Yet Another Sieving Device

(extended version)

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. A compact mesh architecture for supporting the relation collection step of the number field sieve is described. Differing from TWIRL, only isolated chips without inter-chip communication are used. According to a preliminary analysis for 768-bit numbers, with a 0.13 μm process one mesh-based device fits on a single chip of $\approx$(4.9 cm)^2—the largest proposed chips in the TWIRL cluster for 768-bit occupy $\approx$(6.7 cm)^2. A 300 mm silicon wafer filled with the mesh-based devices is $\approx$ 6.3 times slower than a wafer with TWIRL clusters, but due to the moderate chip size, lack of inter-chip communication, and the comparatively regular structure, from a practical point of view the mesh-based approach might be as attractive as TWIRL.

Keywords: factorization, number field sieve, RSA

1 Introduction

Initiated by Bernstein's paper [Ber01], in the last few years several proposals for speeding-up the linear algebra step of the number field sieve (NFS) by means of specialized hardware have been put forward. While Bernstein's original proposal relied on the use of a parallel sorting algorithm, Lenstra et al. derived an improved mesh architecture that relies on a parallel routing algorithm [LSTT02]. Finally, in [GS03b] distributed variants of the proposals in [Ber01,LSTT02] are discussed where the main focus is on deriving a design that can be realized with current standard technology. In summary, building a device that performs the linear algebra step of the NFS for 768- or 1024-bit numbers within a few hours must be considered as doable with current technology.

Using the words from [LSTT02], one can "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." Thus, it is no surprise that several attempts have been made to apply dedicated hardware to speed-up the relation collection step of the NFS, too. In particular, the TWINKLE device [Sha99,LS00] and the mesh-based design of [GS03a] can be seen in this context. However, none of these devices was practically capable of coping with the relation collection step of 1024-bit numbers. A significant step forward has been achieved by Shamir and

Tromer [ST03] recently: the TWIRL device they describe could in principle complete the sieving part of the NFS for 1024-bit numbers in less than a year by means of current technology. However, already for 768-bit numbers chip sizes of up to $\approx(6.7\text{ cm})^2$ (with an irregular layout) have been proposed. Although the proposed TWIRL parameters are not optimized for chip size, actually manufacturing a TWIRL cluster seems extraordinary challenging. For 1024-bit numbers a TWIRL cluster including a wafer-sized silicon chip has been proposed; thus, from a practical point of view the question arises, whether it is possible to do with more regular and smaller chips. Also, avoiding inter-chip communication seems desirable.

In this contribution we discuss a different design which is based on a routing mesh running at 500 Mhz (instead of 1 GHz in TWIRL, where the time-critical operations are simpler). For 768-bit numbers our proposal consists of a mesh of 256×256 almost identical processing units. The layout is rather regular and the estimated silicon area for a complete mesh is about $(4.9\text{ cm})^2$. Counting processing time per wafer, the estimated time needed for the sieving step with 768-bit numbers is about 6.3 times higher than estimated in [ST03] for TWIRL. However, due to the simpler design, manufacturing the device and applying it at least to 768-bit does not seem unrealistic. For 1024-bit numbers we cannot give a reliable answer yet, but as the design presented here allows for a rather compact storage of the factor bases—a critical point in TWIRL—exploring the 1024-bit case in more detail is certainly worthwhile.

2 Preliminaries

2.1 The sieving part of the NFS

A standard reference for an introduction to the number field sieve is [LHWL93]. Here we only recall those aspects of the sieving step which are relevant for describing our device.

In the first step of the NFS two univariate polynomials $f_1(x), f_2(x) \in \mathbb{Z}[x]$ are determined that share a common root m modulo n:

$$f_1(m) \equiv f_2(m) \equiv 0 \pmod n$$

Typically, $f_1(x)$ is of degree $d \geq 5$ and $f_2(x)$ is monic and linear (i. e., $f_2(x) = x - m$). From these two polynomials the bivariate and homogeneous polynomials $F_1(x,y), F_2(x,y) \in \mathbb{Z}[x,y]$ are derived via $F_1(x,y) := y^d \cdot f_1(x/y)$ 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 refered to as the *rational side*. In particular, for given smoothness bounds $B_1, B_2 \in \mathbb{N}_0$ the sets

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

are refered to as algebraic and rational *factor base*, respectively. Following [ST03], for the factorization of a 768-bit number, we assume $B_1 = 10^9$ and $B_2 = 10^8$.

Throughout the relation collection step, pairs of coprime integers $(a, b) \in \mathbb{Z} \times \mathbb{N}$ are to be found, such that the values $F_1(a, b)$ and $F_2(a, b)$ are *smooth*. This means that both $F_1(a, b)$ and $F_2(a, b)$ factor over the primes $< B_1$ resp. $< B_2$, except for a small number of prime factors; the precise number of 'extra' prime factors on the rational and algebraic side is not necessarily identical. The actual computation of (a, b)-pairs where both $F_1(a, b)$ and $F_2(a, b)$ are smooth can be performed by sieving over a rectangular region $-A \le a < A$, $0 < b \le B$ where $A, B \in \mathbb{N}$. For organizing this sieving, different techniques are available; here we focus on so-called *line sieving* which is outlined in Figure 1. At this the threshold bounds T_i correspond to the bitlength of the remaining cofactor on the algebraic resp. rational side. These bounds have to be updated several times throughout the sieving. In an actual implementation the values $\log_2(p)$ are usually replaced by an integer approximation. Also the use of base 2-logarithms is not mandatory; in analogy to [ST03], subsequently we use a 10-bit counter for summing up approximations $\lceil \log_{\sqrt{2}}(p) \rceil$. Finally, note that in the last step of the main loop in Figure 1 it is computationally too expensive to identify the factors of $F_i(a, b)$ through a simple trial-division by the primes in the respective factor base. To cope with this problem the sieving mesh described below reports prime factors of $F_i(a, b)$ that have been found.

2.2 Clockwise transposition routing

An important algorithmic tool used in the sieving device described below is a modification of a fast parallel routing algorithm described in [LSTT02] in the context of fast matrix-vector multiplication. We start by recalling the main ingredients of this *clockwise transposition routing*:

- the hardware platform is a mesh of (rather simple) processing units where each unit is connected to its horizontal and vertical neighbours.
- In each step of the algorithm a processing unit holds no more than *one* packet that is to be routed; only one packet can be sent and received per step.
- At the beginning of the algorithm some mesh nodes contain a data packet (the other nodes contain a *nil* value). Along with each data packet the coordinates of a processing unit in the mesh, the so-called *target node*, are stored, and the goal of the algorithm is to route all packets to the respective target.

```
$b \leftarrow 0$
**repeat**
  $b \leftarrow b + 1$
  **for** $i \leftarrow [1, 2]$
    $s_i(a) \leftarrow 0 \quad (\forall a : -A \le a < A)$
    **for** $(p, r) \leftarrow P_i$
      $s_i(br + kp) \leftarrow s_i(br + kp) + \log_2(p) \quad (\forall k : -A \le br + kp < A)$
  **for** $a \leftarrow \{-A \le a < A : \gcd(a, b) = 1,\ s_1(a) > T_1$, 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
```

Fig. 1. line sieving

– The actual routing is done by repeating the following four steps until the mesh is 'empty', i. e., all packets have reached their target node (where a packet is removed from the mesh):
  1. Each node located on an *odd row* compares its packet with the node *above* it (if any). The packets are exchanged if and only if the distance-to-target of the non-nil value that vertically is farthest away from its target node is reduced in this way.
  2. Each node located on an *odd column* compares its packet with the node to its *right* (if any). The packets are exchanged if and only if the distance-to-target of the non-nil value that horizontally is farthest away from its target node is reduced in this way.
  3. Identical to the first step with 'above' replaced by 'below'.
  4. Identical to the second step with 'right' replaced by 'left'.

A theoretical analysis of this algorithm is still lacking, but experimental results demonstrate its efficiency: e. g., for the situation considered in [LSTT02], the running time of the algorithm did not exceed $2M$ steps, when dealing with an $M \times M$ mesh. In the application below, no 'packet-cancellation' (see [LSTT02]) is used and several parts of the original algorithm have been modified. Again, simulations indicate that the resulting algorithm is rather efficient. However, analogously as in [LSTT02], we cannot provide a theoretical analysis of our approach at the moment.

Although we never encountered an infinite loop in our simulations, it can in principle even happen that our routing algorithm for certain parameter choices does not terminate. But this is of no practical concern: in the sieving procedure described later, only a certain period of time is alloted for sieving a particular range of numbers, and in the (rare) case that the routing cannot be completed within this time limit, say due to an infinite loop, only some (a, b)-candidates of that sieving interval are lost. In contrast to the linear algebra step of the NFS, where an incorrect intermediate result can have devastating consequences, the sieving process is rather robust with respect to such errors.

3 Adapting the Routing Algorithm

Subsequently, we will deal with a mesh of size $M \times M = 2^m \times 2^m$; for 768-bit numbers we can think of $m = 8$. Consequently, the largest distance a packet may have to travel during the routing is $2 \cdot (M - 1) = 2M - 2$. As a first modification, we want to 'connect' the borders of the mesh to get a torus topology and thereby reduce this maximal distance to $2 \cdot (M/2) = M$.

3.1 Using a torus topology

Having in mind an actual mesh of processing units, it is not desirable to install physical connections between the horizontal resp. vertical borders of the mesh, as the wires used for the 'wrap around' had to cross a length of at least $M - 2$ processing units. Instead, a standard trick can be used to derive a layout where connecting wires never cross more than one processing unit:

1. The $2^m \times 2^m$ processing units are arranged in a square, and we denote the i^{th} column in this square by c_i ($1 \leq i \leq 2^m$): $c_1|c_2|c_3|c_4|\dots|c_{2^m-2}|c_{2^m-1}|c_{2^m}$
2. Reversing the order of the columns $c_{2^{m-1}+1}, \dots, c_{2^m}$ followed by applying a perfect shuffle yields the desired column positions:
$$c_1|\mathbf{c_{2^m}}|c_2|\mathbf{c_{2^m-1}}|\dots|\mathbf{c_{2^{m-1}+2}}|c_{2^{m-1}}|\mathbf{c_{2^{m-1}+1}}$$
3. For implementing the vertical wrap around now the same trick is applied to the rows of the resulting arrangement.

This rearrangement of the processing units is 'just' for the ease of implementation and helps to circumvent the handling of extremely unbalanced running times of signals. Thus, in the sequel when discussing algorithmic aspects and the labeling of processing units used in a computation we can ignore this implementation detail and think of an ordinary mesh architecture with wrapped-around borders.

3.2 Finding the route to a target node

Each sieving line is split into subintervals of length S, and the mesh will process these subintervals one by one. For 768-bit numbers, we can think of $S = 2^{24}$, thus in a mesh of size 256×256 each processing unit will be in charge of $256 = 2^{24-16}$ consecutive sieve positions, and we focus on this parameter choice.

When preparing a packet that is to be routed in our sieving procedure, we are given a 24-bit number r that represents a non-negative integer $0 \leq r < S$, and we use only this value to identify the corresponding packet's route to its target node. W. l. o g. we can choose the start s_0 of the sieving subinterval such that $256 \mid s_0$, i. e., once the packet containing r arrived at the processing unit that processes the range $\{s_0 + i \cdot 256, \dots, s_0 + i \cdot 256 + 255\}$ (with $i \in \{0, \dots, 2^{16} - 1\}$), the least-significant 8 bit of r determine which of the sieve positions of that processing unit has to be addressed. Thus, we want to interprete the 16-bit number i, that indicates the number of the processed subinterval, as (x, y)-coordinate ($0 \leq x, y \leq 2^8 - 1$) of the target node.

There are various possibilities to encode these coordinates in the remaining $24 - 8 = 16$ bit of r. For our purposes the following approach is useful: we store the x-coordinate of the target node in the odd-numbered bit positions ($23, 21, 19, 17, \dots, 9$, where bit no. 23 is the most-significant bit) and the y-coordinate in the even bit positions ($22, 20, 18, 16, \dots, 8$). This interleaving of the coordinates can also be interpreted as a Kronecker/tensor product: the leading two bit of r determine in which $2^7 \times 2^7$-(sub)quadrant of the $2^8 \times 2^8$-mesh the target node is located. Similarly, the next two bit determine which (sub)quadrant (of size $2^6 \times 2^6$) inside the $2^7 \times 2^7$-quadrant has to be addressed, etc. To get a better image of the resulting pattern, Figure 2 sketches for small values of i to which processing unit the sieving range $\{s_0 + i \cdot 256, \dots, s_0 + i \cdot 256 + 255\}$ is assigned; e. g., the processing unit at the left border of the third line in the mesh handles the 256 values $\{s_0 + 4 \cdot 256, \dots, s_0 + 4 \cdot 256 + 255\}$.

Given an r-value we can extract the x-coordinate x_t and the y-coordinate y_t of the respective target unit by reading off the odd- resp. even-numbered

0	2	8	10	32	34	40	42
1	3	9	11	33	35	41	43
4	6	12	14	36	38	44	46
5	7	13	15	37	39	45	47
16	18	24	26	48	50	56	58
17	19	25	27	49	51	57	59
20	22	28	30	52	54	60	62
21	23	29	31	53	55	61	63

…

⋮

Fig. 2. assigning sieving ranges to processing units

bit positions. While in the original clockwise transposition routing as described in [LSTT02] storing the target coordinates in each packet is sufficient for deciding efficiently when two packets have two be exchanged, here we also have to take care of the wrapped around borders. For this, we provide one extra bit along each axis which is set if and only if the packet wants to 'cross the border' for reaching its target. In other words, for a node with coordinates (x_0, y_0) in a $2^m \times 2^m$-mesh, the 'horizontal cross border bit' is set if and only if $|x_t - x_0| > 2^{m-1}$. Thus, with each package that is to be routed we store the two (m-bit) target coordinates (x_t, y_t) plus the two 'cross border flags'. Using an 8-bit comparer and a simple circuit that deals with the cross border flags and the most significant bit of the target coordinates, we can easily decide if two adjacent nodes have to exchange their packets. Analogously as in [LSTT02] we assume that no more than one clock cycle is to be used for such a compare/exchange operation.

3.3 Refilling the mesh while routing

Experimentally it turns out, that the routing algorithm performs better if the number of 'travelling' packets is not too high. In our application, a processing unit usually has several packets that have to be output on the mesh for being routed to the respective target node. In principle, a processing unit can release a new packet whenever no other packet has to be stored. However, to avoid a slow-down through congestion of the mesh, in our experiments it turned out to be more efficient to release a new packet only if in the previous two clock cycles no packet was stored in that node. In this way usually $\approx 25\%$ of the processing units are 'free', which—experimentally—allows for a quite efficient routing. Each routed packet represents a divisor of some number in the currently processed sieving range, and the next section explains this connection in more detail.

4 Organizing the sieving

The basic organization of the sieving process is identical as in [GS03a], in particular we use line sieving, and when changing to a new line, i. e., increasing the current b-value, new data has to be loaded into the mesh. For sieving one line $-A \leq a < A$ of ($\approx 3.4 \cdot 10^{13}$) a-values, only local operations inside the mesh nodes are used. As indicated above already, a sieving line is not processed 'as one piece', but divided into consecutive intervals of length $S(= 2^{24})$. Before going into the details of how these subintervals are processed, we have to say how the two factor bases are represented in the mesh—at this we want to exploit the Kronecker/tensor product arrangement explained in Section 3.2.

4.1 Storing the factor bases in the mesh

Each processing unit stores[1] all elements (p, r) of the factor bases where the prime p is smaller than the size of the subinterval of the current sieving line processed by that processor—with the mentioned parameters for 768-bit numbers this translates to the condition $p \leq 2^8$. More precisely, for $p \leq 2^8$ and (p, r) being contained in a factor base, each processing stores the value $(p, (s_0 + r + 2^8 \cdot i) \bmod p)$ where s_0 is the first value in the processed sieving range of length S, and $i \in \{0, \ldots, 2^{16} - 1\}$ is the number of the subinterval of length 256 processed by that unit. The idea here is, that a processing unit will be able to test locally which 'tiny' primes divide an element in the sieving range processed in that node. Next, all pairs (p, r) with primes p that are 'up to 4 times larger'—namely with $2^8 < p \leq 2^{10}$—are stored 'once per 2×2-square' of the mesh. With the numbering from Section 3.2, this means that the prime p (along with the corresponding $(s_0 + r + 2^{10} \cdot i) \bmod p$-values) is stored in all processing units where the 'least significant tensor coordinates'—bits no. 0 and 1 in the binary representation of the number i of the subinterval of length 256—coincide. Again, the idea is to allow for a 'local' handling of prime divisors: a subquadrant of size 2×2 covers a sieving range of $2^2 \cdot 2^8 = 2^{10}$ numbers, and all prime divisors of size $\leq 2^{10}$ are available inside that square.

Next, we proceed analogously for submeshes of size 4×4 and primes $2^{10} < p \leq 2^{12}$. In other words, all processing units where the bits no. 0–3 of the binary representation of the number i of the processed subinterval coincide, store the same pairs (p, r)—where in analogy to the above r is replaced by $(s_0 + r + 2^{12} \cdot i) \bmod p$. So in each 4×4 subquadrant—which corresponds to a sieving range of length $4^2 \cdot 2^8 = 2^{12}$ all primes $\leq 2^{12}$ are 'available'. Going on in this way, in each submesh of size 8×8 we store the pairs (p, r) with $2^{12} < p \leq 2^{14}$, in each submesh of size 16×16 we take care of the primes $2^{14} < p \leq 2^{16}$, in each submesh of size 32×32 we deal with the primes $2^{16} < p \leq 2^{18}$, and in each submesh of size 64×64 we store the pairs (p, r) with $2^{18} < p \leq 2^{20}$. All pairs (p, r) with $p > 2^{20}$ are stored only once in the mesh. We do not consider subquadrants of size 128×128: due to the underlying torus topology, the horizontal or vertical distance between two nodes cannot be larger than 128 anyway.

For an actual implementation, the question of *how* to store the pairs (p, r) is crucial: with smoothness bounds $B_1 = 10^9$ and $B_2 = 10^8$, the rational factor base contains $5,761,456$ pairs (p, r) in total, and the algebraic factor base can

[1] For the moment, we postpone a discussion of *how* to store these pairs.

be assumed to consist of $\approx 50,850,000$ pairs. With the prime distribution just described and leaving some leeway for multiple prime factors, we conclude that each node in a 256×256 mesh has to store $\approx 1,300$ pairs (p, r). Storing them as pairs of 30-bit numbers in DRAM is extraordinary space/area-consuming, and not suitable for our purposes. Thus, before going into algorithmic details we should clarify how to store these pairs more efficiently: for storing its factor base elements, each node is equipped with three rectangular blocks of DRAM, where one DRAM block can be accessed in 'words' of 28 bit, one block can be accessed in 'words' of 31 bit, and the other block allows for access in 'words' of 7 bit. Within each block, sequential access is sufficient—random access is not required. The usage of the memory blocks depends on the size of the processed primes p: in analogy to [ST03], we call p

- **tiny**, if $2 \leq p < 2^{17}$,
- **smallish**, if $2^{17} \leq p \leq S$
- **largish**, if $S < p \leq B_2$.
- **hugish**, if $p > B_2$

For reasons of efficiency, with each tiny or smallish prime we also store the (non-negative) values $S \bmod p$ and $\lceil \log_{\sqrt{2}}(p) \rceil$. The details of the encoding used for storing the four different 'prime types' efficiently are given in Appendix A.

4.2 Sieving a subinterval

Let $\tilde{a}$ be an arbitrary number from a subinterval of length $S = 2^{24}$ and $(p, r) \in P_i$ an element of a factor base. Then $\lceil \log_{\sqrt{2}}(p) \rceil$ is added to the 'length counter' $s_i(\tilde{a})$ during line sieving if and only if $\tilde{a} \equiv br \pmod{p}$ $(i = 1, 2)$. When the factor bases have been loaded into the mesh as described, the mesh is prepared to sieve the first subinterval $-A \leq a < -A + S$ with $b = 1$. Each processing unit is in charge of 256 a-values (see Section 3.2), and conceptually splits into three parts:

The main part contains the DRAM with the stored factor bases along with the necessary logic to read out these elements. In particular, this logic is in charge of a flag which indicates whether currently the unique rational root (which is always stored first) or an algebraic root is processed. Also the 6-bit representation of the current $\lceil \log_{\sqrt{2}}(p) \rceil$-value is stored in this part of the processing unit.

After having retrieved an r-value from the DRAM, first we check whether it is 'relevant' for the current sieving subinterval of length S: as primes are stored repeatedly in the mesh, this 'relevance' does not depend only on r, but also on the size of p. More precisely, for $p \leq 256$ all values $r < 256$ have to be considered, for $256 < p \leq 1024$ all values $r < 1024$ are relevant, etc. For checking this 'relevance condition' efficiently we can make use of several OR gates that check the leading bits of r and a chain of multiplexers that is controlled by $\lceil \log_{\sqrt{2}}(p) \rceil$. If this r-value turns out to be not relevant, then we replace the old r-value in the DRAM by the new

$$r := \begin{cases} r - (S \bmod p) & , \text{if } r - (S \bmod p) \geq 0 \\ r - (S \bmod p) + p & , \text{otherwise} \end{cases} .$$

In this way, r is 'shifted' in the next subinterval of length S (cf. [GS03a]). Note that the value $S \bmod p$ is known already (for tiny and smallish primes it is stored in the DRAM and for largish and hugish primes we have $S = S \bmod p$), so this computation can be implemented efficiently by means of a 30-bit adder. Now the next root or prime can be processed.

On the other hand, if an r-value is identified as relevant, then the (x_t, y_t)-coordinates of the node that is in charge of this value are determined by appending the corresponding odd/even bit positions of r to the respective number of most significant bits of the node's own horizontal/vertical coordinate. In analogy to the relevance test, the precise number of bits that are to be copied from the node's own coordinates depends on p. Then the 'cross border' flags c_x, c_y are determined; for doing so, we may either use general adders with two (8-bit) inputs or an optimized component that can check whether the horizontal resp. vertical coordinate of the current node differs from x_t resp. y_t by more than 128. If the coordinates (x_t, y_t) are not identical with the node's own coordinates, then (x_t, y_t), (c_x, c_y), a 'footprint' of p, $\lceil \log_{\sqrt{2}}(p) \rceil$ (6 bit), the least significant 8 bit of r, and the flag which indicates whether the prime belongs to the algebraic or rational side are written into an output buffer which will be read by—but is not part of—the *mesh part* of the node (see below). What does 'footprint' of p mean? For promising (a, b)-pairs this footprint will be output to the processor that is in charge of the post-processing of the candidate pairs; for primes larger than some predetermined bound B_f, say $B_f = 2^{22}$, it should be possible to recover p from the footprint. In principle we could send the complete value p up to the least significant bit here, however to save some space a different footprint is preferable: we send the coordinates of the current node ($2 \cdot 8$ bit) concatenated with the bits no. 1–10 of p (i. e., the 10 least significant bits after dropping bit no. 0 which is always set). As each processing unit stores only ≈ 850 prime numbers larger than $B_f = 2^{22}$, this determines p in most cases uniquely. If a processing unit contains more than one prime with this footprint, in the postprocessing all primes with this footprint have to be taken into account. In summary, we write x_t, y_t (8 bit each), c_x, c_y (1 bit each), $\lceil \log_{\sqrt{2}}(p) \rceil$ (6 bit), the footprint (26 bit), the least significant 8 bits of r, and a one bit flag that distinguishes between the algebraic and the rational side into a 59-bit output buffer which will be read by the *mesh part* of the node (see below). According to our experiments, it is sufficient to provide space for two 59-bit entries in the output buffer; for storing the buffer entries, we can use latches which require only 4 transistors per bit.[2]

Now the currently processed prime p is added to the current r-value (with a 30-bit adder needing no more than 2 clock cycles), and we have to check as above whether the resulting new value $r := r + p$ is also 'relevant' for the processed sieving subinterval of length S. If this is not the case, then we update the old r-value in the DRAM accordingly for the next sieving subinterval, and otherwise we determine another (x_t, y_t)-pair.

[2] Actually, we could do with fewer bits: as the node's own coordinates (which are part of the 26-bit-footprint) are identical for all buffer entries, we could save some transistors by 'hardcoding' these bits.

Once an (x_t, y_t)-pair with the nodes own coordinates is encountered, the respective r-value has to be handled by the node itself and thus the least significant 8 bit of r, $\lceil \log_{\sqrt{2}}(p) \rceil$ (6 bit), the 26-bit footprint of p, and a 1-bit flag which indicates whether we deal with the rational or algebraic side are written in a 41-bit input buffer that is to be read by—and is part of—the *memory part* of the node (see below). Hereafter, p is added to the current r-value and checked for 'relevance' as already described. In summary, we estimate that realizing the main part requires $\approx 2,750$ transistors. Reading a DRAM entry takes 2 clock cycles, and the retrieved values can be processed in a pipeline structure. Thus, provided that the respective buffer is not full, basically every 4 clock cycles[3] an output is produced or the next p-value is selected. For storing the factor base elements, about $55,000$ bit of DRAM are needed.

The memory part of the node provides two 10-bit DRAM entries for each of the 256 a-values the node has to take care of—one 10-bit counter for the algebraic and one for the rational side. These 10-bit words are initialized with zero and used to store the sum of the $\lceil \log_{\sqrt{2}}(p) \rceil$-values that 'hit' the corresponding a-value during sieving on the algebraic resp. rational side. It is convenient to organize the DRAM for the 10-bit counters in 20-bit words, so that the algebraic and rational counter can be read simultaneously—we will exploit this when checking for simulteneous smoothness on the algebraic and the rational side.

The memory part reads from the mentioned 41-bit input buffer and uses the least significant 8 bit of r and the rational/algebraic flag to address the correct counter. To add the $\lceil \log_{\sqrt{2}}(p) \rceil$-value read from the input buffer to the respective 10-bit value in the DRAM, a 10-bit adder is used. Finally, a different part of the DRAM is needed to store footprints of prime factors larger than the already mentioned predetermined bound B_f, say $B_f = 2^{22}$. As explained above, for storing one footprint we need 26 bits. Moreover, we need 8 bits to identify the precise sieving location within the node, plus 1 bit to distinguish between the rational and the algebraic side. Thus, in total one complete entry occupies 26+8+1=35 bit of DRAM. However, instead of equipping each individual node with DRAM for storing found prime factors, it seems more efficient to share this DRAM among two nodes that are *physical* (cf. Section 3.1) neighbours. Consequently, we add one bit to each entry to identify the processing unit. Of course, the question arises how many prime factors will be encountered per sieving subinterval. According to our experiments for $B_f = 2^{22}$ and 256 targets per node, a DRAM size of $325 \cdot 36$ bit (shared by two nodes) seems reasonable. Further on, the question of choosing the size of the input buffer arises—according to our experiments a buffer with a single 41-bit entry should already be sufficient to avoid a performance bottleneck.

Finally, we have to explain how to identify 'good' (a, b)-pairs and how to output the respective prime factors from the device. For this purpose, the (somewhat dirty) approach explained in Appendix B seems reasonable. In summary, for im-

[3] Largish and hugish primes can usually be processed in 2 clock cycles, as most of them do not 'hit' in the current subinterval of length S.

plementing the memory part $\approx 1,250$ transistors should be sufficient (excluding the DRAM). For incrementing one $\lceil \log_{\sqrt{2}}(p) \rceil$-counter and checking both thresholds we allow 4 clock cycles which should provide enough leeway to store the (footprint of the) prime factor p into the DRAM. In addition to this, on average $\approx 11,000$ bit of DRAM (with random access) are needed for the found prime factors resp. for the $\lceil \log_{\sqrt{2}}(p) \rceil$-counters and the thresholds T_r, T_a.

In the mesh part the complete logic necessary for the clockwise transposition routing is located. In particular, this includes an 8-bit comparison unit plus some circuitry for taking care of the 'cross border flags' in each second node, which allows for an efficient (one clock cycle—cf. Section 3.2) exchange operation. The mesh part contains a register to store a complete packet as transported in the mesh. This register has the same width as the output buffer, and if a packet with (x_t, y_t) being identical to the node's own coordinates is encountered (and the input buffer of the memory part is not full), the 26-bit footprint of p, $\lceil \log_{\sqrt{2}}(p) \rceil$, the least significant 8 bits of r, and the factor base flag are copied into the input buffer of the memory part. New packets that have to be released into the mesh are read from the output buffer (which in turn is filled by the main part of the node as explained above). Implementing the mesh part should 'on average'[4] require no more than $1,100$ transistors.

4.3 Output of the result and moving to the next sieving interval

Once a complete subinterval of size S has been sieved, we have to output the found (a, b)-pairs: for doing so, each processing unit that has set the 'done' flag during the sieving procedure, outputs the footprints of all stored prime factors along with the corresponding factor base indices (1 bit each), the coordinates ($2\times$ 8 bit) of the processing unit that found the factor, and the least significant 8 bit of the corresponding r-value. Note here, that due to the 'cleaning process' the end of the list of factors is marked with an 'all zero' entry. The output values are received by supporting hardware that has to perform the final smoothness testing. Using the available 59-bit bus for this purpose, reading out the results should require less than 700 clock cycles. Of course, before outputing the results, we have to be sure that the sieving of the subinterval is indeed complete. However, there is no need to use a complicated logic for this: from a simulation one can determine a reasonable upper bound for the number of clock cycles that are needed to complete the sieving of a complete subinterval—for $S = 2^{24}$ on a 256×256 mesh such a bound can be $39,500$ clock cycles (this estimation is based on simulations by means of the computer algebra system Magma [BCP97]). After that time we simply instruct each processing unit to clear its input and output buffer, to complete any missing updates of its r-values, and to output its results. In the worst case (say the routing circuit encountered an infinite loop), potentially useful (a, b)-pairs from a single subinterval of length S are lost in this way.

[4] Recall that the comparer is needed only in each second node.

If the next subinterval to be sieved is in the same line, i. e., the b-value does not change, each processing unit simply has to reset its 256×2 10-bit counters to zero now, and is ready to sieve the next subinterval of length S. Analogously as in [GS03a], in this case *no new data has to be loaded into the device*, as all r-values have already been updated during the processing of the last sieving interval. Thus the change into the next sieving interval can be estimated to take no more than 500 clock cycles. Finally, passing to the next b-value requires a replacement of the stored r-values in the processing units, analogously as in [GS03a]. Using the 59-bit bus, 2000 pins per chip, and an I/O clocking rate of 133 MHz, we expect that loading the data into the DRAM takes no more than 0.02 seconds.

5 An improvement

Before discussing the performance of the above design, one may ask about possible improvements of the discussed parameters and the design. E. g., one may think of using larger or smaller mesh sizes, of different values for S or of changing the number of primes that each processing unit deals with locally. In this paper we focus on one (which we think reasonable) parameter choice, but there is still leeway for experimenting here—lacking a theoretical analysis of the underlying routing algorithm, we cannot make reliable theoretical predictions.

In Appendix C a significant improvement is described which affects the physical arrangement of the processing units as discussed in Section 3.1. The basic idea is to use four interleaving meshes instead of one. Certainly there are also other modifications of the above design, but in the estimations given in the next section we restrict to this improvement using four meshes.

6 Space requirements and performance of the device

Due to the 'small' DRAM banks, we take $0.3\mu\text{m}^2$ ($0.5\mu\text{m}^2$) for a DRAM bit with sequential (random) access and $2.8\mu\text{m}^2$ per transistor into account, which is somewhat larger than the estimates in [LSTT02, Table 2] for a specialized 0.13μm DRAM process. Combined with our estimations for the sizes of the node parts, this yields an estimated total size of a 256×256-mesh of $\approx (4.9\text{ cm})^2$. Here the use of four meshes as described in Appendix C is assumed.

Having in mind the comparatively regular layout and that 90nm processes are becoming more widespread, manufacturing such chips does not seem to be unrealistic (recall that for the proposed TWIRL parameters—which are not optimized for chip size—already the algebraic sieve has an estimated size of $\approx(6.7\text{cm})^2$ [ST03, Section 4.4]). In contrast to TWIRL, in the above mesh-based design a single device handles both factor bases, and we do not require any inter-chip communication, which is in particular helpful for cooling the chips. But what about the performance of the above device which we assume to be clocked with 500 MHz (instead of 1 GHz in TWIRL, where the time-critical operations are simpler)? Assuming a sieve line width of $3.4 \cdot 10^{13}$ (see [ST03, Table 1]) and that $40,000$ clock cycles are needed per sieving subinterval of size $S = 2^{24}$, a single chip with a mesh of size 256×256 can process a sieve line in ≈ 163 seconds. This is almost a factor 20 slower than a TWIRL cluster consisting of four rational sieves (each of size $\approx(3.6\text{cm})^2$) and one algebraic sieve (of size $\approx(6.7\text{cm})^2$). However, while only six TWIRL clusters fit on a single 300 mm silicon wafer, we can fit 21 mesh-circuits of size 256×256 on such a wafer. With 21 chips we can handle a sieve line in ≈ 7.8 seconds, which is ≈ 6.3 times more than a wafer with six TWIRL clusters.

Moreover, in [ST03] the authors explain how to exclude sieve regions where a and b have a common divisor 2 or 3. The common divisor 2 can be handled easily by our device—essentially, we have to add $2p$ to r instead of p. Handling the common divisor 3 is in principle possible, but would require significant additional logic, and we do not consider such a modification here. Hence, instead of an 'essentially free' 33% time reduction in TWIRL, we assume only an 'essentially free' 25% time reduction. Thus, for completing all expected $8.9 \cdot 10^6$ sieving lines (see [ST03, Table 1]) for a 768-bit number, with one wafer we expect that ≈ 600 days are needed—roughly 6.3 times more than with TWIRL. But as smaller and regular chips are simpler to produce, and as our design does not rely on inter-chip communication, from a practical point of view the mesh-based approach might be an interesting alternative to TWIRL clusters.

7 Conclusions and further work

The above discussion shows that building a mesh-based sieving device for 768-bit numbers could be feasible with current technology. Depending on the number of chips one is willing to use, performing the sieving step for such numbers within a few months seems feasible. In comparison to the proposed TWIRL clusters (which are not optimized for chip size), the chips in our design are smaller, no inter-chip communication is involved, and the rather regular layout should simplify the production of a detailed hardware layout. A main drawback of the mesh-based approach is a slow-down of a factor ≈ 6.3 compared to TWIRL. However, the simpler hardware requirements might outweigh this drawback. Also, we would like to emphasize that the discussed mesh is certainly not optimal, and modifying some of the paramter choices may yield relevant speed-ups. E. g., experiments show that if one is willing to allow for larger output buffers (which of course increases the chip size), the required number of clock cycles per sieving subinterval can be reduced.

Moreover, to further reduce the chip size, one can think of using a smaller mesh with only 128×128 nodes. According to our experiments, such a design is less efficient, but of course the resulting chips are smaller and producing them can be expected to be cheaper. On the other hand, one can ask whether implementing a larger 512×512 mesh is still feasible and whether a significant speed-up is possible in this way. We have not enough experimental results to give a reliable answer here, but exploring larger meshes is certainly worthwhile, in particular in regard to 1024-bit numbers: it is a natural question to ask to what extent the above mesh-based approach can deal with 1024-bit numbers. We cannot give a satisfying answer here at the moment. However, due to the compact representation of the factor bases in our device, it is certainly worthwhile to explore the 1024-bit case in more detail: the DRAM required for storing the factor bases is one of the critical issues in TWIRL, and it seems to be a very interesting question to explore the potential of the above mesh-based approach for 1024-bit numbers.

Acknowledgements

We are indebted to Eran Tromer and an anonymous referee for several detailed and helpful comments that helped to improve the original manuscript.

References

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma Algebra System I: The User Language. *Journal of Symbolic Computation*, 24:235–265, 1997.

[Ber01] 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.

[GS03a] Willi Geiselmann and Rainer Steinwandt. A Dedicated Sieving Hardware. In Yvo G. Desmedt, editor, *Public Key Cryptography — PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*. Springer, 2003.

[GS03b] Willi Geiselmann and Rainer Steinwandt. Hardware to Solve Sparse Systems of Linear Equations over GF(2). In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems; CHES 2003 Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 51–61. Springer, 2003.

[LHWL93] 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.

[LS00] 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.

[LSTT02] Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, and Eran Tromer. Analysis of Bernstein's Factorization Circuit. In Yuliang Zheng, editor, *Advances in Cryptology — ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2002.

[Sha99] 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.

[ST03] Adi Shamir and Eran Tromer. Factoring Large Numbers with the TWIRL Device. In Dan Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2003.

Appendix

A Storing the factor base elements

For the tiny primes, we store the 17-bit representation of p followed by the most significant 8 bit of $S \bmod p$ and a three-bit 'log-increase value' l_p in the first '28-bit word'. The value l_p is added to the $\lceil \log_{\sqrt{2}}(\tilde{p}) \rceil$-value of the previously processed[5] prime $\tilde{p}$, i. e., we store the primes in increasing order. In the next '28-bit word' we write the (unique!) 17-bit value r with $(p, r) \in P_2$ followed by the least significant 9 bit of $S \bmod p$, a zero bit, and two flag bits b_{26}, b_{27}. W. l. o. g., we assume b_{26} and b_{27} to be the most significant bits, say b_{27} is the most significant one. The flag b_{26} is set if and only if there is also some pair $(p, r') \in P_1$, i. e., a root on the algebraic side belonging to p; in this case, the next 28-bit word contains the (first) algebraic root r' (a 17-bit value) followed by 9 reset bits. If bit no. 26 is reset, then the next entry contains another algebraic root of the currently processed prime, and otherwise the next entry belongs to a new tiny prime. To mark the last tiny prime we set the flag b_{27} resp. the most significant bit in the last root belonging to that prime. Thus, once the control logic encounters such a set most-significant bit, beginning with the next DRAM entry smallish primes have to be processed. Figure 3 surveys the DRAM usage for a tiny prime.

tiny prime p (17 bit)	m.s. 8 bit of S mod p	l_p (3 bit)	
rational root (17 bit)	l.s. 9 bit of S mod p	b_{26} (1 bit)	b_{27} (1 bit)
first algebraic root (17 bit)	$0 \ldots 0$ (9 bit)	b_{26} (1 bit)	b_{27} (1 bit)

...

Fig. 3. storing tiny primes

For smallish primes we use a similar approach, but here we reserve a complete 29-bit word for storing the smallish prime itself. More precisely for $S = 2^{24}$, a smallish prime p is stored in the least-significant 24 bit of the first 28-bit word, followed by a zero bit, and a three-bit 'log-increase value' l_p as described above. The next 28-bit word contains the 24-bit value $S \bmod p$ followed by 4 zero bits. Finally, the subsequent words are used in the same way as for smallish primes, with the only difference that now 24 (instead of 17) bit are used to represent the respective r-value. In particular, the most significant two bits serve again as flags to mark the last root belonging to the prime p and to mark the last smallish prime. This time, a set bit no. 27 indicates that the next prime to be

[5] Before starting the processing of the primes, an initial value—that depends on the number of small primes one wants to ignore and is identical for all processing units—has to be fixed here.

processed is a largish one. The DRAM usage for smallish primes is summarized in Figure 4.

smallish prime p (24 bit)	0 (1 bit)	l_p (3 bit)	
$S \bmod p$ (24 bit)	$0 \ldots 0$ (4 bit)		
rational root (24 bit)	0 0 (2 bit)	b_{26} (1 bit)	b_{27} (1 bit)
first algebraic root (24 bit)	0 0 (2 bit)	b_{26} (1 bit)	b_{27} (1 bit)

...

Fig. 4. storing smallish primes

The majority of the primes on the rational side are largish ones, and it is desirable to use a compact encoding here (without losing the ability to read out the data efficiently). A simple trick to achieve this (and which allows for an easy decoding of the DRAM entries) is to store prime *differences* instead of the prime values themselves: between 2^{24} and $B_2 = 10^8$, the maximal difference between two consecutive primes is 220. Obviously all these differences are even, and thus we can store them by storing only the most significant 7 bits of such a difference. As we are dealing with primes $p > S$ now, storing the value $S \bmod p$ does not make sense, and we use the following encoding: if the first word in the *7-bit DRAM block* is not identically 0, then these 7 bit are the most significant bits of the difference Δp between the currently stored prime p' and the next prime p to be processed. From the leading bits of p' when computing $p = p' + \Delta p$, we can check whether the $\lceil \log_{\sqrt{2}}(p') \rceil$-value has to be increased by one. The next entry in the *28-bit DRAM block* stores the full 27-bit representation of the rational zero corresponding to that prime, followed by a one-bit flag b_{27} which indicates whether for this prime also an algebraic root exists. If this flag is set, then the next 28-bit DRAM entry contains the 27-bit representation of the (first) algebraic root followed by a one bit-flag b_{27} which is set if and only if another algebraic root is stored for this prime. If this flag is reset, then no more data is available for the currently processed prime, and the next entry of the 7-bit DRAM is to be read and processed in the same way.

If a value read from the 7-bit DRAM is zero, then this indicates a ‘big prime difference’: in this case, the prime p to be processed is found in the next 28-bit DRAM entry. Namely, this 28-bit word contains the most significant 26 bit of p (all largish primes are odd) followed by a two-bit ‘log-increase value’ l_p. The subsequent 28-bit words encode the rational and (if existent) the corresponding algebraic roots in the same way as before, i. e., after the 27-bit representation of the r-value a flag indicates whether another (algebraic) root is available for this prime. To recognize the end of the list of largish primes we store the all-one-word in the 7-bit DRAM. Figure 5 summarizes the memory usage for largish primes.

Finally, the hugish primes $< 2^{27}$ can be handled in the same way as the largish ones. The only difference is, that no rational root r is stored here, as the hugish primes are relevant for the algebraic side only. As the 28-bit words are too

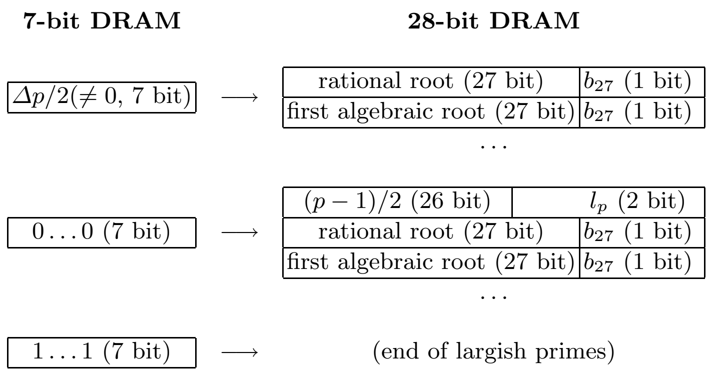

Fig. 5. storing largish primes

small for storing hugish primes $\geq 2^{27}$ conveniently, we use again an all-one-entry in the 7-bit entry to indicate that the remaining hugish primes can be found in the 31-bit DRAM. The storage format used in the 31-bit DRAM is identical to that of the 28-bit DRAM. Eventually, to indicate the end of the list of stored primes, we use another all-one-entry in the 7-bit DRAM; Figure 6 summarizes the memory usage for hugish primes.

For some tiny prime numbers p one may like to include powers p^ν in the sieving process. In principle we can do so easily by just inserting the respective entry at the correct position in the list of tiny primes. However, due to the sieving procedure which we want to use, there is a small subtlety here: when reporting a multiple factor p^ν, our sieving procedure also reports all divisors p^μ with $\mu < \nu$. To compensate for this, multiple factors p^ν should only contribute $\lceil \log_{\sqrt{2}}(p) \rceil$ to the 'length' counter instead of $\lceil \log_{\sqrt{2}}(p^\nu) \rceil$, and therefore we store the multiples p^ν immediately after the tiny prime p with a 'log-increase value' $l_p = 0$. As with the other elements of the factor bases, the number of processing units that store a prime power p^ν depends on the size of p^ν; e. g., if we want to take 2^9 into account, then this value is stored once per 2×2 subquadrant.

B Storing found prime factors

In the memory part, for updating a 10-bit counter on the algebraic resp. rational side, a complete 20-bit word is read from the DRAM which contains also the corresponding rational resp. algebraic counter. Thus, after having performed a counter update (and if necessary stored the most recent prime factors) we can check whether both the rational and the algebraic counter are larger than a smoothness threshold T_r resp. T_a[6] (each requiring 10 bit of DRAM). In this

[6] These smoothness bounds change throughout the sieving process, and we can update them through an external signal when passing to a new sieving subinterval, for instance.

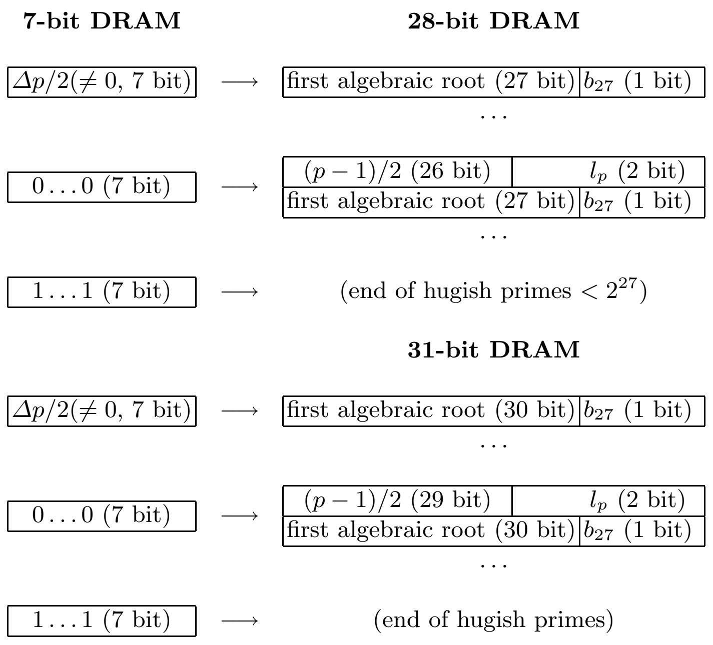

Fig. 6. storing hugish primes

case, a potentially useful (a, b)-pair has been identified, and the processing unit sets a 'done' flag to indicate that a candidate (a, b)-pair has been found. Setting this flag prevents that the processing unit or its physical neighbour writes any further prime factors into the (shared) list of prime factors—counter increments are performed as before. Once this 'done' flag is set, the DRAM with the stored prime factors is 'cleaned', i. e., all entries that do not belong to the promising (a, b)-candidate (these entries are identified by means of a circuit that can decide equality of 8-bit entries) are reset to zero, and the factors belonging to this (a, b)-candidate are stored at the beginning of the memory. This will later facilitate a quick output of the sieving result.

Once the 'DRAM cleaning' has been completed, we can set a 'cleaning ok' flag which means that from now on prime factors can be stored again provided that they belong to a(nother) promising (a, b)-pair. So for the (rare) case that in a range of $2 \cdot 256 = 512$ a-values more than one hit is encountered, we can store at least some of the large prime factors of the additional hit.

C Using four dedicated meshes

The improvement of the original mesh architecture described in this section also affects the physical arrangement of the processing units discussed in Section 3.1. Namely, we want to replace the single torus along each axis by two 'interleaved' tori. To be able to do this, we start with an ordinary $2^m \times 2^m$-array of processing units—without connections among the processing units. Next, we divide the nodes into $2^{m-1} \times 2^{m-1}$ 'macro nodes' where each macro node is comprised of four processing units. Now we form four *separate* meshes where

– Mesh I connects the 'left-upper' processing units of the macro nodes,
– Mesh II connects the 'left-lower' processing units of the macro nodes,
– Mesh III connects the 'right-lower' processing units of the macro nodes, and
– Mesh IV connects the 'right-upper' processing units of the macro nodes.

In each of the four meshes we want to pass to a torus topology without destroying the 2×2-macro nodes. To do so in an implementation we can proceed in exactly the same manner as described in Section 3.1; the only difference is that now the interleaving of rows and columns has to be done with the 2×2-macro nodes instead of the individual processing units. Why keeping the 2×2-blocks 'in one piece'? At the moment, the meshes I–IV, which can be realized on separate metal layers, are not connected at all, and consequently routing a packet from one processing unit to another is in general not possible. To overcome this problem, we replace the four separate output buffers in a macro node with one buffer that is readable for each of the four mesh parts present in the macro node. The mesh part of the 'left-upper' processing unit will read out all packets where both the x- and the y-coordinate are even, etc. In this way, each packet will automatically get into the right mesh. However, each of the four meshes now has a size of $2^{m-1} \times 2^{m-1}$, and once the packet is read by the respective mesh part we can drop the least significant bits of the two target coordinates. E. g., instead of dealing with a 256×256-mesh we now can deal with four 128×128-meshes. This also reduces the size of the comparison units and registers needed in each of the four mesh parts.

Of course, all four main parts of a macro node need write-access to the common output buffer, and the question arises of how large the output buffer should be. According to our experiments, for a 256×256-array (with four meshes) it is enough to allow for eight entries (again 58 bit each, as the least significant bits of the target coordinates are needed in the output buffer to address the mesh). Concerning the space requirement of our device, the size of the main and memory parts of the nodes are basically left unaltered by our modification—combining the four output buffers to one does not change the 'average' size of the nodes significantly. However, the structure of the mesh part has changed—a slightly more complicated logic for reading from the output buffer is needed now, but the comparison unit and the register for storing a packet have become a bit smaller. In summary, for the 256×256-array with four meshes we assume the modified target unit to be realizable with $\approx 1,100$ transistors, too.