Area-time Optimal Division

for $T = \Omega((logn)^{1+\epsilon})$

by

K. Mehlhorn* and F.P. Preparata**

* Fachbereich 10, Informatik, Universität des Saarlandes, 6600, Saarbrücken, West Germany.

* Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801.

This work was supported by the DFG, SFB 124, VLSI Entwurf und Parallelität, and by NSF Grant ECS-84-10902.

A family of area-time (AT^2) optimal networks for the computation of the inverse of an n-bit number (referred to here as "dividers") has been proposed some time ago by Mehlhorn [1]. A network of this type can be constructed for each computation time T in the range $[\Omega(\log^2 n), 0(\sqrt{n})]$. Since then considerable progress has been made in the design of faster dividers [2], culminating in the result of Beame-Cook-Hoover [3] illustrating an $O(\log n)$-time divider (i.e. , a time-optimal network in the hypothesis of bounded-fan-in components). However the Beame-Cook- Hoover network (referred to here as the BCH network) does not achieve area optimality. Thus, it is natural to ask the question of the existence of area-time optimal dividers for $T = o(\log^2 n)$. This paper provides an affirmature answer for $T \in [\Omega((\log n)^{1+\epsilon}), O(\log^2 n)]$ for any positive constant $\epsilon \leq 1$. It must be pointed out that the proposed networks are so complicated - notwithstanding their area-time-optimality - that they are exclusively of theoretical interest.

The network (see Figure 1) consists of $\lceil 1/\epsilon \rceil + 2$ cascaded modules. (For simplicity we assume that $1/\epsilon$ is an integer.) The first $1/\epsilon$ modules are modified dividers of the BCH type, computing a sequence of approximations of the inverse with increasing numbers of bits $l_1 \leq l_2 \leq \ldots \leq l_{1/\epsilon} \leq n$.

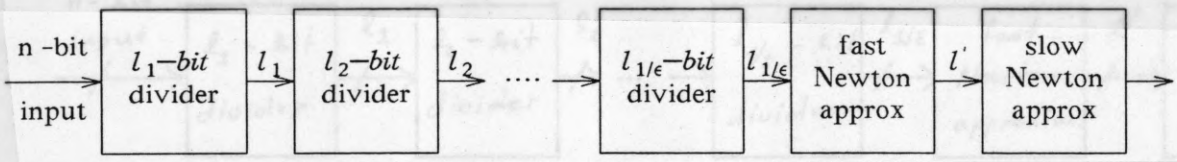

Fig. 1: Block structure of the divider

The last two modules are designed to complete the build-up of the result size from $l_{1/\epsilon}$ to n bits by implementing the Newton approximation method, which, at each iteration doubles the length of the result. This is carried out in two phases, respectively executed by the "fast" and "slow" approximators. The fast approximator basically consists of a single area-time optimal fastest multiplier, used to execute the initial iterations; the slow approximator is instead a cascade of affordably slow multipliers, each executing one of the final iterations. Both approximators execute $0(loglogn)$ iteration steps. Note that the cascade of the two Newton approximators structurally coincides with Mehlhorn's divider [1].

The paper is organized as follows. In section 2 we present a more efficient implementation of the BCH method leading to a circuit referred to as "modified BCH divider". In Section 3 we discuss an alternative method for the computation of the inverse, which uses the modified BCH method as a subroutine. Finally, in

Section 4 we illustrate the combination of the previous techniques with the Newton approximation, to yield our proposed network, while Section 5 contains a few closing remarks.

2.. An efficient implementation of the BCH method.

In this section we first describe (a variant of) the BCH method [3] and then modify it so as to reduce its area requirement.

The original BCH method computes the inverse of an n-bit number x by adding the first n powers of $u = 1 - x$ and truncating the n^2-bit result to its leading n-bits. Each power of u is computed individually and the n powers are subsequently added together; so we just consider the computation of u^n. The approach consists of taking the "logarithm" of u, multiplying it by n, and then taking the "antilogarithm".

Since taking logarithms of large numbers is very hard, the method resorts to a modular representation and works as follows:

Algorithm INVERSE1(x)

Input: an n-bit number x in the range $[1/2, 1)$. Given are primes $p_1, \ldots, p_m$ such that

$$\prod_{j=1}^{m} p_j \geq 2^{(n^2)} \quad \text{(Note that } m \simeq n^2/\log n\text{)}$$

(n is assumed to be a power of two)

Output: an $(n+2)$-bit number v in the range $(1, 2]$, so that $v \times x = 1 + \delta$ with $\delta < 2^{-n-1}$ (v is given by the first $n+2$ bits of $\sum_{i=0}^{n-1}(1-x)^i$)

(1) **begin** $u := (1-x)2^n$;($*u$ is an integer $*$)
(2) **for** $j, 1 \leq j \leq m$
(3) **pardo** $b_j := u mod p_j$;
(4) compute r_j so that $a_j^{r_j} = b_j$, where
a_j is a generator of the multiplicative group of $\mathbf{Z}^*_{p_j}$;
(5) **for** $l = 0$ **to** $\log n - 1$
(6) **do** $m_j^{(l)} := a_j^{r_j 2^l mod(p_j - 1)}$ $(*m_j^{(l)} = u^{2^l} mod p_j *)$
(7) **od**;
(8) $v_j := \prod_{l=0}^{\log n - 1}(m_j^{(l)} + 1) mod p_j$
$(*v_j = \sum_{l=0}^{n-1} u^l \mod p_j *)$
(9) $V_j := v_j M_j mod(p_1 \ldots p_m)$
(* Chinese remaindering *)
(10) **odpar**;

(11) $v := \sum_{j=1}^{m} V_j mod(p_1 \dots p_m)$;
(12) $v :=$ truncate v to the first $m+2$ bits and set point after the second bit from the left
(13) **end**

Let us next describe the different steps of this algorithm in more detail. In this description we will make frequent use of the following two facts.

1) One can multiply two k-bit integers in time T and area A where $AT^2 = O(k^2)$ and $T \in [\Omega(\log k), O(\sqrt{k})]$. This is the result of [6].

2) One can add m k-bit integers in time $O(\log m + \log k)$ and area $O(km \cdot \log m)$. This can be achieved by expressing the m integers in redundant representation (see, e.g. [4,5,6]) and then adding them in a tree-like fashion. The tree has depth $O(\log m)$ and requires area $O(m \log m)$ for every bit position. Each level of the tree introduces a delay of just $O(1)$ thanks to the redundant number representation.

We are now ready to describe the circuit in more detail. We start with the parallel loop, lines 2-10.

Line 3: This line is easily executed in time $O(\log n)$ and area $O(n(\log n)^2)$ by expressing u by its binary expansion $u = \sum_{t=0}^{\log n - 1} u_t 2^t, u_t \in \{0,1\}$, storing the numbers $2^t mod p_j$ in a table and performing the required additions in redundant number representation. We leave the details of this step to the reader.

Line 4: Step 4 is realized by a table-look-up, i.e. by a loop-up in a table which gives the value of r_j for each possible value of b_j. Since p_j can certainly be expressed using $2 \log n$ bits this table has n^2 extries of $2 \log n$ bits each. We realize this table by $2 \log n$ H-trees each requiring area $O(n^2)$. Thus the total area is $O(n^2 \log n)$ and a table-look-up takes time $O(\log n)$.

Note that the $2 \log n$ slices of the table are accessed in parallel. Also note that this circuit is pipelinable, (its period is $O(1)$ in technical terms) and therefore $O(\log n)$ look-ups can also be performed in time $O(\log n)$ using the same area. This observation is important for step 6.

Line 5,6,7: Consider a fixed l first. We first compute

$$R_j^{(l)} = r_j 2^l mod(p_j - 1)$$

as outlined in line 3. Note that the l-place shift does not have to be executed explicitly; it only determines which powers of two need to be looked-up. The computation of $R_j^{(l)}$ takes time $O(\log n)$ and area $O(n(\log n)^2)$. We perform this computation in parallel for all l, $0 \leq l \leq \log n - 1$.

The integer $m_j^{(l)}$ is computed from $R_j^{(l)}$ by look-up in a table of "antilogarithms". The $\log n$ look-ups are pipelined and take time $O(\log n)$ and area $O(n^2 \log n)$ (refer to the description of line 3).

Finally note that $m_j^{(l)} = a_j^{r_j 2^l \bmod(p_j-1)} = b_j^{2^l} \bmod p_j = u^{2^l} \bmod p_j$.

Line 8: We use a tree of multipliers. This tree has depth $O(\log\log n)$ and has $\log n$ nodes. Each node contains a circuit multiplying two $2\log n$ bit numbers and reducing the result $\bmod p_j$ in time $O(\log\log n)$ and area $O((\log n)^2)$. This shows that step 8 takes time $(\log n)$ and area $O(n)$. Both estimates are very generous.

Finally note that

$$\prod_{l=0}^{\log n-1}(1+m_j^{(l)}) = \prod_{l=0}^{\log n-1}(1+u^{(2^l)}) = \sum_{l=0}^{n-1} u^l$$

Line 9: Let $M_j = [(p_1 \dots p_m)/p_j]^{p_j-1} (\bmod p_1 \dots p_m)$. Then M_j is the coefficient of v_j required for Chinese remaindering [7]. The number M_j is precomputed and stored in a register of length $O(n^2)$. We multiply v_j by M_j by dividing M_j into $n^2/\log n$ pieces of length $O(\log n)$, performing $n^2/\log n$ multiplications in parallel and then summing the results. This can certainly be done in time $O(\log n)$ and area $O(n^2 \log n)$. Also the reduction $\bmod(p_1 \dots p_m)$ can be done in that area and time.

Summary: Line (3) to (9) take time $O(\log n)$ and area $O(n^2 \log n)$ for each p_j. Since u^n has n^2 bits we have $m = \Theta(n^2/\log n)$ and each modulus is representable in $2\log n$ bits. We realize loop (2) to (10) by having a module for each modulus and hence the loop takes time $O(\log n)$ and area $O(n^4)$.

Line 11: In Line 11 we add m numbers of n^2 bits each. This takes time $O(\log n)$ and area $O(m \log m \cdot n^2) = O(n^4)$.

Lemma 1. There exists a circuit which computes the n-bit inverse of an n-bit number in time $O(\log n)$ and area $O(n^4)$.

Proof: Immediate from the discussion above. ∎

The enormous space requirement of the method sketched above is essentially due to the fact that the powers of u are computed with $\Theta(n^2)$ bits of precision. However, only the leading $n + \log n$ bits are truly needed for the computation of v. This observation is the key to the "modified" BCH method, to be described next. In the modified method we compute the powers of an l-bit integer u in m rounds (this m has nothing to do with the m in algorithm INVERSE1), where m is a design parameter to be selected. In each round we compute the sum of $s = (l)^{1/m}$ consecutive powers using the method of Lemma 1. We call s the *depth* of the method. This takes time $O(\log l)$ and area $O((ls)^2)$ and yields a result of $O(ls)$ bits. The space requirement results from the fact that only $ls/\log(ls)$ different prime moduli, each of length $2\log(ls)$ bits, must be used. We truncate this result to $l + \lceil \log 12m \rceil$ bits and start the next round. The details are as follows.

Algorithm INVERSE2(x)

Input: an l-bit number $x \in [1/2, 1)$ and an integer $s = (l)^{1/m}$.
Output: an $(l+2)$-bit number $v \in (1, 2]$

begin $u_o := 1 - x$;
  for $i = 0$ **to** $m - 1$ **do**
    begin compute $u_i, u_i^2, \ldots, u_i^{s-1}, u_i^s$;
      $\sigma_i := \sum_{j=0}^{s-1} u_i^j$
      $u_{i+1} :=$ truncate u_i^s to $q = l + \lceil \log 12m \rceil$ bits right of point;
    end;
  $v :=$truncate $\sigma_0 \sigma_1 \ldots \sigma_{m-1}$ to l bits right of point;
end.

To prove the correctness of this algorithm we must show that v gives the $(l+2)$ leading bits of $1/(1-u)$ (of which the rightmost l bits represent the fractional part). To this end, we must show that the error of the approximation is $< 2^{-l}$.

For any variable a used by the above algorithm let $\tilde{a}$ denote the corresponding exact value (note that, since all numbers are nonnegative, the truncation mechanism gives $\tilde{a} \geq a$), and $\delta(a)$ the absolute error on a, such that $a = \tilde{a} - \delta(a)$. Recall also that $\delta(a \cdot b) < \delta(a)\tilde{b} + \delta(b)\,\tilde{a}$ and that $\delta(a+b) = \delta(a) + \delta(b)$. Using these relationships, we readily have

$$\delta(\sigma_0 \ldots \sigma_{m-1}) < \tilde{\sigma}_0 \ldots \tilde{\sigma}_{m-1} \left(\frac{\delta(\sigma_0)}{\tilde{\sigma}_0} + \ldots + \frac{\delta(\sigma_{m-1})}{\sigma_{m-1}} \right)$$

Since $\tilde{\sigma}_0 \ldots \tilde{\sigma}_{m-1} < 3$ and $\tilde{\sigma}_i > 1 (i = 0, \ldots, m-1)$, we obtain

$$\delta(\sigma_0 \ldots \sigma_{m-1}) < 3(\delta(\sigma_0) + \ldots + \delta(\sigma_{m-1})).$$

From $\tilde{\sigma}_i = \sum_{j=0}^{s-1} \tilde{u}_i^j$ we have

$$\delta(\sigma_i) = \sum_{j=0}^{s-1} \delta(u_i^j) < \sum_{j=0}^{s-1} j \tilde{u}_i^{j-1} \delta(u_i) < \delta(u_i)/(1 - \tilde{u}_i)^2 < 4\delta(u_i)$$

since $\tilde{u}_i < 1/2$ for $i = 1, \ldots, m-1$. (Obviously $\delta(u_0) = 0$.)

Thus $\delta(\sigma_0 \ldots \sigma_{m-1}) < 12m \max \delta(u_i)$ and the condition

$$12m \max \delta(u_i) < 2^{-l}$$

ensures the correctness of the method. We claim that $\delta(u_i) < 2^{-q}$ as a result of truncating to q bits right of the point. Indeed $\delta(u_1) < 2^{-q}$, trivially. For $i \geq 1$, assuming $\delta(u_i) < 2^{-q}$, let $u_{i+1}^* = u_i^s$ (before the truncation). Then

$$\delta(u_{i+1}^*) < s\tilde{u}_i^{s-1}\delta(u_i) < \frac{s}{2^{s-1}}\delta(u_i)$$

since $u_i < 1/2$. If we assume $s \geq 4$, then $\delta(u_{i+1}^*) < 2^{-q-1}$, which shows that its $(q+1)$ bits to the right of the point are correct. Thus, the prescribed truncation yields $\delta(u_{i+1}) < 2^{-q}$, and the induction step is complete. In conclusion, we choose

$$q \geq l + \log 12m$$

(Note that for any choice of s, $\lceil \log 12m \rceil < 4 + \log\log l$ by the definition of m.)

Noting that $m \cdot O(\log l) = O(\log l / \log s)$, we have:

Lemma 2. For any $2 \leq s \leq l$ there exists a circuit computing the l-bit inverse of an l-bit number in time $O(\log^2 l / \log s)$ and with area $O((ls)^2)$.

The AT^2-performance of the above circuit is given by

$$AT^2 = O\left(l^2 \log^4 l \cdot \frac{s^2}{\log^2 s}\right) \tag{1}$$

By choosing the depth s as $s = l^\epsilon (\epsilon > 0)$, the resulting circuit achieves $T = O((1/\epsilon)\log l)$ and $AT^2 = O(l^{2(1+\epsilon)})$, i.e. it is a moderately AT^2- suboptimal divider still achieving $T = O(\log l)$, for fixed ϵ. We are aware that this result had been previously obtained by F. T. Leighton [8], presumably by a similar argument.

3.. An Accelerating Technique

We now describe an alternative approach to the computation of the inverse of an l-bit number, which capitalizes on the presence of leading zeros in the representation of the number to be inverted. This method is best described for an l-bit integer $x \in [1, 2)$.

The number $x \in [1, 2)$ can be written as

$$x = x_1 + 2^{-l_1} w$$

where x_1 is an l_1−bit number (the leading l_1 bits of x) and w is an $(l - l_1)$-bit number (the trailing $l - l_1$-bits of x). Then $x_1 \in [1, 2)$ and $2^{-l_1} w \in [0, 2)$. Let v_1 be an l_1-bit approximation to x_1 (i.e. $x_1 v_1 = 1 + \eta, \eta < 2^{-l_1}$). Then

$$v_1 x = v_1 x_1 + v_1 w 2^{-l_1} = 1 + \eta + v_1 w 2^{-l_1},$$

that is, $v_1 x$ has at least $l_1 - 1$ consecutive 0's immediately to the right of the point. Define

$$y = 1/v_1 x$$

Then, if v_2 denotes an approximation of $1/y$, we have $v_1 v_2 \simeq 1/x$. Also, if $v_1 y = 1 + \eta'$ then $v_1 v_2 x = 1 + \eta'$, i.e. $v_1 v_2$ is an approximation of precision η'. The process can be iterated for the computation of the inverse of y, thereby obtaining

$$1/x = v_1 v_2 \ldots v_k$$

This leads to the following algorithm:

Algorithm INVERSE3(x)

Input: an l-bit number $x \in [1, 2)$, and an integer sequence $l_1 < l_2 < \ldots < l_k = l$.
Output: an l-bit number $v \in (1/2, 1]$, such that $vx = 1 + \epsilon, \epsilon < 2^{-l}$

(1) **begin** $z := x$
(2) **for** $i = 1$ **to** k **do**
(3) **begin** $x_i :=$ leftmost l_i bits of z_i;
(4) $v_i :=$ $(l_i + 1)$-bit inverse of x_i;
(5) $z_{i+1} := z_i v_i$
end;
(6) $v := v_1 v_2 \ldots v_k$
end

The correctness of the method follows from the fact that $\delta(v) = \tilde{v}_1 \ldots \tilde{v}_{k-1} \cdot \delta(v_k) < 2 \cdot 2^{-l-1} = 2^{-l}$.

Step 4 is the crucial action in the above algorithm. To analyze its performance, we need the following result.

Lemma 3. If an l-bit number $x \in [1, 2)$ has $l_1 - 1$ zeros immediately to the right of the point, the l-bit inverse of x can be computed in time $T = O(\log(l/l_1) \cdot \log l / \log s)$ and area $A = O((ls)^2)$, for any $2 \leq s < l/l_1$. (Note that this result subsumes Lemma 2 for $l_1 = 1$.)

Proof: Indeed $u = 1 - x$ is a (negative) number with l_1 zeros immediately to the right of the point. This implies that $u^{\lceil l/l_1 \rceil} < 2^{-l}$, so that only the first $\lceil l/l_1 \rceil$ consecutive powers of u need to be computed. ∎

The numbers $x_i, i = 1, \ldots, k$, used in Step 4 meet the conditions of Lemma 3, since $1 - z_i v_i$ is a (negative) number with l_i leading zeros. Step 4 is therefore carried out by applying Algorithm INVERSE2 so that the i-th iteration is characterized by length l_i and depth s_i. An implementation of this accelerating technique is therefore completely specified by the two sequences:

$$l_1, l_2, \ldots, l_k$$

and

$$s_1, s_2, \ldots, s_k,$$

Before closing this section we note that Step 5 involves a multiplication of $(l_i + 1)$-bit numbers at the i-th step; thus this operation is no more complex than the execution of the homologous Step 4, and will not be further mentioned in this discussion.

4. The Divider Network

We have all the premises to illustrate in detail the structure of the divider sketched in Figure 1.

The first $1/\epsilon$ stages are collectively designed to implement the accelerating technique; each module implements the modified BCH algorithm. For $i = 1, 2, \ldots, 1/\epsilon$, let l_i be the (output) operand length, s_i the depth, $A_{1,i}$ the area, and $T_{1,i}$ the time of the i-th module. We seek a solution where all such modules have identical area (i.e. $A_{1,i} = A'$ for $i = 1, \ldots, 1/\epsilon$) and identical computation time, equal to the target time (i.e. , $T_{1,i} = \theta((\log n)^{1+\epsilon}), i = 1, \ldots, 1/\epsilon$). By the requirement of optimality, we have

$$\sqrt{A_{1,i}} = \frac{n}{T_{1,i}} = \frac{n}{(\log n)^{1+\epsilon}}. \tag{2}$$

We also choose:

$$l_i = \frac{n}{(\log n)^{1+\epsilon} s_i},$$

$$s_i = 2^{(\log n)^{1-\epsilon}/(1+(\log n)^{\epsilon})^{i-1}} \quad (i = 1, \ldots, 1/\epsilon).$$

Since the area of the i-th module is $\theta((l_i s_i)^2)$, condition (2) is obviously verified. Next note that $\log l_i = O(\log n), i = 1, \ldots, 1/\epsilon$. We therefore infer from Lemma 2:

$$\begin{aligned} T_{1,1} &= O(\log l_1 \frac{\log l_1}{\log s_1}) \\ &= O((\log n)^2/(\log n)^{1-\epsilon}) \\ &= O((\log n)^{1+\epsilon}) \end{aligned}$$

and for $i = 2, \ldots, 1/\epsilon$

$$\begin{aligned} T_{1,i} &= O(\log \frac{l_i}{l_{i-1}} \cdot \frac{\log l_i}{\log s_i}) \\ &= O(\log \frac{s_{i-1}}{s_i} \cdot \frac{\log l_i}{\log s_i}) \qquad \text{since} \quad l_i s_i = l_{i-1} s_{i-1} \\ &= O(\log l_i \cdot \frac{\log s_{i-1}}{\log s_i}) \qquad \text{since} \quad s_i \geq 1 \\ &= O(\log n \cdot \frac{(\log n)^{1-\epsilon}(1 + (\log n)^{\epsilon})^{i-1}}{(1 + (\log n)^{\epsilon})^{i-2}(\log n)^{1-\epsilon}}) \\ &= O((\log n)^{1+\epsilon}) \end{aligned}$$

thus verifying the objective for the computation time.

With these choices, each module of the chain is AT^2-optimal, and the global computation time is $c_1(1/\epsilon)(\log n)^{1+\epsilon} = \theta((\log n)^{1+\epsilon})$, for some constant c_1. The value of $l_{1/\epsilon}$, the number of bits of the result, is bounded from below as follows:

$$l_{1/\epsilon} = \frac{n}{(logn)^{1+\epsilon} 2^{(\log n)^{1-\epsilon}/(1+(\log n)^{\epsilon})^{1/\epsilon - 1}}} > \frac{n}{(\log n)^{1+\epsilon} \cdot 2}.$$

This value $l_{1/\epsilon}$ represents the length of the operand supplied to the cascade of the two Newton approximators, to be described next.

Starting with the downstream approximator, we recall (see figure 2) that this module is in turn the cascade of p submodules (p is an integer to be defined shortly), where the i-th submodule has area and time $A_{3,i}$ and $T_{3,i}$, respectively, and

$$A_{3,i} = 2A_{3,i-1}, \quad T_{3,i} = \sqrt{2} T_{3,i-1} \quad i = 2, 3, \ldots, p.$$

With this choice (originally proposed in [1]), the global area and time of the slow approximator are respectively proportional to the area $A_{3,p}$ and time $T_{3,p}$ of

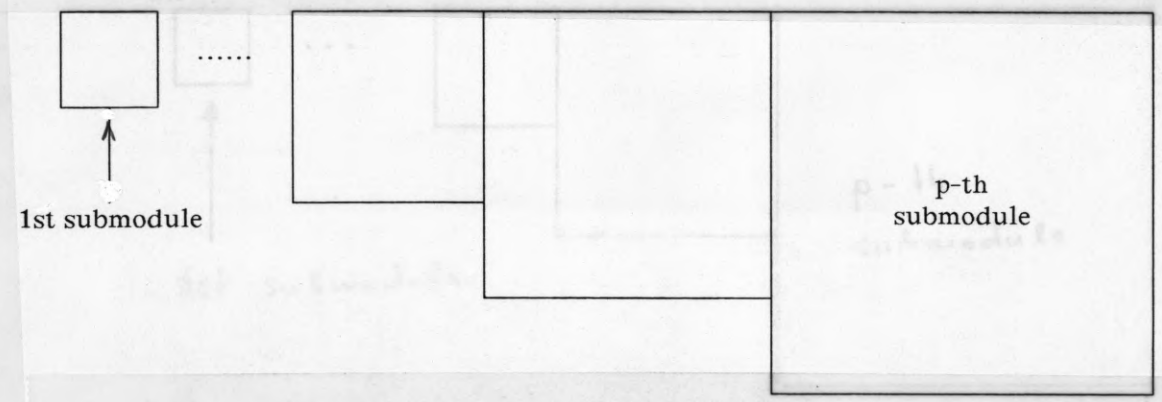

Fig. 2: The module structure of the slow "Newton approximator".

the p-th (last) submodule. Since we are aiming for an AT^2-optimal network with computation time $O(T)$, we must have

$$A_{3,p}T_{3,p}^2 = O(n^2)$$

and

$$T_{3,p} = T.$$

This condition enables us to specify the parameter p. Indeed, the speed of the submodules increases as we proceed upstreams (by decreasing submodule index), and each submodule must satisfy the condition that its multiplication time is at least logarithmic in the operand length. Since the operand length is halved in going from index i to index $i-1$ (due to the mechanism of the Newton approximation), and the most stringent condition occurs for $i = 1$, we have

$$\frac{T}{(\sqrt{2})^p} \leq \log(\frac{n}{2^p}),$$

which is certainly satisfied if we select p as follows:

$$p = 2\log\left(\frac{T}{\log n}\right) = 2\epsilon \log\log n. \tag{3}$$

Finally we turn our attention to the "fast approximator". This module receives an approximation of length $l_{1/\epsilon} = n/(\log n)^{1+\epsilon} \cdot 2$ and delivers an approximation of length $n/(\log n)^{2\epsilon}$. Thus, this module must execute $(1-\epsilon)\log\log n + 1$ iteration steps, each of them within time $\theta(\log n)$. The module essentially consists of a "fastest" multiplier of numbers of length $n/(\log n)^{2\epsilon}$, and can be realized with area A_2 such that $A_2(\log n)^2 = \Theta((n/2^p)^2)$, i.e. , $A_2 = \Theta((n/(\log n)^{1+2\epsilon})^2)$. Thus, the resulting AT^2-measure for this module is

$$A_2T^2 = \Theta((\frac{n}{(\log n)^{1+2\epsilon}}\log n \cdot (1-\epsilon)\log\log n)^2) = O(n^2)$$

and the optimality condition is clearly satisfied.

Since each of the three major units of our divider - the chain of modified BCH dividers, the fast Newton approximator and the slow Newton approximator - has area $O((n/(\log n)^{1+\epsilon})^2)$ and time $O((\log n)^{1+\epsilon})$, we conclude with the following result:

Theorem 1 For any fixed $1 \geq \epsilon > 0$, the n-bit inverse of an n-bit number can be calculated with optimal AT^2-performance for any $T \in [\Omega((\log n)^{1+\epsilon}), O((\log n)^2)]$.

5. Conclusion.

We constructed an AT^2-optimal divider with computation time $(\log n)^{1+\epsilon}$ for any $\epsilon > 0$. The reader may wonder whether one can choose ϵ as a decreasing function of n (tending to zero as n goes to infinity). This is indeed the case if the construction is slightly modified. In the construction as it is now we use a chain of modified BCH dividers each with the same area and speed. Thus both area and time grow as $1/\epsilon$ and hence AT^2 grows (at least) as $(1/\epsilon)^3$.

If ϵ is chosen as a function of n, then this simple chain of equally sized modules does not suffice. Rather one has to use a chain of increasingly larger (and slower) modules as we did for the Newton iteration. Omitting the tedious and not particular illuminating details we have:

Theorem 2. There is an AT^2-optimal divider for n-bit integers for any $T \in [\Omega(\log n \cdot 2^{(\log\log n)^{3/4}}), O((\log n)^2)]$.

Note that $2^{(\log\log n)^{3/4}} = O((\log n)^\epsilon)$ for any $\epsilon > 0$.

References

[1] K. Mehlhorn: "AT^2-optimal VLSI Integer Division and Integer Square Rooting", *Integration*, 2, 163-167, 1984.

[2] J. Reif: "Logarithmic Depth Circuits for Algebraic Functions", 24th FOCS, 138-145, 1983.

[3] P.W. Beame, S.A. Cook, H.J. Hoover: "Log Depth Circuits for Division and Related Problems", 24th FOCS, 1-6, 1984.

[4] W.K. Luk, J. Vuillemin: "Recursive Implementation of Optimal Time VLSI Integer Multipliers", VLSI 83, Trondheim, Norway, 1983.

[5] O. Spaniol: "Arithmetik in Rechenanlagen", Teubner Verlag, Stuttgart, 1976.

[6] K. Mehlhorn, F.P. Preparata: "AT^2-optimal VLSI Integer Multiplier with Minimum Computation Time", *Information and Control*, 58, 1-3, 137-156, 1983.

[7] D.E. Knuth: "The Art of Computer Programming", Vol. 2: Seminumerical Algorithms, Addison-Wesley, Reading, Mass. 1981, 2d ed.

[8] F.T. Leighton, personal communication, May 1985.