MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A. I. LABORATORY

Artificial Intelligence
Memo #257

May 1979

A TWO COUNTER MACHINE CANNOT CALCULATE 2^N

Rich Schroeppel

Work reported herin was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0003.

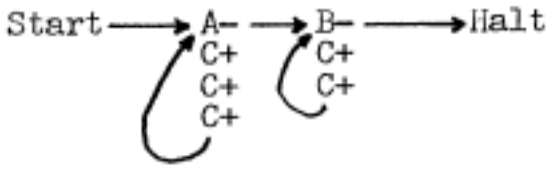
This note proves that a two counter machine cannot calculate 2^N.

An N counter machine has N counters, each of which contains a nonnegative integer. There is also a program, with four kinds of instructions:

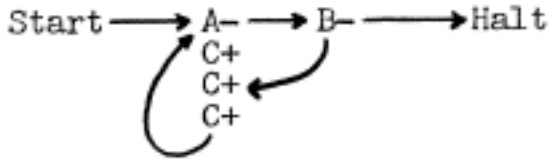
Type 1: Add 1 to a specified counter. Written C+.
Type 2: Examine a specified counter. If it contains 0, don't change it, but jump to an instruction out of the normal sequence. Otherwise, subtract 1 from the counter and continue in the normal program sequence. Written C-, with an arrow at the right indicating where to go in the zero case. This arrow is called a zero branch arrow.
Type 3: (Unconditional) jump. Written with an arrow.
Type 4: Halt.

Counter machines are also known as register machines or program machines. I use "counter" and "register" interchangeably, but usually a counter is incremented or decremented, while a register is subject to more exotic transformations, such as doubling.

Example: A three counter machine (3CM) computes 3X+2Y. At Start, X is in counter A, Y is in counter B, and counter C is zero. At Halt, the answer is in C. (In the diagrams, the normal direction of program flow is down the page.)

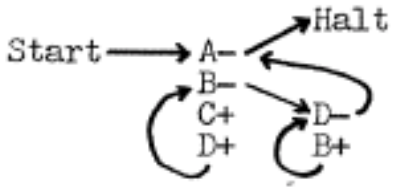


This can be bummed to:



(a,b) is the greatest common divisor of a and b.
d|n means d is a divisor of n.
p usually denotes a prime number.
x<=y means x is less than or equal to y.
[x] is the greatest integer <=x.
sqrt(x) is the square root of x.
log2 x is the logarithm of x to the base 2.
Fib(N) is the Nth Fibonacci number.
F(S), where F is a function and S a set, is {F(e) | e in S}.

Example: Multiply (the contents of) counter A by (the contents of) counter B; the answer is in C. C and D are zero at Start.



Hard problem: Multiply two numbers using only three counters.
Easier problem: Square a number using three counters.
The solutions are at the end of the memo.

Theorem: A three counter machine can simulate a Turing machine.

Proof (abridged): The counter machine has counters A, B, and C. The Turing machine alphabet is two characters, 0 and 1. There are only a finite number of 1s on the tape.

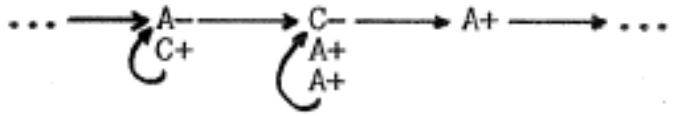
The tape is split into three pieces: Tape to the left of the read head, tape to the right of the read head, and the tape square under the read head. The left half of the tape is interpreted as a binary number, which goes into counter A. Counter B represents the right half of the tape, interpreted as a binary number in reverse. Thus, if the tape contained

...0000001101 1 011010000000...

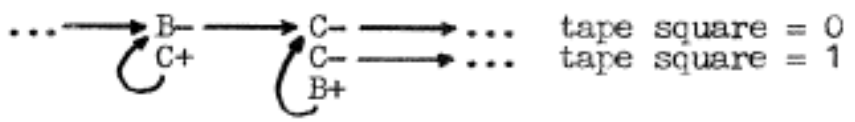
counter A would contain 13, and counter B would contain 22. The 1 under the read head would be reflected in what part of its program the counter machine was executing.

Each TM operation is simulated by an open subroutine in the 3CM. Suppose our TM is to print a 1 and move the read head one square to the right:

To simulate writing a 1 on the left half of the tape, the 3CM doubles the contents of A and adds 1. Counter C is used as a temporary in this operation.



To simulate moving the read head right, reading one square of tape, counter B is divided by two and the remainder tells what character was on the tape. Again, C is used as a temporary.



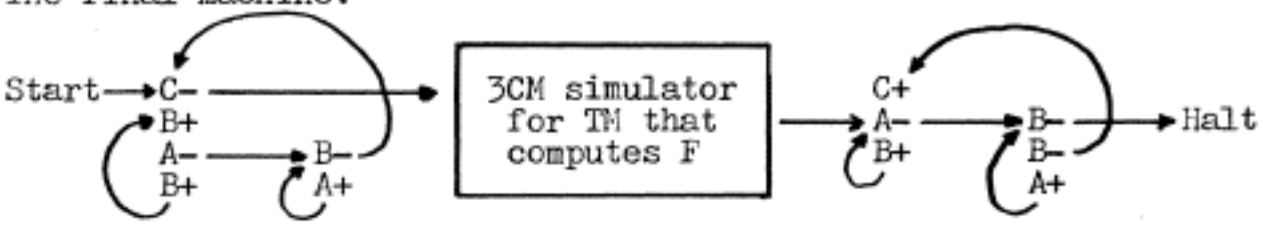
Each (state, character under read head) pair of the Turing machine corresponds to about 10 counter machine instructions (ignoring jumps). The proof can be modified to allow a larger alphabet by changing the radix of the simulation. QED

If the Turing machine has an argument, the usual convention is to use "unary" notation. The argument is written as a string of 1s on the tape immediately to the left of the read head. The argument convention for CMs is to put the argument in one of the counters. When a CM computes a function F(N), it starts with N in a specified counter, with the other counters zero. If F(N) is defined, the CM halts with F(N) in a particular counter. If F(N) is not defined, the CM does not halt.

Theorem: A 3CM can compute any partial recursive function of one variable. It starts with the argument in a counter, and (if it halts) leaves the answer in a counter.

Proof: There is some Turing machine which computes the given function. Construct a 3CM to simulate that TM. An argument of N is given to the TM as a string of N 1s immediately to the left of the read head. The simulating 3CM will expect counter A to contain a binary number consisting of N consecutive 1s; i.e., $2^N - 1$. But our theorem requires that the 3CM start with N. So we attach an input converter at the beginning of the simulator. The converter changes N into $2^N - 1$. If the TM halts, its answer will be a string of F(N) consecutive 1s immediately to the left of the read head. So we attach an output converter to the simulator, which converts $2^{F(N)} - 1$ to F(N).

The final machine:



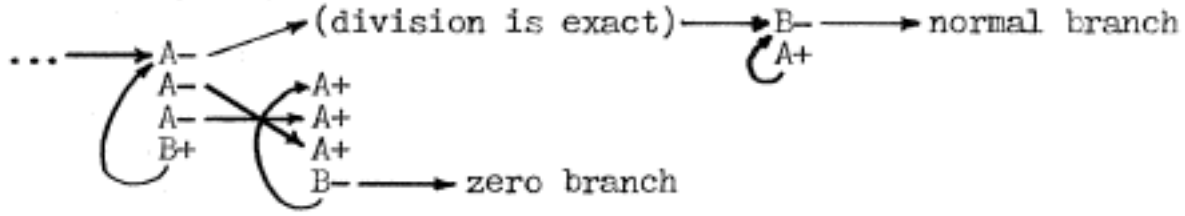
The machine starts with N in counter C, and A and B zero. If it halts, F(N) is in counter C.

Theorem: Any counter machine can be simulated by a 2CM, provided an obscure coding is accepted for the input and output.

Proof: Suppose the 2CM has counters A and B, and that we want to simulate a 4CM with counters named W, X, Y, and Z. The contents of all four counters are coded into one number, $2^W 3^X 5^Y 7^Z$. This number is kept in counter A. Counter B is used as a temporary. Each 4CM instruction is simulated by an open subroutine in the 2CM.

To simulate X+, the number in counter A is multiplied by 3.

To simulate X-, the number in A is divided by 3, the quotient going into B. If the division is exact, B is moved to A. If the division is not exact, the original number in A is restored, and the simulated zero branch of the X- instruction is taken.



QED

Note for future use: We need only three counters to simulate a Turing machine. If we are simulating a 3CM that computes a function F(N), the 2CM input would be $2^N 3^0 5^0$, or 2^N. The 2CM output would be $2^{F(N)}$. A peculiarity of this simulation scheme is that if the 2CM is started with $2^N K$, where (K,30)=1, it will halt with $2^{F(N)} K$.

Corollary: The Halting Problem for 2CMs is unsolvable.

Corollary: A 2CM can compute any partial recursive function of one argument, provided the input is coded as 2^N and the output (if the machine halts) is coded as 2^{answer}.

This corollary provides the rationale for the Input Problem and the Output Problem.

The Input Problem: Find a 2CM that computes 2^N when started with N in one counter.

The Output Problem: Find a 2CM that computes N when started with 2^N in one counter.

The goal of these two problems is to make a 2CM without the coding restrictions in the corollary. This would be done by putting a TM simulator in between an input converter and an output converter. This memo proves that the Input Problem is impossible. The Output Problem seems to involve some difficult number theory questions.

Before proceeding with the impossibility proof, we should mention another way around the coding difficulties. Another convention sometimes used for arguments is to have a special instruction which reads one character of the argument, and branches one of two ways. (This assumes a two character input alphabet.) If our 2CM gets its input this way, and writes its answer with similar "write one character of output" instructions, then the coding problems go away.

Next, we introduce the idea of an MP1RM. The best way to think about 2CMs is to think about MP1RMs instead.

The Replacement Lemma: Any two counter machine started with zero in one counter can be simulated by a more powerful one register machine (MP1RM). The register is denoted by R, and contains a nonnegative integer, also denoted by R. The MP1RM has a better instruction set than a CM. The instructions available are:

Add K	Adds a constant K to the register.
Mul K	Multiplies the register by K.
Sub1	If R=0, jump out of the normal instruction sequence. If R>0, subtract one from R. (This is the same as the counter machine instruction R-.)
Div K	Divide R by K>0, the quotient going into R. Jump to one of K different places, depending on the remainder. A divide instruction has K arrows coming out of it.
Jump	Jump to someplace else in the program.
Halt	Halt; the answer is in R.

K denotes a nonnegative integer; K may be different in different instructions; the program may not alter any K.

We introduce the notation Jump* for the infinite loop Jump  .

Proof of the Replacement Lemma:
Before reading the proof, it is well to have the main point in mind. Besides adding or subtracting constants from the counters, a 2CM can carry out only one useful kind of computation step:

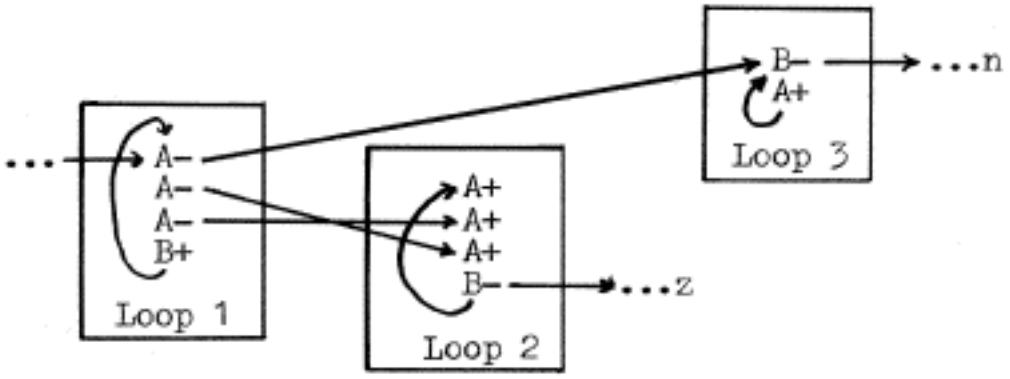
Multiply the contents of one counter by some rational number P/Q, putting the result in the other counter. (The first counter is cleared.) Jump to one of Q different places, depending on the remainder mod Q.

This computation step is accomplished by a strong loop. The first part of the proof changes the 2CM into a form that makes the proof of this statement easy.

Let's look at the 2CM we have to simulate. Assume the counters are A and B, and that B is zero at Start.

First we define the phrase strong loop. Suppose that we take the diagram of the 2CM program, and temporarily erase all zero branch arrows. Any loops remaining are called strong loops. A strong loop will be some sequence of A+, A-, B+, B-, and Jump instructions. Note that strong loops cannot have subloops, for the inside loop would have to exit with a zero branch arrow, which would be part of the strong loop.

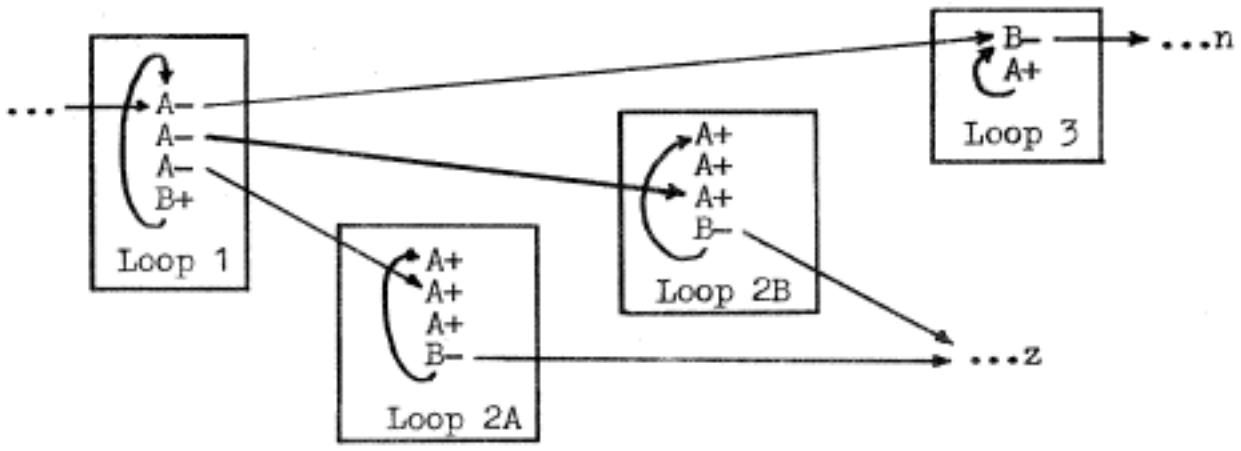
(Now restore the erased arrows.)



In the diagram, there are three strong loops.

An instruction in a strong loop is called an entry point if it can be reached (in one instruction) from Start, or from some instruction outside the loop, or even from a zero branch of a "-" instruction within the loop. In the diagram above, Loops 1 and 3 have one entry point, and Loop 2 has two.

We want to alter the 2CM so that all strong loops have only one entry point. For each strong loop with more than one entry point, make as many copies of the loop as there are excess entry points. Each copy, and the original, has a different entry point designated as its single entry point. Improper entries into the original loop are disconnected and routed to the appropriate copy. Exits from the copies are routed together with the corresponding exits in the original loop. This transformation will cause a lot of exit arrows to be copied, but will not create any new strong loops (except the copies), or any new entry points to any strong loop. The modified program will be somewhat larger than the original, but will have the same computational behavior.



A regular loop is a single entry strong loop in which all "-" instructions come immediately after the entry point, and precede all "+" instructions. The point after the -s and before the +s is called the midpoint of the loop. A single entry strong loop can be made regular by some finite sequence of instruction exchanges of Types 1 and 2.

Type 1:
X+
X- ⟶ ...
=
delete

Type 2:
X+
Y- ⟶ ...
=
Y- ⟶ X+ ⟶ ...
X+

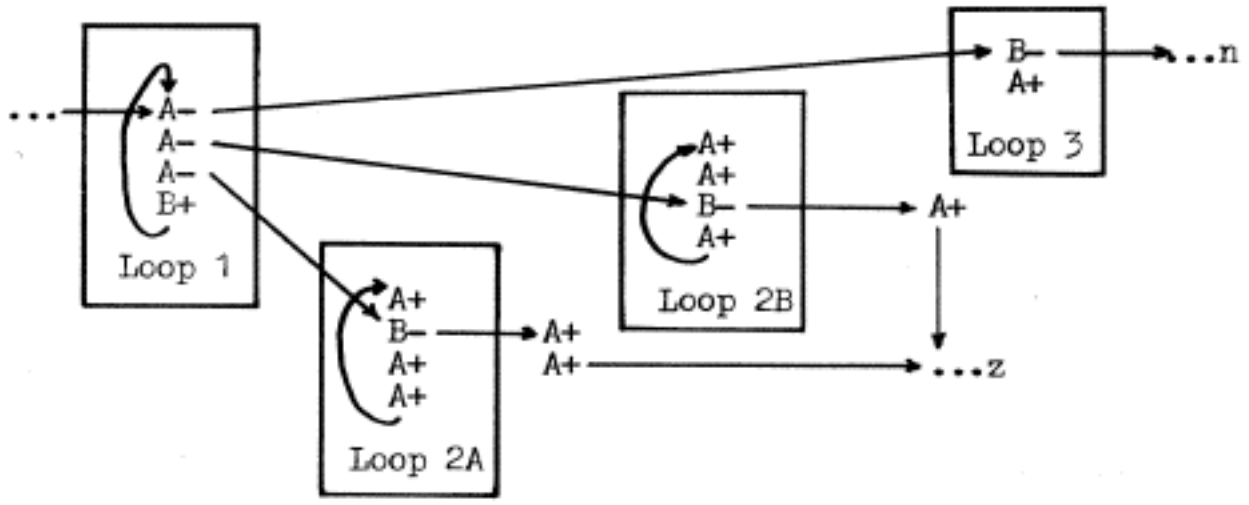
Type 3:
X+
Y+
=
Y+
X+

Type 4:
X- ⟶ ...u
Y- ⟶ ...v
=
Y- ⟶ X- ⟶ ...u
X- ⟶ Y+ ⟶ ...v

Type 5:
X- ⟶ ...
X+
=
no replacement

Type 6:
X- ⟶ ...
Y+
=
Y+
X- ⟶ Y- ⟶ (impossible)
...

Types 3-6 are included for completeness. Note that types 2, 4, and 6 are invalid if X and Y are the same counter. The exchanges are also invalid if the second instruction of the pair is an entry point. Since we have made our loops have only one entry point, the necessary exchanges can be carried out without crossing the entry point.



Suppose all strong loops are regularized. The program may be a little longer, but no new entry points are created, nor any new strong loops.

The number of instructions in the expanded 2CM is denoted by I.

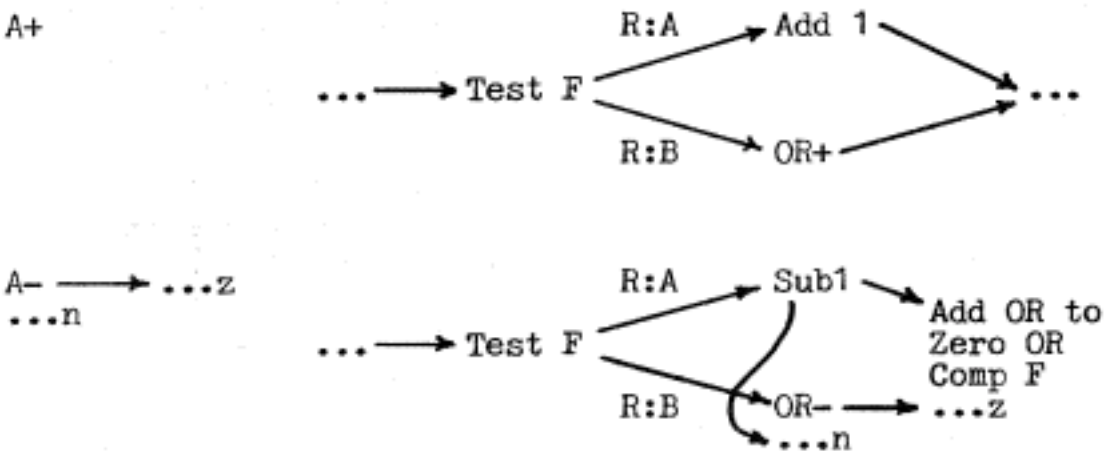
Next, we introduce an intermediate machine: The EMP1.5RM (Even More Powerful 1.5 Register Machine). We will prove that a 2CM can be simulated by an EMP1.5RM, and that an EMP1.5RM can be simulated by an MP1RM.

An EMP1.5RM is an MP1RM with a certain amount of additional memory. It has a flag F indicating whether R is representing counter A or counter B at the moment. It also has another register called OR (for Other Register) which is capable of holding any integer from 0 to I+1. OR will represent whichever counter is not repesented by R. We also include the additional instructions OR+, OR-, Add OR to R, Set OR to zero, Test F, and Complement F.

OR+ is illegal if OR contains I+1. At various places in the proof it is necessary to verify that OR<=I. To help accomplish this, we will specify that whenever the EMP1.5RM simulates the 2CM taking a zero branch arrow, the condition OR=0 is satisfied. Taking a zero branch arrow implies at least one counter is zero; if OR is representing this counter, fine. If not, we exchange OR and R (and complement F).

Now the simulation: At Start, R represents A, OR represents B (which contains zero), and F reflects R:A. 2CM instructions outside of strong loops are translated to equivalent EMP1.5RM routines:

Jump represents Jump, and Halt, Halt.



Similarly for B+ and B-. (This translation introduces a Test F instruction for every 2CM instruction. We could remove most of the Test F instructions by rearranging the flow of control slightly.)

We have blithely translated A+ to a routine containing OR+. To make this step valid, we must verify that OR<=I. In fact, we assert that OR<=I at all times:
First, OR=0 in three cases:

1) At Start;
2) Whenever a zero branch arrow is simulated;
3) When simulating an exit from a strong loop. (When we specify the simulation of a strong loop, this condition will be met.)

At any place in the program outside of a strong loop, no more than I 2CM instructions have been executed since one of these events occurred. (For, if I+1 instructions had been executed, some instruction would have been executed twice. But the path taken between the two executions of the repeated instruction could not contain a zero branch arrow; so the path would be a strong loop. But we are outside of a strong loop.) Since at most I 2CM instructions have been executed since OR was 0, it could have increased to at most I.

To simulate a regular loop: Determine the six constants Aplus, Bplus, Aminus, Bminus, dA, and dB. Aplus is the number of A+ instructions in the loop; Aminus is the number of A- instructions; and dA is the net change in A in one cycle of the loop.
dA = Aplus - Aminus. Similarly for Bplus, Bminus, and dB.
At the entry point of the loop:
1) OR<=I by the argument above.
2) The program will exit instead of completing the next cycle if and only if A<Aminus or B<Bminus.

For the first section of the loop, from the entry point to the midpoint, we use the substitution given above for "-" instructions. If the program exits from the loop in this section, it will satisfy the OR=0 condition, since the "-" routine does. F will not be changed unless the loop exits, so we can remove all but one of the Test F instructions if we desire.
Suppose that the program reaches the midpoint of the loop. We do a Test F; each branch from the test transfers to a copy of the appropriate routine listed below. We will describe the R:A and OR:B branch.
Since F has not changed since the loop was entered, OR:B at the entry point. Therefore, when the loop was entered, B<=I.

There are three cases:

Case 1: dA >= 0 and dB >= 0. Then the 2CM program will loop indefinitely. Replace code here with Jump*.

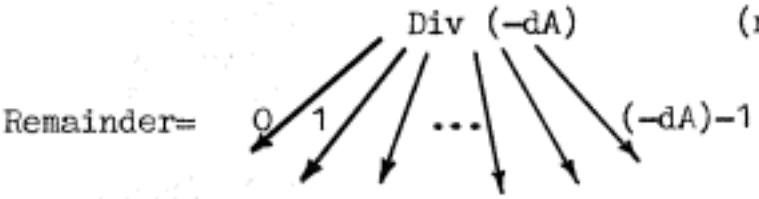
Case 2: dB < 0. We may just copy the rest of the loop, translating A+ to Add 1 and B+ to OR+. When the program gets back to the loop's entry point, B will be less than it was when the loop was entered. (Specifically, B at return to entry point = B at original entry point + dB.) But B has been increasing since the midpoint, so it is <=I everywhere in the loop. Thus, OR<=I everywhere in the simulated loop. When the loop finally exits, it will take a zero branch arrow, and satisfy the OR=0 condition. Note that we cannot say without more calculation whether the exit will be via an A- or a B-.

Case 3: dA < 0 and dB >= 0. If the program reaches the midpoint the first time around the loop, it will never exit via a B-, since B will not decrease on successive cycles of the loop.

Suppose A=Amid and B=Bmid at the midpoint. One cycle of the loop adds dA to A and dB to B. After K cycles, counting from midpoint to midpoint,

$$A = Amid + K\,dA = Amid - K\,(-dA), \text{ and } B = Bmid + K\,dB.$$

(Since dA<0, -dA will be positive.) The 2CM program will return to the midpoint exactly L=[Amid/(-dA)] times—— as long as A>=0 at the midpoint. The routine below simulates L cycles of the loop, from midpoint to midpoint. We close off the loop by simulating the 2CM going through the +s for the last time, and then jump to the entry point.



(A copy of this routine appears at the end of every divide arrow.)

```
Mul dB
Add OR to R          (R contains Bmid+LdB)
Comp F
Set OR to Zero
Set OR to Rem        (many OR+s)
                     (OR=Rem=Amid-L(-dA)=Amid+LdA)
Add Bplus (to R)     (These instructions simulate
                     the 2CM going through the +s
                     for the last time.)
Add Aplus to OR      (more OR+s)
Jump to the top of   the loop
```

The loop will exit before returning to the midpoint again. We must verify that OR<=I when we jump to the top of the loop:

$$OR = Aplus+Rem < Aplus+(-dA) = Aplus-(Aplus-Aminus) = Aminus <= I$$

This also confirms that the loop will exit before reaching the midpoint again. The exit will be via an A-, and the OR=0 condition will be satisfied.

We have taken care of both regular loops, and instructions outside of regular loops. EMP1.5RM simulates the regularized 2CM.

But we can convert the EMP1.5RM to an MP1RM. For each state of the EMP1.5RM, we create 2(I+2) MP1RM states; one state for each combination of values of F and OR. The instructions OR+, OR-, Zero OR, Test F, and Comp F become Jumps between different MP1RM states. Add OR to R becomes an Add K. QED

Addendum to the Replacement Lemma: The Ks in the MP1RM need never be greater than the number of instructions in the original 2CM program.

Proof: Add K can always be broken up into a group of Add 1s. For Mul K and Div K, we note that the construction given above for regularizing strong loops never makes a strong loop longer than it was in the original 2CM. The Ks in the Mul and Div instructions are never greater than the length of the regular loop they simulate, which is in turn bounded by the size of the 2CM being simulated. QED

Theorem: An MP1RM can be simulated by a 2CM.

Corollary: F(N) is compuatable by a 2CM if and only if it is computable by an MP1RM.

Notice that if the program of an MP1RM ever takes the zero branch of a Sub1 instruction, the total state of the machine is known. We know that R is 0, and we know which instruction the MP1RM is about to execute. We could replace the code following the zero branch with code setting R to the final answer and halting. (Or going into an infinite loop, if that is what the code in the original machine would do.) The code would be "Add K, Halt" or "Jump*". A similar argument shows that we can replace code following a Mul 0 instruction.

A Sub1 zero branch or a Mul 0 instruction is called evaluated if it is immediately followed by either "Add K, Halt" or "Jump*". (We may change Mul 0, Jump* to Jump*.) Evaluation of a Sub1 zero branch often allows us to ignore the branching of a Sub1 instruction, and to consider only the main line of code.

Theorem: There is no two counter machine that calculates 2^N.

Proof sketch: Assume that we have a 2CM which computes 2^N from N. We convert it to an equivalent MP1RM, and evaluate Sub1 zero branches. (Most of these branches will probably have powers of two as their answers.)

Since there are infinitely many powers of two, and only finitely many Sub1 instructions, we can find some N such that when the MP1RM is started with N, it halts with 2^N, but the Halt instruction is not at the end of a Sub1 zero branch.

Examine the path taken through the program. This path will always take the nonzero branch of any Sub1 instruction that it goes through.

We make a note of all the divide instructions the path goes through. Multiply together all the divisors, counting a divisor several times if the path goes through the instruction several times. Call the product D. D is not zero, since no divisor is zero. Now look at the multiply instructions in the path; multiply together the multipliers, counting repeated multipliers as with divides; call the product M.

Now we claim: Start the MP1RM with N + jD, j>=0. Then the MP1RM will follow the same path as it did when started with N, and will get as answer $2^N + jM$. (The proof of this claim is spelled out in the next theorem.)

But this is impossible, for the function 2^x grows more rapidly than any arithmetic progression. Hence the proposed 2CM cannot exist. QED

Arithmetic Progression Theorem: Suppose a 2CM computes some function F(N). (F may be partial.) Then the range of F contains a finite subset S, such that for any N for which F(N) is defined and outside of S, there exist D>0 and M>0 such that

$$F(N+jD) = F(N) + jM \qquad \text{for all } j>=0.$$

Moreover, D and M have no prime factors greater than I, the number of instructions in the 2CM.

The theorem is vacuous unless F has infinite range.

Proof: Replace the 2CM by an equivalent MP1RM. Evaluate Sub1 zero branches and Mul 0 instructions. Take S to be the set of values at the end of Sub1 zero branches or Mul 0 instructions.

Consider an N for which F(N) is defined and not in S. Start the MP1RM with N. It will halt after some time t with R=F(N). We must supply an M and D such that F(N+jD) = F(N)+jM.

Look at the path taken by the computation. Suppose the nth instruction in the path is denoted by I(n) or by In; 1<=n<=t; I(t) is Halt.

Define Mn, the nth multiplier, by:

If In = Mul K, then Mn=K. Otherwise, Mn=1.

Similarly, the nth divisor, Dn, is defined by:

If In = Div K, then Dn=K. Otherwise, Dn=1.

Rn is the contents of R after executing In. R0 is the starting value of R.

Put M = M1 M2 M3 ... Mt and D = D1 D2 D3 ... Dt.

M>0, since all the Mi must be >0. If some Mi were 0, the MP1RM would halt immediately, with R containing an element of S. But F(N) is not in S. D>0, since all Di>0. Each Mi and Di is <=I, so M and D have no prime factors >I. Now we claim:

> If the MP1RM is started with N+jD, it follows the same path as with N, and it Halts with R = F(N)+jM.

Let Nn be the value of Rn when the machine is started with N. Then N0=N and Nt=F(N). The proof of the claim is by induction, along the common path of the computations of F(N) and F(N+jD), of the formula

$$Rn = Nn + j\ M1\ M2 \ldots Mn\ D(n+1) \ldots Dt. \qquad \text{(EQ1)}$$

First, R0 = N0 + jD, by hypothesis. Now suppose that we have verified that EQ1 is true for n-1, and that both computations have followed the same path so far. We consider all possible cases of what the next instruction, In, might be:

If In is Add K: $Nn = N(n-1) + K$, and $Mn = Dn = 1$.

$$Rn = R(n-1) + K$$
$$= N(n-1) + j\ M1 \ldots M(n-1)\ Dn\ D(n+1) \ldots Dt + K$$
$$= Nn + j\ M1 \ldots M(n-1)\ Mn\ D(n+1) \ldots Dt$$

If In is Sub1: $N(n-1) > 0$, since the computation of F(N) always takes the nonzero branch of Sub1 instructions. $Nn = N(n-1) - 1$. All Mi and Di are >0; j>=0; so $R(n-1) >= N(n-1) > 0$; hence the R computation will take the nonzero branch also. $Rn = R(n-1) - 1$.

If In is Mul K: $Nn = N(n-1) * K$ $\quad Mn = K,\ Dn = 1$

$$Rn = R(n-1) * K$$
$$= \{N(n-1) + j\ M1 \ldots M(n-1)\ Dn \ldots Dt\} * K$$
$$= Nn + j\ M1 \ldots M(n-1)\ K\ 1\ D(n+1) \ldots Dt$$
$$= Nn + j\ M1 \ldots M(n-1)\ Mn\ D(n+1) \ldots Dt$$

If In is Div K: $Dn = K,\ Mn = 1$

$Nn = [N(n-1)/K]$ and the remainder is $N(n-1) - K\ Nn$.

$Rn = [R(n-1)/K]$ and the remainder is $R(n-1) - K\ Rn$.

$$Rn = [\ \{N(n-1) + j\ M1 \ldots M(n-1)\ Dn \ldots Dt\} / K\ ]$$
$$= [N(n-1)/K] + j\ M1 \ldots M(n-1)\ D(n+1) \ldots Dt$$
$$= Nn + j\ M1 \ldots M(n-1)\ Mn\ D(n+1) \ldots Dt$$

$$R(n-1) - K\ Rn = N(n-1) + j\ M1 \ldots M(n-1)\ Dn \ldots Dt - K\ (Nn + j\ M1 \ldots Mn\ D(n+1) \ldots Dt)$$
$$= N(n-1) - K\ Nn$$

The remainders are the same in both computations, so they will take the same branch out of the divide instruction.

If In is Jump or Halt, Mn=Dn=1, so EQ1 remains true.

We have proved that all types of instruction preserve EQ1, and that when instructions which branch are executed, both computations will take the same branch. So at time t, both computations will halt. $Rt = Nt + j\ M1 \ldots Mt = F(N)+jM$, as required. QED

Arithmetic Series Condition: Let F(N) be a (perhaps partial) function. Let Sp, where p is prime, be the set of all N satisfying the following two conditions:

(1) N is a nonnegative integer, and F(N) is defined.
(2) For all D>0 with no prime factors >p, if F(N+jD) is defined for all j>=0, then it is a nonlinear function of the j>=0.

Let Sp′ be the set of all N satisfying (1) and (2′):

(2′) For all D>0 with no prime factor >p, F(N+jD), for those values of j>=0 at which it is defined, is a nonlinear function of j.

(So Sp′ is a subset of Sp. Also, if q is a prime >p, then Sp contains Sq, and Sp′ contains Sq′.) Suppose that for all p, F(Sp) is an infinite set. Then F is not computable by a 2CM. Furthermore, if, for all p, F(Sp′) is an infinite set, then no extension of F is 2CM computable.

This theorem is essentially a contrapositive of the Arithmetic Progression Theorem.

Proof: Suppose that F(Sp) is infinite for all p, and that some 2CM computes F. Then by the AP Theorem, there is a finite set S such that if F(N) is not in S, there are D>0 and M>0 such that F(N+jD)=F(N)+jM. Choose a prime p which is greater than the number of instructions in the 2CM. Choose an N in Sp such that F(N) is not in S. Find D and M. Then F(N+jD)=F(N)+jM, and D has no prime factor >p. This contradicts (2).
Suppose now that F(Sp′) is infinite for all p, and that some 2CM computes G, an extension of F. Find S such that G(N) defined and outside S implies the existence of D and M. Choose an N in Sp′ such that F(N) is not in S. Since G is an extension of F, G(N)=F(N). So G(N) is not in S. So there exist M>0 and D>0, D having no prime factor >p, such that G(N+jD)=G(N)+jM, for all j>=0. Hence any restriction of G, in particular, F, will satisfy the equation if it is defined for N+jD. But this contradicts (2′). QED

We define the arithmetic series with zeroth element A>=0 and common difference D>0 to be the set {A+jD, where j>=0}. Arithmetic series with A<0 or D<=0 are disqualified; the series starting with A does not include any elements preceding A, such as A-D. The series starting with A is distinct from the series starting with A+D; the former has an extra element.

We now give several "working" corollaries to the ASC.

Corollary: Suppose a 2CM computes F(N). There is some threshold T, such that the set {N | F(N)>T} is the disjoint union of a (possibly empty or infinite) collection of arithmetic series. The common differences of these series have bounded prime factors. F maps each of the series into another arithmetic series. The common differences of the image series also have bounded prime factors. The range of F is the union of a finite set and a (possibly empty or infinite) collection of arithmetic series.

In particular, if F(N) is unbounded as N approaches infinity, both the domain and the range of F must contain an arithmetic series.

Note: The corollary appears to distinguish between the domain and the range in saying that the domain is a disjoint union, and the range merely a union. This distinction is not real, however, since if a set is the union of a collection of arithmetic series, it is the disjoint union of some other collection of arithmetic series.

Theorem: The functions Exactsqrt and Exactlog2 defined below are not 2CM computable:

Exactsqrt(N) = sqrt(N) if N is a perfect square;
undefined otherwise.

Exactlog2(N) = log2 N if N is a power of 2;
undefined otherwise.

Proof: Both functions are unbounded, but their domains do not contain arithmetic sequences. QED

In the following theorems, the phrase F(N) has nonlinear growth rate means that F(N) is defined infinitely often, and that F(N)/N approaches either 0 or infinity as N approaches infinity. If F(N) has nonlinear growth rate, and F(A+jD) is defined for infinitely many j>=0, then F(A+jD), where it is defined, is a nonlinear function of j.

Theorem: If F(N) is total, monotonic, and unbounded, and has nonlinear growth rate, F is not 2CM computable.

Corollary: No 2CM can compute the functions N^2, 2^N, [log2 N], [sqrt(N)], Fib(N), etc.

The corollary doesn't solve the Output Problem, because the function [log2 N] is stronger than we need. The Output Problem only requires that a 2CM map 2^N into N; i.e., that exact powers of 2 come out correctly. Nothing is said about the 2CM's behavior when started on non-powers of 2. In fact, we know that any 2CM that solves the Output Problem will compute a function whose domain contains an arithmetic series.

Theorem: If F(N) is unbounded and monotonic, but perhaps partial, with a nonlinear growth rate; and for infinitely many N, (F(N) is defined, and for all D>0, there is a j>0 such that F(N+jD) is defined), then F and its extensions are not 2CM computable.

Proof: The condition (for all D>0, there is a j>0 such that F(N+jD) is defined) implies (for all D>0, there are infinitely many j>0 such that F(N+jD) is defined). If, for particular N and D', J were the last j, we could consider a new D: 2JD'. F(N+j2JD') is defined for some j>0; but then 2jJ is a new j for D'.
Hence, for the N in the theorem, for all D>0, F(N+jD) is a nonlinear function of j. But now the ASC applies. QED

Another way to state this theorem: Suppose F(N) is unbounded, monotonic, perhaps partial, and has nonlinear growth rate. N is disqualified by D if the domain of F contains only finitely many numbers =N (mod D). If F(N) is defined, and N is not disqualified by any D, N is qualified. Suppose that some (necessarily infinite) subset of the domain of F is qualified. Then F is not 2CM computable.

Theorem: No 2CM can compute sqrt(N) even if N is guaranteed to be a square.

Proof: Choose N to be a square, say z^2. Then for any D>0, there are infinitely many j such that N+jD is a square. Take $j = 2zh+Dh^2$, where h is any nonnegative integer. QED

This theorem asserts more than the theorem about Exactsqrt. It asserts that no extension of Exactsqrt is 2CM computable. We would like to prove a corresponding theorem about Exactlog2; this would solve the Output Problem.

Theorem: No 2CM can compute the inverse function of Fib(N), even if the input is guaranteed to be a Fibonacci number.

Proof: We show that for any N and any D>0, there is a j>0 such that Fib(N) + jD is a Fibonacci number. We prove that there is a Q such that $0 < Q <= D^2$ and Fib(N) = Fib(N+Q) (mod D).
Consider ordered pairs (Fib(I),Fib(I+1)) (mod D). There are at most D^2 such pairs; so for some J and K, $0 <= J < K <= D^2$, Fib(J)=Fib(K) (mod D) and Fib(J+1)=Fib(K+1) (mod D). But this implies Fib(J+X)=Fib(K+X) (mod D) for all positive and negative X; we may take X=N-J; then Q=K-J. QED

The technique of the last two theorems fails on the Output Problem: It is not true that for all N and D, there is an E>N such that $2^N = 2^E$ (mod D). Trouble arises if $2^{N+1} | D$.

Theorem: If F(N) is monotonic, unbounded, and 2CM computable, then F(N)/N, where it is defined, approaches a positive rational number P/Q as a limit. Moreover, QF(N)-PN is bounded whenever it is defined.

Conjecture: If F(N) is monotonic, unbounded, and 2CM computable, then there exist P and Q>0 such that (for large N) QF(N)-PN depends only on N (mod Q); F may be undefined at some residues mod Q.

Disproof: $F(N) = \begin{cases} N-1 & \text{if } 3 \nmid N \text{ and } N = (2K+1)4^J, \text{ J and K integers;} \\ N & \text{otherwise.} \end{cases}$

Conjecture: If F(N) is 2CM computable and there exist real numbers X and Y such that for large N, $X > F(N)/N > Y > 0$, whenever F(N) is defined, then F is computable by an MP1RM without loops (except for Jump*), and with only one divide instruction.

Disproof: $F(N) = \begin{cases} N & \text{if } 3 \mid N \text{ or } 5 \mid N; \\ 3^{[J/\log_2 3]}(2K+1) & \text{if } N = 2^J(2K+1). \end{cases}$

This function says, "Suppose that powers of 3 and 5 happen to be free for doing a computation. Suppose that 2^J is the largest power of 2 that divides N. Then convert 2^J to the largest power of $3 <= 2^J$." Since powers of 3 and 5 are available to simulate counters, we can carry out the computation necessary to determine the correct power of 3.

Below is an example of a 2CM computable total function F for which F(N)/N has arbitrarily large and small values:

$$F(N) = \begin{cases} 0 & \text{if } N=0; \\ N & \text{if } (N,6)=1; \\ F(5N/2) & \text{if } 2 \mid N; \\ F(N/3) & \text{if } 3 \mid N. \end{cases}$$

This function converts each 2 in the factorization of N to a 5, and removes all 3s. $F(N)>N^2$ infinitely often, and F(N)=1 infinitely often.

More generally, suppose G is any partial recursive function. We can define a 2CM computable function H:

$$H(N) = \begin{cases} \text{undefined} & \text{if } 3 \mid N \text{ or } 5 \mid N; \\ 2^{G(A)} & \text{if } N = 2^A K \text{ and K is odd.} \end{cases}$$

Such an H can grow as rapidly as any partial recursive function.

The following theorem is the reason for carrying along the condition about prime factors in the AP Theorem and subsequent theorems.

Theorem: The function SPF(N), the smallest prime factor of N, is not 2CM computable.

Proof: One proof is to note that the range of SPF does not contain an arithmetic sequence.

Here is an alternate proof: Refer to the ASC for the definition of Sp. We show that Sp contains all primes >p. Suppose q is a prime >p. Then SPF(q+jD) is a nonlinear function of j. For (q,D)=1, since D has no prime divisors >p. So the sequence q+jD contains infinitely many primes; at these points, SPF(q+jD)=q+jD. On the other hand, whenever q|j, SPF(q+jD)<=q. So SPF(q+jD) is not a monotonic function of j, and consequently not linear. QED

For a proof of Dirichlet's Theorem (if (A,D)=1, the arithmetic series starting with A and having common difference D contains an infinity of primes) see Davenport, Multiplicative Number Theory.

Consider the sets Sinf = $\bigcap_{p=2}^{\infty}$ Sp and Sinf' = $\bigcap_{p=2}^{\infty}$ Sp'. In all of our examples except SPF, Sinf or Sinf' has been an infinite set. In the case of SPF, Sinf is empty, but each Sp is infinite. Fortunately this weaker condition is enough to show that SPF is not 2CM computable.

Suppose F is a (perhaps partial) function. FI(N) is called a quasi-inverse of F if, for all N in the range of F, FI(N) is defined and F(FI(N))=N.

It seems curious that the following statement has only the status of a conjecture, but I do not know of a proof.

Conjecture: There is some 2CM computable function F with no 2CM computable quasi-inverse.

We digress a moment to mention a candidate for the simplest unsolvable problem: the 3N+1 Problem. Suppose we define a function F on the integers:

$$F(N) = \begin{cases} N/2 & \text{if } N \text{ is even;} \\ 3N+1 & \text{if } N \text{ is odd.} \end{cases}$$

Consider the sequence N, F(N), F(F(N)), F(F(F(N))), ... Numerical evidence, and a probability argument, suggest that for all N, the sequence is eventually bounded.

There are five known loops:

A: 1, 4, 2, 1

B: 0, 0

C: -1, -2, -1

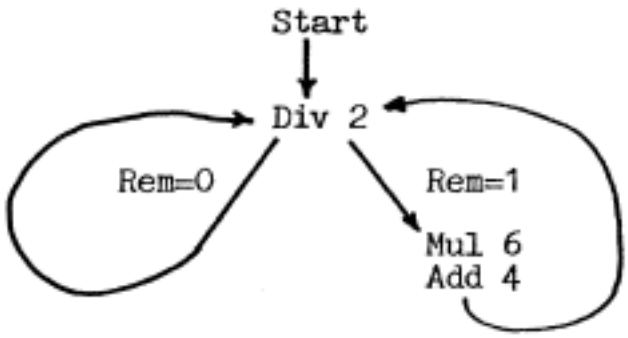
D: -5, -14, -7, -20, -10, -5

E: -17, -50, -25, -74, -37, -110, -55, -164, -82, -41, -122, -61, -182, -91, -272, -136, -68, -34, -17

All N>0 seem to lead to loop A; and all N<0 seem to lead to C, D, or E. Although extensive numerical data has been gathered, virtually nothing has been proved about the problem. Open questions include: Are there any more loops? Do all numbers get into a loop? Do any numbers go to infinity? What percentage of integers fall into each loop? Do the percentages exist? It has not even been shown that some set of positive density falls into one of the known loops.

Roger Banks wrote a computer program that verified that all numbers between -100,000,000 and 60,000,000 decay into one of these loops.

The 3N+1 Problem is an interesting example of what a simple MP1RM can compute. The basic program is simple:



We can put in a check for 1 if we want to know whether N has converged to the 1 loop.

There is an isomorphism between the 3N+1 Problem and the 3N-1 Problem; the 3N+1 Problem for positive numbers is identical to the 3N-1 Problem for negative numbers, and vice versa.

The following function is interesting: Given a negative number, which loop does it fall into? This can be coded for an MP1RM by translating to the 3N-1 Problem for positive numbers.

The next few theorems show that many MP1RMs are equivalent to MP1RMs that resemble the one above.

We give names to a few common substructures of MP1RM programs.

A Sub1 chain is a (possibly null) sequence of Sub1 instructions, the main branch of each leading to the next. An evaluated Sub1 chain is a Sub1 chain in which the zero branch arrows are evaluated; i.e., each zero branch arrow is either Jump* or Add K, Halt. A threshold routine is an evaluated Sub1 chain followed by Jump*.

An MP1RM routine is called linear if it consists of one (optional) Mul instruction, followed by an (optional) Add instruction. An eventually linear routine is a linear routine preceded by an evaluated Sub1 chain.

A routine is called modular if it is a Div instruction, every branch of which is an evaluated Mul 0; i.e., either Jump* or Mul 0, Add K, Halt. An eventually modular routine is a modular routine preceded by an evaluated Sub1 chain.

Theorem: An MP1RM subroutine with one entry point, containing no Div instructions, but possibly including loops, can be converted to either an eventually modular, an eventually linear, or a threshold routine.

Proof: The routine must consist of Adds, Sub1s, Muls, Jumps, and Halts. Suppose that all Sub1 zero branches and Mul 0 instructions are evaluated. (To do this, we suppose that the subroutine is part of a larger program.) Delete Mul 1s.

Now consider what possible loops remain: A loop can only exit with a Sub1. Therefore, at most one loop is possible; there are no subloops of loops.

Suppose that the loop exists; then we want to replace it with equivalent loop-free code.

If the loop does not contain a Mul, it must consist only of Sub1s, Adds, and Jumps. Either one trip around the loop diminishes R, or it does not. If it does, all R will go to zero; which Sub1 zero branch becomes the exit will depend on a remainder condition. The loop can be replaced by a modular routine. If R is not diminished in a circuit of the loop, then all R that are large enough to make one circuit will loop forever. So we can disconnect the last instruction in the loop from the entry point, and route it to a Jump*.

Now suppose that the loop does contain Muls: If R is greater than some threshold T, one circuit of the loop will change it to AR+B, where A>1 and B may be negative. If R>max(T,-B), one circuit of the loop will increase R, and hence will loop forever. So we replace the loop with a chain of 1+max(T,-B) Sub1s followed by a Jump*.

This eliminates all loops.

Now we use simplifications from the list on the next page to put the resulting loop-free code into the desired form:

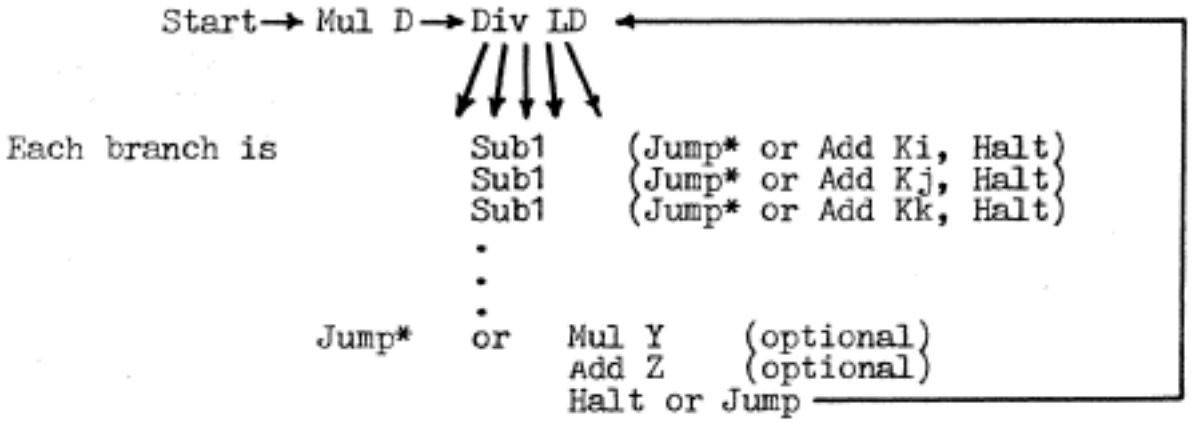
```
Add K, Add L    Add K+L
Add 0           delete
Add K, Sub1     Add K-1  if K>0
Add K, Mul L    Mul L, Add KL
Add K, Div L    Div L; branch Rem is Add [(K+L-1-Rem)/L], followed
                by old branch -K (mod L)
Add K, Jump*    Jump*
Mul 0, Sub1     Mul 0, then Jump to zero branch of the Sub1
Mul K, Sub1     Sub1, Mul K, Add K-1  if K>0
Mul K, Div L    Div L; branch Rem is Mul K, Add [KRem/L], followed
                by old branch KRem (mod L)
Mul K, Jump*    Jump*
```

Assume that we have applied as many simplifications as possible. Then any Sub1s will precede any Mul, which will precede any Add. If the main line of code ends in Jump*, any Add or Mul can be simplified away, giving a threshold routine.

If a Div was introduced to remove a loop, every branch of the Div starts with either Jump* or Mul 0. Any Mul or Add preceding the Div will be converted to Muls or Adds on the branches of the Div. These will be absorbed by the Jump* or Mul 0. This gives an eventually modular routine.

If the main line does not end in Jump* or Div, we have an eventually linear routine. This is the only case in which the routine will return to the calling program. QED

Theorem: Any MP1RM can be simulated by an MP1RM of a very special type. The special MP1RM has only one divide instruction; each branch of the divide is either a threshold routine, or an eventually linear routine that either halts or jumps back to the divide. The path from Start contains a Mul, and then goes to the divide. This makes the MP1RM into a generalized 3N+1 Problem.



Proof:
We do several modifications to the MP1RM to get our simulator.

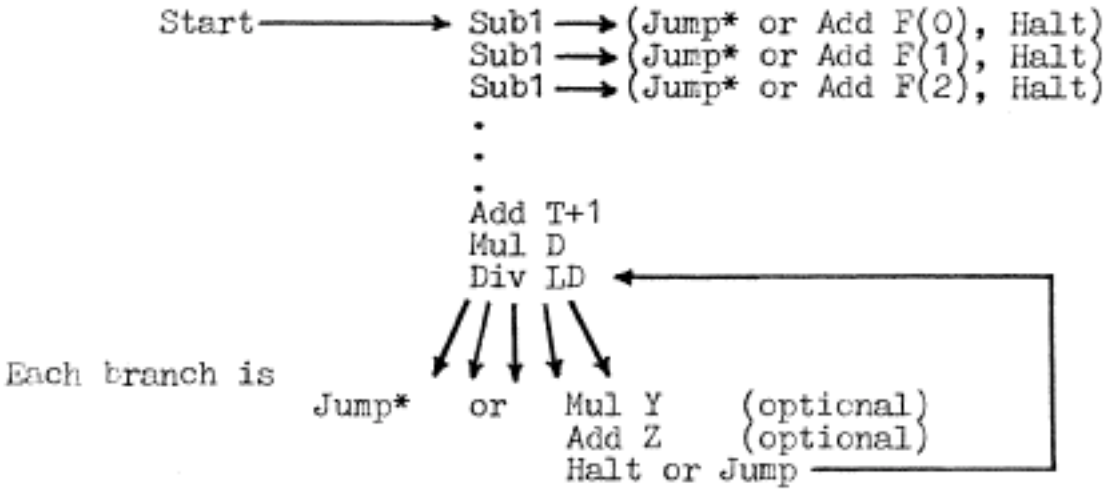
If the first instruction after Start is not a Div, we put in a Div 1.
Next, we convert every branch out of a Div instruction to either a threshold routine, an eventually modular routine, or an eventually linear routine. The modular routines will introduce new Div instructions into the program; the branches out of these divides are either Jump* (a threshold routine), or Mul 0, Add K, Halt (a linear routine).

We make all the divisors of the Div instructions the same. Let L be the least common multiple of all the divisors. Every arrow or instruction going into a Div K has a Mul L/K appended; and the Div K is changed to a Div L. Branch Rem from the Div K becomes branch (L/K)Rem of the new Div L. If L>K, there will be other branches of the Div L which are unassigned; these are inaccessible by the program, and may be filled in with Jump*.
Now that all the divisors are the same, we combine all the divide instructions. Assign a state number to each divide, giving state 0 to the dummy Div 1 at Start. Let D be the total number of Div instructions. Every arrow or instruction going into the divide with state number S has a Mul D, Add S appended, and is routed to the superdivide, a Div LD. Remainders from the superdivide will range from 0 to LD-1. Express each remainder as DX+S, with 0<=X<L and 0<=S<D. To branch DX+S, we attach branch X from divide number S.
The various Muls and Adds we have introduced are then simplified as in the preceding theorem, bringing the MP1RM to the correct form. QED

Addendum to theorem: The divisor of the single divide instruction may be chosen to be a number without square factors.

Proof: An MP1RM divide instruction with a composite divisor may be replaced by a tree of divide instructions with prime divisors. If we make this modification to an MP1RM, and then apply the preceding theorem, then the least common multiple of the divisors, L, will be a number without square factors. We may choose D to be any large prime greater than the number of Div instructions, so LD will be square-free. QED

Theorem: Suppose that the function computed by an MP1RM approaches infinity, and is undefined for only a finite number of arguments. Suppose that the definition of the Add instruction is extended to allow a negative addend. (It is an error if an Add instruction produces a negative number.) Then the Sub1s may be eliminated from the divide branches, and replaced with an evaluated Sub1 chain immediately after Start, followed by an Add.



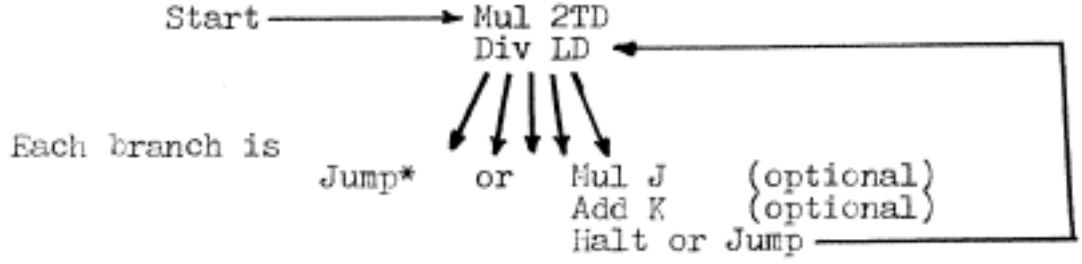
Proof: Construct the single divide MP1RM of the preceding theorem. Let S be the set of values at the end of Sub1 zero branches. Let T be the largest N such that F(N) is undefined or F(N) is in S. Put a length T+1 Sub1 chain after Start in the MP1RM, followed by an Add T+1. The Sub1 chain will evaluate F(N) correctly for N<=T. If N>T, the extra instructions will go unnoticed. In this new machine, once execution has gotten past the initial Sub1 chain, the computation will never take a Sub1 zero branch. We may combine all of the Sub1s in a length C chain into an Add -C. A divide branch will be either

```
Add -C      or      Add -C
Jump*               Mul J
                    Add K
                    Halt or Jump to the divide
```

Since no Sub1 zero branches are used by the computation, the Add -C will not produce negative numbers. The first type of branch above may be changed to Jump*. The second type may be changed to Mul J, Add K-JC, and then Halt or Jump. The quantity K-JC will be of indeterminant sign. QED

In the cases I have examined, it is always possible to avoid negative addends in the final MP1RM. Can this always be done?

Theorem: Suppose that some MP1RM solves the Output Problem. Then we may convert that MP1RM into the form below, where the addend of an Add instruction may be negative. The new MP1RM may be inequivalent to the starting MP1RM, but will still solve the Output Problem. We specify that if an Add produces a negative number, the function is undefined.



Proof: Assume that we have an MP1RM that solves the Output Problem. We apply the transformations given in the preceding two theorems, while saying that we don't care what happens to any input that is not a power of two. The number T in the preceding theorem is two to the largest number at the end of a Sub1 zero branch. We put a Mul 2T after Start and an Add -log2 2T before all Halt instructions. Powers of two larger than T will skip over the initial Sub1 chain, so we may delete it and combine the Mul 2T and the Mul D. The Adds introduced before Halt instructions are combined with any preceding Add instructions. QED

We have assumed that our CMs always begin with most of their registers zero. The possibility remains that we could compute 2^N if perhaps the 2CM were started with N in one counter, and suitable "help" in the other counter—— perhaps another copy of N, or somesuch. The following discussion should eliminate that possibility.

Suppose we have an N CM with an I instruction program. The numbers in the counters of the CM can be considered to be the coordinates of a point in N-space. One CM instruction has the effect of moving 1 step in N-space. We can think of the CM as a finite state bug that crawls around in the region of N-space with positive coordinates, and has the capability of detecting when it is on a wall. (This corresponds to some counter being 0.) The point of the Replacement Lemma is that when a CM is far from a wall, it is in a loop which repeatedly moves it a constant vector.

If V is a vector in N-space, the norm of V, denoted by |V|, is the sum of the absolute values of the coordinates of V.

For reasons explained below, the set of points whose coordinates are all >=I is called the inaccessible region. It is not true that a CM is excluded from the inaccessible region.

A CM is a constant adder if, whenever it is started in the inaccessible region, it moves a constant vector V, and halts. (The CM may compute an interesting function when started outside the inaccessible region.) In a constant adding CM, |V|<=I.

If, for some input, a CM halts in the inaccessible region, then the CM is a constant adder.

If a CM is not a constant adder, then there is a vector V, with 0<|V|<=I, such that any two points in the inaccessible region which differ by V are equivalent: Either the CM does not halt with either input, or it halts with both, at the same point in N-space.

We note briefly the computing power of a 1CM:

Theorem: A 1CM that is not a constant adder has a threshold T and a period P such that N>T implies F(N)=F(N+P). A 1CM is either a constant adder or is eventually modular.

If N>2, an N CM can compute any computable function of N-1 variables. The CM starts with the arguments in N-1 counters, and the remaining counter zero. The trick is for the CM to code two of the variables into one number; this provides the necessary two free counters to carry out a Turing machine simulation. The details of the coding are given in the first part of the solution to the multiplication problem, at the end of the memo. A similar trick permits an N CM to compute up to N-1 outputs; only one counter need be <I when the CM halts.

The definitions below refer to a set S of positive integers.

$$\mathrm{Rho}(S,N) = (\text{number of elements of } S \leq N)/N$$

Rho(S,N) is called the density of S at N.

$$\mathrm{Rho}(S) = \lim_{N\to\infty} \mathrm{Rho}(S,N)$$

Rho(S) is the density of S. The limit may not exist.

$$\mathrm{LRho}(S) = \lim_{N\to\infty} \inf_{J>N} \mathrm{Rho}(S,J)$$

$$\mathrm{URho}(S) = \lim_{N\to\infty} \sup_{J>N} \mathrm{Rho}(S,J)$$

LRho(S) is the lower density of S. URho(S) is the upper density of S. The lower and upper densities always exist; they are equal if and only if the density exists.

Theorem: If the range of a 2CM computable function is infinite, both the domain and range have positive lower densities.

It is not true that a union of arithmetic series must have a density.

Conjecture: If the range of a 2CM computable function is infinite, both the domain and the range have densities.

If the range is finite, the domain need not have a density. For example,

$$F(N) = \begin{cases} 0 & \text{if } [\log_2 N] \text{ is odd;} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

In this example, LRho=1/3 and URho=2/3. Several similar examples prompt the following conjecture:

Conjecture: If the range of a 2CM computable function is finite, and the domain does not have a density, then LRho and URho are rational.

We will discuss briefly finite-range functions computed by 2CMs. We call these functions partitions. A partition P(N) divides the nonnegative integers into a finite number of classes; two integers A and B are in the same class if P(A)=P(B), or if both P(A) and P(B) are undefined. A total partition is defined on all nonnegative integers. The values taken by a partition are usually unimportant. A predicate is a partition that takes only the values 0 and 1.

Some predicates that a 2CM can decide are:
Is N a power of 2?
Is N either a power of two or a power of three?
Is N the sum of three distinct powers of 2?
Does the decimal represention of N contain a 3 and a 7?
Is [log2 N] even or odd?
Are there an odd number of 1s in the binary expansion of N?
(This is the parity function.)
In general, any Finite State Machine function of the digits of N is 2CM computable.
Is N a Mersenne $(2^{p}-1)$ or a Fermat $(2^{2^{K}}+1)$ prime?
Is N a prime of the form $2^{A}3^{B}+1$? Since there are probably infinitely many primes of this form, it seems likely that a 2CM can recognize some infinite class of primes.
Does counter 1 countain a larger number than counter 2?

Some questions:
Can a 2CM decide if its argument is a prime number?
Can a 2CM decide if its argument is a square?
Can a 2CM calculate the Nth digit of pi?
My conjecture is that all three questions get No answers.
Can a 2CM do the Or or And of two predicates?
What about the amalgamation or corefinement of two partitions?
My guess here is also No, since many predicates seem to require the destruction of the input data.

Presumably 2CMs cannot compute all possible partitions, so it is natural to look for theorems that distinguish 2CM computable partitions.

A natural conjecture is: One of the classes of the partition must contain an infinite arithmetic series. This is false:

P(N) = 0 if N=0 or if [log2 N] is even;
1 if [log2 N] is odd.

Neither class of this partition contains an infinite arithmetic series. Neither class has a density; the density of each oscillates between LRho=1/3 and URho=2/3

Definition: Suppose that $A+B>=0$, $B>=0$, $C>1$, $D>0$, $D|A+B$, and $D|A+BC$. Then $\{(A+BC^K)/D \mid K>=0\}$ is called an offset geometric progression (OGP). All elements of an OGP are integers: $D|A+B$ and $A+BC$, so $D|B(C-1)$; hence $D|B(C^K-1)$; and $D|A+BC^K$.

Theorem: An OGP contains an infinity of composite numbers.

Proof: Suppose that $p|(A+BC^K)/D$ for some K. Suppose that $(p,C)=(p,D)=1$. Then $p|(A+BC^{K+h(p-1)})/D$ for any $h>=0$. By Fermat's Theorem, (see Hardy and Wright, Introduction to Number Theory, Theorem 71), $C^{p-1}=1 \pmod p$, so $C^{K+h(p-1)}=C^K \pmod p$. QED

It would be nice if each infinite class of a partition had to contain an offset geometric progression: This would prove that a 2CM could not decide if its argument is prime. We note that each of classes of the 3N-1 problem contains an OGP. Unfortunately, this conjecture is false:

Theorem: The predicate Q defined below is 2CM computable.

$$Q(N) = \begin{cases} 1 & \text{if } N = 2^{2^K}; \\ 0 & \text{otherwise.} \end{cases}$$

Proof: First the 2CM verifies that $3 \nmid N$. Then it loops, doing the transformation $2^{2J}V$ (with V odd) to $3^J V$ to $2^J V$, until J is odd. Now, $Q(N)=1$ only if $J=V=1$. QED

A real number X is effectively computable if there is an algorithm A, which, for any positive integer N, calculates an integer A(N) such that $0 <= NX - A(N) <= 2$.

Theorem: Suppose that X is effectively computable and $0<=X<=1$. Then there is a recursive function B, defined on the integers greater than 1, and taking only the values 0, 1, and 2, such that

$$X = \sum_{N=2}^{\infty} 2^{-N} B(N)$$

Proof: Let A be an algoritm for X. Suppose that we have determined B(N) for N<J.

Let $X(K) = \sum_{N=2}^{K} 2^{-N} B(N)$, and X(1)=0.

Let $E(K) = A(2^{K}) - 2^{K} X(K-1)$. E(K) can be calculated without knowing B(K).

Let B(J) = 0 if E(J)<=0;
1 if E(J)=1;
2 if E(J)>=2.

We must show that $\lim_{K\to\infty} X(K) = X$. We prove by induction that

$$2^{1-J} >= X - X(J) >= 0.$$

For J=1, we have $1 >= X - X(1) >= 0$.

Suppose $2^{2-J} >= X - X(J-1) >= 0$; we want to derive

$$2^{1-J} >= X - X(J) >= 0.$$

We abbreviate B(J) to B, E(J) to E, and $A(2^{J})$ to A.
We know that

$$X(J-1) + 2^{2-J} >= X >= X(J-1)$$

$$A + 2 >= 2^{J} X >= A$$

$$X(J) = X(J-1) + 2^{-J} B$$

$$A = E + 2^{J} X(J-1)$$

First we show that X>=X(J). If E<=0, B=0, and X(J)=X(J-1)<=X. If E>=1, then E>=B.

$$2^{J} X >= A = 2^{J} X(J-1) + E >= 2^{J} X(J-1) + B = 2^{J} X(J).$$

Next we show that $X(J) + 2^{1-J} >= X$. If E<2, then E<=B.

$$2^{J} X <= A + 2 = 2^{J} X(J-1) + E + 2 <= 2^{J} X(J-1) + B + 2 = 2^{J} X(J) + 2.$$

If F>=2, then B=2.

$$X(J) + 2^{1-J} = X(J-1) + 2^{-J} B + 2^{1-J} = X(J-1) + 2^{2-J} >= X.$$

QED

Theorem: Suppose that X is effectively computable and 0<=X<=1. Then there is a 2CM computable total partition with a class of density X.

Proof: Put $X = \sum_{J=2}^{\infty} 2^{-J} B(J)$.

Define C(J) = 1 if B(J)=1 or 2;
0 otherwise.

and D(J) = 1 if B(J)=2;
0 otherwise.

Then B(J) = C(J) + D(J).
Now define the partition

$$P(N) = C(J+2) \text{ if } N = 2^{J}(4K+1);$$
$$D(J+2) \text{ if } N = 2^{J}(4K+3);$$
$$0 \text{ if } N=0.$$

The partition is 2CM computable: The CM first removes any factors of 5, 13 or 17 from N. This provides the room to simulate a 4CM using powers of 2, 5, 13, and 17. The 4CM starts with J in its first counter and calculates B(J+2). Then it zeros all four of its counters, and takes one of three exits, depending on the value of B. If B=0, P=0. If B=2, P=1. If B=1, we examine whether the number remaining in the 2CM after the simulation is congruent to 1 or 3 (mod 4); P=1 or 0, respectively.

Now we show that the set S = {N | P(N)=1} has density X. The number of numbers <=N and =1 (mod 4) is [(N+3)/4]; and the number =3 (mod 4) is [(N+1)/4]. We can write

$$N\ Rho(S,N) = \left[\frac{N+3}{4}\right] C(2) + \left[\frac{N+1}{4}\right] D(2) + \left[\frac{N+3}{8}\right] C(3) + \left[\frac{N+1}{8}\right] D(3) + \ldots$$

If we remove the brackets and fractions, we introduce an error of at most 2 per term. The number of terms is about 2log2 N, so the error is less than about 4log2 N.

$$N\ Rho(S,N) = \frac{N}{4} C(2) + \frac{N}{4} D(2) + \frac{N}{8} C(3) + \frac{N}{8} D(3) + \ldots + \text{error}$$

$$= N \left\{ \frac{B(2)}{4} + \frac{B(3)}{8} + \frac{B(4)}{16} + \ldots \right\} + \text{error}$$

$$Rho(S,N) = X + \text{error}/N$$

Since error/N approaches 0 as N approaches infinity, Rho(S)=X. QED

Some thoughts on the output problem:

Let Zp be the set of numbers with no prime factor greater than p; for example, $Z5 = \{2^A 3^B 5^C\}$.

Lemma: Suppose p is fixed. There is a 2CM which maps 2^Q into Q for all Q in Zp.

Proof: Suppose p is 7. There is a 6CM with counters U, V, W, X, Y, and Z, which starts with $Q = 2^A 3^B 5^C 7^D$ in counter U, and halts with A in U, B in V, C in W, D in X, and Y=Z=0. If this 6CM is simulated by a 2CM, the 2CM will map 2^Q into Q. QED

Theorem: Suppose p is fixed. Any partial recursive function from Zp into Zp is 2CM computable.

Proof: Along the lines used above.

Theorem: Suppose that for some Zp, there is a 2CM computable function F which maps Zp onto the nonnegative integers. Then the Output Problem can be solved.

Proof: We can construct a Turing machine which takes an input N and searches for an element Q of Zp for which F(Q)=N. We can simulate the Turing machine with a 2CM which takes as input 2^N and has output 2^Q. From 2^Q we can get to Q, and thence to N. QED

Unfortunately, the reasonable candidates for such a function do not seem to work. Consider the function G:

$$G(K) = \begin{cases} K/7 & \text{if } 7 \mid K; \\ G([K/7]) & \text{otherwise.} \end{cases}$$

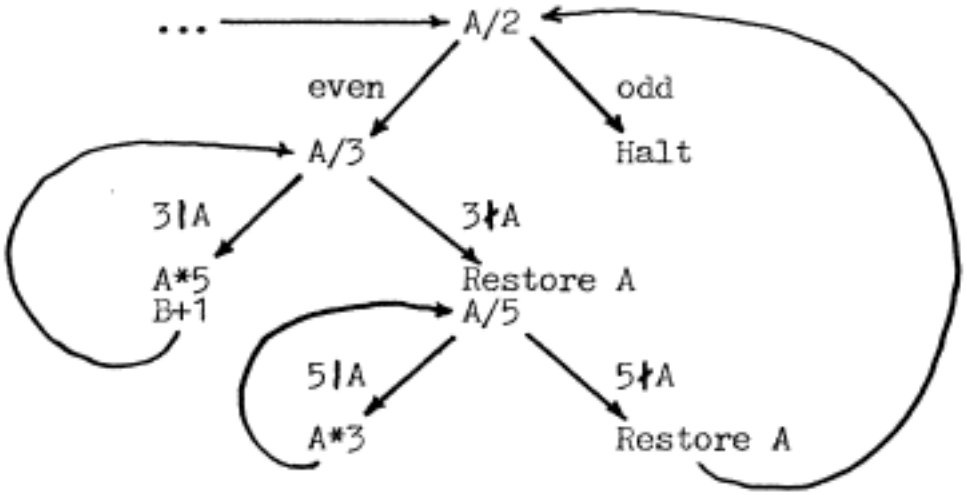
G is clearly 2CM computable. A number is G-representable if it is in G(Z5). There is a probability argument which indicates that G(Z5) should have density zero.

I did a small numerical experiment. I examined numbers up to 100 to see which were G-representable. The search was terminated if no representation $< 10^{20}$ was found. The following numbers appear to be unrepresentable: 36, 45, 49, 56, 60, 70, 72, 75, 80, 84, 90, 96, 98, and 100.

Below are the solutions to the multiply and square problems given as exercises on page 2.

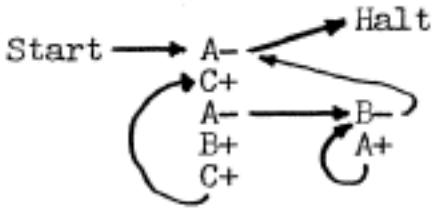
My solution to the multiply problem is ugly; I would be happy to hear of any improvements.

Suppose X is in A, and Y is in B. C is used as a temporary for the multiplications and divisions. First, we calculate $2^{XY}3$. Set B to 2Y+1. Double B, X times, meanwhile counting A down to zero. Now B contains $2^{X}(2Y+1)$. Set A to 1. Loop, Halving B, and doubling A each time around. Exit when B gives remainder 1. Now A contains 2^{X} and B contains Y. Triple A, Y times, counting down B to zero. Now A contains $2^{XY}3$. B and C are zero. Now the following program will create XY in B.



Is there some way that takes less time?

The solution to the squaring problem is much easier:



The program uses the fact that $N^2 = (2N-1) + (2N-3) + \ldots + 3 + 1$.

An introduction to Counter Machines is given in Chapters 11 and 14 of Minsky's Computation - Finite and Infinite Machines, Prentice-Hall, 1967.

My proof of the non-computability of 2^N was discovered in September 1970. Frances Yao independently proved the non-computability using a similar method in April 1971.