July 1989 Report No. STAN-CS-89-1268

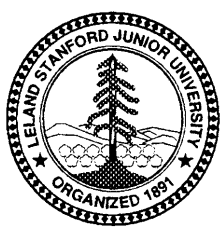
Addition Machines

by

Robert W. Floyd and Donald E. Knuth

Department of Computer Science

Stanford University
Stanford, California 94305



Addition Machines

Robert W. Floyd and Donald E. Knuth
Stanford University

An addition machine is a computing device with a finite number of registers, limited to the following six types of operations:

read x	{input to register x}
x $\leftarrow y$	{copy register y to register x}
$x \leftarrow x + y$	{add register y to register x}
$x \leftarrow x -$ y	{subtract register y from register x}
if $x \geq y$	{compare register x to register y}
write x	{output from register x}

The register contents are assumed to belong to a given set A, which is an additive subgroup of the real numbers. If A is the set of all integers, we say the device is an integer *addition machine*; if A is the set of all real numbers, *we* say the device is a *real addition machine.*

We will consider how efficiently an integer addition machine can do operations such as multiplication, division, greatest common divisor, exponentiation, and sorting. We will also show that any addition machine with at least sis registers can compute the ternary opera tion $x\lfloor y/z \rfloor$ with reasonable efficiency, given x, y, $z \in A$ with $z \neq 0$.

Remainders. As a first example, consider the calculation of

$$x \bmod \text{y} = \begin{cases} \mathbf{x} - y\lfloor x/y \rfloor, & \text{if y} \neq \mathbf{0:} \\ x, & \text{if } \mathbf{y=O.} \end{cases}$$

This binary operation is well defined on any additive subgroup A of the reals, and we can easily compute it on an addition machine as follows:

P_1:
```
read x; read y; z ← z − z;
if y ≥ z then
    if z ≥ y then {y = 0, do nothing}
    else if x ≥ z then while x ≥ y do x ← x − y
        else repeat x ← x + y until x ≥ z
else if 3 ≥ x then while y ≥ x do x ← x − y
    else repeat x ← x + y until z ≥ x;
write x.
```

This research was supported in part by the National Science Foundation under grant, CCR-86-10181, and by Office of Naval Research contract N00014-87-K-0502.

(There is implicitly a finite-state control. A pidgin Pascal program such as this one is easily converted to other formalisms; cf. [1].)

Program P_1 handles all sign combinations of x and y; therefore it is rather messy. In the special case where $x \geq 0$ and y > 0, a much simpler program applies:

```
P2:   read x; read y;        {assume that x ≥ 0 and y > 0}
      while x ≥ y do x ← x − y;
      write x.
```

Any program for this special case can be converted to a program of comparable efficiency for the general case by using the identities

$$-x = (x - x) - x;$$

$$(-x) \bmod (-y) = -(x \bmod y);$$

$$(-x) \bmod y = \begin{cases} y - (x \bmod y), & \text{if } x \bmod y \neq 0; \\ 0, & \text{if } x \bmod y = 0. \end{cases}$$

General programs for multiplication, division, and gcd can be constructed similarly from algorithms that assume positive operands. We shall therefore restrict consideration to positive cases in the algorithms below.

Program P_2 performs $\lfloor y/x \rfloor$ subtractions. Can we do better? Yes; here, for example, is a. program that uses a doubling procedure to **subtract** larger multiples of y:

```
P3:   read x; read y;        {assume that x ≥ 0 and y > 0}
      while x ≥ y do
          begin z ← y;
          repeat w ← 3; z ← z + z; until not x ≥ z;
          x ← x − w;
          end;
      write x.
```

This program repeatedly subtracts $w = 2^k y$ from x, where $k = \lfloor \log_2(x/y) \rfloor$; thus, it implicitly computes the binary representation of $\lfloor x/y \rfloor$, from left to right. The total running time is bounded by $O(\log(x/y))^2$, which is considerably smaller than $\lfloor x/y \rfloor$ when $\lfloor x/y \rfloor$ is large.

Further improvement, to a running time that is $O(\log(x/y))$ instead of $O(\log(x/y))^2$, appears at first sight to be impossible, because an addition machine has only finitely many registers and it cannot divide by 2. Therefore the numbers y, $2y$, $4y$, $8y$, . . . must all apparently be computed again and again if we want to use a trick based on doubling.

A Fibonacci method. Remainders can, however. be computed with the desired efficiency $O(\log(x/y))$ if we implicitly use the Fibonacci representation of $\lfloor x/y \rfloor$ instead of the binary

representation. Define Fibonacci numbers as usual by

$$F_0 = 0; \quad F_1 = 1; \quad F_n = F_{n-1} + F_{n-2}, \qquad \text{for } n \geq 2.$$

Every nonnegative integer n can be uniquely represented [9] in the form

$$n = F_{l_1} + F_{l_2} + \cdots + F_{l_t}, \qquad l_1 \gg l_2 \gg \cdots \gg l_t \gg 0,$$

where $t \geq 0$ and $l >> 1'$ means that $l - 1' \geq 2$. If $n > 0$, this representation can be found by choosing l_1 such that

$$F_{l_1} \leq n < F_{l_1+1},$$

so that $n - F_{l_1} < F_{l_1+1} - F_{l_1} = F_{l_1-1}$, and by writing

$$n = F_{l_1} + (\text{Fihonacci representation of } n - F_{l_1}).$$

We shall let

$$\lambda n = l_1 \qquad \text{and} \qquad \nu n = t$$

denote respectively the index of the leading term and the number of terms, in the Fibonacci representation of n. By convention? $\lambda 0 = 1$.

Fibonacci numbers are well suited to addition machines because we can go from the pair $\langle F_l, F_{l+1} \rangle$ up to the nest pair $\langle F_{l+1}, F_{l+2} \rangle$ with a single addition, or down to the previous pair $\langle F_{l-1}, F_l \rangle$ with a single subtraction. Furthermore, Fibonacci numbers grow exponentially, about 69% as fast as powers of 2. They have been used as power-of-2 analogs in a variety of algorithms (see, for instance, "Fibonacci numbers" in the index to [3]), and in Matijasevich's solution to Hilbert's tenth problem [6].

If we let two registers of an addition machine contain the pair of numbers $\langle yF_l, yF_{l+1} \rangle$, where y is an implicit parameter, it is easy to implement the operations

$$l \leftarrow 1, \qquad l \leftarrow l + 1, \qquad l \leftarrow l - 1$$

and to test the conditions

$$x \geq yF_l, \qquad x < yF_{l+1}, \qquad l = 1.$$

Therefore we can compute x mod y efficiently by implementing the following procedure:

```
read x; read y;        {assume that x ≥ 0 and y > 0}
if x ≥ y then
    begin l ← 1;
    repeat l ← l + 1 until x < yF_{l+1};
    repeat if x ≥ yF_l then x ← x − yF_l;
        l ← l − 1;
    until l = 1;
    end;
write x.
```

The first **repeat** loop increases l until we have

$$yF_l \le x < yF_{l+1},$$

i.e., until $l = \lambda n$, where $n = \lfloor x/y \rfloor$. The second loop decreases l while subtracting

$$yF_{l_1} + yF_{l_2} + \cdots + yF_{l_t} = n$$

from x according to the Fibonacci representation of n. The result, $x - n\,y = x \bmod y$, has been computed with

$$2\lambda n - 2 + \nu n = O(\log(x/y))$$

additions and subtractions altogether.

Here is the same program expressed directly in terms of additions and subtractions, using only three registers:

```
P4:   read x; read y;        {assume that x ≥ 0 and y > 0}
      if x ≥ y then
          begin z ← y;
          repeat ⟨y, z⟩ ← ⟨z, y + z⟩ until not x ≥ z;      { x ≥ y still holds}
          repeat if x ≥ y then x ← x − y;
              ⟨y, z⟩ ← ⟨z − y, y⟩;
          until y ≥ z;
          end:
      write x.
```

The multiple assignment '$\langle y, z\rangle \leftarrow \langle z, y + x\rangle$' is an abbreviation for the operation 'set $y \leftarrow y + z$ and interchange the roles of registers y and z in the subsequent program': the assignment '$\langle y, z\rangle \leftarrow \langle z - y, y\rangle$' is similar. By making two copies of this program code. in one of which the variables y and z are interchanged, we can jump from one copy to the other and obtain a legitimate addition-machine program; cf. [4, Esample 7].

A formal proof of correctness for program P_4 would establish the invariant relation

$$\exists\, l \ge 1 \ (y = y_0 F_l \text{ and } z = y_0 F_{l+1})$$

in the case $x_0 \ge y_0$, where x_0 and y_0 are the initial values of x and y.

Multiplication and division. We can use essentially the same idea to compute the ternary operation $x\lfloor y/z \rfloor$efficiently on any addition machine. This time we accumulate multiples of x as we discover the Fibonacci representation of $\lfloor y/z \rfloor$:

```
read x; read y; read z;        {assume that y ≥ 0 and z > 0}
w ← 0;
```

```
    if y ≥ z then
        begin l ← 1;
        repeat l ← l + 1 until not y ≥ zF_{l+1};
        repeat if y ≥ zF_l then ⟨w, y⟩ ← ⟨w + xF_l, y − zF_l⟩;
            l ← l − 1;
        until l = 1;
        end;
    write w.
```

The actual addition-machine code requires six registers, because we need Fibonacci multiples of x as well as z:

```
P5:     read x; read y; read z;        { assume that y ≥ 0 and z > 0
        w ← w − w;
        if y ≥ z then
            begin u ← x; v ← z;
            repeat ⟨u, x⟩ ← (x, u + x); ⟨v, z⟩ ← ⟨z, v + z⟩;
            until not y ≥ z;        {y ≥ v still holds}
            repeat if y ≥ v then ⟨w, y) ← ⟨w + u, y − v⟩;
                ⟨u, x⟩ ← ⟨x − 11, u⟩; ⟨v, 3) ← ⟨z − v, v);
            until v ≥ z;
            end;
        write w.
```

The key invariant relations, in the case $y_0 \geq z_0$, are now

$$3\, l \geq 1\ (u = x_0 F_l\,,\ x = x_0 F_{l+1}\,,\ v = z_0 F_l\,,\ z = z_0 F_{l+1})\,;$$

$$\exists n \geq 0\ (w = x_0 n\,,\ y = y_0 - z_0 n)\,.$$

If we suppress x, u, and w from this program, the **repeat** statements act on $\langle y, v, z\rangle$ exactly as the **repeat** statements in our previous program act on $\langle x, y, z\rangle$. Therefore, if $y_0 \geq z_0$, we have $y = y_0 \bmod z_0 = y_0 - z_0 \lfloor y_0/z_0 \rfloor$ after the **repeat** statements in the new program. Hence $w = x_0 \lfloor y_0/z_0 \rfloor$ as desired. The total number of additions and subtractions is

$$4\lambda n - 3 + 2\nu n = O(\log(y_0/z_0))$$

where $n = \lfloor y_0 / z_0 \rfloor$.

An integer addition machine can make use of the constant 1 by reading that constant, into a. separate, dedicated register. Then we can specialize the ternary algorithm by setting $z \leftarrow 1$ (for multiplication) or $x \leftarrow 1$ (for division). Thus we can compute the product. xy i n $O(\log \min(|x|, |y|))$ operations, and the quotient $\lfloor y/z \rfloor$ in O(log $|y/z|$) operations, using only addition, subtraction, and comparison of integers. (Multiplica tion and division

clearly cannot be clone unless such constants are used, since any function $f(x, y, \ldots)$ computed by an addition machine that inputs the sequence of values $\langle x$, y, . . .) must satisfy $f(\alpha x, \alpha y, \ldots) = \alpha f(x$, y, . . .) for all $\alpha > 0$.)

Greatest common divisors. Euclid's algorithm for the greatest common divisor of two positive integers x and y can be formulated as follows:

read x; **read** y; {assume that $x > 0$ and y ≥ 0}
while $y > 0$ **do** **(.I',** $y\rangle \leftarrow$ **(y,** x mod y);
write x.

The **while** loop preserves the invariant relation gcd(x, y) = gcd(x_0, y_0). After the first iteration, we have $x > y \geq 0$; the successive values of x are strictly decreasing and positive, so the algorithm must terminate.

We can therefore use our method for computing x mod y to calculate gcd(x, y) on an integer addition machine:

P_6: **read** x; **read** y; {assume that x > 0 and y ≥ 0}
$z \leftarrow y$; $z \leftarrow z + z$;
while not $y \geq z$ **do** {equivalently, y > 0, since $z = 2y$}
begin while $x \geq z$ **do** $\langle y, z\rangle$ t $\langle z, y + z\rangle$;
repeat if $x \geq y$ **then** x t **x** − y;
$\langle y, z\rangle \leftarrow \langle z - y, y\rangle$;
until $y \geq z$;
(,I'. $y\rangle \leftarrow \langle y,$ **x);** $z \leftarrow$ **y;** z **t** z **+** z;
end:
write x.

(Here the operation (x, y) ← (y , $x\rangle$ should not really be performed; it means that the roles of registers x and y should be interchangecl. The implementation jumps between sis copies of this program, one for each permutation of the register names x, y, z.)

This algorithm will compute gcd(x, y) correctly on a general addition machine, whenever the ratio y/x is rational. Otherwise it will loop forever.

The total number of operations performed by program P_6 is

$$T(x,y) = f(q_1) + f(q_2) + \cdots + f(q_m) + 6,$$

where $q_1, q_2, \ldots, q_m$ is the sequence of quotients $\lfloor x/y \rfloor$ in the respective iterations of Euclid's algorithm. and where $f(q)$ counts the number of operations in one iteration of the outermost **while** loop. If q = 0 (this case can occur only on the first iteration), we have one assignment, one addition. one subtraction, and four comparisons; so $f(0) = 7$. If $q > 0$ we have one assignment, λq − **1** additions, $\lambda q + \nu q$ − **1** subtractions, and $3\lambda q - 2$

comparisons; so

$$f(q) = 5\lambda q + \nu q - 3.$$

We have f(1) = 8, f(2) = 13, and, in general, $f(F_l) = 51 - 2$ for all $l \geq 2$.

This three-register algorithm for greatest common divisor turns out to be quite efficient, even though it uses only addition, subtraction, and comparison. Indeed, the numbers in the registers never exceed $2 \max(x, y)$, where x and y are the given inputs, and we can obtain rather precise bounds on the running time.

Theorem 1. Let $N = \max(x, y)/ \gcd(x, y)$. *The* number of' operations $T(x, y)$ *performed by program* P_6 *satisfies*

$$3 \log_\phi N + \alpha \leq T(x, y) \leq 13.5 \log_\phi N + \beta,$$

for some constants α *and* β, *where* $\phi = (1 + \sqrt{5})/2$.

Proof. We can assume that x > y; then all the q's are positive. If $F_l \leq q < F_{l+1}$ *we* hase $\lambda q = l$ and $1 \leq \nu q \leq l/2$, hence

$$5l - 2 \leq f(q) \leq 5.5l - 3.$$

Furthermore we have $\phi^{l-2} \leq F_l \leq \phi^{l-1}$, hence

$$5 \log_\phi(q + 1) - 2 \leq f(q) \leq 5.5 \log_\phi q + 8.$$

Summing over all values $q_1, \ldots, q_m$ gives

$$5 \log_\phi\big((q_1 + 1) \ldots (q_m + 1)\big) - 2m \leq T(x, y) - 6 \leq 5.5 \log_\phi(q_1 \ldots q_m) + 8m.$$

Now let the values occurring in Euclid's algorithm be $x_0, x_1, \ldots, x_{m+1}$, where $x_0 = x$, $x_1 = y$, $x_{j+1} = x_{j-1} \bmod x_j$, $x_m = \gcd(x, y)$, and $x_{m+1} = 0$. Then $q_j = \lfloor x_{j-1}/x_j \rfloor$ for $1 \leq j \leq m$, and we have

$$q_1 q_2 \ldots q_m \leq \frac{x_0}{x_1}\frac{x_1}{x_2} \cdots \frac{x_{m-1}}{x_m} < (q_1 + 1)(q_2 + 1) \ldots (q_m + 1).$$

The product $(x_0/x_1)(x_1/x_2) \ldots (x_{m-1}/x_m) = x_0/x_m$ is just what we have called N. Furthermore we have $m \leq \log_\phi N$ by a well-known theorem of Lamé [2, Theorem 4.5.3F]. This suffices to complete the proof.

When the inputs are consecutive Fibonacci numbers $\langle x, y \rangle = \langle F_m, F_{m+1} \rangle$ with $m \geq 2$, we have $q_1 = 0$, $q_2 = \ldots = q_{m-1} = 1$, $q_m = 2$, and the total running time is

$$T(F_m, F_{m+1}) = 7 + 8(m - 2) + 13 + 6 = 8m + 10.$$

This appears to be the worst case, in the sense that it seems to maximize $T(x, y)$ over all pairs $\langle x, y \rangle$ with $\max(x, y) \leq F_{m+1}$. Computations for small n support this conjecture, which (if true) would imply that the upper bound in Theorem 1 could be improved to Slog, $N + \beta$.

Stacks. Euclid's algorithm defines a one-to-one correspondence between pairs of relatively prime positive integers (x, $y\rangle$ with x > y and sequences of positive integers $\langle q_1, \ldots, q_m \rangle$ where each $q_j \geq 1$ and $q_m \geq 2$. We can push a new integer q onto the front of such a sequence by setting $\langle x, y \rangle \leftarrow (qx + y, x)$; we can pop $q_1 = \lfloor x/y \rfloor$ from the front by setting $\langle x, y \rangle \leftarrow \langle y, x \bmod y \rangle$.

Therefore an integer addition machine can represent, a stack of arbitrary depth in two of its registers. The operation of pushing or popping a positive integer q can be clone with $O(\log q)$ operations, using a. few auxiliary registers.

Here, for example, is the outline of an integer addition program that reads a sequence of positive integers followed by zero and writes out those positive integers in reverse order:

$\langle x, y \rangle \leftarrow \langle 2, 1 \rangle$; {the empty stack}
repeat read q;
  if $q \geq 1$ **then** $\langle x, y \rangle \leftarrow (qx + y, x)$;
until not $q \geq 1$;
repeat $\langle q, x, y \rangle \leftarrow \langle \lfloor x/y \rfloor, y, x \bmod y \rangle$;
  if $y \geq 1$ **then write** q;
until not $y \geq 1$.

This program uses the algorithms for multiplication and division shown earlier. The running time to reverse the input $\langle q_1, q_2, \ldots, q_m, 0 \rangle$ is $O(m + \log q_1 q_2 \ldots q_m)$.

We can sort a given list of positive integers $\langle q_1, q_2, \ldots, q_m \rangle$ in a similar way, using the classical algorithms for merge sorting with three or more magnetic tapes that can be "read backwards" [3, Section 5.4.4]. The basic operations required are essentially those of a stack; so we can sort in 0 ((m + log $q_1 q_2 \ldots q_m$) log m) steps if there are at least 12 registers.

Exponentiation. We can now show that an integer addition machine is able to compute

$$x^y \bmod z$$

in $O((\log y)(\log z) + \log(x/z$ j) operations. The basic idea. is simple: We first form the numbers

$$x_l = x^{F_l} \bmod z$$

for $2 \leq l \leq \lambda y$; this requires one multiplication mod z for each new value of l, once $x_2 = x \bmod z$ has been found in 0 (log(x/z)) operations. Then we use the Fibonacci

representation of y to compute $x^y \bmod z$ with $\nu y - 1$ further multiplications mod z. For example, $x^{11} \bmod z$ is computed by successively forming the powers

$$x^1, \quad x^2, \quad x^3, \quad x^5, \quad x^8, \quad x^{8+3}$$

modulo z.

There is, however, a difficulty in carrying out this plan with only finitely many registers, since the method we have used to discover the Fibonacci representation of y determines the relevant terms F_l in reverse order from the way we need to calculate the relevant factors x_l.

One solution is to push the numbers $x_2, x_3, \ldots, x_{\lambda y}$ onto a simulated stack as they are being computed. Then we can pop them off in the desired order as we discover the Fibonacci representation of y. Each stack operation takes $O(\log z)$ time, since each x_l is less than z; hence the stacking and unstacking requires only $O((\log y)(\log z))$ operations, and the overall running time changes by at most a constant factor.

But the stacking operation forms extremely large integers, having $\Theta((\log y)(\log z))$ bits, so it is not a practical solution if we are concerned with the size of the numbers being added and subtracted as well as the number of additions and subtractions. An algorithm that needs only $O((\log y)(\log z))$ additions and subtractions of integers that never get much larger than z would be far more useful in practice.

We can obtain such an algorithm if we first compute the "Fibonacci reflection" of y, namely the number

$$y^R = F_{2+\lambda y-l_1} + F_{2+\lambda y-l_2} + \cdots + F_{2+\lambda y-l_t}$$

when y has the Fibonacci representation

$$y = F_{l_1} + F_{l_2} + \cdots + F_{l_t}.$$

Then we can use the Fibonacci representation of y^R to determine the relevant factors x_l as we compute them; no stack is needed.

Here is a program that computes y^R, assuming that $y > 0$ and that both y and the constant 1 have already been read into registers named y and 1.

$u \leftarrow 1$; $v \leftarrow 1$; $w \leftarrow y$; $\quad \{u = F_l, v = F_{l+1}, l = 1\}$
repeat $\langle u, v \rangle \leftarrow \langle v, u + v \rangle$ **until not** $w \geq v$; $\quad \{u = F_l, v = F_{l+1}, y \geq u\}$
$\{u = F_l, v = F_{l+1}, l = \lambda y\}$
$l' \leftarrow 1$; $s \leftarrow 1$; $t \leftarrow t - t$;
repeat if $w \geq u$ **then**
begin $w \leftarrow w - u$; $t \leftarrow t + s$;

end;

$\langle u, \mathbf{v}) \leftarrow (\mathbf{v} - u, u\rangle$; $\langle r, s\rangle \leftarrow \langle s, r + s\rangle$; $\quad \{l \leftarrow l - 1\}$

until $u \geq v$.

Throughout this program we have $u = F_l$ and v = F_{l+1}, where l begins at 1, rises to λy, and returns to 1. During the second **repeat** statement we have also

$$r = F_{1+\lambda y-l}, \qquad s = F_{2+\lambda y-l}, \qquad t = (y - w)^R.$$

The program terminates with l = 1 and w = 0; hence we have

$$r = F_{\lambda y}, \qquad s = F_{\lambda y+1}, \qquad t = y^R.$$

The full program for x^y mod z can now be written as follows, using routines described earlier:

read x; **read** y; **read** z;
$\langle r, s, t\rangle \leftarrow \langle F_{\lambda y}, F_{\lambda y+1}, y^R\rangle$;
$x \leftarrow x \bmod z$; $w \leftarrow x$; $u \leftarrow 1$; $\quad \{x = x_l,\ w = x_{l+1},\ l = 1\}$
repeat if $t \geq r$ **then**
begin $t \leftarrow t - r$; $u \leftarrow (uw) \bmod z$;
end;
$\langle r, s\rangle \leftarrow (s - r, r\rangle$; (x, w) ← (w, (xw) mod $z\rangle$; $\quad \{l \leftarrow l + 1\}$
until $r \geq s$;
write u.

The invariant relations

$$x = x_l, \qquad w = x_{l+1}, \qquad r = F_{1+\lambda y-l}, \qquad s = F_{2+\lambda y-l}$$

are maintained in the final **repeat** loop as l increases from 1 to λy.

For example, if $y = 11 = 8+3 = F_6+F_4$, we have $\lambda y = 6$ and $y^R = F_2+F_4 = 1+3 = 4$. Hence r = 8, s = 13, t = 4, u = 1, and $x = w = x_0 \bmod z_0$ at the beginning of the final **repeat.** The registers will then contain the following respective values at the moments when the final **until** statement is encountered:

r	s	t	u	x	w
5	**8**	**4**	1	$x_0 \bmod z_0$	$x_0^2 \bmod z_0$
3	**5**	**4**	1	$x_0^2 \bmod z_0$	$x_0^3 \bmod z_0$
2	**3**	1	$x_0^3 \bmod z_0$	$x_0^3 \bmod z_0$	$x_0^5 \bmod z_0$
1	2	1	$x_0^3 \bmod z_0$	$x_0^5 \bmod z_0$	$x_0^8 \bmod z_0$
1	1	0	$x_0^{11} \bmod z_0$	$x_0^8 \bmod z_0$	$x_0^{13} \bmod z_0$

The statement '$u \leftarrow (uw) \bmod z$' can be implemented by first forming uw and then t aking the remainder mod z, using the multiplication and division algorithms presented earlier. But we can do better by changing the multiplication algorithm so that the quantities being added together for the final product are maintained modulo z: We simply change appropriate operations of the form $\alpha \leftarrow \alpha + \beta$ to the sequence

$$\alpha \leftarrow \alpha + \beta;$$
$$\textbf{if } \alpha \geq z \textbf{ then } \alpha \leftarrow \alpha - z.$$

Then the register contents never get large. In fact, if x_0 and y_0 are initially nonnegative and less than z_0, all numbers in the algorithm will be nonnegative and less than $2z_0$. We have proved the following result:

Theorem 2. *If* $0 \leq x, y < z \leq 2^{n-1}$, the quantity $x^y \bmod z$ can be computed *from* x, y, *and* z *with* $O((\log y)(\log z))$ *additions and subtractions of integers in the interval* $[0..2^n)$, *on a machine with finitely many registers.*

Indeed, the constant implied by this O is reasonably small. The algorithm just sketched may therefore find practical application in the design of special-purpose hardware for $x^y \bmod z$, which is the fundamental operation required by the RSA scheme of encoding and decoding messages [7].

Lower bounds. Some of the algorithms presented above can be shown to be optimal, up to a constant factor. For example, we obviously need $\Omega(\log \min(x, y))$ additions to compute the product xy; we cannot compute any number larger than $2^k \max(x, y)$ with k additions, and if $2^k < \min(x, y)$ this is less than $\min(x, y) \max(x, y) = xy$.

Logarithmic time is also necessary for division and gcd, even if we extend addition machines to *addition-multiplication machines* (which can perform multiplication as well as addition in one step). An elegant proof of this lower bound was given by L. J. Stockmeyer in an unpublished report [8]. We reproduce his proof here for completeness.

Theorem 3 (Stockmeyer). An *integer addition-multiplication* machine *requires* $\Omega(\log x)$ *arithmetic* operations to compute $\lfloor x/2 \rfloor$, $x \bmod 2$, *or* $\gcd(x, 2)$, *for infinitely many* x.

Proof: If we can compute $\lfloor x/2 \rfloor$ or $\gcd(x, 2)$ in t steps, we can compute $x \bmod 2 = x - 2\lfloor x/2 \rfloor = 2 - \gcd(x, 2)$ in at most $t + 2$ steps. So it suffices to prove that $x \bmod 2$ requires $\Omega(\log x)$ steps.

Any computation of an integer addition-multiplication machine on a given input x forms polynomials in x and compares polynomial values. A t-step computation defines at most 2^t different *computation paths*, depending on the results of **if** tests. For convenience we assume that each statement of the form '**write** w' is changed to

$$\textbf{if } 0 \geq w \textbf{ then write } w \textbf{ else write } w.$$

Then a program that computes x mod 2 must take a different path when x is changed to $x + 1$.

Each computation path is defined by a sequence of polynomial tests

$$q_1(x) : 0, \qquad q_2(x) : 0, \ . \ . \ . \ , \ q_s(x) : 0$$

made at times $t_1 < t_2 < \ . \ . \ . \ < t_s \leq t$. (Different paths have different polynomials in general, although q_1 (x) will be the same on each path.) If $q_j(x)$ corresponds to a test at time t_j, the degree of $q_j(x)$ is at most $2^{t_j - 1}$. Therefore the sum of the degrees of the $q_j(x)$ is less than 2^t. Therefore the total number of roots of all the polynomials $q_j(x)$, taken over all computation paths of length t, is less than 2^{2t}.

Let m be the least integer $\geq 2^{2t}$ such that none of the polynomials described in the previous paragraph has a root in the closed interval [m, m + 1]. Each root can esclucle at most two values of *m;* therefore m $\leq 2^{2t} + 2^{2t+1}$. By definition, the addition-multiplication program takes the same computation path when it is applied to x = m and to $x = m + 1$; therefore it is does not compute x mod 2 on both of these values. Therefore there is an integer x_t in the interval $[2^{2t}, 2^{2t+2}]$ such that the value x_t mod 2 has not been computed at time t on any of the computation paths. Therefore there are infinitely many x for which the time to compute x mod 2 is $\Omega(\log x)$.

So far we have counted both arithmetic operations and conditional tests as steps of the computation. This also gives a lower bound on the number of arithmetic operations, since we can assume without loss of generality that no computation path makes more than $\binom{k}{2}$ consecutive conditional tests when there are k registers. This completes the proof.

Notice that Stockmeyer's argument establishes the lower bound $\Omega(\log x)$ on the total computation time even if the number of registers is unbounded, and even if the programs are allowed to introduce arbitrary constants. A straightforward generalization of the proof shows that an integer addition-multiplication machine needs $\Omega(\log(x/y))$ steps to compute x mod y, uniformly for all $y > 0$ and for infinitely many x when y is given. However. the argument does not apply to machines with unbounded registers and indirect addressing; for this case Stockmeyer [8] used a more complex argument to obtain the lower bound $\Omega(\log x / \log\log x)$. It is still unknown whether indirect addressing can be exploited to do better than $O(\log a)$. When integer division is allowed, as well as addition and multiplication, the bound $\Omega(\log\log\log \min(x, y))$ on arithmetic operations needed to compute $\gcd(x, y)$ has been proved by Mansour, Schieber, and Tiwari [5].

Our efficient constructions have all been for addition machines that contain at least three registers. The following theorem shows that Z-register addition machines cannot do much:

Theorem 4. *Any algorithm that computes* $\gcd(x, y)$ on an *integer* addition machine *with only two registers* needs $\Omega(n - 1)$ operations to compute $\gcd(n, 1)$.

Lemma. *Consider a. graph on unordered pairs* $\{x, y\}$ *of nonnega tive integers, where* $\{x, y\}$ *is adjacent to* $\{x, x + y\}, \{x + y, y\}, \{|x - y|, y\}$, *and* $\{x, |x - y|\}$. *The shortest path from* $\{n, 1\}$ *to* $\{1, 1\}$ *in this graph has length* $n - 1$, *for all* $n \geq 1$.

Proof of the lemma (by Tomás Feder). Consider the following four operations on unordered pairs $\{x, y\}$:

$\underline{A}$. Replace $\min(x, y)$ by $x + y$.

$\overline{A}$. Replace $\max(x, y)$ by $x + y$.

$\underline{S}$. Replace $\min(x, y)$ by $\max(x, y) - \min(x, y)$.

$\overline{S}$. Replace $\max(x, y)$ by $\max(x, y) - \min(x, y)$.

Then $\underline{A}\,\overline{S} = \overline{A}\,\underline{S} = \underline{S}\,\underline{S}$ = identity and $\underline{S}\,\overline{S} = \overline{S}$. Furthermore $\underline{S}$ is either $\overline{S}\,\underline{A}$ or $\underline{S}\,\overline{A}$, hence $\underline{A}\,\underline{S}$ and $\overline{A}\,\underline{S}$ are either $\underline{A}$ or $\overline{A}$. Any minimal sequence of operations must therefore begin with $\overline{S}$'s and end with A's. But $\overline{S}^k$ applied to $\{n, 1\}$ yields $\{n - k, 1\}$, for $k < n$; and A's do not decrease anything. Therefore the shortest path is $\overline{S}^{n-1}$.

Proof of the theorem. As in the proof of Theorem 3, the sequence of **if** tests made by an addition machine defines a computation path, dependent on the inputs. We say that the test **if** $x \geq y$' is *critical* if it is performed at a moment when the contents of registers x and y happen to be identical.

Let M be a Z-register addition machine that produces the output $M(a, b)$ when applied to inputs $\langle a, b\rangle$. We assume that a and b are initially present in the two registers; therefore the computation path corresponding to $\langle a, b\rangle$ will be the computation path corresponding to $\langle ma, mb\rangle$ for a.11 integers $m \geq 1$.

Every computation path defines constants α and β such that $M(a, b) = \alpha a + \beta b$ for all $\langle a, b\rangle$ leading to this path. If M never encounters a critical test when applied to $\langle a, b\rangle$, it will follow the same path on inputs $\langle am, bm\rangle$ and $\langle am + 1, bm\rangle$ for all sufficiently large values of m. Therefore we will have $M(am + 1, bm) = M(am, bm) + \alpha$ for all large m; and M cannot. be a valid program for computing the gcd. We have proved that every Z-register gcd program must make a critical test before it produces an output.

Nest we show that every Z-register gcd machine must make a critical test before it uses any instruction of the form $x \leftarrow x - x$ or $x \leftarrow x + x$. Suppose M performs such an instruction when it is applied to inputs $\langle a, b\rangle$; these inputs determine a computation path defining constants α and β such that the other register, y, contains $\alpha a + \beta b$ when $x \leftarrow x - x$ or $x \leftarrow x + x$ is performed. If no critical tests have occurred, the same computation path will be followed when the inputs are $\langle a^2 bm + 1, ab^2 m\rangle$ and $\langle a^2 bm, ab^2 m + 1\rangle$, for a.11 sufficiently large m. But $\gcd(a^2 bm + 1, ab^2 m) = \gcd(a^2 bm, ab^2 m + 1) = 1$; hence y must contain an odd value when M is applied to $\langle a^2 bm + 1, ab^2 m\rangle$ or $\langle a^2 bm, ab^2 m + 1\rangle$. (If y is even when x is being set, to $x - x$ or $x + x$, both registers will contain an even value; hence M cannot subsequently output '1'.) Hence $\alpha(a^2 bm + 1) + \beta(ab^2 m)$ and $\alpha(a^2 bm) + \beta(ab^2 m + 1)$ are

oclcl, for all sufficiently large m; hence α and β are both odd. But $\gcd(2a^2bm + 1, 2ab^2\ m + 1)$ is odd, and the inputs $\langle 2a^2\ bm + 1, 2ab^2m + 1\rangle$ follow the same path as $\langle a, b\rangle$ for all large m; hence $\alpha(2a^2bm + 1) + \beta(2ab^2m + 1)$ must be oclcl, a contradiction.

Therefore every %-register gcd machine must make a critical test, before which it has performed only operations of the forms $x \leftarrow x \pm y$, y $\leftarrow$ y $\pm$ x. Such operations correspond to the transformations considered in the lemma..

Suppose M is applied to the inputs **(n.** 1). When the first critical test occurs, we have x = y; and gcd(x, y) = gcd(n, 1) = 1, because gcd(x, y) is preserved by all of the operations x $\leftarrow$ x $\pm$ y or $y \leftarrow y \pm$ x that have been performed so far. Thus $x = y = \pm 1$; the algorithm must have followed a path from $\{n, 1\}$ to $\{1, 1\}$ in the sense of the lemma,. So the algorithm must have performed at least $n - 1$ operations before reaching the first critical test. This completes the proof.

Further restrictions. A "minimalist" definition of addition machines would eliminate the copy operation x $\leftarrow$ y, because this operation can be achieved by

$$x \leftarrow \text{x} - x\,; \qquad x \leftarrow \text{x} + y\,.$$

We can also simplify the **if** tests, allowing only the one-register form **'if** $x \geq 0$**';** because a general two-register comparison **'if** x $\geq$ y **then** α **else** β' can be replaced by

x +--x-y;
if x $\geq$ 0 then begin x $\leftarrow$ x + y; α end
else begin $x \leftarrow$ **x + y;** β **end.**

Similarly, we can do away with addition, if we add a new register ξ, because $x \leftarrow x + y$ can be achieved by three subtractions:

$$\xi \leftarrow \xi - \xi\,; \qquad \xi \leftarrow \xi - y\,; \qquad x \leftarrow x - \xi\,.$$

Addition cannot be eliminated without increasing the number of registers, in general. For we can prove that the operation $x_1 \leftarrow x_1 + x_2$ cannot be achieved by any sequence of operations of the forms $x_i \leftarrow x_i - x_j$, for $1 \leq i.\ \text{j} \leq r$. The proof can be formulated in matrix theory as follows:

Let E_{ij} be the matrix that is all zeroes except for a 1 in row i and column j. We want to show that the matrix $I + E_{12}$ cannot be obtained as a product of matrices of the form $I - E_{ij}$. Clearly we cannot use the matrices $I - E_{jj}$, whose determinant is zero: so we must use $I - E_{ij}$ with $i \neq$ j. But the inverse of $I - E_{ij}$ is $I + E_{ij}$, when $i \neq j$. So if

$$I + E_{12} = (I - E_{i_1 j_1}) \ldots (I - E_{i_m j_m})$$

we have, taking inverses,

$$I - E_{12} = (I + E_{i_m j_m}) \dots (I + E_{i_1 j_1}),$$

which is patently absurd since the right side contains no negative coefficients.

Open questions.

1) Can the upper bound in Theorem 1 be replaced by $8 \log_\phi N + \beta$?

2) Can an integer addition machine with only 5 registers compute x^2 in $O(\log x)$ operations? Can it compute the quotient $\lfloor y/z \rfloor$ in $O(\log y/z)$ operations?

3) Can an integer addition machine compute $x^y \bmod z$ in $o((\log y)(\log z))$ operations, given $0 \leq x, y < z$?

4) Can an integer addition machine sort an arbitrary sequence of positive integers $\langle q_1, q_2, \dots, q_m \rangle$ in $o((m + \log q_1 q_2 \dots q_m) \log m)$ steps?

5) Can the powers of 2 in the binary representation of x be computed and output by an integer addition machine in $o(\log x)^2$ steps? For example, if $x = 1.3$, the program should output the numbers S, 4, 1 in some order.

6) Is there an efficient algorithm to determine whether a given r x r matrix of integers is representable as a product of matrices of the form $I + E_{ij}$?

Acknowledgment. We wish to thank Baruch Schieber for calling our attention to Stockmeyer's paper [7].

References

[1] Donald E. Knuth and Richard H. Bigelow, "Programming languages for automata," *Journal* of the *ACM* 14 (1967), 615-635.

[2] Donald E. Knuth, The *Art of Computer Programming,* Vol. 2: *Seminumerical* *Algorithms,* (Reading, Mass.: Addison-Wesley 1969).

[3] Donald E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching* (Reading, Mass.: Addison-Wesley 1973).

[4] Donald E. Iinuth, "Structured programming with **goto** statements," Computing *Surveys* 6 (1974), 261–301.

[5] Yishay Mansour, Baruch Schieber, and Prasoon Tiwari, "Lower bounds for integer greatest common divisor computations," *Proc. 29th* IEEE Symp. *Foundations of Compu ter Science* (1988), 54-63.

[6] Yuri V. Matijasevich, "Enumerable sets are cliophantine," *Dokl. Akad. Nauk SSSR* 191 (1970), 279–282; s*oviet Math. Dokl.* 11 (1970), 354-357.

[7] R. L. Rivest, A. Shamir, and L. Adleman, "A met hod for obtaining cligi t a.1 signatures and public-key cryptosystems," *Comm. ACM* 21 (1978), 120–126.

[S] Larry J. Stockmeyer, "Arithmetic versus Boolean operations in idealized register machines," IBM Thomas J. Watson Research Center report RC 5954, 21 April 1976.

[9] E . Zeckendorf, "leprésentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas," *Bull. Soc. Roy. Sci. Liège* 41 (1972), 179–182.