E0 309: Topics in Complexity Theory **Spring 2015**

Lecture 5: Feb 4, 2015

Lecturer: Neeraj Kayal *Scribe: Sumant Hegde*

5.1 Recap

In the last lecture we showed a constant-depth polynomial-size boolean circuit (of unbounded fanin) that computed the sum of two n-bit integers, implying the parallel time complexity of Integer Addition is constant. Then we wondered whether the following problems also have constant parallel time complexity: Iterated Integer Addition, Integer Multiplication and Iterated Integer Multiplication.

In the case of iterated addition, we observed that the least significant bit (LSB) of the sum of the n n-bit inputs is actually the parity of LSB's of the n inputs. But parity of n bits cannot be computed by any constant-depth boolean circuit:

Theorem 5.1 *Any Δ-depth circuit computing the parity of n bits using $\wedge, \vee, \neg$ gates (of unbounded fanin) must have size $2^{n^{1/\Delta}}$* [1].

(Proof not covered here.) Therefore, parallel time complexity of Iterated Addition is not constant.

5.2 Classes $\mathsf{AC}^0, \mathsf{TC}^0$

Definition 5.2 AC^0 *is the class of boolean functions computed by boolean circuits having $\vee, \wedge$ and $\neg$ gates, with constant depth and polynomial size, having unbounded fanin for $\wedge, \vee$.*

From the discussion above, integer addition $\in \mathsf{AC}^0$ and Iterated Integer Addition$\notin \mathsf{AC}^0$.

Theorem 5.3 *Integer Multiplication $\notin \mathsf{AC}^0$*

Proof: We show an AC^0 reduction from PARITY to Integer Multiplication.
Let the input be $x_1, \ldots, x_n$. We consider the binary string $x_1 \ldots x_n$. Between each x_i and x_{i+1} for $i \in [n-1]$, we insert $\log n$ many zeros to get another binary string y. y is of length $n + (n-1)\log n$. Let z be the binary string of length $|y|$ such that it has 1 at the positions corresponding to x_i's of y, and 0 everywhere else. An AC^0 circuit can easily form y, z. Treating y, z as integers, let w be their product with bits w_0(LSB),$w_1, w_2, \cdots$. From the naive algorithm it is clear that $w_{n-1+(n-1)\log n} = PARITY(x_1, \ldots, x_n, c_{n-1+(n-1)\log n})$, where c_i is the i^{th} carry bit. A closer inspection reveals that $c_{n-1+(n-1)\log n} = 0$: Columns $n-2+(n-1)\log n$ through $n-1+(n-2)\log n$ "absorb" all carries propagated from the lower columns. Thus $w_{n-1+(n-1)\log n} = PARITY(x_1, \ldots, x_n)$. ■
It follows that Iterated Integer Multiplication$\notin \mathsf{AC}^0$.

We now consider *threshold gates* (slightly more powerful than the basic gates) and see if they give us constant-depth polynomial-sized circuits for Iterated Integer Addition, Integer Multiplication and Iterated Integer Multiplication.

Definition 5.4 *For inputs $x_1, \dots, x_n \in \{0,1\}$, the output of a threshold gate is*

$$Th(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } a_1x_1 + \cdots + a_nx_n \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

where $\theta, a_1, \dots, a_n \in \mathbb{Z}$. $\theta, a_1, \dots, a_n$ may depend on n, but they do not depend on the input $x_1, \dots, x_n$.

Remarks
1. If we set $a_i = 1\ \forall i \in [n]$ and $\theta = 1$ then we get an $\vee$ gate.
2. If we set $a_i = 1\ \forall i \in [n]$ and $\theta = n$ then we get an $\wedge$ gate.
3. If we set $a_1 = -1$ and $\theta = 0$ then we get a $\neg$ gate.
4. If we set $a_i = 1\ \forall i \in [n]$ and $\theta = n/2$ then we get a MAJORITY gate. A MAJORITY gate outputs 1 if at least half the (boolean) inputs are 1 and outputs 0 otherwise.
5. One of the motivations to study threshold gates comes from Artificial Neural Networks. In a simplistic view, neurons behave like threshold gates. A neuron receives signals from several neurons, and fires if the weighted sum of inputs exceeds some threshold.

Definition 5.5 TC^0 *is the class of boolean functions computed by constant-depth poly(n)-size circuits with threshold gates.*

Observation From the remarks above it follows that $\mathsf{AC}^0 \subseteq \mathsf{TC}^0$. It turns out that the containment is strict, i.e., $\mathsf{AC}^0 \subsetneq \mathsf{TC}^0$, since MAJORITY $\in \mathsf{TC}^0$ (from Remark 4 above) and

Theorem 5.6 *MAJORITY* $\notin \mathsf{AC}^0$.

(Proof not covered here.)

5.3 Iterated Integer Addition $\in \mathsf{TC}^0$

Definition 5.7 (symmetric boolean functions) *A boolean function $f : \{0,1\}^n \to \{0,1\}$ is symmetric if for all permutations $\sigma : [n] \to [n]$, $f(x_{\sigma(1)}, x_{\sigma(2)}, \dots, x_{\sigma(n)}) = f(x_1, \dots, x_n)$.*

Theorem 5.8 *Every symmetric boolean function $f(x_1, \dots, x_n) \in \mathsf{TC}^0$.*

Proof: We observe that

$\exists S = \{b_1, \dots, b_m\} \subseteq [n]$ such that

$$f(x_1, \dots, x_n) = 1 \iff \sum_{i \in [n]} x_i \in S. \text{ Hence we can rephrase } f \text{ as}$$

$$f(x_1, \dots, x_n) = \bigvee_{j \in [m]} \Big(\sum_{i \in [n]} x_i = b_j \Big) \qquad \triangleright \text{It's a conditional } =.$$

$$= \bigvee_{j \in [m]} \Big(\Big(\sum_{i \in [n]} x_i \geq b_j \Big) \wedge \Big(\sum_{i \in [n]} (-x_i) \geq -b_j \Big) \Big)$$

The two inequalities in the last line can be implemented by two threshold gates, one with $a_i = 1\ \forall i \in [n]$, $\theta = b_j$ and the other with $a_i = -1\ \forall i \in [n]$, $\theta = -b_j$, respectively. Combining these two threshold gates with an $\wedge$ gate gives a subcircuit C_j of size 3 and depth 1. Making m such subcircuits, one for each b_j, and combining them in parallel using an $\vee$ gate results in a circuit of size $3m + 1$ and depth 2. Since $m \leq n$, it's a poly-size constant depth circuit. Finally we can turn $\wedge$ and $\vee$ into threshold gates as explained before and get a TC^0 circuit. ■

Since PARITY is symmetric, we have

Corollary 5.9 *PARITY* $\in \mathsf{TC}^0$.

Theorem 5.10 *Iterated Addition* $\in \mathsf{TC}^0$.

Proof:

Input: n integers $a_1, \dots, a_n$ of n bits each.
Output: $s = a_1 + \cdots + a_n$.
Let $c_0, c_1, \dots, c_n$ be the carries as shown:

$$\begin{array}{ccccccc}
c_n & c_{n-1} & c_{n-2} & \cdots & c_2 & c_1 & c_0 = 0 \\
 & a_{1(n-1)} & a_{1(n-2)} & \cdots & a_{12} & a_{11} & a_{10} \\
 & + \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
 & + a_{n(n-1)} & a_{n(n-2)} & \cdots & a_{n2} & a_{n1} & a_{n0}
\end{array}$$

Just like in the algorithm of addition of two integers, we would like to first compute all the carry bits in parallel and then compute parities in parallel. But there are two differences here. First, the carries $c_1, \dots, c_n$ are not single bits; they are small integers. So, our parity computation step now looks like $\forall 0 \le i \le n,\ s_i = \mathit{PARITY}(c_{i,0}, a_{1,i}, \dots, a_{n,i})$, where $c_{i,j}$ is the j^{th} LSB of c_i. The second difference is that for each output bit we have to compute parity of $n+1$ bits, not 3 bits. But this is doable in constant depth with threshold gates as we saw earlier. We now show that computation of carries also is possible in constant depth with threshold gates.

As an example we discuss computing $c_{n-1,0}$ and generalize it for all $c_{i,0}$'s later.

Let us first estimate the size of c_{n-1}. For this we consider $t = b_1 + b_2 + \cdots + b_n$, where each b_i is an integer obtained by retaining $n-1$ LSBs of a_i and dropping the rest (here $a_{i,n-1}$). The reasoning is that c_{n-1} depends only on $n-1$ LSBs of a_i's.
Since $b_i < 2^{n-1}$ $\forall i \in [n]$, clearly $t < n2^{n-1}$. And the number of bits in $t = \log t \le \log n + n - 1$. Dropping the $n-1$ LSBs of t we get c_{n-1} which has at most $\log n$ bits.
Thinking of binary representation of t ,

$$t = t_m \cdot 2^m + \cdots + t_{n-1} \cdot 2^{n-1} + \cdots + t_1 \cdot 2 + t_0,$$

the bits of c_{n-1} are $t_m, t_{m-1}, \dots, t_{n+1}, t_n$. We can safely assume c_{n-1} has exactly $\log n$ bits. Thus, m is fixed for a fixed n.
Now we show how we can compute these bits using small number of threshold gates. We observe that

$$t_m = 1 \iff \qquad 2^{m+1} - 1 \ge t \ge 2^m \tag{5.1}$$

$$\begin{aligned} t_{m-1} = 1 \iff \qquad & (2^m - 1 \ge t \ge 2^{m-1}) \\ & \vee (2^m + 2^m - 1 \ge t \ge 2^m + 2^{m-1}) \end{aligned} \tag{5.2}$$

$$\begin{aligned}
t_{m-2} = 1 \iff \quad & (2^{m-1} - 1 \geq t \geq 2^{m-2}) \\
& \vee(2^{m-1} + 2^{m-1} - 1 \geq t \geq 2^{m-1} + 2^{m-2}) \\
& \vee(2^{m} + 2^{m-1} - 1 \geq t \geq 2^{m} + 2^{m-2}) \\
& \vee(2^{m} + 2^{m-1} + 2^{m-1} - 1 \geq t \geq 2^{m} + 2^{m-1} + 2^{m-2}) \qquad (5.3) \\
\dots & \\
t_{n} = 1 \iff \quad & (2^{n} - 1 \geq t \geq 2^{n-1}) \vee \\
& \vdots \\
& \vee(2^{m} + \cdots + 2^{n+1} + 2^{n} + 2^{n} - 1 \geq t \geq 2^{m} + \cdots + 2^{n+1} + 2^{n} + 2^{n-1}) \qquad (5.4)
\end{aligned}$$

Clearly it is t_n that requires the maximum number of inequalities. Specifically, t_n requires $2 \cdot 2^{m-n} \leq 2 \cdot 2^{\log n} = O(n)$ inequalities. Also the bounds in the inequalites are fixed for a fixed n. This means LSB of c_{n-1} can be computed by $O(n)$ threshold gates, with depth 2 (taking into account the $\vee$ gate as well).

The above procedure can be generalized to compute any $c_{i,0}$ by defining b_i as the $i-1$ LSBs of a_i. One can verify that c_i has at most $\log n$ bits for all i. Thus we can compute all $c_{i,0}$'s in parallel using n subcircuits each of size $O(n)$ and depth 2.

On top of this layer, finally, for $0 \leq i \leq n$ we can introduce TC^0 parity circuits to compute $s_i = PARITY(c_{i,0}, a_{1,i}, \dots, a_{n,i})$. This makes total size $O(n^3)$ and depth 5. For $n+1 \leq i \leq m$, $s_i = t_i$ which too can be computed by inequalities like 5.1, 5.2, 5.3 in parallel with the main computation. Thus we have a constant depth polynomial size threshold circuit for the problem. ■

Remark The above TC^0 circuit is not the optmial one. TC^0 circuits with smaller size for iterated addition are known.

Corollary 5.11 *Integer Multiplication* $\in \mathsf{TC}^0$

Proof: The grade school algorithm for multiplying two n-bit integers first performs n^2 bitwise multiplications to get n n-bit integers and then performs iterated addition over them (with appropriate shifts). The first step is clearly in TC^0. The second step also is in TC^0 from Theorem 5.10. ■

5.4 Analogy between Integers and Polynomials

We informally discuss some similarities between integers and univariate polynomials as it will be helpful in the later discussions.
Consider the ring of integers $\mathbb{Z} = \{\{0, \pm 1, \pm 2 \dots\}, +, \times\}$ and the ring of polynomials $\mathbb{F}[X]$ where $\mathbb{F}$ is $\mathbb{R}, \mathbb{C}$, or $\mathbb{F}_p$ (where p is prime). We can compare the two on the following aspects:

Size and Degree While the size of an integer is the number of bits required to represent it, the degree of a polynomial determines the number of field elements required to represent it.

Prime numbers and Irreducible polynomials A prime number is an integer greater than 1 such that it is not a product of two integers greater than 1. Similarly, an irreducible polynomial over a field is a nonconstant polynomial such that it is not a product of two nonconstant polynomials in that field.

Unique factorization Any integer greater than 1 can be written as a unique product of prime numbers (ignoring the order). Similarly, any nonconstant polynomial over a field can be written as a unique product of irreducible polynomials (ignoring the order) in that field.

Remainder is unique Given two integers $a, b,\ b \neq 0$, there exist unique integers q, r such that $a = bq + r$ and $0 \leq r < |b|$. Similarly, given two polynomials $a, b,\ b \neq 0$ over a field, there exist two unique polynomials q, r such that $a = bq + r$ and $0 \leq deg(r) < deg(b)$.

5.5 Iterated Integer Multiplication $\in \mathsf{TC}^0$

Input: n-bit integers $a_1, \ldots, a_n$
Output: $P = a_1 \cdot a_2 \cdots a_n$
Since $a_i < 2^n\ \forall i \in [n]$, we have $P < 2^{n^2}$, i.e., P has at most n^2 bits.
Let us translate the problem to the world of polynomials and analyze it.

5.5.1 Iterated Polynomial Multiplication

Input: n polynomials of degree $(n-1)$ each:

$$\begin{aligned} a_1(x) =& a_{1,0} + a_{1,1}x + \cdots + a_{1,n-1}x^{n-1} \\ \cdots& \\ a_n(x) =& a_{n,0} + a_{n,1}x + \cdots + a_{n,n-1}x^{n-1} \end{aligned}$$

Output: $p(x) = a_1(x)a_2(x)\cdots a_n(x)$
The naive approach requires in total exponentially many (n^n) multiplications and thus does not give us a TC^0 circuit.
However, we can try the interpolation technique which we encountered before. For now we assume addition, subtraction and scalar multiplication are free. Here are the steps:
1. Choose n^2 points $\alpha_1, \ldots, \alpha_{n^2}$ from the field and get evaluation lists for a_i's:

$$\begin{aligned} a_1(x) \equiv& (a_1(\alpha_1), a_1(\alpha_2), \ldots, a_1(\alpha_{n^2})) \\ \cdots& \\ a_n(x) \equiv& (a_n(\alpha_1), a_n(\alpha_2), \ldots, a_n(\alpha_{n^2})) \end{aligned}$$

Each evaluation is essentially an addition, so if we allow addition gates with unbounded fanin then this step costs depth 1.
2. Do pointwise multiplication to get

$$p(x) \equiv (\prod_{i\in[n]} a_i(\alpha_1), \prod_{i\in[n]} a_i(\alpha_2), \ldots, \prod_{i\in[n]} a_i(\alpha_{n^2}))$$

If we allow multiplication gates with unbounded fanin then this step costs depth 1.
3. Interpolate p to get the coefficients $p_0, \ldots, p_{n^2-n}$. Again, allowing addition gates with unbounded fanin, this step costs depth 1.
Thus we have a depth 3 circuit with addition gates and multiplication gates of unbounded fanin, for the problem of iterated multiplication of polynomials.

5.5.2 Chinese Remaindering Theorem

The counterpart of the above technique in the integer world is really Chinese Remainder Theorem (CRT) which we describe now. Henceforth we assume that $\forall i \in [n]\ a_i \neq 0$, since otherwise the product is trivially 0.

Step 1. Pick a few distinct (and small) prime numbers $\alpha_1, \ldots, \alpha_m$ such that their product exceeds 2^{n^2}, the upper bound of $\prod_{i\in[n]} a_i = P$(say). Let's say we pick the first n^2 prime numbers.

Fact 5.12 *For each a_i, the tuple $(a_i \mod \alpha_1, \ldots, a_i \mod \alpha_m)$ uniquely determines a_i.*

Compute the tuples for all a_i's.

Step 2. For each $\alpha_j,\ j \in [m]$, let $b'_j := \prod_{i\in[n]} a_i \mod \alpha_j$. Compute $b_j := b'_j \mod \alpha_j$.

Fact 5.13 *The tuple $(b_1 \mod \alpha_1, \ldots, b_m \mod \alpha_m)$ uniquely determines P.*

Step 3. For all $j \in [m]$ let $Q_j := \prod_{t\in[m], t\neq j} \alpha_t$. For all $j \in [m]$, let $R_j \in [1, \alpha_j - 1]$ such that $Q_j R_j \equiv 1 \mod \alpha_j$. Call R_j the inverse of Q_j.

Fact 5.14 *For each Q_j there exists exactly one inverse R_j.*

Let $p' := \sum_{j\in[m]} b_j Q_j R_j$. Compute $p' \mod (\alpha_1\alpha_2\cdots\alpha_m)$, which is P.

Back to analogy. Steps 1,2 and 3 of CRT are the analogues of evaluation, pointwise multiplication and interpolation respectively. Let's elaborate the analogy for Step 1.
Evaluation of a polynomial $p(x)$ at a point α is equivalently the remainder obtained by dividing $p(x)$ by $(x-\alpha)$. i.e., $p(x) \equiv p(\alpha) \mod (x-\alpha)$. (For example, suppose $p(x) = x^2+1$ and $\alpha = 1$. Then $p(1) = 2$. Also, since $x^2+1 = (x-1)(x+1)+2$, we have $p(x) \equiv 2 \mod (x-1)$.) Noticing that $x-\alpha$ is irreducible, the above operation naturally translates to taking modulo a prime number in the integer world.

5.5.3 Discrete Logarithms

We need one more trick before designing the circuit. Step 2 of CRT still involves iterated multiplications of numbers, albeit modulo small primes. The trick is to reduce this problem into iterated additions of their discrete logarithms as follows.
For a prime number α, the set of positive integers modulo α forms a cyclic group under multipication modulo α. Thus there exists an element $2 \leq g \leq \alpha - 2$ such that for every $1 \leq a \leq \alpha - 1$ there exists $0 \leq e \leq \alpha - 2$ such that $g^e \equiv a \mod \alpha$. (g is called a generator.) Call e the discrete logarithm of a (to the base g). Now, given $1 \leq a_1, \ldots, a_n \leq \alpha - 1$, their product b modulo α is

$$\begin{aligned} b = a_1 \cdots a_n \mod \alpha &= g^{e_1} \cdots g^{e_n} \mod \alpha \qquad\qquad \text{where } g^{e_i} \equiv a_i \mod \alpha, \forall i \in [n] \\ &= g^{e_1 + \cdots + e_n} \mod \alpha. \end{aligned}$$

Thus the problem is reduced to iterated addition $e_1 + \cdots + e_n$ (at the added overhead of logarithm computation and exponentiation though).

5.5.4 The circuit

We are now ready to detail the TC^0 circuit for iterated integer mutiplication. Broadly, the levels in the circuit can be partitioned into 3 layers, one for each step of CRT. For simplicity, when explaining layers 1 and 2 (steps 1 and 2 of CRT) we pretend there is only one prime number α to work with, instead of n^2-many. The idea is that the circuitry described for α can be replicated $m = n^2$ times in parallel, one for each α_i. This replication causes only polynomial blowup in size and no change in depth.

Layer 1: Task: Given a_i, compute $a_i \mod \alpha$. Let $a_{i,k}$ denote, as always, k'th LSB of a_i. The solution starts with having the following values precomputed: $2^{n-1} \mod \alpha, 2^{n-2} \mod \alpha, \ldots, 2^1 \mod \alpha, 2^0 \mod \alpha$. Then, computing the sum

$$A_i = a_{i,n-1} \cdot 2^{n-1} \mod \alpha + a_{i,n-2} \cdot 2^{n-2} \mod \alpha + \cdots + a_{i,1} \cdot 2 \mod \alpha + a_{i,0} \cdot 1 \mod \alpha$$

costs a standard TC^0 circuit for the iterated addition. Call this circuit ϕ. Crucially, ϕ gives us A_i that is much smaller than a_i and has the property $A_i \equiv a_i \mod \alpha$. Specifically, $A_i \leq n\alpha$, a polynomial upper bound, which lets us take the "exhaustive approach" to find $A_i \mod \alpha$ ($= a_i \mod \alpha$), as follows.

We can have with us the following values precomputed: $\alpha, 2\alpha, \ldots, n\alpha$. We'll have small TC^0 circuits $C_1, \ldots, C_n$ in parallel (on top of ϕ) such that C_l computes $A_i - l\alpha$. Exactly one of them outputs a value in $[0, \alpha - 1]$. And that is our $a_i \mod \alpha$. (There is another level above C_l's to do this range-checking.) Throughout, we have used polynomially many threshold gates and incurred constant depth.

Layer 2: Task: Implement Discrete Logarithms described in section 5.5.3. Input is of course $a_1 \mod \alpha, a_2 \mod \alpha, \ldots, a_n \mod \alpha$. We assume $a_i \neq 0 \ \forall i \in [n]$, since otherwise the product is trivially 0.

We will have the following values precomputed: g (a generator), $g^2 \mod \alpha, g^3 \mod \alpha, \ldots, g^{\alpha-1} \mod \alpha$. (Precomputation is possible because g depends only on α which is fixed.) We'll have small TC^0 circuits $D_1, \ldots, D_{\alpha-1}$ in parallel such that D_l computes $(a_i \mod \alpha) - g^l \mod \alpha$. Exactly one of them, say $D_{l'}$, outputs 0. l' is our e_i, i.e., $g^{e_i} \equiv a_i \mod \alpha$. (There is one more level above D_l's to "select" l' and present it as e_i to the upper level.)
The depth of layer 2 so far is a constant. The size of layer 2 so far is roughly $O(poly(\alpha))$. In the worst case α is n^2th prime number, which is less than n^3. Thus the size of layer 2 so far is $O(poly(n))$.

We now have $e_1, \ldots, e_n$ as outputs. We simply place a standard iterated addition TC^0 circuit over them to compute $s = \sum\limits_{i \in [n]} e_i$.
The last step is to compute $g^s \mod \alpha$ (which equals $b \mod \alpha$). Let $s' = s \mod (\alpha - 1)$. Notice that $g^{s'} \mod \alpha = g^s \mod \alpha$, since $g^{\alpha-1} \equiv g^0 \equiv 1 \mod \alpha$. Thus it suffices to compute $g^{s'} \mod \alpha$.

We need to find s' from s. For this we observe that $s \leq n(\alpha - 1)$, and hence decide to have the following multiples of $\alpha - 1$ precomputed: $\alpha - 1, 2(\alpha - 1), \ldots, n(\alpha - 1)$. Now, very similar to C_l's mentioned before, there will be a setup of n-many TC^0 circuits that can find the right s' using the precomputed multiples of $\alpha - 1$.

Now that we know s', computing $g^{s'} \mod \alpha$ is a matter of merely selecting the right element from the lot $\{g^0, g^1, g^2, \ldots, g^{\alpha-2} \mod \alpha\}$, all of which are precomputed. A bunch of $poly(\alpha)$ many threshold gates wired in parallel can do this. Thus we have computed $g^{s'} \mod \alpha$ which is $b \mod \alpha$.
One can verify that layer 2 in total costs polynomial size and constant depth. This is true even after taking into account the fact that the above procedure has to be replicated (in parallel) for all n^2 prime numbers.

Layer 3: Input: $b_1 \mod \alpha_1, \ldots, b_m \mod \alpha_m$. We borrow the notations from (step 3 of) section 5.5.2. α_j are fixed. So, for all $j \in [m]$, product $Q_j R_j$ can be precomputed. Further, we can have the following multiples of Q_jR_j precomputed: $Q_jR_j, 2Q_jR_j, \dots, (\alpha_j - 1)Q_jR_j$. Then computing $b_jQ_jR_j$ amounts to selecting the appropriate multiple which can be done using a TC^0 circuit. As usual we have m such circuits in parallel one for each j. On top of them we have an iterated addition circuit to compute $p' := \sum_{j\in[m]} b_jQ_jR_j$. It remains to compute $P = p' \mod (\prod_{j\in[m]} \alpha_j)$. This task is essentially the same as what we did with α in layer 1, hence we replicate that here. Of course, this time the divisor is larger: $\prod_{j\in[m]} \alpha_j \leq \alpha_m^m \leq m^{2m} = (n^2)^{2n^2} < 2^{n^3}$ implying that it has at most n^3 bits. Accordingly we have to make a few changes, like precomputing the multiples up to the factor of n^3 instead of $n-1$ or so, etc.

One can verify that for layer 3, and thus altogether, the size is polynomial in n and the depth is constant.■

5.6 References

[1] M. Furst, J. B. Saxe, M. Sipser, Parity, circuits, and the polynomial-time hierarchy. Mathematical Systems Theory, 17(1):13–27, Apr. 1984

[2] Neil Immerman,Susan Landau, The complexity of iterated multiplication, Information and Computation 116(1) (1995), 103-116,

[3] Paul W. Beame,Stephen A. Cook,H. James Hoover, Log Depth Circuits for Division and Related Problems, SIAM(1986) Vol. 15 No. 4.

[4] John Reif, On Threshold Circuits and Polynomial Computation, Second Annual Structure in Complexity Theory Symp(1987), 118-123

[5] David Cox,John Little,Donal O'Shea, Ideals, Varieties and Algorithms, Second Edition, Springer (1996)