j} b_{j+1} \end{aligned}$$

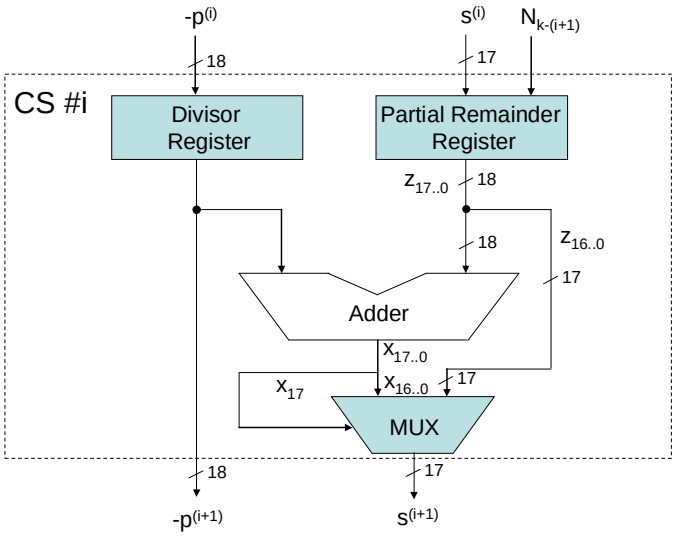
This is effectively computable in time $O(c^3 r)$. We calculated the distribution values for $B = 100,000$ and $c = 10$, and these results are shown in Table 2 along with our experimental results.

Table 2. Probability q_c of c prime factors less than $100,000$ in a random number determined by experimentation and calculated results.

c	Calculated q_c	Experimental q_c
0	0.048753	0.048570
1	0.169584	0.168777
2	0.261423	0.261627
3	0.244033	0.243688
4	0.157985	0.157970
5	0.076659	0.077459
6	0.029327	0.029584
7	0.009167	0.009327
8	0.002404	0.002369
9	0.000540	0.000510
10	0.000105	0.000104
11	0.000018	0.000016
12	0.000003	0.000000
> 12	0.000000	0.000000



(a) Array Divider



(b) CS Cell

Figure 3. Basic layout of pipelined array divider (a) composed of controlled subtractor cells (b)

Data: N (k-bit integer), $-p$ (where p is a small prime $p < 2^{17}$)
Result: $s = N \bmod p$
$s^{(1)} = 000 \cdots 0 N_{k-1}$; *17 zeros followed by* N_{k-1}
for *i=1 to k-1* **do**
  $s^{(i)} = s^{(i)} || N_{k-(i+1)}$; *equivalent to* $s^{(i)} = 2 \cdot s^{(i)} | N_{k-(i+1)}$
  if $(s^{(i)} + (-p^{(i)})) > 0$ **then**
    $s^{(i+1)} = s^{(i)} + (-p^{(i)})$
  else
    $s^{(i+1)} = s^{(i)}$
  end
  $s = s^{(k)}$
end
Note: $||$ *denotes concatenation,* $|$ *denotes bitwise or*

Algorithm 2: Array Divider Pseudocode

3.3 Array Divider

The array divider occupied most of the circuit area and provided the basis for its high throughput. It was designed to determine the divisibility of a single dividend with each of the 9592 different divisors from the set of possible prime factors. Our analysis indicated that most of these prime divisors would not divide the dividend, and as a result we discarded the quotient and only stored the remainder output. We developed our design with a target dividend size of 512 bits however, through modification of a generic value we were also able to produce designs with dividends of 350 bits and 216 bits. The size of the dividend input did not change the critical path in the array divider; instead, it changed the degree of pipelining. Algorithm 2 describes the operation of the array divider. The block diagram of the circuit implementing this algorithm is shown in Figure 3. A k-bit dividend required k-1 pipeline stages, where each pipeline stage consisted of a divisor register, a partial remainder register, a ripple-carry adder, and a multiplexer. These components combined to form the controlled subtractor cell.

The negated divisors were stored in two's complement form and required 18 bits to store values in the range of -2 down to $-100,000$. The dividend was zero extended by one less bit than the length of the divisor (i.e., by 17 zeros) in order to support a quotient as large as the dividend. In the first pipeline stage the upper 18 bits from the zero-extended dividend were provided as input to the partial remainder register. The two register values were added together and if the result was positive then the lower 17 bits of the result were appended with the next bit from the dividend and provided as the partial remainder result to the next stage. If the result of the addition was negative then the lower bits from the existing partial remainder were appended with the next dividend bit and provided as input to the partial remainder for the next stage. The value stored in the dividend register was kept constant during the processing of a

number, while a new negated divisor value, -p, was provided each clock cycle. By keeping the dividend constant we limited the data stored in each pipeline stage to an 18-bit partial remainder and an 18-bit divisor associated with that remainder.

3.4 Sequential Divider

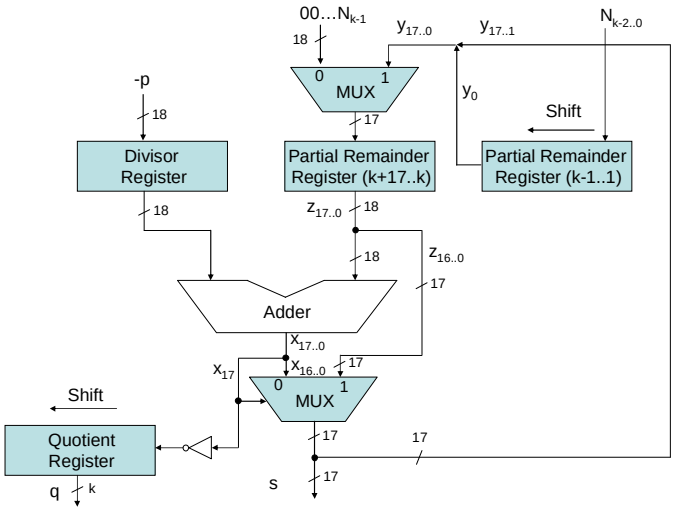
The sequential divider shown in Figure 4 was designed to minimize area without increasing the clock period, and was implemented as a shift/subtract restoring divider [12]. Algorithm 3 shows the classical shift/subtract restoring divider algorithm that we based our implementation on. In Figure 4a we present the direct translation of Algorithm 3 into hardware. In this circuit, all 18 bits of the negated divisor, $-p$, are used, the adder is 18-bit wide, and the most significant bit of the sum is used to determine the new value of the partial remainder. In Figure 4b, an optimized circuit, implementing the same algorithm is shown. The most significant bit of the negated divisor, $-p$, is not used, because this bit is always 1. The adder width is reduced from 18 bits to 17 bits. The most significant (18th) bit of the sum is calculated with two simple logic gates (OR and inverter), based on the most significant bit of the partial remainder and the carry out signal from the 17-bit adder. The two designs are functionally equivalent, with the design in Figure 4b using slightly less resources and having a slightly shorter critical path. The majority of this critical path is the time necessary to perform a 17-bit ripple carry addition. Both designs shown in Figure 4 require k clock cycles to process one full division. In our final optimization, the number of required clock cycles was reduced from k to $k-1$ based on the fact that the divisor p is always greater or equal 2, and thus $q_{k-1} = 0$. Although the circuit was designed to process 512-bit numbers, the clock period was determined by the size of the divisors. Thus, changing the dividend size to 350-bits or 216-bits increased latency but did not change the critical path, just as in the array divider.

3.5 Software Implementation

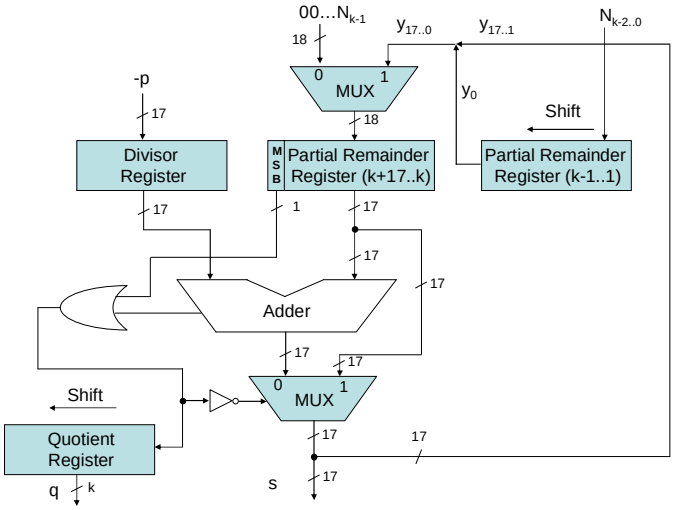
We developed a software implementation to perform trial division using the same algorithm that we implemented in hardware as shown in Algorithm 1. The software implementation was initially used as a source of test vectors but was later optimized to provide performance metrics to compare to our hardware implementation. The software was written in C using the GNU Multiple Precision Arithmetic Library (GMP) and was optimized for performance.

4 Results

We synthesized our design, using the Xilinx ISE Foundation tool, for the Virtex-4 and Spartan-3 FPGA families. Table 3 shows the resources required to implement the circuit in the two families of Xilinx FPGAs. The limiting resource was



(a) Sequential restoring divider



(b) Optimized sequential restoring divider

Figure 4. Basic shift/subtract sequential restoring divider (a) and optimized version (b)

Data: N (k-bit integer), $-p$ (where p is a small prime $p < 2^{17}$)
Result: $q = N/p; s = N \bmod p$
$s^{(0)} = 000 \cdots 0 || N$; 18 zeros followed by N
for *i=1 to k* **do**
  if $(2s^{(i-1)} + 2^k(-p)) > 0$ **then**
    $q_{k-i} = 1$
    $s^{(i)} = 2s^{(i-1)} + 2^k(-p)$
  else
    $q_{k-i} = 0$
    $s^{(i)} = 2s^{(i-1)}$
  end
end
$q = q_{k-1 \cdots 0}$
$s = s^k_{k+16 \cdots k}$

Algorithm 3: Sequential Divider Pseudocode

CLB slices and the pipelined array divider dominated the total circuit area. For the Spartan-3 family the circuit could be implemented in the XC3S1500 device for the dividend sizes of 216 and 350 bits, but it required the XC3S2000 device for the 512 bit dividend. For the Virtex-4 family the XC4VLX25 device was used for the 216 bit dividend, but the XC4VLX40 device was needed for the 350 and 512 bit dividends.

Table 3. Circuit area in CLB slices

	Spartan-3		Virtex-4	
Dividend	Total	Array Divider	Total	Array Divider
512 bits	16,922	15,323	16,895	15,323
350 bits	11,614	10,462	11,578	10,462
216 bits	7,216	6,441	7,182	6,441

Table 4 shows the timing results for the various different size dividends. The clock period for an 18-bit ripple carry adder and multiplexer is included as our target critical path through the controlled subtractor cell. The array divider area dominates the size of the circuit, and it would not be possible to reduce the critical path through the controlled subtractor cells without a significant increase in circuit area. However, initial results indicate significant opportunities to further optimize our circuit before we reach this limitation. Currently the circuit's critical path is in the sequential divider, which has a very simple implementation and could be optimized so that it is not in the critical path. We also have a simple state-machine control unit implementation that has a larger critical path than the controlled subtractor cell. While optimizing the state machine would be more complex, it should not result in a significant increase in circuit area.

Table 4. Circuit minimum clock period

Circuit	Spartan-3	Virtex-4
18-bit RC adder and MUX	4.483 ns	2.046 ns
512-bit circuit	11.028 ns	4.182 ns
350-bit circuit	10.917 ns	4.182 ns
216-bit circuit	11.084 ns	4.006 ns

Table 5 shows the throughput in terms of numbers fully processed per second. The optimal throughput is based on the clock period estimate using the critical path in the controlled subtractor cell, while the actual throughput is based on the clock period results after synthesis. Each number takes 9592 cycles plus one additional cycle for each pipeline stage to fully process. In addition, required synchronization cycles are estimated at 10 cycles per number. The processing of the different primes dominates the total delay, and the overall throughput for processing a 512-bit number is very similar to the throughput for a 216 bit number.

Table 5. Circuit throughput in numbers per second

	Spartan-3		Virtex-4		Software
Dividend	Optimal	Actual	Optimal	Actual	Version
512 bits	22,057	8,966	48,329	25,425	434
350 bits	22,416	9,205	49,116	24,029	594
216 bits	22,722	9,190	49,786	23,644	884

The results for the software version were determined experimentally by running our program on a Xeon 2.8 GHz CPU system with 512KB cache and 4GB of RAM. We used the fastest available GMP routines and compiled our program with all compiler optimizations turned on. It is likely that additional performance improvements could be obtained if the code was optimized for a particular architecture; however, we believe this implementation is near optimal for an implementation written in a portable coding style and we expect any further performance gains achieved would be less than 10%. While the hardware implementation has nearly constant throughput, the software implementation is linearly dependent on the size of the dividend. As a result, the hardware speedup is greater for the larger dividend sizes.

Our circuit is able to achieve a significant speedup over a software implementation of the same algorithm because the operations it performs are optimized at the digital logic layer for a specialized application for which a general purpose microprocessor is not optimized. The problem our circuit solves would only be really useful as part of a larger application running on a reconfigurable computer

Table 6. Circuit speedup over software

	Spartan-3		Virtex-4	
Dividend	Optimal	Actual	Optimal	Actual
512 bits	51	21	111	56
350 bits	37	15	83	40
216 bits	26	10	56	27

able to perform the NFS factoring algorithm. Such a reconfigurable computer would be implemented using a combination of CPU and FPGA processing elements as well as memory, network interfaces, circuit board interconnects, power supply, cases, and other components. We have limited our analysis to the cost of the processing elements when performing our cost analysis; however, we expect that the peripheral costs associated with an FPGA device, such as the Spartan 3, typically used in embedded applications, would be less than those of a microprocessor, such as an Intel Xeon, designed for use in high performance servers. Table 7 provides an estimate of the cost of the FPGA and CPU elements based on advertised prices at the time of publication. Pricing for FPGAs and CPUs is volatile and is dependent on the number of devices ordered, however, advertised prices allow for an estimate of cost/performance between the different device families. The prices listed in Table 7 are for FPGA devices purchased in quantities of 100 from two different vendors, one for the Spartan 3 devices [13], and a different one for the Virtex 4 devices [14]. The price listed for Xeon CPU is based on [15], and is for CPUs purchased in single unit quantities. The results show that even the most expensive FPGA devices cost less than the CPU used for our software implementation. They also indicate that the low-cost Spartan 3 device will provide the best overall performance/price ratio. The Intel Xeon MP 2.8 GHz CPU is listed because this is the CPU in the servers we used for testing our software implementation. It is likely that a different CPU would provide a better performance/cost ratio than the one selected. However, even with an optimized CPU selection we expect that the FPGA devices would offer a significant performance/cost improvement for the problem our circuit solves.

Table 7. Cost of FPGA and CPU devices

Device	Cost
Spartan 3 XCS1500	$61.55
Spartan 3 XCS2000	$50.49
Virtex 4 XC4VLX25	$222.00
Virtex 4 XC4VLX40	$457.50
Intel Xeon MP 2.8 GHz	$399.00

Table 8 shows the throughput/cost ratio comparison between the two FPGA device families and the software implementation. The results in Table 7 show an

anomaly of the volatile pricing for FPGA devices. Typically the larger XCS2000 device would be expected to cost more than the XCS1500 device, however, when we surveyed the advertised prices the XC2000 device was less expensive. The results in Table 8 were calculated using the XCS1500 for the 216 and 350 bit dividend values; however, the throughput/cost ratio could have been improved if the XCS2000 device was used. These results show that the low-cost Spartan 3 device offers the best throughput/cost ratio, and they further highlight the advantages of the FPGA implementation over a software implementation.

Table 8. Circuit throughput/cost in (num/sec)/dollar

	Spartan-3		Virtex-4		Software
Dividend	Optimal	Actual	Optimal	Actual	Version
512 bits	437	178	106	56	1.09
350 bits	364	149	107	53	1.49
216 bits	369	149	224	107	2.22

One proposed implementation of the TWIRL sieving device by Geiselmann, et al. [6], provides a performance estimate of an ASIC implementation of a trial division pipeline. This implementation requires $3 \cdot k + 50$ clock cycles per prime factor analyzed where k is the number of bits in the dividend. For a 512-bit dividend this implementation would require 1586 clock cycles to check a single prime factor. In contrast, our implementation requires only a single clock cycle to check each next prime for divisibility once the pipeline is full. The analysis of TWIRL concluded that the trial division pipeline could produce results at a faster rate than the ECM unit could process them. In this paper we focused on developing a trial division pipeline with maximum throughput and combining this implementation with the FPGA ECM implementation described in [5] is a subject of future work. However, any realistic reconfigurable computer used for factoring would contain a large number of FPGAs for specialized computationally intensive tasks combined with general purpose microprocessors to provide control functions. As a result, the balance between producing results from the trial division pipeline and processing them in ECM units can be optimized by controlling the ratio of FPGAs assigned to these different operations.

5 Conclusion

Our design implemented a method to perform trial division by small primes in two FPGA families, using a technique that provides a significant speedup over an optimized software implementation of the same algorithm. The task of division by small primes allowed us to implement a highly pipelined technique that determines the divisibility of a number using one pipeline stage for each bit of the dividend. The area and latency of each pipeline stage was dependent only on the smaller sized divisor, and this allowed for nearly constant throughput

independent of the dividend size. The circuit area was linearly dependent on the number of bits in the dividend.

Reconfigurable computers provide a cost-effective means to implement specialized algorithms at the digital logic layer for use in a relatively small number of devices. In a reconfigurable computer, general purpose microprocessors are used for control purposes, and FPGAs are used for specialized, computationally intensive tasks. Our results demonstrate the cost-performance advantage of an FPGA-based implementation of the specialized problem of trial division by small primes over a microprocessor-based implementation. We believe that the most cost-efficient means of breaking RSA encryption is likely to be developed using large numbers of low-cost FPGA devices, as demonstrated by the COPACOBANA device in breaking DES encryption. While our design is too large to fit on the FPGA devices used in COPACOBANA, we expect that FPGA devices with larger numbers of transistors will become increasingly affordable and that our design will be practical for low-cost devices.

Division is an inherently sequential operation, it is difficult to pipeline, and it is typically the slowest basic arithmetic operation implemented in general purpose microprocessors. Existing research on developing a solution to the cofactoring problem using reconfigurable computers has performed the trial division step using general purpose microprocessors, and focused on developing FPGA based designs to implement algorithms such as ECM. We believe that our design provides a useful addition to the cofacotoring problem, by providing a technique to perform the computationally intensive trial division step using a specialized FPGA design.

Acknowledgments: We would like to thank Paul Kohlbrenner for assistance in developing an optimized software implementation of the algorithm used for trial division.

References

[1] A. Shamir, "Factoring large numbers with the TWINKLE device (extended abstract)," in *CHES*, ser. Lecture Notes in Computer Science, Ç. K. Koç and C. Paar, Eds., vol. 1717. Springer, 1999, pp. 2–12.

[2] A. Shamir and E. Tromer, "Factoring large numbers with the TWIRL device," in *Advances in Cryptology - CRYPTO 2003*, ser. Lecture Notes in Computer Science, 2003, pp. 1–26.

[3] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, and C. Stahlke, "SHARK: A realizable special hardware sieving device for factoring 1024-bit integers," in *CHES*, ser. Lecture Notes in Computer Science, J. R. Rao and B. Sunar, Eds., vol. 3659. Springer, 2005, pp. 119–130.

[4] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, "Breaking ciphers with COPACOBANA - A cost-optimized parallel code breaker," in

CHES, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds., vol. 4249. Springer, 2006, pp. 101–118.

[5] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi, "Implementing the elliptic curve method of factoring in reconfigurable hardware," in *CHES*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds., vol. 4249. Springer, 2006, pp. 119–133.

[6] W. Geiselmann, F. Januszewski, H. Köpfer, J. Pelzl, and R. Steinwandt, "A simpler sieving device: Combining ECM and TWIRL," in *ICISC*, ser. Lecture Notes in Computer Science, M. S. Rhee and B. Lee, Eds., vol. 4296. Springer, 2006, pp. 118–135.

[7] M. Khaleeluddin, "Hardware implementation of the elliptic curve method of factoring," Master's thesis, George Mason University, August 2006. [Online]. Available: http://ece.gmu.edu/courses/Crypto_resources/web_resources/theses/gmu_theses.htm

[8] R. Bachimanchi, "FPGA and ASIC implementation of rho and p-1 methods of factoring," Master's thesis, George Mason University, May 2007. [Online]. Available: http://ece.gmu.edu/courses/Crypto_resources/web_resources/theses/gmu_theses.htm

[9] J. Pelzl, M. Šimka, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer, and C. Paar, "Area-time efficient hardware architecture for factoring integers with the elliptic curve method," *IEE Proceedings Information Security*, vol. 152, no. 1, pp. 67–78, Oct. 2005.

[10] M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer, and C. Paar, "Hardware factorization based on elliptic curve method," in *IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM 2005, 18-20 April 2005, Napa, California*, J. Arnold and K. L. Pocek, Eds., 2005, pp. 107–116.

[11] P. Zimmermann and B. Dodson, "20 years of ECM," 2006. [Online]. Available: http://hal.inria.fr/inria-00070192/en/

[12] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.

[13] "Spartan 3 advertised prices quantities of 100," August 2007. [Online]. Available: http://www.em.avnet.com

[14] "Virtex 4 advertised prices quantites of 100," August 2007. [Online]. Available: http://www.nuhorizons.com

[15] "Intel Xeon-MP 2.8ghz CPU advertised price," August 2007. [Online]. Available: http://www.compuvest.com