CS277: Concrete Complexity	Fall 1996
Lecture 3: September 3, 1996	
Lecturer: Uri Zwick	*Scribe: Anupam Gupta*

3.1 Turing Machines and Boolean Circuits

3.1.1 Boolean Circuits that simulate TMs

Theorem 3.1 (Schnorr, Pippenger) *If $L \subseteq \{0,1\}^*$ can be recognized using a Turing Machine which runs in $O(T(n))$ time, then the corresponding sequence $\{f_n\}$ of Boolean functions has the property that the n^{th} function can be realized using a circuit of size $O(T(n)\log T(n))$.*

We shall not prove this theorem, but instead shall prove a weaker bound of $O(T(n)^2)$ — actually of $O(T(n)S(n))$, where $S(n)$ is the space complexity of the TM. This will also give us an idea of where we can reduce the number of gates required, and hence give us an idea of the proof of theorem 3.1.

A point to be noticed here is that a lower bound for Boolean circuits to recognize any language of $\omega(n \log n)$ will imply a super-linear lower bound for recognizing the language on a Turing Machine. This will be better than the lower bounds we currently have for languages in $\mathcal{NP}$.

Theorem 3.2 *If $L \subseteq \{0,1\}^*$ can be recognized using a TM whose running time is bounded above by $T(n)$, then the corresponding sequence $\{f_n\}$ of Boolean functions has the property that the n^{th} function can be ralized using circuit of size $O(T(n)^2)$.*

Proof: Let the Turing Machine be a one-sided tape TM $M = (Q, q_o, Q_f, \Sigma, \delta)$, where

- Q is the set of states,
- q_o is the initial state,
- $Q_f \subseteq Q$ is the set of accepting states,
- Σ is the finite alphabet (we assume the input alphabet and the tape alphabet to be the same) and
- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$ be the transition function.

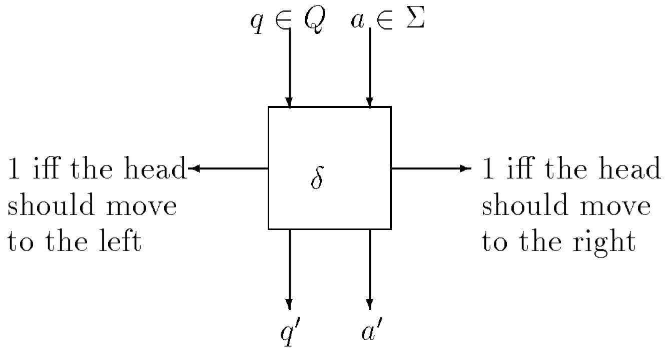

Figure 3.1: Inputs and outputs of the δ circuit

This transition function can be easily implemented as a constant size Boolean circuit, as in figure 3.1. This takes in input the Boolean encodings of the initial state q and symbol a, and outputs the new state q', output symbol a', and also the L or R bit is set depending on whether the head should move left or right.

Using this as the building block, we can start constructing the Boolean circuit for simulating the action of the Turing Machine. For this, consider a table of size $T(n) \times T(n)$. The i^{th} row of the this table represents the state of the Turing machine at the i^{th} time instant. Each cell of this table has the information given in 3.2.

For the first row : The *data* parts of the first n cells in the first row are filled in with the input x, one cell holding the encoding of one character of the input. The rest being filled in with the encoding of the blank character. The *head-present* bit of the first cell is set (we can assume that the TM starts from the leftmost end of the tape). The *state* of the first cell is set to q_o, the rest are arbitrarily set, they having no consequence as the *head-present* bit is not set.

Computing the values of the cells in subsequent rows : To compute the values of subsequent rows, it can be seem that the values of a cell in the i^{th} row depend on the three cells directly above it, and that it can be computed by a constant sized circuit δ'. This is a slight augmentation of the circuit computing δ, simply looking at whether the *head-present* bit of the cell directly above is set: if it is, then the *data* value is set to be the data output of the δ circuit above it, else the *data* is simply copied from the cell above (if the head is not in a cell of the tape, then the contents of the cell cannot change). It also checks whether the head has moved into this cell from an adjacent one (by examining the head movement outputs of the adjacent δ-circuits); in which case, it sets its state from the output of the corresponding adjacent δ-circuit and the *head-present* bit to 1. Otherwise the state is set to an arbitrary value and the *head-present* bit to 0.

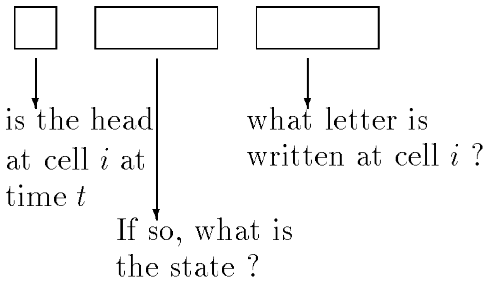

Figure 3.2: Information stored in a cell of the table

We also assume, *wlog*, that if the TM reaches an accepting state, it does not stop but just moves the head to and fro while staying in an accepting state.

The Acceptance Criterion : If the *state* of the cell with the head is an accepting state, the circuit outputs 1 in case of a match, 0 otherwise. ■

It can be easily seen that the total size of the circuit will be $O(T(n)^2)$. Furthermore, the depth is $O(T(n))$, which is the best we can hope for. A careful look shows us that the gates below the cells which do not have the *head-present* bit set are totally wasted and can be removed if we can fix the position of the head at any time instant regardless of the input. Machines for which this can be done are called *oblivious TMs*.

Theorem 3.3 *Any Turing machine whose running time is $O(T(n))$ can be simulated using an oblivious TM which runs in time $O(T(n) \log T(n))$.*

Using the above result, we can cut down on the useless gates and have a proof of theorem 3.1.

3.1.2 TMs that "take advice"

A *TM that takes advice* has the property that for every input **size**, the input to the TM is the input word x and an "advice" string $w_{|x|}$. The important fact is that the advice depends only on the length of the input (dependence on the input itself would trivialize the problem, because we could just give as advice one bit indicating the presence or absence of the string in the language).

P/poly is the class of languages which can be recognized by TMs that take advice in time polynomial in the length of the original input x (note that this allows the TM to inspect a section of the advice string of a polynomial length only).

Theorem 3.4 *A language L is in P/poly if and only if L can be recognized by a family of circuits* $\{f_1, f_2, \ldots\}$ *such that* $size(f_n)$ *is polynomial in* n.

Proof: ($\Rightarrow$) If L is in *P/poly* then there exists a poly-time TM that takes advice. For any polytime TM, there is a family of poly-sized circuits (Theorem 3.1). In these circuits, we can hard-wire the inputs corresponding to the advice, as the advice depends only on the length of the input, giving us a required family of circuits.

($\Leftarrow$) The poly-sized circuit recognizing the inputs of size n can be encoded to form the advice string for inputs of length n. The TM can then simply evaluate the value of the circuit on the input string x in poly-time. This gives us that L is in *P/poly*. ■

Thus, whenever we attempt to prove that a probem like the Hamiltonan Path problem requires circuits of super-polynomial size, we are not just attempting to prove that $P \neq NP$, but the stronger statement that $P/poly \neq NP$.

3.2 The Complexity of Basic Arithmetic Operations

3.2.1 Addition

We want, given two n-bit numbers in binary encoding, to output the $n+1$-bit sum of the two numbers.

We shall try several methods to add them, attempting to reduce the size and depth of circuits required for this operation. The basic unit used here is the Full Adder (FA), which takes 3 bits as input, and outputs the 2 bit sum of the 3 bits in binary. It can be realized using the circuit given in 3.3 using 5 gates.

3.2.1.1 Ripple-carry Adders

This is the most basic method for addition. For each two bits x_i and y_i of the input and the carry-in c_{i-1} from the provious position, the sum z_i and the carry-out c_i are computed using a FA. The carry-in to the rightmost bit is 0, and hence this FA can be replaced by a Half Adder (which takes 2 bits and outputs their sum), allowing us to save on 3 gates (it requires only one XOR and one AND gate). This gives us the size of the construction to be $5n-3$, which has been proved to be the optimal. However the depth is an apalling $2n-1$.

We now attempt to reduce the depth to $O(\log n)$, which is optimal. To do this, notice that the bottleneck is in the calculations of the carries, and computing the carries $c_0, c_1, \ldots c_n$ in $O(\log n)$ will allow us to get the required depth.

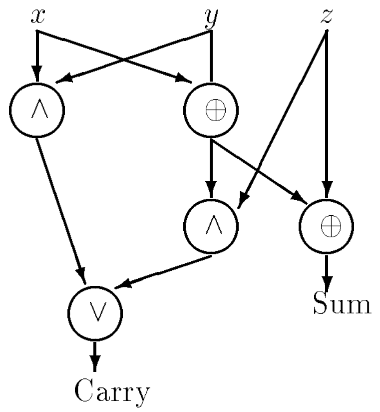

Figure 3.3: Full Adder implementation

3.2.1.2 Carry look-ahead Adders

At every position during the addition, the carry is either *generated* (if $x_i = y_i = 1$), *propagated* ($x_i \oplus y_i = 1$) or *cleared* ($x_i = y_i = 0$). Thus define

$$g_i = (x_i \wedge y_i) \quad , \quad p_i = (x_i \vee y_i)$$

It is then easy to see that the carry c_i is set iff there is a position $j \leq i$ where a carry is generated, and then propagated all the way through, i.e.

$$c_i = \bigvee_{j \leq i} (g_j \wedge p_{j+1} \wedge \ldots \wedge p_i)$$

This is a conjunction of at most n terms, each term having at most n disjunctions, and thus can be computed in depth $O(\log n)$. However, the size of each of these circuits is $O(n)$, and thus the circuit size jumps to $O(n^2)$.

3.2.1.3 Smaller and better things

We use the method of computing *Parallel Prefix Sums* to get a better circuit for addition with depth $O(\log n)$ and size $O(n)$.

The Prefix Sums problem is defined as follows: Let $\circ : D \times D \rightarrow D$ be an associative operator. We want to build an $\{\circ\}$-circuit with inputs $x_0, x_1, \ldots x_{n-1} \in D$ which outputs $x_0,\ x_0 \circ x_1,\ \ldots\ , x_0 \circ x_1 \circ \ldots \circ x_{n-1}$.

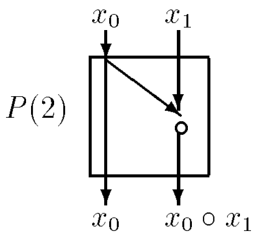

Figure 3.4: Construction of a circuit for $P(2)$

In our problem, $D = \{c, p, g\}$, the inputs are $f_i \in D$ where

$$f_i = \begin{cases} c & \text{if } x_i = y_i = 0 \\ p & \text{if } x_i \oplus y_i = 1 \\ g & \text{if } x_i = y_i = 1 \end{cases}$$

and the operation is given by the following table :

$\circ$	c	p	g
c	c	c	c
p	c	p	g
g	g	g	g

The carries can be then calculated by the rule :

$$c_i = 1 \textit{ iff } f_i \circ f_{i-1} \circ \ldots \circ f_o = g$$

Since $\circ$ can be implemented using a small constant sized circuit, f_i can be easily computed, all the c_i can be computed using a circuit of size O(size of circuit computing prefix sums).

To compute prefix sums with a circuit $P(n)$ of size $O(n)$ and depth $O(\log n)$, let us assume that $n = 2^k$. $P(2)$ is constructed as shown in figure 3.4. For $P(n)$, we use $P(\frac{n}{2})$ recursively (as in figure 3.5). The size and depth of this construction is given by

$$\begin{aligned} C(n) &= C(\frac{n}{2}) + (n-1) \\ C(2) &= 1 \end{aligned}$$

$$\begin{aligned} D(n) &= D(\frac{n}{2}) + 2 \\ D(2) &= 1 \end{aligned}$$

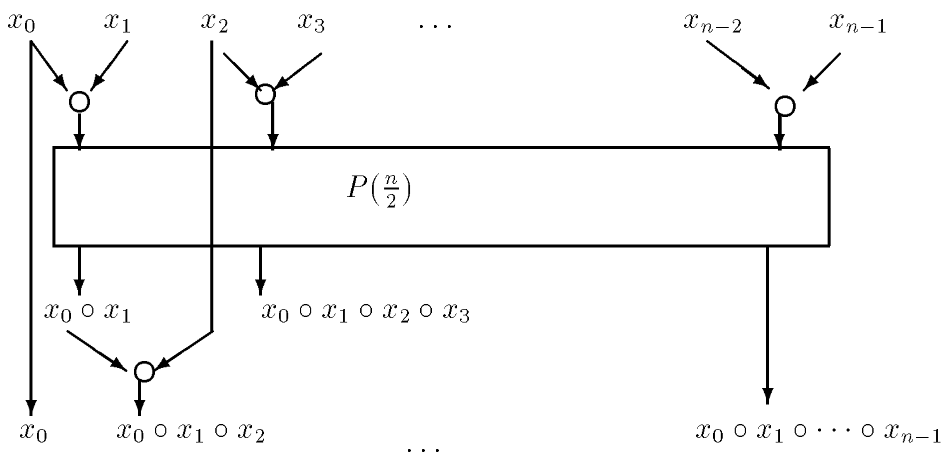

Figure 3.5: The recursive construction of $P(n)$

which on solving, give us

$$
\begin{aligned}
C(n) &= 2n - (k+2) \quad \text{and} \\
D(n) &= 2k - 1
\end{aligned}
$$

CS277: Concrete Complexity **Fall 1996**

Lecture 4: September 5, 1996

Lecturer: Uri Zwick *Scribe: Anupam Gupta*

4.1 The Complexity of Basic Arithmetic Operations

4.1.1 Circuits for Addition (contd.)

We can actually improve the construction given in the last lecture to get a depth of *exactly* $\log n$ at the expense of a small increase in size using a construction of Ladner and Fischer. For this, we look at the divide-and-conquer method to compute prefix sums (which would give us a depth of $O(\log n)$, but a bad size of $O(n \log n)$). Note that the output $x_0 \circ \ldots x_{\frac{n}{2}-1}$ is critical, because it is required at the $\log n - 1^{th}$ time instant, as opposed to the others, which can come one time unit later.

So we compose two different circuits to get the construction -

- A circuit $P_0(n)$ of depth exactly $\log n$, and
- A circuit $P_1(n)$ of depth $\log n + 1$, but such that the last output $x_0 \circ x_1 \circ \ldots \circ x_{n-1}$ is output at depth $\log n$.

To compose them, we proceed thus: $P_0(2) = P_1(2) = P(2)$, and given $P_0(\frac{n}{2})$ and $P_1(\frac{n}{2})$, proceed as in figure 4.1 (to construct $P_0(n)$) and figure 4.2 (to construct $P_1(n)$). To calculate the size,

$$\begin{aligned} C_0(n) &= C_0(\frac{n}{2}) + C_1(\frac{n}{2}) + (n-1) \\ C_1(n) &= C_0(\frac{n}{2}) + (n-1) \\ C_0(2) &= C_1(2) = 1 \end{aligned}$$

which on solving, gives us

$$\begin{aligned} C(n) &= 4n - F_{\log n + 5} + 1 = 4n - o(n) \text{ and} \\ C(n) &= 3n - F_{\log n + 4} = 3n - o(n) \end{aligned}$$

This trend of circuit size falling with a constant increase in the depth can help us to have circuits of size very close to $2n$ with depth $\log n + \gamma$ for some constant γ.

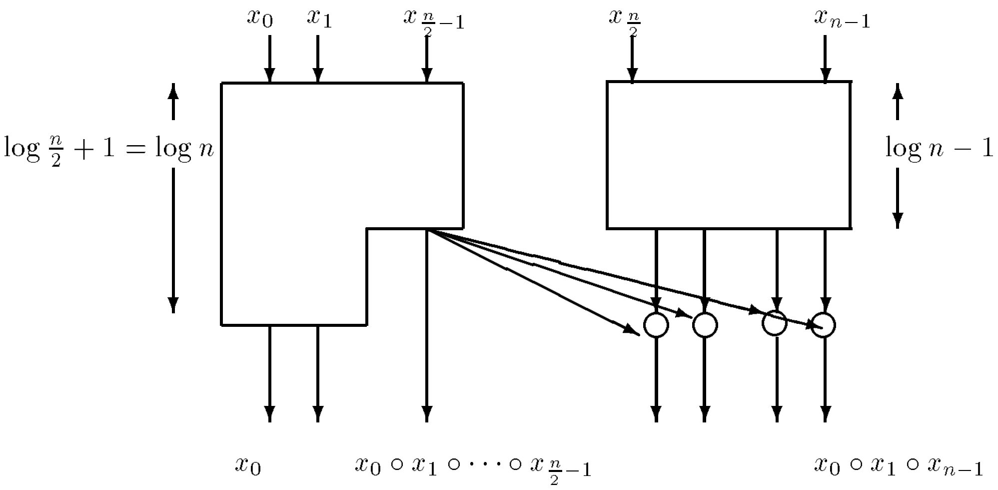

Figure 4.1: Recursive construction of $P_0(n)$

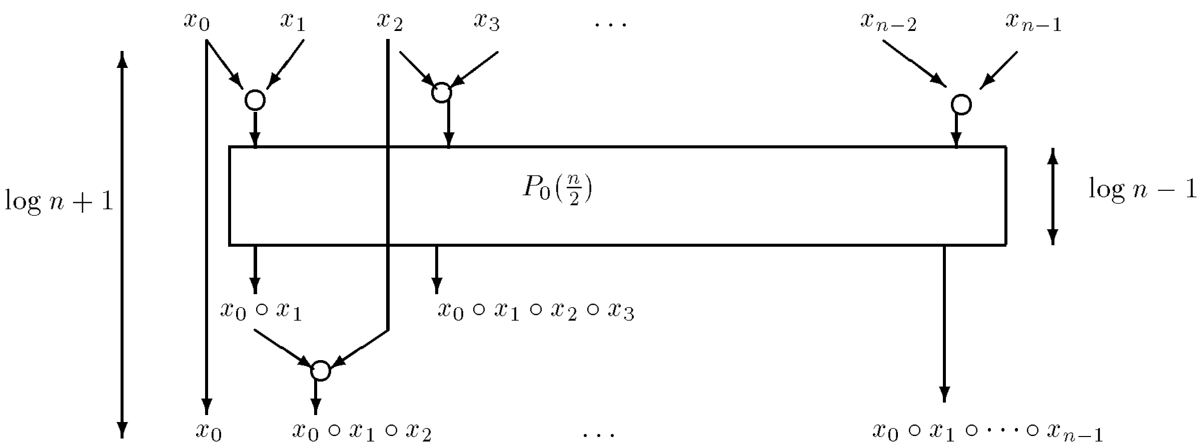

Figure 4.2: Recursive construction of $P_1(n)$

$$
\begin{array}{r}
x_{n-1} \ldots x_1\, x_0 \\
\times \quad y_{n-1} \ldots y_1\, y_0 \\
\hline
x_{n-1}y_0 \ldots x_1y_0\; x_0y_0 \\
x_{n-1}y_1 \ldots x_1y_1\; x_0y_1 \qquad\quad \\
\vdots \qquad\qquad \\
x_{n-1}y_{n-1} \ldots x_1y_{n-1}\; x_0y_{n-1} \qquad\qquad\qquad \\
\hline
z_{2n-1}\, z_{2n-2} \qquad\qquad \ldots \qquad\qquad z_0
\end{array}
$$

Figure 4.3: Naive multiplication

4.1.2 Multiplication

The naive method of multiplying two n-bit numbers requires us to add together n sub-products (as in figure 4.3) which would require circuits of size $O(n^2)$ and depth $O(\log^2 n)$ (which could be achieved by adding them pairwise in a binary-tree fashion).

4.1.2.1 Reducing the size

The trick to reduce the size is similar in spirit to Strassen's matrix multiplication trick, and is due to Ofman and Karatsuba. In this, we divide the given numbers into two parts each (as in figure 4.4), and compute the products recursively. But doing this naively again would be disastrous, as we would get the relation

$$
\begin{aligned}
C(n) &= 4C(\frac{n}{2}) + O(n) \quad \text{which would give} \\
C(n) &= O(n^2)
\end{aligned}
$$

But we notice that we only need the term $x_0y_1 + y_0x_1$, which allows us to do the following : multiply $z_1 = x_0y_0, z_2 = x_1y_1$ and $z_3 = (x_0 + x_1)(y_0 + y_1)$, and then the term $x_0y_1 + y_0x_1 = z_3 - z_2 - z_1$. This gives us :

$$
\begin{aligned}
C(n) &= 3C(\frac{n}{2}) + O(n) \quad \text{which would give} \\
C(n) &= O(n^{\log_2 3}) = O(n^{1.59..})
\end{aligned}
$$

This also suggests a method to multiply 2 complex numbers together using three multiplications, thus saving on one multiplication.

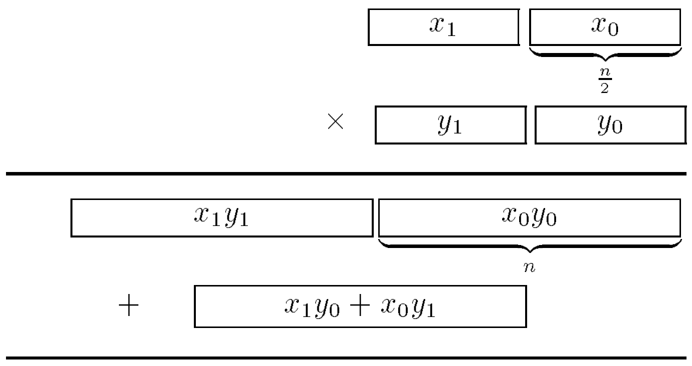

Figure 4.4: An improved multiplication procedure

We could continue in this vein and divide the numbers into three parts, and we would see that we do not need to perform all nine multiplications. Splitting into more and more parts leads to the Fast Fourier Transform (FFT), which is the best known method for multiplication.

Theorem 4.5 (Schönhage, Strassen) *Two n-bit numbers can be multiplied by a Boolean circuit of size $O(n \log n \log \log n)$.*

To obtain FFT, which involves complex numbers, with (finite precision) circuits, we can either use enough bits to ensure that rounding-off gives the correct answer (a linear number of bits suffice), or use finite rings.

4.1.2.2 Sinking to lower depths

To reduce the depth of the circuits, we look at the naive method of multiplying two numbers, and note that we need a low depth circuit to add n $2n$-bit numbers together. Two n-bit number require a circuit of depth at least $\log n$ to find their sum. However, if we want to add three numbers x, y and z to get two numbers u and v such that $x + y + z = u + v$, we can do this by a constant depth circuit. For this, corresponding bits x_i, y_i and z_i are added using a full adder, and results are placed thus: $u_i = sum_i$ and $v_{i+1} = carry_i$ (see figure 4.5. This is called *Carry-Save Addition*.

An obvious question is: where does the enhanced power come from ? It is from the fact that we are representing the numbers in a format which has redundancy (each number has many representations), which gives us more maneuverability while performing additions (but not so while performing comparisons). Thus we can now add the numbers three by three, each addition requiring a constant depth, until we have two numbers left, which can be added by a conventional adding circuit.

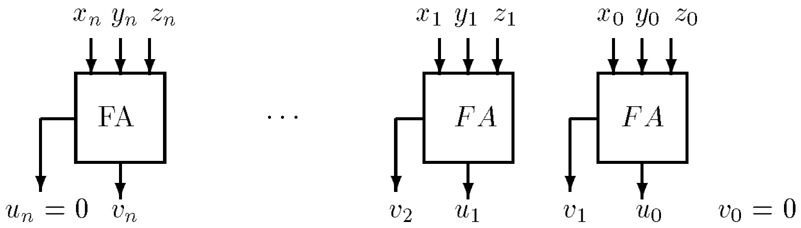

Figure 4.5: Carry-save multiplication

The depth of this circuit is ($\log_{\frac{3}{2}} n \times d_{csa}$) (for adding n numbers using the carry-save adders, which have depth d_{csa}) + $\log n$ (for adding the last 2 numbers), giving us multiplication circuits of depth $O(\log n)$ and size $O(n^2)$. In fact, this same trick can be used in the FFT to get a circuit of size $O(n \log n \log\log n)$ and depth $O(\log n)$.

4.1.3 Division

Division is equivalent to the problem of inversion (given an n-bit number x, to compute, say, the first $2n$ bits of $\frac{1}{x}$). Hence, we shall henceforth study the problem of how to invert efficiently.

For inversion, we use *Newton-Raphson's method* of finding roots to get the solution. We would want to find the root of the equation

$$f(y) = \frac{1}{y} - x$$

Let y_0 be the initial estimate, then the next estimates would be given by

$$\begin{aligned} y_{n+1} &= y_n - \frac{f(y_n)}{f'(y_n)} \\ &= y_n + (\frac{1}{y_n} - x)y_n^2 \\ &= 2y_n - x \cdot y_n^2 \\ &= y_n(2 - x \cdot y_n) \end{aligned}$$

We can, w.l.o.g., assume that $\frac{1}{2} \leq x < 1$ (if it is not, we can design a simple circuit that on input x, outputs k such that $x' = x/2^k$ would satisfy the constraint above, and a simple circuit can be used to convert x to x'). Let the error after the n^{th} iteration be ϵ_n. Then,

$$\frac{1}{x} - y_{n+1} = \frac{1}{x} - y_n(2 - x.y_n)$$

$$
\begin{aligned}
&= x\left(\frac{1}{x} - y_n\right)^2 \\
\Rightarrow \epsilon_{n+1} &= x \cdot \epsilon_n^2 < \epsilon_n^2
\end{aligned}
$$

If we choose $y_0 = \frac{3}{2}$, then the error $\epsilon_0 \leq \frac{1}{2}$. Also, with each iteration, the number of digits of precision doubles, implying that $O(\log n)$ iterations would be enough. This method gives division circuits of depth $O(\log^2 n)$. In fact, we can improve this to get

Theorem 4.6 *If there exist multiplication circuits of size $M(n)$, then there are division circuits of size $O(M(n))$.*

This can be done by using the fact that the number of digits of precision at the early stages of the algorithm are few, and smaller multiplication circuits would suffice.