A Dedicated Sieving Hardware

Willi Geiselmann and Rainer Steinwandt

IAKS, Arbeitsgruppe Systemsicherheit, Prof. Dr. Th. Beth
Fakultät für Informatik, Universität Karlsruhe
Am Fasanengarten 5, 76 131 Karlsruhe, Germany

Abstract. We describe a hardware device for supporting the sieving step in integer factoring algorithms like the quadratic sieve or the number field sieve. In analogy to Bernstein's proposal for speeding up the linear algebra step, we rely on a mesh of very simple processing units. Manufacturing the device at moderate cost with current hardware technology on standard wafers with 200 mm or 300 mm diameter should not provide any major obstacle.
A preliminary analysis of the parameters for factoring a 512-bit number with the number field sieve shows that the design considered here might outperform a TWINKLE device.

Keywords: factorization, number field sieve, RSA

1 Introduction

Current factoring algorithms like the quadratic sieve (QS) or the number field sieve (NFS) involve a so-called *sieving step* that is usually considered to be the most time-consuming part of the whole algorithm. Consequently, the question arises whether a specialized hardware can be used to achieve a significant speed-up in this step. For the QS a proposal for such a specialized hardware is due to Pomerance, Smith and Tuler [8]; according to [7] this special-purpose quadratic sieve processor "was built but never functioned properly. The point later became moot due to the exponential spread of low-cost, high-quality computers."

A more recent proposal for a dedicated sieving hardware, due to Shamir, is the TWINKLE device [10]. In [5] Lenstra and Shamir analyze this proposal in more detail and discuss the use of such a hardware in conjunction with the NFS. A major practical drawback of the TWINKLE device is the fact that it relies on the use of (expensive) Gallium Arsenide technology with optoelectronic components; a satisfactory silicon based approach without optical components seems not to be known.

Recently, Bernstein [1] proposed the use of a 'classical' mesh-architecture, which does not rely on the use of optoelectronic components, for the *linear algebra step* of the NFS. For a discussion of this approach we refer to [6]. Here we want to dwell on the question whether a hardware design similar to the one considered by Bernstein can be used for speeding-up the sieving step. It turns out that this approach seems possible indeed, and that it might be more efficient than using TWINKLE devices.

Y.G. Desmedt (Ed.): PKC 2003, LNCS 2567, pp. 254–266, 2003.

Our contribution is organized as follows: after recalling the sieving step of the NFS to the extent necessary for the sequel, we describe the algorithm we want to use for sieving, along with the corresponding hardware requirements. Thereafter we give a rough analysis of the occurring parameter sizes when dealing with 512-bit numbers. Some remarks on possible improvements and further work conclude the paper.

2 The Sieving Step in the NFS

For an introduction to the number field sieve we refer to [4]; here we recall only those aspects of the relation collection/sieving step relevant for the sequel. The importance of this step is illustrated by the following comment from [6], for instance: "We conclude that from a practical standpoint, the security of RSA relies exclusively on the hardness of the relation collection step of the number field sieve."

In the first step of the NFS two univariate polynomials $f_1(x), f_2(x) \in \mathbb{Z}[x]$ are chosen that share a common root m modulo n. Typically, $f_1(x)$ is of degree 5 and $f_2(x)$ is monic and linear:

$$f_1(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$
$$f_2(x) = x - m$$

where $f_1(m) \equiv f_2(m) \equiv 0 \pmod{n}$. From these two polynomials two bivariate and homogeneous polynomials $F_1(x, y), F_2(x, y) \in \mathbb{Z}[x, y]$ are derived via $F_1(x, y) := y^5 \cdot f_1(x/y)$ resp. $F_2(x, y) := y \cdot f_2(x/y)$. Now everything related to $f_1(x)$ resp. $F_1(x, y)$ is said to belong to the *algebraic side*, and everything related to $f_2(x)$ resp. $F_2(x, y)$ is referred to as the *rational side*. In particular, we refer to the sets

$$P_i := \{(p, r) : f_i(r) \equiv 0 \pmod{p},\ p \text{ prime},\ p < 2^{24},\ 0 \le r < p\} \subseteq \mathbb{N}^2 \quad (i = 1, 2)$$

as algebraic and rational *factor base*, respectively. The upper bound 2^{24} used here aims at the factorization of a 512-bit number and is taken from [5].

The aim of the relation collection step is to find pairs of coprime integers $(a, b) \in \mathbb{Z}^2$ such that b is positive and the values $F_1(a, b)$ and $F_2(a, b)$ are *smooth*. Having in mind 512-bit numbers, according to [5] a sensible definition of 'smoothness' is the following:

Algebraic Side: $F_1(a, b)$ factors over the primes $< 2^{24}$, except for possibly 3 primes $< 10^9$

Rational Side: $F_2(a, b)$ factors over the primes $< 2^{24}$, except for possibly 2 primes $< 10^9$

Of course, these parameter choices are debatable: e. g., in [2] a significantly larger algebraic factor base along with only two (instead of three) large primes is used. Also, it is common to exclude small primes from the factor bases to speed up

the sieving step (cf., e. g., [3]). In Section 3 we shall see that for the approach to sieving considered in this paper, such variations also prove useful.

For finding pairs (a, b) with $F_1(a, b)$ and $F_2(a, b)$ being smooth, typically some kind of sieving process is applied to a rectangular region $-A \leq a < A$, $0 < b \leq B$ where $A, B \in \mathbb{N}$. Different techniques are available for organizing this sieving process. For sake of simplicity, here we focus on so-called *line sieving*; some comments on *special q sieving* are given in Section 5. Denoting by T_1, T_2 appropriate threshold values for the algebraic and rational side, respectively, a rough outline of line sieving reads as follows:

$b \leftarrow 0$
repeat
  $b \leftarrow b + 1$
  for $i \leftarrow [1, 2]$
    $s_i(a) \leftarrow 0 \quad (\forall a : -A \leq a < A)$
    for $(p, r) \leftarrow P_i$
      $s_i(br + kp) \leftarrow s_i(br + kp) + \log_2(p) \quad (\forall k : -A \leq br + kp < A)$
  for $a \leftarrow \{-A \leq a < A : \gcd(a, b) = 1,\ s_1(a) > T_1, \text{ and } s_2(a) > T_2\}$
    check if both $F_1(a, b)$ and $F_2(a, b)$ are smooth
until enough pairs (a, b) with both $F_1(a, b)$ and $F_2(a, b)$ smooth are found

E. g. in [5] it is pointed out that in the last step of the main loop, i. e., when testing $F_1(a, b)$ and $F_2(a, b)$ for being smooth, it is computationally too expensive to use a simple trial-division over the primes in the factor base. The sieving device described below takes this problem into consideration: when reporting that some $F_1(a, b)$ and $F_2(a, b)$ values are smooth, also prime factors hereof that have been found during sieving are reported.

3 A Sieving Device

3.1 Schimmler's Sorting Algorithm

An essential algorithmic tool we will use for sieving is Schimmler's sorting algorithm: assume we are given an $M \times M$ mesh of processing units $(Q_{i,j})_{1 \leq i,j \leq M}$ where $M := 2^n$ and each processing unit Q_{ij} stores some integer $q_{i,j}$. Then Schimmler's sorting algorithm can sort these M^2 numbers in $8M - 8$ 'steps' according to any of the following orders on the indices (i, j) of the processing units $Q_{i,j}$:

left-to-right: $(1, 1) \leq (1, 2) \leq \ldots \leq (1, M) \leq (2, 1) \leq \ldots \leq (M, M)$
right-to-left: $(1, M) \leq (1, M - 1) \leq \ldots \leq (1, 1) \leq (2, M) \leq \ldots \leq (M, 1)$
snakelike: $(1, 1) \leq (1, 2) \leq \ldots \leq (1, M) \leq (2, M) \leq (2, M - 1) \leq \ldots \leq (M, 1)$

A detailed explanation of this algorithm is given in [9, 1], for instance. Here it is sufficient to recall an elementary 'step' of the algorithm: in analogy to the well-known odd-even transposition sorting, in a single step each processing unit Q_{ij} communicates with exactly one of its horizontal or vertical neighbours. So let

$\hat{Q}$, $\tilde{Q}$ be two communicating processing units, and denote by $\hat{q}$, $\tilde{q}$ the integers stored in $\hat{Q}$, $\tilde{Q}$, respectively. At the end of one 'elementary step' one of the two processing units, say $\hat{Q}$, must hold $\min(\hat{q}, \tilde{q})$ while the other one has to store the value $\max(\hat{q}, \tilde{q})$. To achieve this we can proceed as follows:

1. $\hat{Q}$ sends $\hat{q}$ to $\tilde{Q}$, and $\tilde{Q}$ sends $\tilde{q}$ to $\hat{Q}$. E. g., if the stored integers represent natural numbers $< 2^{24}$, this operation can be completed in one clock cycle via a unidirectional 24-bit bus in each direction.
2. Both $\hat{Q}$ and $\tilde{Q}$ compute the boolean value *exchange* $:= (\tilde{q} < \hat{q})$. E. g., if $\hat{q}$ and $\tilde{q}$ are 24-bit numbers, this comparison can be done in one clock cycle.
3. If *exchange* evaluates to true, then $\hat{Q}$ stores $\tilde{q}$ and deletes $\hat{q}$. Analogously, $\tilde{Q}$ keeps $\hat{q}$ and deletes $\tilde{q}$, in this case. If *exchange* evaluates to false, then both $\hat{Q}$ and $\tilde{Q}$ keep their old values and delete the values received in the first step. Again, for 24-bit integers this operation does not require more than one clock cycle. In fact it is feasible to integrate this step into the previous one without requiring an additional clock cycle.

In summary, when dealing with natural numbers $< 2^{24}$, Schimmler's sorting algorithm enables us to sort M^2 numbers in less than $8M$ steps where each step takes 2 clock cycles. If pairs of integers (q, r) have to be sorted according to the size of q, this can easily be done within the same time. In Step 2 and Step 3 the bus is not used and thus can transmit r. On behalf of *exchange* this value is stored or ignored by the appropriate unit.

Before applying these observations to the sieving step of the NFS, for sake of completeness, we would like to point out that the idea of making use of Schimmler's sorting algorithm within the sieving step is also mentioned (without further explanation) in [1, Section 5, 'Plans.'].

3.2 Sieving with a Fast Sorting Hardware

For the ease of exposition we first describe the basic outline of our algorithm and postpone a more detailed discussion of the hardware requirements and the resulting performance to a separate paragraph.

The Sieving Algorithm Assume that in a precomputation step the factor bases P_1 and P_2 have been computed. For the moment we also assume that all primes that do not exceed some bound S are excluded from the two factor bases; for 512-bit numbers we may think of $S = 2^{22}$, i. e., the 295,947 smallest primes do not occur in P_1 and P_2—of course we must not ignore that many small primes, and we will discuss later how to lower the bound S.

Now we load into each processing unit of the sorting network one of the $|P_1| + |P_2|$ tuples $(p, r, i) \in \mathbb{N}^2 \times \{1, 2\}$ with $(p, r) \in P_i$. In other words, each processing unit holds an element from one of the factor bases along with a flag i that indicates to which of the two factor bases the stored value belongs. For sake of simplicity let us assume that $|P_1| + |P_2|$ coincides with the number M^2 of processing units.

For the actual sieving of an interval $-A \leq a < A$ (with b fixed), at first we divide this interval into $2A/S$ subintervals, each of size S (throughout we assume that $S \mid 2A$). Then we know that for $(p, r) \in P_1 \cup P_2$ arbitrary, each of these subintervals contains no more than one element $\tilde{r}$ with $r \equiv \tilde{r} \pmod{p}$. Further on, for $\tilde{a}$ an arbitrary number from a given subinterval and $(p, r) \in P_i$, the value $\lfloor \log_2(p) \rfloor$ is added to $s_i(\tilde{a})$ during line sieving if and only if $\tilde{a} \equiv br \pmod{p}$ $(i = 1, 2)$. The initial values stored in the processing units correspond to $b = 1$, and to identify potentially useful $(a, 1)$-pairs we first apply Schimmler's algorithm to 'collect equal residues', namely we perform the following computation:

(I) Sort the triples (p, r, i) in snakelike order (smaller values first) according to the following order: $(p_0, r_0, i_0) < (p_1, r_1, i_1) \iff r_0||i_0 < r_1||i_1$, where $r_j||i_j$ denotes the value obtained by concatenating the registers where r_j and i_j are stored.

Now all pairs (p, r) that share the same r- and i-values are neighbours of each other. Thus, all (p, r)-pairs that belong to the same factor base and that contribute a value $\log_2(p)$ to the same counter $s_i(-A+1+r)$ during line sieving are neighbours. In Step (II)–(VI) we determine (by means of local computations) whether for both factor bases the value $\sum_{p|(-A+r)} \lfloor \log_2(p) \rfloor$ (with p ranging over P_1 resp. P_2) exceeds the corresponding threshold value T_1 resp. T_2. Within each sequence of identical r-values an *ok*-flag will be set if and and only if both threshold values are exceeded.

To do so, first of all each processing unit determines the approximate bit length of the currently stored prime number:

(II) Each processing unit initializes an internal counter c to $\lfloor \log_2(p) \rfloor$ by counting the leading zeroes in p (where (p, r, i) is the currently stored value).

Next, for each sequence of equal r-values referring to the same factor base, we want to identify the first resp. last element of the sequence—in dependence of the stored factor base index i. The processing units holding these values play a distinguished role in the sequel.

(III) Each processing unit sends $r||i$ to its two neighbours in the snakelike order. A processing unit with $i = 1$ receives an $r||i$-value different from its own (from its successor) if and only if it is the last one in a sequence of equal $r||i$-values. Analogously, processing units with $i = 2$ can decide whether they are at the beginning of a sequence of equal r-values with $i = 2$.

For each sequence of r-values we want to compute $\sum_{p|(-A+r)} \lfloor \log_2(p) \rfloor$ for p ranging over the primes in the algebraic and the rational factor base, respectively:

(IV) Each processing unit not at the end of an r-sequence $(i = 1)$ or not at the beginning of an r-sequence $(i = 2)$, sends its $\lfloor \log_2(p) \rfloor$-value c to its neighbours in the snakelike order. Further on, these processing units receive and store the c-value from their predecessor $(i = 1)$ resp. successor $(i = 2)$. Processing units at the end of an r-sequence $(i = 1)$ or at the beginning

of an r-sequence ($i = 2$) send 0 and add the value received from their predecessor resp. successor to their current counter c.
Depending on the expected maximal number of prime divisors of the $F_i(a, b)$ within a factor base, this step is repeated several times (for 512-bit numbers say ≈ 10).

Due to the order used during sorting in Step (I), for an r-value that occurs both with factor base index $i = 1$ and $i = 2$, the two processing units storing $\sum_{p|(-A+r)} \lfloor \log_2(p) \rfloor$ with p ranging over P_1 and P_2 respectively, are neighbours. Consequently, for deciding whether the value $-A + r$ belongs to a potentially useful $(a, 1)$-pair, these two processing units compare their counter c with the corresponding 'smoothness threshold' T_i; these threshold values are identical for all processing units and adapted for new a-subintervals (through an external signal) if required.

(V) The processing units that summed up the $\lfloor \log_2(p) \rfloor$-values in their counter c compare c with the corresponding threshold value T_i. If $c > T_i$ then a flag *ok* is set to 1, otherwise *ok* is set to 0.
Thereafter, in case of factor base index $i = 1$, the stored $ok||r$-value is sent to the successor. Dually, for $i = 2$ the stored $ok||r$-value is sent to the predecessor. Finally, the stored *ok*-value is left unaltered, if the stored $ok||r$-value coincides with the received one, otherwise *ok* is set to 0.

Hereafter, the *ok*-flags of the two processors 'in the middle of an r-sequence' are set if and only if both smoothness conditions are met, i. e., a potentially useful $(a, 1)$-pair has been found. As we do not want to loose the prime factors found during sieving, we next broadcast these (identical) *ok*-flags to all elements of the corresponding r-sequence:

(VI) Repeating Step (III) with the roles of $i = 1$ and $i = 2$ interchanged, the processing units with $i = 1$ notice whether they are at the beginning of an r-sequence. Dually, for $i = 2$ the end is recognized.
These 'border units' send 0 in the sequel to ensure that only the correct prime factors are marked. The other processing units send their *ok*-flag. The processing units with $i = 1$ receive and store the *ok*-flag from their successor and those processing units with $i = 2$ from their predecessor. Similarly as in Step (V), in dependence on the number of expected prime factors, this operation is repeated several times.

At the end of Step (VI) exactly those (p, r, i)-triples are marked with $ok = 1$ that we want to have as output to examine the smoothness of $F_1(-A + r, 1)$ and $F_2(-A + r, 1)$ in more detail:

(VII) To output the marked triples, we sort in snakelike[1] order (larger values first) according to the following order: $(p_0, r_0, i_0, ok_0) < (p_1, r_1, i_1, ok_1) \iff$

[1] left-to-right or right-to-left would work as well here

$ok_0||r_0 < ok_1||r_1$. Then, depending on the maximal number of 'hits' expected, a fixed part of the processing units writes their (p, r, i, ok)-values in parallel into the output buffer.
As there are only very few hits expected within one subinterval of size S, this step can be simplified: first sort the columns according to the order above; thereafter with high probability all marked triples are within the first few rows (for 512-bit numbers say ≈ 4)[2]. Next, we sort these first few rows left-to-right, and write them into an output buffer—possibly several rows in parallel.

The output will serve as basis for more precise smoothness tests discussed in Section 4. Here we focus on the problem of how to continue with the sieving of the next subinterval once the first subinterval $[-A, \ldots, -A + S - 1]$ has been sieved as described above. For passing to the next subinterval each processing unit must adapt its r-values to the new start $-A + S$ of the sieve subinterval. This can be done with simple 'local' computations, i. e., *there is no need to load new data into the mesh*. At this we exploit that all processed prime numbers are larger than the length S of the sieving interval:

(VIII) Each processing unit performs the following operation: r is replaced by $r - S$, and in case of $r - S < 0$ the prime p is added.

Now we are in essentially the same position as before Step (I), but now the processing units are initialized for sieving an interval of length S that starts at $-A + S$. Consequently, we apply the Steps (I)–(VIII) again, and continue in this way until the complete interval $-A \leq a < A$ of a-values has been sieved.—Without having to load new data into the mesh. Once the complete interval has been processed, the current value of b must be increased by 1. For doing so, new data is loaded into the network: in analogy to the case $b = 1$, each processing unit is initialized with a triple of the form $(p, b \cdot r \pmod p, i)$ where $(p, r) \in P_i$. In this way the sieving is continued until enough coprime values a, b with $F_1(a, b)$ and $F_2(a, b)$ being smooth are found.

We are still left to explain in more detail how the smoothness testing is actually done; in particular all prime numbers $\leq S$ have not been taken into account so far. But before dwelling onto this topic, we take a more detailed look on the hardware characteristics and the possible performance of the above device when dealing with 512-bit numbers.

Hardware Requirements To analyze the technical limits of the above algorithm we first estimate the number of transistors required for one processing unit when dealing with 512-bit numbers. Each of these units requires

2 registers with 24 bit each (p, r),

[2] For the very few subintervals with $|a|$, $|b|$ very small a different procedure should be used, as otherwise too many useful candidates might be lost. This part is so small, however, that even a few hours on a normal PC are sufficient to take care hereof.

1 adder/compare unit for 25-bit numbers,
2 registers with 1 bit each (i, ok),
3 registers with 8 bit each (c, T_1, T_2), and
2 unidirectional 25-bit connections with each of the 4 neighbours; one to send and one to receive data

to perform the algorithm above. One D Flip-Flop can be realized with 8 transistors, and an adder of the appropriate size with about 40 transistors per bit. In total, around 1600 transistors are required for these basic elements. Together with 4 multiplexers for the 25-bit connections (4 transistors per bit each) and the additional program-logic, e. g., to store internal states like first/last unit in a sequence of equal r-values, we estimate the number of transistors required per processing unit as 2500 transistors. (In [6] a slightly simpler unit was assumed to need 2000 transistors.) Within each cell only a very small program logic is required; most commands determine how to set the multiplexers and are identical within one row resp. column. These commands can be generated outside the square area of processing units and distributed through a couple of connections. The only part of the procedure that requires more logic within the units is the calculation of $\lfloor \log_2(p) \rfloor$; it might be more efficient to store and communicate this 8-bit value through the complete calculation.

With current 0.13 μm technology (used for the Intel Pentium 4 "Northwood" processor) more than 400,000 transistors and thus at least 160 processing units fit on 1 mm^2. On the square area of a wafer with 200 mm diameter 3.2 million units can be placed; on a 300 mm wafer, as used for the Pentium 4 'Northwood' processor, 7.2 million processing units fit. Thus it is certainly possible to produce a mesh of $2^{11} \times 2^{11}$ processing units with current technology.

This estimation does not take into account the problem of defective cells. Additional rows and columns and some logic have to be added to the layout to bypass complete rows and columns with defective units. On a 300 mm wafer there should be enough space left to take care of this problem.

Performance If a mesh with $M^2 = 2^{22}$ processing units is given we want to analyze its use for the sieving step in the NFS for a 512-bit number. The size of the factor bases and the regions to be sieved are taken from [5] to easily compare the suggested hardware with the version of the TWINKLE device adapted for the NFS.

With subintervals of length $S = 2^{22}$, on a mesh with 2^{22} processors a large part of the factor bases, namely all pairs $(p, r) \in P_i$ $(i = 1, 2)$ with $2^{17} < p < 2^{24}$, can be processed in the form described in Section 4: for all these primes the triples $(k \cdot p, r + (l-1) \cdot p, i)$ with $1 \le l \le k := \lceil 2^{22}/p \rceil$ are stored. In summary, then $2,025,624$ processors are needed for the rational factor base and about the same number for the algebraic factor base. The $12,251$ primes smaller 2^{17} have to be processed by the trial division pipeline described in Section 4.

With a conservative estimation the mesh can be expected to work at a clock rate of 500 MHz; the communication across the border of the wafer is slower by a factor 4. Using a 48-bit I/O bus for loading the data for the 2^{22} cells

sequentially onto the wafer requires 2^{24} clock cycles or 0.034 seconds; with $4 \cdot 48$ I/O pins this initialization can be finished in less than 0.01 seconds.—If an additional powerful (standard) processor and 32 MB of memory are placed on the remaining $\approx 35\%$ of the wafer outside the square area, this operation can be speeded up significantly.

Schimmler's sorting algorithm has to be performed in Step (I) and requires about $8M$ steps with 2 clock cycles each. The reduced version in Step (VII) works with about $2M$ steps. The rest of the algorithm, Steps (II)–(VI) and (VIII) require a small fixed number of clock cycles (less than 100) and are neglected for the following estimations.

For $M = 2^{11}$ an interval of length $S = 2^{22}$ can be sieved within $20 \cdot 2^{11}$ clock cycles. A line of the sieving region required for factoring a 512-bit number with the NFS has a length of $1.8 \cdot 10^{10}$; it can be sieved in 4292 runs of the above algorithm within 0.36 seconds. The modified TWINKLE device with 2 LEDs per cell, working at a clock rate of 10 GHz, requires 1.8 seconds for the same interval or 18 seconds, when working at the lower speed of 1 GHz; there are about 10 wafers with 6 inches (≈ 152 mm) required for this TWINKLE architecture.

To sieve the complete region necessary for the 512-bit factorization, $9 \cdot 10^5$ such lines are required and can be processed in less than 4 days with the device described here.

Alternatively, a different (slower) setting of the parameters can be chosen to reduce the false alarms by allowing only two large primes on the algebraic side instead of three (cf. [2]). In this case the computing time of the further processing (like trial division and checking for large primes) is reduced. To find still enough relations one has to increase the size of the algebraic factor base. This can be achieved if the size of the subinterval in the above procedure is reduced to $S := 2^{21}$: in the 2^{22} processing units the part of the factor bases related to the primes p with $2^{16} < p < 2^{24}$ can be stored analogously as above. Then $1,577,786$ processors are required for the rational factor base and about the same number for the algebraic factor base related to the primes $< 2^{24}$. The remaining $1,030,000$ processors can be used to deal with the $985,818$ primes up to 2^{25}. In this setting only 6542 primes smaller 2^{16} have to be taken into account for the trial division pipeline, and the size of the algebraic factor base has nearly doubled. Compared to the previous parameters the time for sieving has increased by a factor 2 to less than 8 days.

4 Handling Small Primes and Testing Smoothness

The assumption that the mesh handles only primes $p > S$ is overnecessarily restrictive. If we are willing to dedicate more than one processing unit to a pair $(p, r) \in P_1 \cup P_2$, then at least primes that are not 'much' smaller than S can be dealt with: let $(p, r) \in P_1 \cup P_2$ and $k \in \mathbb{N}$ minimal with $k \cdot p > S$. Then we use k processing units for representing the triple (p, r, i) where the l^{th} unit $(1 \leq l \leq k)$ holds the value $(k \cdot p, r + (l-1) \cdot p, i)$. Note here that this approach introduces an inaccuracy into the summation of the $\lfloor \log_2(p) \rfloor$-values: if we proceed during

the summation as described in Section 3.2, we essentially add $\lfloor \log_2(kp) \rfloor$ instead of $\lfloor \log_2(p) \rfloor$. If one is not willing to accept this inaccuracy, one may think of adding a fourth component to the triples $(k \cdot p, r + (l-1) \cdot p, i)$ to store the value $\lfloor \log_2(p) \rfloor$. Of course, whenever two processing units exchange their currently stored values throughout Schimmler's sorting algorithm, the additional fourth component must also be taken care of then. To do so without sacrificing time, one can use a broader bus between the processing units. E. g., if $\lfloor \log_2(p) \rfloor$ is represented with 8 bits, then 4 extra bits on the bus are sufficient to transmit this value within two clock cycles.—Step (II), and therewith also parts of the program logic, obviously become superfluous when $\lfloor \log_2(p) \rfloor$ is stored explicitly.

Using $S = 2^{22}$, on the rational side all 1,065,620 primes p with $2^{17} < p < 2^{24}$ can be represented with 2,025,624 processing units in this way.—And analogously on the algebraic side. Using $S = 2^{21}$, the 1,071,329 primes p with $2^{16} < p < 2^{24}$ on the rational side can be handled with 1,577,786 processing units. But even with the modification just described, for $S = 2^{22}$ resp. $S = 2^{21}$ so far we still ignore the smallest 12,251 resp. 6,542 prime numbers; these remaining primes can be dealt with in a 'trial division pipeline':

Once the sieving of a subinterval is completed, the output of the sieving device tells us promising (a, b)-pairs (note that the a-value corresponding to a (p, r, i)-triple of the output is given by $a = -A + (u-1) \cdot S + r$ with $u \geq 1$ denoting the number of the subinterval sieved). A separate processor reads the output buffer of the sieving device and stores a and b along with the corresponding (p, i)-values found during sieving, if $\gcd(a, b) = 1$ holds.

If in the earlier $\lfloor \log_2(p) \rfloor$-summation the inaccuracies—due to multiples $j \cdot p$—have not been taken into account, this processor can also recompute the sum of the logarithms of the potential prime factors in each factor basis and compare them with the corresponding threshold value T_i. In this way the number of candidates that have to be explored in more detail can be reduced. For all (a, b)-pairs that passed the tests so far, the result of the two $\lfloor \log_2(p) \rfloor$-summations is appended to the already stored list of prime factors of the $F_i(a, b)$. The values $F_1(a, b)$ and $F_2(a, b)$ are also stored and passed to a 'trial division pipeline' that divides out small prime factors (in the above example with $S = 2^{22}$ resp. $S = 2^{21}$ the first 12,251 resp. 6,542 primes have to be dealt with here): this device has a simple pipeline structure where in each step of the pipeline a small prime factor is divided out if this divisor is actually present, and the result is passed on to the next stage of the pipeline. For very small primes one may also devote several stages of the pipeline to one prime factor; e. g., to divide out all powers of 2 up to 2^7 one can use three division units that try to divide out 2^4, 2^2, and 2^1, respectively.

The output of this pipeline is forwarded to a processor that determines the bit length of the result and subtracts the corresponding $\lfloor \log_2(p) \rfloor$-sum hereof. (Note that by construction the output of the division pipeline alternately refers to the algebraic and rational side, and the algebraic and rational $\lfloor \log_2(p) \rfloor$-sum have been stored right before passing $F_1(a, b)$, $F_2(a, b)$ into the division pipeline.) If the obtained value is too large for being the length of a number consisting

of two resp. three large primes, the candidate pair (a, b) is not processed any further—in the corresponding table entry a 'to do' flag will be set to zero in this case. Otherwise the factors $\tilde{F}_1$ and $\tilde{F}_2$ of $F_1(a, b)$ and $F_2(a, b)$ obtained from the division pipeline are appended to the table entry for (a, b) and the table entry is marked by setting a 'to do' flag.

While the two processors before and after the division pipeline fill the table row by row in this way, the entries marked with 'to do' will be processed by several additional processors (working in parallel) as explained in a moment. In particular the processing of these entries—which might, e. g., be done by a PC network—has to be fast enough so that at the time when the table would be full, a wrap around is possible; entries already processed will then have a reset 'to do' flag.

As indicated already, the actual factorization of $F_1(a, b)$ and $F_2(a, b)$ is performed by several additional processors. These processors are assumed to be able to do long integer arithmetics: they look for set 'to do' flags in the table, read out the table entry and reset the 'to do' flag. Now the $\tilde{F}_i$ values are divided by all prime factors that have been found during sieving. Finally, the resulting cofactor is checked for being composed of only two or three primes. If this is the case, the small factors removed by the division pipeline will have to be recovered—say by factoring the values $F_i(a, b)/\tilde{F}_i$ $(i = 1, 2)$ with trial division. Once a complete (a, b)-pair has been processed (and if necessary the resulting factorizations been written in an output buffer), the next table entry with set 'to do' flag is looked for and processed.

Although several details are ignored in the above discussion, we think it gives ample evidence that neglecting smaller prime values during the actual sieving does not provide fundamental difficulties. In fact, for 512-bit numbers one might think about locating the division pipeline and the supporting processors (possibly including the hardware that provides the $(p, (b \cdot r) \bmod p, i)$-triples to be loaded into the processing units) on the same 300 mm wafer as the actual sieving circuit.

5 Improvements and Further Work

The sieving device described above is certainly not optimal, and several questions seem worth to be explored. We would like to mention some interesting issues here:

- A crucial point is the reinitialization of the device when incrementing b. In particular, if one thinks about implementing so-called q sieving (instead of line sieving) this point becomes important, as here the a-intervals are shorter. If one is willing to store the initial r-values and to add some additional hardware for an externally controlled shift-and-add procedure, updating the b-values should be possible within ≈ 100 clock cycles (in parallel in each processing unit). Of course, the additionally stored r-value also has to be taken care of during the sorting.

- A well-known optimization of the line-sieving procedure that is exploited in normal PC implementations is to exclude sieving regions with both a and b even. Assuming w. l. o. g. the numbers A and S to be even one can think of increasing the bit size of the processed numbers by one and using a similar trick in the above device: whenever b is even, we subtract S twice (and if necessary add $2p$) to get into the next subinterval.
- Also the question of scalability should be explored further, as the above design exploits that the complete sieving circuit is located on a single wafer.

6 Conclusions

We have described a dedicated hardware for supporting the sieving step in factoring algorithms like the NFS. The given rough analysis for the case of 512-bit numbers gives evidence that it might be feasible to manufacture such a circuit on the basis of a standard wafer, and that such an approach could possibly be preferable to the optoelectronic approach used for the TWINKLE device.

Acknowledgement

We are indebted to an anonymous referee for various valuable comments on the original manuscript.

References

[1] Daniel J. Bernstein. Circuits for Integer Factorization: a Proposal. At the time of writing available electronically at `http://cr.yp.to/papers.html#nfscircuit`, 2001. 254, 256, 257

[2] Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Walter Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, Paul Leyland, Joël Marchand, François Morain, Alec Muffet, Chris Putnam, Craig Putnam, and Paul Zimmermann. Factorization of a 512-bit RSA Modulus. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2000. 255, 262

[3] Marije Elkenbracht-Huizing. An Implementation of the Number Field Sieve. *Experimental Mathematics*, 5(3):231–253, 1996. 256

[4] Arjen K. Lenstra and Jr. Hendrik W. Lenstra, editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer, 1993. 255

[5] Arjen K. Lenstra and Adi Shamir. Analysis and Optimization of the TWINKLE Factoring Device. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2000. 254, 255, 256, 261

[6] Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, and Eran Tromer. Analysis of Bernstein's Factorization Circuit. At the time of writing available electronically at `http://www.cryptosavvy.com/mesh.pdf`, 2002. 254, 255, 261

[7] Carl Pomerance. A Tale of Two Sieves. *Notices of the AMS*, 43(12):1473–1485, 1996. 254

[8] Carl Pomerance, J. W. Smith, and Randy Tuler. A pipeline architecture for factoring large integers with the quadratic sieve algorithm. *SIAM Journal on Computing*, 17:387–403, 1988. 254

[9] Manfred Schimmler. Fast sorting on the instruction systolic array. Technical Report 8709, Christian Albrecht Universität Kiel, Germany, 1987. 256

[10] Adi Shamir. Factoring Large Numbers with the TWINKLE Device. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems. First International Workshop, CHES'99*, volume 1717 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 1999. 254