COMPUTING FIBONACCI NUMBERS WITH GATES

CHRIS KALTWASSER

August 12, 1994

ABSTRACT. We consider computing Fibonacci numbers in base 2 using binary logic circuits. Such a circuit consists of a number of simpler components called *gates*. We consider only circuits that use two-input gates, which take the number k as input in binary representation, and produce f_k as output, again in binary, for any k, $0 \leq k < n$. Let $\lambda = \frac{1+\sqrt{5}}{2}$, $\gamma = \log_2 \lambda$. The number of gates required to compute $f_0, \ldots, f_n$ is shown to be bounded below by $n \log_2 \lambda - \frac{1}{2} \log 5$ and above by $\gamma n^2 + n \log n - (\gamma + 1)n$. Working solutions for $f_0, \ldots, f_3$, $f_0, \ldots, f_7$, and $f_0, \ldots, f_{15}$ are considered. From these solutions we conjecture that the required number of gates might be no better than $O(n \log n \log \log n)$.

1. INTRODUCTION

We will be considering the problem of computing Fibonacci numbers for positive n in base 2 using a logic circuit. Previous approaches to the problem of the computation of these numbers has mostly concentrated on the arithmetic model of computation. By considering the digital model of computation we hoped to shed some new light on the problem. The problem of computing the nth Fibonacci number is interesting because the Fibonacci sequence is generated by a very simple difference equation, making it useful for investigating general methods for computing solutions of difference equations.

1.1 PREVIOUS RESULTS

In their paper [Cull 1989], Cull and Holloway consider several algorithms for computing the f_n, comparing the time in bit operations each requires. By repeated addition, the nth Fibonacci number may be found using $\frac{1}{2}\gamma n(n-1)$ bit operations, whereas the best algorithm they examined required only $\frac{3}{2}\gamma n \log \gamma n \log\log \gamma n$ bit operations. At this point, the computational complexity of generating f_n has been reduced to a number of multiplications, which can be done with $O(n \log n \log\log n)$ bit operations. Babb [Babb 1990] found a new sequence of algorithms for computing f_n, which in the limit reduce the initial constant. During the 1993 Undergraduate Summer Research in Mathematics, Karro [Karro 1993] applied automata theory to the problem of computing f_n given n, and showed that the Fibonacci sequence cannot be generated by a finite state or push-down automaton, and that any algorithm will require at least $\log f_n$ bits of random-access memory. He also looked for patterns in the binary representations of the Fibonacci numbers which might prove helpful in computing them in linear time, but did not find any useful bit patterns. The algorithms of Cull, Holloway, and Babb still do no better than $O(n \log n \log\log n)$ bit operations, and the addition algorithm is still quite decent for smaller values of n. Ideally, we would like to find some algorithm to compute f_n in $O(n)$ bit operations because then the solution of any difference equation of the form

$$u_{n+1} = \alpha_0 u_n + \cdots + \alpha_k u_{n-k}$$

might be as efficiently computed by a similar algorithm. Our approach is to look for a family of circuits which will complute the Fibonacci numbers with a minimal number of gates because we know that for any sequencial algorithm the time in bit operations should be proportional to the number of gates needed to simulate the algorithm. Hence, if the Fibonacci numbers can be computed by circuits of $O(n)$ gates, there should be a corresponding algorithm which requires $O(n)$ bit operations.

1.2 Definitions

Let f_n denote the nth Fibonacci number, satisfying the recurrence relation

$$f_{n+1} = f_n + f_{n-1} \tag{1}$$

where $f_0 = 0, f_1 = 1$, $n \geq 2$. Let l_n denote the nth Lucas number, where

$$l_0 = 2, l_1 = 1, l_{n+1} = l_n + l_{n-1}, n \geq 2.$$

It is well known that the roots of the characteristic equation for (1) are the golden ratio and its conjugate, so let $\lambda = \frac{1+\sqrt{5}}{2}$ and $\overline{\lambda} = \frac{1-\sqrt{5}}{2}$. Binet's formula satisfies the Fibonacci difference equation, giving a non-recursive formula for f_n:

$$f_n = \frac{\lambda^n - \overline{\lambda}^n}{\sqrt{5}} \tag{2}$$

As in [Cull 1989], we will use $\log n$ to denote the base-2 logorithm of n and let $\gamma = \log \lambda$ so that γn is approximately equal to the number of digits in the base-2 representation of f_n, since f_n is asymptotic to $\lambda^n/\sqrt{5}$ by (2).

A *gate* is a device which computes a Boolean function of two input variables. A *Boolean circuit* is a directed acyclic graph with a gate at each node. Since each gate will compute a function of two variables, the in-degree of each vertex (except the sources) will be two. The sources will correspond to the inputs. Each gate is allowed to have unlimited fan-out (out-degree). A *truth table* is a list of all possible input values and the corresponding output values for a logic circuit. By examining the circuits that compute $f_0, \ldots, f_n$ we hope to find a family of circuits which require a determined number of gates. If the number of gates required were found to be linear in n, then it may be that any similar difference equation may be easily solvable. We are considering gates with two inputs, $f : \mathbb{Z}_2{}^2 \rightarrow \mathbb{Z}_2$. There are 2^{2^2}, or 16 such functions which are enumerated in table 1.

Theorems 1 and 2 list the major properties of Boolean algebra. These are important for the reduction of Boolean expressions, allowing the reduction of the gates required to implement a function.

TABLE 1. Binary functions of two inputs and one output.

X Y	0 1 0 1 0 0 1 1	**Common Name**	**Common Notation**
(1)	0 0 0 0		0
(2)	0 0 0 1	AND	XY or $X \wedge Y$
(3)	0 0 1 0		$\overline{X}Y$ or $(\neg X) \wedge Y$
(4)	0 0 1 1		Y
(5)	0 1 0 0		$X\overline{Y}$ or $X \wedge (\neg Y)$
(6)	0 1 0 1		X
(7)	0 1 1 0	EXOR	$X \oplus Y$
(8)	0 1 1 1	OR	$X + Y$ or $X \vee Y$
(9)	1 0 0 0	NOR	$\overline{X + Y}$ or $\neg(X \vee Y)$
(10)	1 0 0 1	NEXOR	$\overline{X \oplus Y}$ or $X \Leftrightarrow Y$
(11)	1 0 1 0		$\overline{X}$ or $\neg X$
(12)	1 0 1 1		$\overline{X} + Y$ or $(\neg X) \vee Y$
(13)	1 1 0 0		$\overline{Y}$ or $\neg Y$
(14)	1 1 0 1		$X + \overline{Y}$ or $X \vee (\neg Y)$
(15)	1 1 1 0	NAND	$\overline{XY}$ or $\neg(X \wedge Y)$
(16)	1 1 1 1		1

Theorem 1. *(Properties of Boolean Algebra) Given $A, B, C \in \mathbb{Z}_2$:*

$$
\begin{array}{lrcll}
\text{(a)} & \left.\begin{array}{r} AB \\ A+B \\ A\oplus B \end{array}\right. & \begin{array}{c} = \\ = \\ = \end{array} & \left.\begin{array}{l} BA \\ B+A \\ B\oplus A \end{array}\right\} & (\textit{Commutativity}) \\
\text{(b)} & \left.\begin{array}{r} (AB)C \\ (A+B)+C \\ (A\oplus B)\oplus C \end{array}\right. & \begin{array}{c} = \\ = \\ = \end{array} & \left.\begin{array}{l} A(BC) \\ A+(B+C) \\ A\oplus(B\oplus C) \end{array}\right\} & (\textit{Associativity}) \\
\text{(c)} & \left.\begin{array}{r} A(B+C) \\ A+(BC) \\ A(B\oplus C) \end{array}\right. & \begin{array}{c} = \\ = \\ = \end{array} & \left.\begin{array}{l} AB+AC \\ (A+B)(A+C) \\ AB\oplus AC \end{array}\right\} & (\textit{Distributivity}) \\
\text{(d)} & \left.\begin{array}{r} A\overline{A} \\ A+\overline{A} \end{array}\right. & \begin{array}{c} = \\ = \end{array} & \left.\begin{array}{l} 0 \\ 1 \end{array}\right\} & (\textit{Complement}) \\
\text{(e)} & \left.\begin{array}{r} AA \\ A+A \end{array}\right. & \begin{array}{c} = \\ = \end{array} & \left.\begin{array}{l} A \\ A \end{array}\right\} & (\textit{Idempotency}) \\
\text{(f)} & \left.\begin{array}{r} 0A \\ 1+A \end{array}\right. & \begin{array}{c} = \\ = \end{array} & \left.\begin{array}{l} 0 \\ 1 \end{array}\right\} & (\textit{Constant}) \\
\text{(g)} & \left.\begin{array}{r} A+0 \\ 1A \\ A\oplus 0 \\ A\oplus 1 \end{array}\right. & \begin{array}{c} = \\ = \\ = \\ = \end{array} & \left.\begin{array}{l} A \\ A \\ A \\ \overline{A} \end{array}\right\} & (\textit{Identity}) \\
\text{(h)} & \left.\begin{array}{r} A(A+B) \\ A+(AB) \end{array}\right. & \begin{array}{c} = \\ = \end{array} & \left.\begin{array}{l} A \\ A \end{array}\right\} & (\textit{Absorption})
\end{array}
$$

Theorem 2. *(DeMorgan's Laws)* Given $A, B, C, \ldots, N \in \mathbb{Z}_2$ the following identities are true:

(a) $$\overline{ABC\cdots N} = \overline{A} + \overline{B} + \overline{C} + \cdots + \overline{N}$$

(b) $$\overline{A+B+C+\cdots N} = \overline{A}\,\overline{B}\,\overline{C}\cdots\overline{N}$$

Hence, one can change between the AND and OR operators and invert the inputs and outputs, and preserve the function value.

1.3 Disjunctive Normal Form

A circuit is designed with gates by selecting a basis of one or more gates and combining those components into the desired function. There are a number of such bases used, one of which is called the *disjunctive normal form* (DNF), or "sum of minterms". Given inputs $x_0, \ldots, x_m$ let m_i be the product over the AND function of each x or its complement, $\overline{x}$, such that if $(x_0, \ldots, x_m)$ is the binary representation for i, $0 \leq i < 2^m$, then m_i has value

1. For example, $m = 3$:

$$
\begin{aligned}
m_0 &= \overline{x}_0\overline{x}_1\overline{x}_2\\
m_1 &= x_0\overline{x}_1\overline{x}_2\\
m_2 &= \overline{x}_0x_1\overline{x}_2\\
m_3 &= x_0x_1\overline{x}_2\\
m_4 &= \overline{x}_0\overline{x}_1x_2\\
m_5 &= x_0\overline{x}_1x_2\\
m_6 &= \overline{x}_0x_1x_2\\
m_7 &= x_0x_1x_2
\end{aligned}
$$

If $x_2 = 0$, $x_1 = 1$, $x_0 = 1$ $(0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3)$, then $M_3 = x_0\, x_1\, \overline{x}_2 = 1$. We have that for any function $f : \mathbb{Z}_2{}^m \to \mathbb{Z}_2$,

$$f(x_0, \ldots, x_m) = \bigvee_{\substack{f(m_i(x_0,\ldots,x_m))=1\\ 0\le i<2^m}} m_i(x_0, \ldots, x_m) \tag{3}$$

where $\vee$ denotes the sum over the OR function. This result is obvious from examination of the truth table.

TABLE 2. Truth Table for Example 1

x_2	x_1	x_0	g
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Example 1. The truth table for a simple function of three inputs, g, is shown in table 2. The DNF of g is found by ORing together the three minterms for which g has a value of one, namely m_1, m_4, m_5. This gives the straightforward equation

$$g = x_0\overline{x}_1\overline{x}_2 + \overline{x}_0\overline{x}_1x_2 + x_0\overline{x}_1x_2 \tag{4}$$

which can be implemented with 2-input gates by grouping the terms in twos:

$$g = ((x_0\overline{x}_1)\overline{x}_2 + (\overline{x}_0\overline{x}_1)x_2) + (x_0\overline{x}_1)x_2.$$

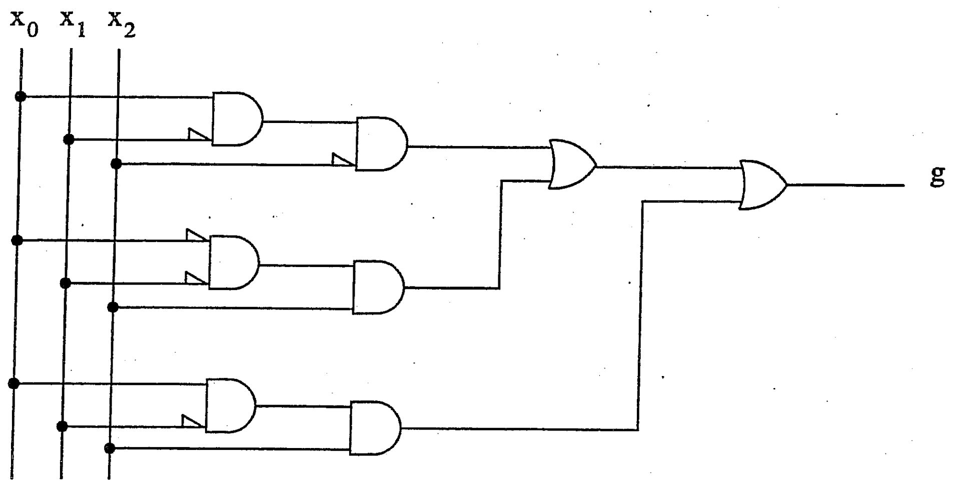

FIGURE 1. Circuit for Example 1

The corresponding gates are shown in figure 1.

By factoring and sharing terms when possible, the number of gates can very often be reduced. In this example, equation 4 can be factored like so:

$$\begin{aligned} g &= x_0\overline{x}_1\overline{x}_2 + (\overline{x}_0 + x_0)\overline{x}_1 x_2 \\ &= x_0\overline{x}_1\overline{x}_2 + 1\overline{x}_1 x_2 \\ &= \overline{x}_1(x_0\overline{x}_2 + x_2) \\ &= \overline{x}_1(x_0 + x_2) \end{aligned}$$

This saves a total of six gates, giving a 2-gate implementation of g.

1.4 CONJUNCTIVE NORMAL FORM

A very similar basis can be arrived at by applying Theorem 2 to the DNF form, but first we define the "minterms" of $x_0, \ldots, x_m$. Let M_i denote the sum over the OR function of each x or its complement. $M_i(x_0, \ldots, x_m) = 0$ when $(x_0, \ldots, x_m)$ is the binary representation of i. For example, $M_6 = x_0 + \overline{x}_1 + \overline{x}_2$.

$$f(x_0, \ldots, x_m) = \bigvee_{\substack{f(m_i(x_0,\ldots,x_m))=1 \\ 0 \le i < 2^m}} m_i(x_0, \ldots, x_m) \qquad \text{(from eq. 3)}$$

$$= \overline{\bigvee_{\substack{f(m_i(x_0,\ldots,x_m))=0 \\ 0 \le i < 2^m}} m_i(x_0, \ldots, x_m)}$$

$$= \bigwedge_{\substack{f(M_i(x_0,\ldots,x_m))=0 \\ 0 \le i < 2^m}} M_i(x_0, \ldots, x_m) \qquad \text{(by Thm. 2)}$$

Example 2. Using the same g as in example 1, we can find the *Conjunctive Normal Form* (CNF) of g by ANDing together the maxterms corresponding to the zeroes in the truth table:

$$g = (x_0 + x_1 + x_2)(x_0 + \overline{x}_1 + x_2)(\overline{x}_0 + \overline{x}_1 + x_2)(x_0 + \overline{x}_1 + \overline{x}_2)(\overline{x}_0 + \overline{x}_1 + \overline{x}_2)$$

This gives a 14-gate implementation of g, and by factoring we can again reduce the number of gates considerably by factoring and reduction:

$$\begin{aligned} g &= (x_0 + x_2 + x_1\overline{x}_1)(\overline{x}_0 + \overline{x}_1 + x_2)(x_0\overline{x}_0 + \overline{x}_1 + \overline{x}_2) \\ &= (x_0 + x_2)(\overline{x}_1 + (\overline{x}_0 + x_2)\overline{x}_2) \\ &= (x_0 + x_2)(\overline{x}_1 + \overline{x}_0\overline{x}_2 + x_2\overline{x}_2) \\ &= (x_0 + x_2)\overline{x}_1 + x_0\overline{x}_0\overline{x}_2 + x_2\overline{x}_0\overline{x}_2 \\ &= (x_0 + x_2)\overline{x}_1 \end{aligned}$$

Due to the commonalities of the two bases, it should not be surprising that the two forms reduce to the same equation.

1.5 Polynomial Basis

The other basis we used for finding circuits for Boolean functions is polynomials of the form:

$$f(x_1, \dots, x_m) = a_0 \oplus a_1x_1 \oplus a_2x_2 \oplus a_{12}x_1x_2 \oplus a_3x_3 \oplus a_{13}x_1x_3 \oplus a_{23}x_2x_3 \oplus a_{123}x_1x_2x_3 \oplus \dots$$

For this ordering of terms we can form the matrix equation:

$$(a_0, a_1, a_2, a_{12}, a_3, a_{13}, a_{23}, a_{123}) \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \\ x_3 \\ x_1x_3 \\ x_2x_3 \\ x_1x_2x_3 \end{pmatrix}$$
$$= (f(0), \dots, f(7))$$

for $m = 7$. In general, $P_k \in \mathbb{Z}_2 \times \mathbb{Z}_2$ is defined recursively as

$$P_0 = [1], \qquad P_{k+1} = \begin{pmatrix} P_k & P_k \\ 0 & P_k \end{pmatrix}, \qquad k \geq 0$$

Since $P_k = P_k^{-1}$, we can find the coefficients recursively by splitting the vector of function values in half and applying the matrix multiplication in blocks:

$$\begin{aligned} (a_0, \dots, a_{1\dots k}) &= \left(f(0), \dots, f(2^k - 1)\right) P_k \\ &= (\mathbf{f}_1, \mathbf{f}_2) \begin{pmatrix} P_{\frac{k}{2}} & P_{\frac{k}{2}} \\ 0 & P_{\frac{k}{2}} \end{pmatrix} \\ &= \left(\mathbf{f}_1 P_{\frac{k}{2}}, (\mathbf{f}_1 + \mathbf{f}_2) P_{\frac{k}{2}}\right) \end{aligned}$$

This is repeated until the $k = 0$.

Example 3. As an example of how to find the polynomial coefficients, consider again the function g from examples 1 and 2. We can continually divide the vector of values, copying the first half of each block into the next row, and then putting the sum of the two halves next to it:

$$\begin{array}{ccccccccccccccc}
0 & & 1 & & 0 & & 0 & & 1 & & 1 & & 0 & & 0 \\
0 & & 1 & & 0 & & 0 & | & 1 & & 0 & & 0 & & 0 \\
0 & & 1 & | & 0 & & 1 & | & 1 & & 0 & | & 1 & & 0 \\
0 & | & 1 & | & 0 & | & 1 & | & 1 & | & 1 & | & 1 & | & 1
\end{array}$$

So for g we get the polynomial equation

$$g = x_1 \oplus x_0x_1 \oplus x_2 \oplus x_0x_2 \oplus x_1x_2 \oplus x_0x_1x_2.$$

2. General Bounds on Number of Gates

Consider the circuit that will generate any Fibonacci number from f_0 through f_n. Using Binet's equation (2) we know that the number of bits in f_n is going to be

$$\lceil \log(\frac{\lambda^n}{\sqrt{5}}) \rceil = \lceil n \log \lambda - \frac{1}{2} \log 5 \rceil \tag{5}$$

but for convenience we are using γn to denote the number of bits in f_n.

The obvious lower bounds on the number of gates required is γn, provided that none of the output lines is identical to another, or to one of the inputs. This follows from the fact that each gate has exactly one output, so there must be at least one gate for every output in the circuit. By two different methods we have arrived at upper bounds that are $O(n^2)$, one which uses a combination of the DNF and CNF methods described in section (1–3) and (1–4), the other of which simulates the addition algorithm for computing the f_k.

2.1 Look-Up Table Circuit

The first general circuit class we considered was the circuit to simulate a simple lookup in which all of the Fibonacci numbers up to f_n are known and are encoded in the circuit. By applying the general DNF method to each of the digits of $f_0, \ldots, f_n$ (ORing together the maxterms that correspond to 1's in the truth table) we can construct a circuit which uses only AND gates (plus possible inverted inputs) and OR gates. This circuit design is shown in figure 2.

To AND together $\log n$ lines requires $\log n - 1$ gates. There will be n of these AND trees: m_0 will never generate any 1's. There will then be some fraction of these for each output, so we will say there are at most n lines to OR together for each output. This is a very poor over-estimate, but we did not have time to investigate a better estimate. By using the known results for the residues of $f_n \mod 2^k$ we suspect the number of lines necessary could probably be reduced by at least $\frac{1}{2}$. Assuming at most n lines for each of the γn outputs gives us another $\gamma n^2 - n$ gates for an upper bound of

$$\gamma n^2 + n \log n - (\gamma + 1)n \tag{6}$$

for the entire circuit.

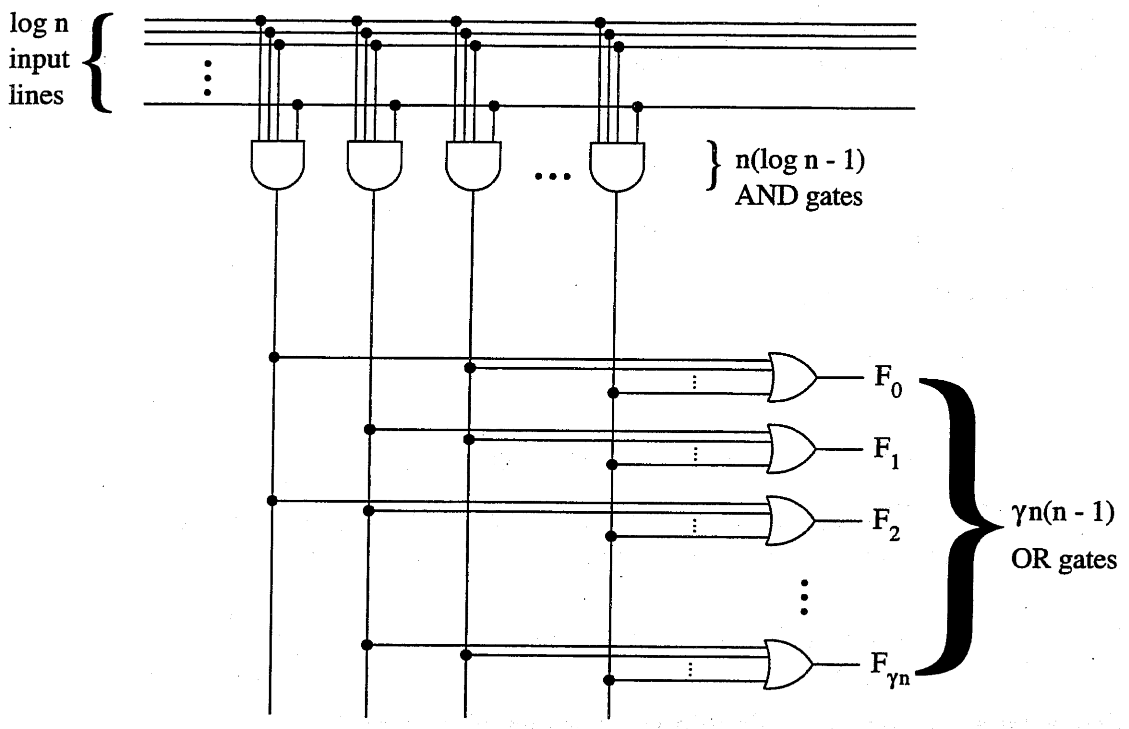

FIGURE 2. Simulated Look-Up Table Circuit

2.2 ADDITION ALGORITHM CIRCUIT

The other circuit we considered would simulate the addition algorithm. The look-up table circuit has the obvious disadvantage that it does not actually *compute* the $n + 1$ Fibonacci numbers, but simply connects the correct value to the inputs. Instead, we would like a circuit which will generate the Fibonacci numbers up to any n, even if we do not know them all ahead of time.

To do this, we construct a circuit with $n - 1$ addition circuits to compute the sums according to equation 1. This will require $5\gamma n - 3$ gates per adder for a total of $(5\gamma n^2 - (5\gamma + 3)n + 3$ gates. To get the correct Fibonacci number from this sequence of adders we will use a full decoder (for all $n + 1$ Fibonacci numbers), an array of AND gates for enabling the outputs, and an OR tree to combine these values for each of the outputs. This totals

$$7\gamma n^2 + n \log n - (5\gamma + 4)n - \log n + 4 \tag{7}$$

gates which is obviously much less efficient than the look-up table algorithm, though it is still $O(n^2)$.

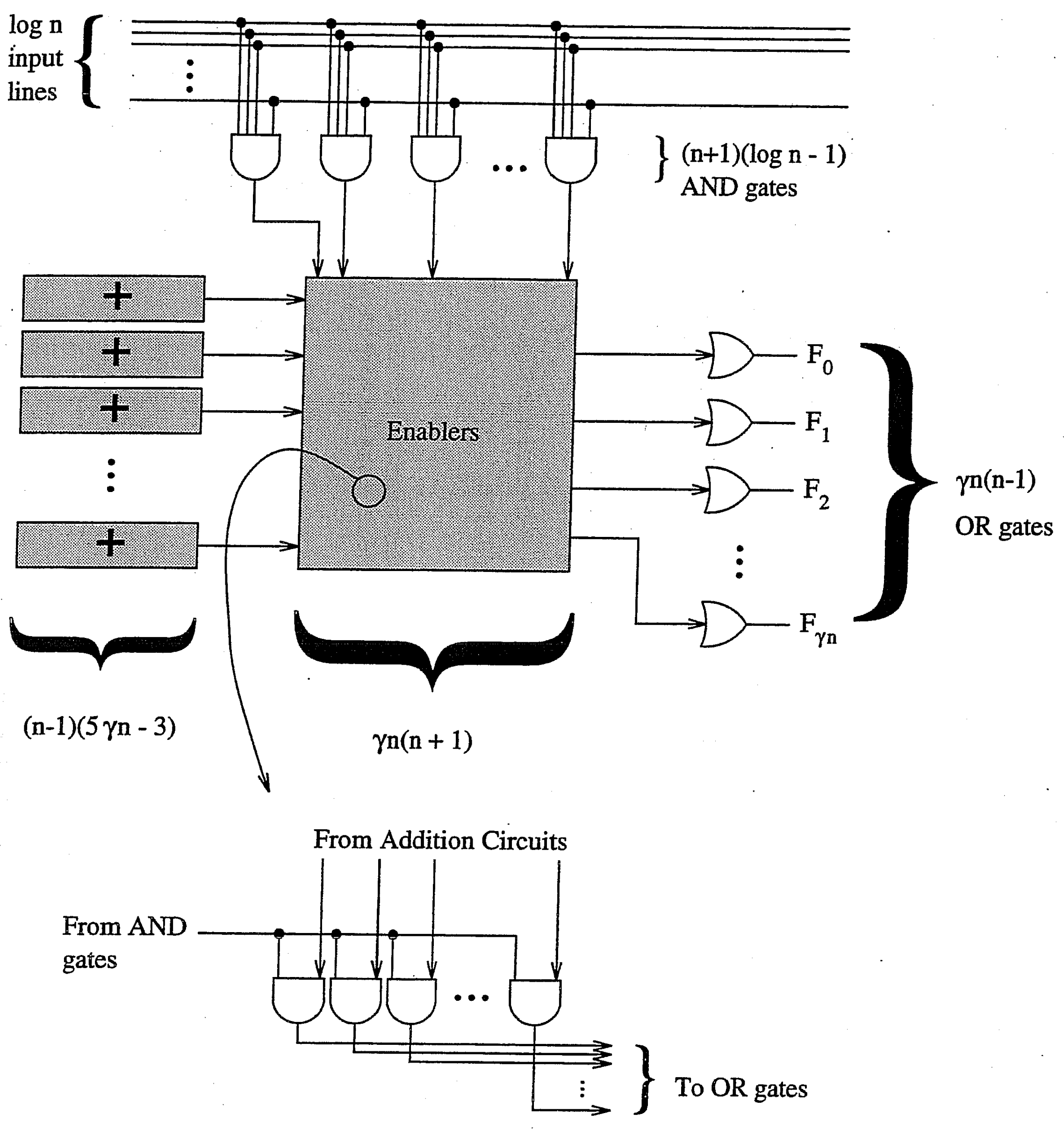

FIGURE 3. Simulated Addition Algorithm Circuit

3. CIRCUITS FOR SMALL n

We begin by considering three base cases for circuit designs, looking for structure in the circuits which might lead to a method of constructing any of a family of circuits for computing $f_0 \dots f_n$ with the fewest number of gates. We consider the cases $n = 3$, $n = 7$, and $n = 15$.

3.1 The Minimal Circuit for $n = 3$

For $n = 3$, we have that the number of inputs is 2 as is the number of outputs. Any such function can be implemented with two or fewer gates since there exists one of the 16 functions of the two inputs for each output. The function can be represented with the simple truth table shown in table 3.

Table 3. Truth Table for $n = 3$

x_1	x_0	F_1	F_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

By inspection, it is clear that the function $F_1(x_0, x_1)$ is equivalent to the AND function shown in table 1 and that the function $F_0(x_0, x_1)$ is equivalent to the EXOR function. Since neither of these corresponds to one of the outputs, and since each is independent, it follows that the minimal circuit for the case $n = 3$ corresponds to the equations

$$F_0 = x_0 \oplus x_1$$
$$F_1 = x_0 x_1$$

This circuit is shown in figure 4.

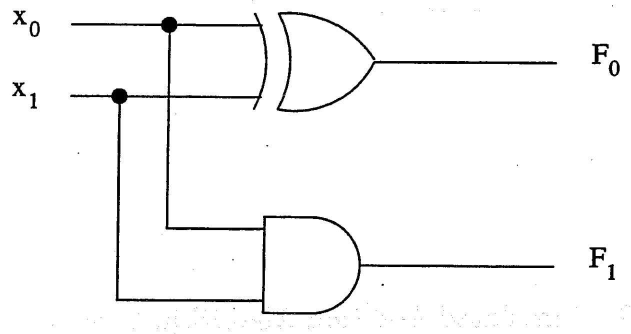

Figure 4. Minimal Circuit for $n = 3$

TABLE 4. Truth Table for $n = 7$

x_2	x_1	x_0	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	0	1	1
1	0	1	0	1	0	1
1	1	0	1	0	0	0
1	1	1	1	1	0	1

3.2 CIRCUIT FOR $n = 7$

The case for $n = 7$ is nontrivial. By equation 5 we know that for $n = 7$ the circuit must have 4 outputs. Table 4 shows the truth table for this case.

By applying the DNF method described in section (1–3) we get a circuit using 13 gates, the CNF method results in 14 gates, and a well-chosen combination of the two gives a 12-gate circuit. However, by inspection it is not difficult to find a circuit that uses 10 or fewer gates so these two bases are clearly not optimal for this case. By applying the polynomial method described in the same section it is possible to find a circuit of only 8 gates. The polynomial method yields the following equations:

$$\begin{aligned} F_0 &= x_0 \oplus x_1 \oplus x_2 \oplus x_0x_2 \oplus x_0x_1x_2 \\ F_1 &= x_0x_1 \oplus x_2 \oplus x_0x_2 \oplus x_1x_2 \\ F_2 &= x_0x_2 \\ F_3 &= x_1x_2 \end{aligned}$$

These equations can be factored to give:

$$\begin{aligned} F_0 &= x_1 \oplus x_2 \oplus x_0(1 \oplus x_2 \oplus x_1x_2) \\ &= (x_1 \oplus x_2) \oplus x_0\overline{\overline{x}_1x_2} \\ F_1 &= (x_0 \oplus x_2)(x_1 \oplus x_2) \\ F_2 &= x_0x_2 \\ F_3 &= x_1x_2 \end{aligned}$$

For the two most significant digits, it is clear that the minimal implementation will be one AND gate for each, but it is less clear how digits 0 and 1 are most optimally implemented. Without factoring or sharing, the number of gates required is 15. With the above factorization plus sharing of common terms, the number of gates required is reduced to 8, which is shown in figure 5.

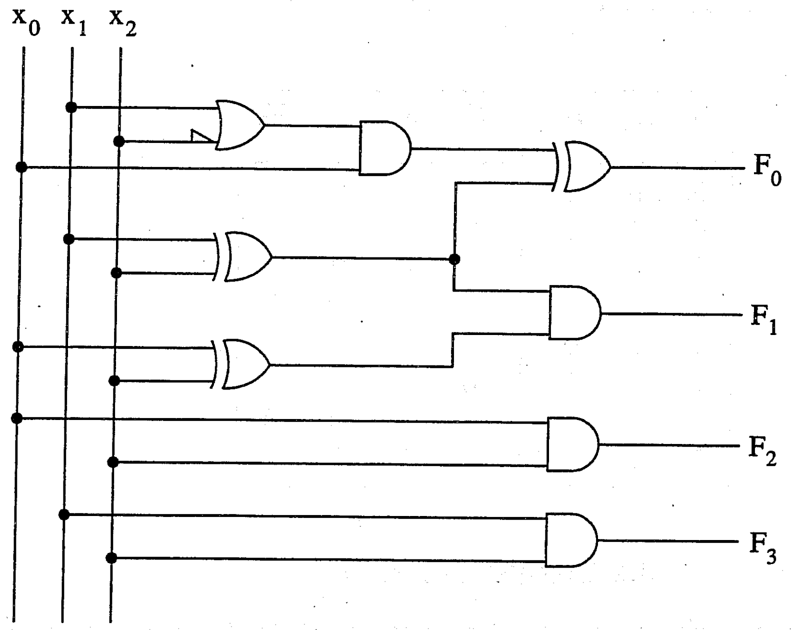

FIGURE 5. Eight-gate Circuit for $n = 7$

3.3 Circuit for $n = 15$

The truth table for the case of $n = 15$ is shown in table 5. The DNF and CNF methods give unwieldy equations that do not reduce well. The polynomial method, on the other hand, yields a 36-gate circuit in which the sharing of terms is obviously a large factor in the simplification of the circuit. This implementation is shown in figure 6. Without sharing of terms, the circuit would require 47 gates. Those gates which are shared are marked with a bullet (•).

TABLE 5. Truth Table for $n = 15$

x_3	x_2	x_1	x_0	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	0	0	0	0	1	1
0	1	0	1	0	0	0	0	0	0	0	1	0	1
0	1	1	0	0	0	0	0	0	0	1	0	0	0
0	1	1	1	0	0	0	0	0	0	1	1	0	1
1	0	0	0	0	0	0	0	0	1	0	1	0	1
1	0	0	1	0	0	0	0	1	0	0	0	1	0
1	0	1	0	0	0	0	0	1	1	0	1	1	1
1	0	1	1	0	0	0	1	0	1	1	0	0	1
1	1	0	0	0	0	1	0	0	1	0	0	0	0
1	1	0	1	0	0	1	1	1	0	1	0	0	1
1	1	1	0	0	1	0	1	1	1	1	0	0	1
1	1	1	1	1	0	0	1	1	0	0	0	1	0

The complete polynomial equations for $n = 15$, with the factorization I use are:

$$\begin{aligned}
F_0 &= x_0 \oplus x_1 \oplus x_2 \oplus x_0x_2 \oplus x_0x_1x_2 \oplus x_3 \oplus x_1x_3 \oplus x_0x_1x_3 \oplus x_0x_2x_3 \oplus x_1x_2x_3 \\
&= (x_0 \oplus x_1) \oplus (\overline{x_0\overline{x_1}})x_2 \oplus x_3(x_0 \oplus x_1)(x_1 \oplus x_2) \\
F_1 &= x_0x_1 \oplus x_2 \oplus x_0x_2 \oplus x_1x_2 \oplus x_0x_3 \oplus x_1x_3 \oplus x_0x_1x_3 \oplus x_2x_3 \oplus x_0x_1x_2x_3 \\
&= (x_0 \oplus x_2)(x_1 \oplus x_2) \oplus x_3 \left[x_0(\overline{x_1\overline{x_2}}) \oplus (x_1 \oplus x_2)\right] \\
F_2 &= x_0x_2 \oplus x_3 \oplus x_0x_3 \oplus x_2x_3 \\
&= (x_0 \oplus x_3)(x_2 \oplus x_3) \\
F_3 &= x_1x_2 \oplus x_0x_1x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_2x_3 \\
&= (x_0x_3)(x_1 + x_2) \oplus x_1x_2 \\
F_4 &= x_3 \oplus x_0x_3 \oplus x_0x_1x_2x_3 \\
&= x_3(\overline{x_0(\overline{x_1\overline{x_2}})}) \\
F_5 &= x_0x_3 \oplus x_1x_3 \oplus x_0x_1x_2x_3 \\
&= x_3(x_1 \oplus x_0(\overline{x_1x_2})) \\
F_6 &= x_0x_1x_3 \oplus x_0x_2x_3 \oplus x_1x_2x_3 \\
&= x_3(x_0(x_1 \oplus x_2) \oplus x_1x_2
\end{aligned}$$

$$\begin{aligned} F_7 &= x_2x_3 \oplus x_1x_2x_3 \\ &= \overline{x}_1x_2x_3 \\ F_8 &= x_1x_2x_3 \oplus x_0x_1x_2x_3 \\ &= \overline{x}_0x_1x_2x_3 \\ F_9 &= x_0x_1x_2x_3 \end{aligned}$$

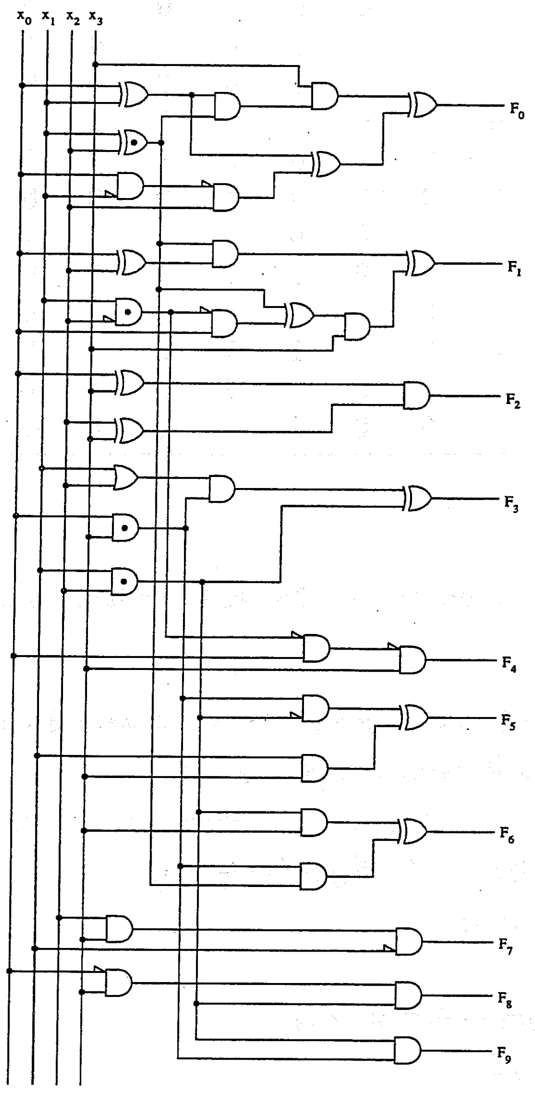

FIGURE 6. 36-gate Implementation for $n = 15$

4. Conclusions

From the three base cases examined, it looks as though the number of gates required to compute f_n given n may be $O(n^2)$. That is, when we double n, the number of gates required appears to increase by a factor of 4. There are still a number of unanswered questions:

(1) Can we find a binary circuit structure which gives a better upper bound? We considered only the possibilities that correspond to a table look-up and a direct addition algorithm. Another possibility would be to construct a circuit which simulates one of the various multiplication algorithms known for computing f_n. Such a circuit ought to require only $O(n \log n \log\log n)$ gates, corresponding to the number of bit operations for these algorithms.

(2) Is there an algorithm to find the circuit with the fewest number of gates? We considered trying a search of all possible binary circuits, but the search space was too daunting without better upper bounds on the number of gates.

Bibliography

Brendan J. Babb, *Computing Fibonacci Numbers Rapidly*, M. S. Paper, Oregon State University, Corvallis, Oregon (1991).

David J. Comer, *Digital Logic and State Machine Design, 2nd ed.*, Saunders College Publishing, 1990.

Paul Cull and J. L. Holloway, *Computing Fibonacci Numbers Quickly*, Information Processing Letters **32** (1989), 143–149.

Paul E. Dunne, *The Complexity of Boolean Networks*, Harcourt Brace Jovanovich Publishing, New York, 1988.

Eliot Jacobson, *Almost Uniform Distribution of the Fibonacci Sequence*, Fibonacci Quarterly **27, no. 4** (1989), 335–337.

John Karro, *The Complexity of Computing Fibonacci Numbers*, Undergraduate Summer Research in Mathematics, Oregon State University, Corvallis, Oregon (1993), (unpublished).

N. N. Vorob'ev, *Fibonacci Numbers*, Blaisdell, New York, 1961.

Department of Mathematics, Oregon State University, Corvallis OR 97331

E-mail: kaltwasc@math.orst.edu