MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge, Massachusetts

PROJECT MAC

ARTIFICIAL INTELLIGENCE PROJECT
MEMO 89

MEMORANDUM MAC-M-262
September, 1965

A THEORY OF COMPUTER INSTRUCTIONS

WARD DOUGLAS MAURER

Introduction

This paper has arisen from an attempt to determine the nature of computer instructions from the viewpoint of general function and set theory. Mathematical machines, however the term is understood, are not adequate models for the computers of today; this is true whether we are talking about Turing machines, sequential machines, push-down automata, generalized sequential machines, or any of the other numerous machine models that have been formulated in the last fifteen years. Most of these models are either not general enough, as the sequential or Turing machines with their single input and output devices; or capable of accurately reproducing only one important programming feature; or in a sense too general (see the discussion of sequential machines in Chapter 10 below). On the other hand, modern computers, whether they are binary, decimal, or mixed, whether they have one or two instructions per word, or one instruction covering several words, have several important common features. All of their instructions have input, output, and affected regions (in the sense of Definitions B and K below). The study of the input and output regions and the structure of affected regions of all the instructions on a given computer can provide a key to its logical efficiency.

Various directions of further study may suggest themselves. This computers introduced here seem to cover at least two situations which have nothing to do with "hardware": the construction of an algorithm (such as a flow chart) and

A THEORY OF COMPUTER INSTRUCTIONS

ERRATA

P. 15, line 14: "$\prod_{x \in M} B_x$"

P. 19, line 10: "if $x \in OR(I_1)$ but ... "

P. 19, line 22: "if $x \notin OR(I_2)$,"

P. 20, line 3: "$S_1 | IR(J) = S_2 | IR(J)$"

P. 20, line 18: "$x \in OR(I_1)$"

P. 20, line 19: "$x \notin OR(I_2)$"

P. 27, bottom: "by construction of $\mathfrak{I}$ "

P. 29, line 10: "$S_1(z) = S_2(z)$ for $z \notin IR(I)$,"

P. 33, third line from bottom was omitted; it should read:

"as the intersection of all subsets N of M which possess the predicate $AP_2(M', I)$;"

P. 34, line 9: "M', I_1), so that"

P. 34, tenth line was omitted; it should read:

"I_1), I_2) = $I_2(I_1(S_2)) | M - AR_2(AR_2(M', I_1), I_2)$. Therefore $AR_2(AR_2(M', $ "

P. 34, seventh line from bottom: "$AR_2(M', I) = AR(M', I) \cup$ ..."

P. 34, second line from bottom: "This shows that (M' - OR(I))"

P. 34, bottom: "$(M - (OR(I) - AR(M', I))) \cap (M - Z)$ possesses ..."

P. 35, line 2-4; replace with:

"$AR(M', I) \cup (M' - OR(I) - Z)$. To show the converse relation, we note that $AR(M', I) \subseteq AR_2(M', I)$ follows directly from the definitions; also if we choose $S_1 | M-M'$ "

P. 35, line 5: "= $S_2 | M-M'$ "

P. 46, line 1: "input, output, and affected"

the construction of a program in a computer language such as ALGOL. Also, we may be able to prove far-reaching theorems by imposing restrictions on the computers herein defined.

1. A Simple Model

Most modern computers are either binary or decimal; but some are a combination of the two, and theoretically there is no reason why a computer could not be constructed to the base 3, 7, 16, etc. To say that a computer is constructed to the base n means that each "element" of the computer (i. e., each bit position or decimal digit in a computer word) is capable of "assuming" the values 0 through n-1. For the sake of convenience, we shall now make a restriction (which is removed in Chapter 4) that the base n is constant over the whole computer.

All computers have a memory, which is a finite collection of "elements" of the above type. We may now drop the quotes and speak of the memory of a computer as a finite set M, whose elements are permitted to assume values from 0 through n-1. A particular state (sometimes known as an "instantaneous description") of the computer is then a specification of such a value to each element of M; i.e., a function from M into the set of all integers from 0 to n-1. This suggests that our treatment of the "number base" n of a computer, above, is inadequate, and that we ought instead to consider a set B, called the base space, whose cardinality is the base of the computer. A state of the computer is then an arbitrary map from M into B.

Some computers have accumulators, index registers, "Q-registers", and/or other special-purpose registers. It is important to note that we are regarding all such registers as subsets of M. Each register has its own "elements" (bit positions or decimal digits), which are regarded here on the same basis as the corresponding "elements" of a standard core memory cell; i.e., as elements of the set M.

Almost all computers have input-output devices. It is possible, of course, for computers to compute without using input-output devices, and one might expect that such devices are not necessary from this point of view. In fact, input-output devices and the instructions governing them are included in the present model, by extending the memory to be infinite; this is discussed in Chapter 10. We shall therefore postpone discussion of input and output, and assume that our computer has no such devices.

Passage from one state to the next is carried out by means of instructions. An instruction, then, is a method of passing from one state to another state; i.e., a map $I: \mathcal{S} \to \mathcal{S}$ where $\mathcal{S}$ is the set of all states under consideration. Let us assume for the moment that $\mathcal{S}$ is in fact the set of all maps from M into B. Let us, in turn, denote the set of all instructions in a given computer by $\mathcal{I}$. We then have the following definition.

DEFINITION A. Let M and B be finite sets, let $\mathcal{S}$ be the set of all mappings $S: M \to B$, and let $\mathcal{I}$ be a set of maps $I: \mathcal{S} \to \mathcal{S}$. Then the 4-tuple $(M, B, \mathcal{S}, \mathcal{I})$ is a finite complete computer. The set M is the memory of the computer; the set B is the base space; the members $S \in \mathcal{S}$ are the states; and the members $I \in \mathcal{I}$ are the instructions.

2. Input and Output Regions of an Instruction

Each instruction $I: \mathcal{S} \to \mathcal{S}$ has associated with it two subsets of M, in a manner dictated by intuitive considerations.

As an example, let us consider a computer whose memory contains a "core cell" Y and an "accumulator" AC, i.e., $Y \subset M$ and $AC \subset M$, and there is an instruction (possibly called "clear and add Y," "load Y," or "zero and add Y") which moves the data in Y to AC. When we speak of "the data in Y" we are implying that the computer is in a given state $S: M \to B$, and the restriction of this map to Y (denoted by $S \mid Y$), which is a map from Y into B, is a code representation of a number, one or more characters, or the like. What we seek is a rigorous formulation of the phrase "moves the data." Stated another way: The instruction I = (CLA Y), or "clear and add Y," is a map from $\mathcal{S}$ into $\mathcal{S}$, where $\mathcal{S}$ is the set of all maps $S: M \to B$. Yet there are clearly associated with the instruction I, two subsets of M; one is Y, and the other is AC. What is the precise relation between I and these two subsets?

Before we answer this question, we would like to make two wishes:

(a) We would like to call Y the "input region" and AC the "output region" of I. Each instruction, then, may "take" data only from its input region, and may "place" data only in its output region.

(b) We would like the definition of "input and output region" to give meaningful results when applied to <u>any</u> instruction--at least, any common instruction on a real computer. In order to make this precise, let us mention certain commonly used instructions, with their (putative) input and output regions:

I. A store instruction (STO Y), which stores data from AC in the cell Y. Input region, AC; output region, Y.

II. An add instruction (ADD Y), which adds the data in Y to the current contents of the AC, and leaves the result in the AC. Input region, Y $\cup$ AC; output region, Y. (note: The same holds if "add" is replaced by any other binary operation, such as: subtract, multiply, divide; logical and, or, exclusive or.)

III. A shift instruction (RQL 6) which rotates register MQ left by six places. Input region, MQ; output region MQ.

Without giving any more examples at the moment, we proceed to our definitions.

DEFINITION B. Let $(M, B, \mathscr{S}, \mathscr{I})$ be a computer, and let $I \in \mathscr{I}$. Then the input region $IR(I) \subseteq M$ and the output region $OR(I) \subseteq M$ are defined as follows:

$$OR(I) = \{x \in M: \exists\ S \in \mathscr{S}\ \exists\ \ S(x) \neq I(S)(x)\}$$

$$IR(I) = \{x \in M: \exists\ S_1, S_2 \in \mathscr{S}, y \in OR(I)\ \ni\ S_1(z) = S_2(z),\ z \neq x,\ \text{and } I(S_1)(y) \neq I(S_2)(y)\}$$

Roughly speaking, OR(I) is the set of all elements of M which "can be affected" by I; if $x \notin OR(I)$, then x is "unaffected," i.e., the state of x before the instruction, S(x), equals the state of x after the

instruction, $I(S)(x)$, for any state $S \in \mathcal{S}$. The input region IR(I) is the set of all elements of M which can affect OR(I); an element x is in IR(I) if there exists a state S_1 such that, by changing it on x alone (and obtaining S_2) one gets different results on some element of OR(I) after the instruction I. Note that the definition of IR(I) depends on OR(I); this seems to be unavoidable.

The following property of input and output regions is fundamental.

PROPOSITION I. Let S_1 and S_2 be any two states and let I be any instruction. If $S_1 \mid IR(I) = S_2 \mid IR(I)$, then $I(S_1) \mid OR(I) = I(S_2) \mid OR(I)$.

Proof: Let IP(I) (the <u>input property</u>) be the following predicate of subsets of M: M' possesses IP(I) $\Longleftrightarrow$ ($S_1 \mid M' = S_2 \mid M' \Rightarrow I(S_1) \mid OR(I) = I(S_2) \mid OR(I)$ for all $S_1, S_2 \in \mathcal{S}$). We must prove that IR(I) possesses IP(I). We first prove that the predicate IP(I) is preserved under the taking of intersections. Let M'_1 and M'' be two subsets of M, each of which possesses IP(I). Let S_3 and S_4 be any two states such that $S_3 \mid M' \cap M'' = S_4 \mid M' \cap M''$. Let S_5 be defined by $S_5(x) = S_3(x)$, $x \in M'$; $S_5(x) = S_4(x)$, $x \notin M'$. We have $S_5 \mid M' = S_3 \mid M'$ and, as can easily be verified, $S_5 \mid M'' = S_4 \mid M''$. Since M' and M'' each possess IP(I), we have $I(S_5) \mid OR(I) = I(S_3) \mid OR(I)$ and also $I(S_5) \mid OR(I) = I(S_4) \mid OR(I)$. Therefore $I(S_3) \mid OR(I) = I(S_4) \mid OR(I)$. This shows that $M' \cap M''$ also possesses IP(I).

Now take the intersection of <u>all</u> subsets of M which possess IP(I). Since M is finite, there will be only a finite number of these, and the intersection will therefore possess IP(I). We show that this intersection (call it M_1) is equal to IR(I). Note that the definition of IR(I) can be

rewritten: $IR(I) = \{ x \in M: M - \{x\} \text{ does not possess } IP(I) \}$. Thus we need only prove: for each $x \in M$, $M - \{x\}$ possesses IP(I) if and only if $x \notin M_1$. But if $M - \{x\}$ possesses IP(I), then $M_1 \subseteq M - \{x\}$ by definition, so $x \notin M_1$; conversely, if $x \notin M_1$, then there exists some set M_2 possessing IP(I) to which x does not belong, and if M_2 possesses IP(I), then since $M - \{x\} \supseteq M_2$, $M - \{x\}$ certainly possesses IP(I). This completes the proof.

Thus, if two states agree on the input region of an instruction, the results on applying I agree on the output region. This would seem to follow immediately from the definitions; yet the length of the above proof is not artificial. The result, in fact, does not hold if M is infinite (unless other changes are made).

3. A Few Generalizations

At this point we may ask: How general can we make the sets M, B, $\mathcal{A}$, and $\mathcal{S}$ without losing its most important properties? The only important property of computers we have at the moment is Proposition I; however, it turns out that the same conditions which ensure Proposition I also are sufficient for our other purposes. In Chapter 6 we shall discuss what happens when these conditions are relaxed.

As we have seen, there is no reason to suppose that B has either 2 or 10 elements. Of course, if B is empty, then so is $\mathcal{A}$, while if B has exactly one element, there can be only one state S (and hence only one instruction I). Since these two cases are uninteresting, we may postulate that <u>there exist at least two elements in B</u>. In fact, this postulate will become essential later on.

Even for real computers, we may consider other cardinalities for B than 2 and 10. We may, for example, regard a character machine as a computer with the separate 6-bit (or 7-bit, or 8-bit) characters as elements of M, and give B a cardinality of 64 (128, 256). For a binary computer with 36-bit words, we can consider the words as elements of M (provided there are no 15-bit index registers) and consider B as having a cardinality of 2^{36}. This raises the further question: Can we allow B to be infinite? Certainly. In fact, in the previous example, we could let the words of a computer be the elements of a set M, and make the (idealized)

assumption that a word may contain any integer (or any real number). Thus we get a computer in which the base space is the integers or the reals. None of the theory discussed here precludes the case in which the base space is infinite. The only condition imposed on B is the one above: that its cardinality be at least 2.

Is it possible to consider $\mathcal{S}$ as only a subset of the set of all maps S: M→B? If it were possible, consistently with $\mathcal{D}$ (i.e., $S \in \mathcal{S}$ implies $I(S) \in \mathcal{S}$, for all $I \in \mathcal{D}$) then the choice of $\mathcal{S}$ might have an effect on the input and output regions of an instruction. It turns out that it is necessary to impose one condition:

(P1) If $S_1, S_2 \in \mathcal{S}$, and M' is any subset of M, then the state S_3, with

$$S_3(x) = S_1(x), \qquad x \in M',$$
$$S_3(x) = S_2(x), \qquad x \notin M',$$

is a member of $\mathcal{S}$.

Thus for a decimal machine, it is possible to consider families $\mathcal{S}$ such as the following: $S \in \mathcal{S}$ if and only if S can only take the values 3 or 7 on a subset M_1 of M; only the values 0 or 1 on another subset M_2; and may take any value elsewhere. If M is finite, however, no other conditions need be imposed on $\mathcal{S}$.

However, if we allow M to be infinite, a new situation comes up. In this case, in fact, it is unwise to admit even the entire class of maps S from M into B. This raises the question as to whether we should ever let M be infinite: shouldn't we stick to finite computers, in which M (and therefore $\mathcal{S}$) is finite, just as is usually done in sequential

machine theory? Of course, $\mathcal{S}$ can only be finite if B is finite; but there is another consideration.

Let us consider the following example of an (infinite) computer. The memory M consists of the integers. The base space, B, is the union of three finite sets, K, Σ, and Δ. A map S: M $\longrightarrow$ B is said to be in $\mathcal{S}$ if and only if:

$$S(0) \in K;$$

$$S(x) \in \Delta, x > 0;$$

$$S(x) \in \Sigma, x < 0.$$

Note that this family $\mathcal{S}$ satisfies the condition given above. There is exactly one instruction $I \in \mathcal{I}$. For $x \neq 0$ or 1, I(S)(x) is defined to be S(x-1). Let us be given two arbitrary maps, $\delta : K \times \Sigma \rightarrow K$, and $\lambda : K \times \Sigma \rightarrow \Delta$. Then we define

$$I(S)(0) = \delta(S(0), S(-1)),$$

$$I(S)(1) = \lambda(S(0), S(-1)).$$

It should be evident that we have given a realization of a sequential machine [1] as a computer. The states of the sequential machine are the values S(0). The inputs enter at -1, and the outputs appear at 1. The entire input and output "tapes" are part of the memory, although the instruction makes no use of this memory (except for -1) other than to move it forward by one square. Without going into the merits of this particular realization, we see that infinite computers are, in fact, interesting. However, for infinite computers, it becomes necessary to

impose another condition on $\mathcal{S}$:

(P2) If $S_1, S_2 \in \mathcal{S}$, then $\{x \in M: S_1(x) \neq S_2(x)\}$ is finite.

No special conditions at all are needed on the set $\mathcal{I}$. We may therefore make our general definition of a computer as follows:

DEFINITION C. Let M be arbitrary, let B have at least two elements, let $\mathcal{S}$ be a set of maps S: $M \to B$ satisfying conditions (P1) and (P2) given above, and let $\mathcal{I}$ be a set of maps I: $\mathcal{S} \to \mathcal{S}$. Then the 4-tuple $(M, B, \mathcal{S}, \mathcal{I})$ is a <u>computer</u>. As before, M is the <u>memory</u>, B is the <u>base space</u>, the members of $\mathcal{I}$ are called <u>states</u>, and the members of are called <u>instructions.</u>

The reason for introducing the conditions (P1) and (P2) has to do with Proposition I. We should like to know, not only that the conditions insure that Proposition I can be properly extended, but that they are likewise necessary for this purpose.

PROPOSITION II. Let $(M, B, \mathcal{S}, \mathcal{I})$ be a computer and let $I \in \mathcal{I}$. If S_1 and S_2 are any two states of $\mathcal{S}$, then $S_1 \mid IR(I) = S_2 \mid IR(I)$ implies $I(S_1) \mid OR(I) = I(S_2) \mid OR(I)$. Conversely, let M and B be arbitrary sets, and let $\mathcal{S}$ be a set of maps S: $M \to B$ which fails to satisfy (P2). Then there exists a map I: $\mathcal{S} \to \mathcal{S}$ and two states $S_1, S_2 \in \mathcal{S}$ such that $S_1 \mid IR(I) = S_2 \mid IR(I)$, but $I(S_1) \mid OR(I) \neq I(S_2) \mid OR(I)$.

Proof: Let IP(I) be as in Proposition I. The proof that the predicate IP(I) is preserved under intersections is exactly as in Proposition I, although we note that the construction of the state S_5

from S_3 and S_4 is now possible because $\mathcal{L}$ satisfies condition (P1). Now take the intersection of all subsets of M which possess IP(I). This intersection will be equal to IR(I), just as in Proposition I, and it remains only to prove that M_1, the intersection, possesses IP(I). Let S_1 and S_2 be any two states such that $S_1 \mid M_1 = S_2 \mid M_1$. Let $M_2 = \{x \in M: S_1(x) \neq S_2(x)\}$. Since $\mathcal{L}$ satisfies condition (P2), M_2 is finite, and $M_1 \cap M_2 = \emptyset$. Let x_i, $1 \leq i \leq n$, be the elements of M_2. Since $x_i \notin M_1$, $M - \{x_i\}$ possesses IP(I) (just as in Proposition I), and, by applying condition (P1) repeatedly, we see that $M - M_2$ possesses IP(I). But $S_1 \mid M - M_2 = S_2 \mid M - M_2$, so that $I(S_1) \mid OR(I) = I(S_2) \mid OR(I)$. Therefore, M_1 possesses IP(I).

Now let S_1 and S_2 be any two states of S such that $M_1 = \{x \in M: S_1(x) = S_2(x)\}$ is infinite. Let $y \in M_1$, and define a map $I: \mathcal{L} \to \mathcal{L}$ as follows: $I(S)(z) = S(z)$, $z \neq y$; $I(S)(y) = S_2(y)$ if $\{x \in M: S(x) \neq S_1(x)\}$ is finite; $I(S)(x) = S_1(y)$, otherwise. Clearly y is the only element of M that can be in OR(I), and since $S_1(y) \neq I(S_1)(y)$, we have $OR(I) = \{y\}$. We claim that $IR(I) = \emptyset$. To see this, suppose $x \in IR(I)$, and consider states S_1' and S_2' as in the definition of IR(I), with $S_1'(z) = S_2'(z)$ for $z \neq x$. Since $\{x \in M: S_1'(x) \neq S_1(x)\}$ and $\{x \in M: S_2'(x) \neq S_1(x)\}$ are either both finite or both infinite, we must have $I(S_1')(y) = I(S_2')(y)$, or $I(S_1') \mid OR(I) = I(S_2') \mid OR(I)$, a contradiction. Therefore $S_1 \mid IR(I) = S_2 \mid IR(I)$ is true vacuously, but $I(S_1) \mid OR(I) \neq I(S_2) \mid OR(I)$. This completes the proof.

4. The Product Model

We might also consider how to take account of computers which are part binary and part decimal; i.e., in which there are two base spaces B_1 and B_2, and the elements S(x) are in B_1 for $x \in M'$ (where $M' \subset M$ is the "binary part" of the memory) and in B_2 for $x \notin M'$. One way to do this is to consider a single base space $B = B_1 \cup B_2$, and to put a corresponding restriction on $\mathcal{S}$: $S \in \mathcal{S}$ if and only if $S(x) \in B_1$, $x \in M'$, and $S(x) \in B_2$, $x \notin M'$. This family $\mathcal{S}$ satisfies (P1).

Another way is to re-examine the structure of the set $\mathcal{S}$. The set of all maps S: $M \to B$ can be thought of as the cartesian product of copies of B, over M as the index set. The set of states $\mathcal{S}$ given in the preceding paragraph can be thought of as the cartesian product of two cartesian products: one of copies of B_1 over index set M'; the other, of copies of B_2 over index set M-M'. Might we consider a cartesian product of <u>arbitrary</u> sets B_x, for $x \in M$, where M is some index set? This would correspond to a set of maps S: $M \to B$, where B is the union of the (distinct) B_x, and $S \in \mathcal{S}$ if and only if $S(x) \in B_x$ for each $x \in M$. This, however, raises the question as to whether such a family $\mathcal{S}$ always satisfies (P1). For finite computers, the following proposition answers this question in a very strong manner.

PROPOSITION III. Let $(M, B, \mathcal{S}, \mathcal{L})$ be a finite computer (i.e., the memory M is finite). For each $x \in M$, let $B_x = \{b \in B:\ S(x) = b$ for some $S \in \mathcal{S}\}$. Then $\mathcal{S}$ is in fact the set of all maps S: $M \to B$ such that $S(x) \in B_x$ for all $x \in M$.

PROOF: Let $S: M \to B$ be any map such that $S(x) \in B_x$ for all $x \in M$. We wish to prove that $S \in \mathcal{S}$. Since $S(x) \in B_x$, there exists, for each $x \in M$, a map $S_x \in \mathcal{S}$ such that $S_x(x) = S(x)$. The proof is completed by applying condition (P1) repeatedly to the states S_x. Since the condition need only be applied a finite number of times (one for each element of M), the state S will be in $\mathcal{S}$; clearly no other states can be in $\mathcal{S}$.

Thus, if M is finite, the cartesian product of arbitrary sets B_x, over M as index set, corresponds to the set of states $\mathcal{S}$ of a computer, which satisfies (P1). Furthermore, in this case, we gain no generality by considering a subset of $\mathcal{S}$. For, if the subset $\mathcal{S}'$ satisfies (P1), then it defines its own subsets B'_x in the first place. Thus we are led to an alternate definition of a finite computer.

DEFINITION D. Let M be finite, and for each $x \in M$, let B_x be a non-empty set. Let $\mathcal{S} = \prod_{x \in M} B_x$. If $\mathcal{I}$ is a set of maps $I: \mathcal{S} \to \mathcal{S}$, then $(M, \mathcal{S}, \mathcal{I})$ is a <u>finite computer</u>. The index set M is the <u>memory</u>; the elements of $\mathcal{S}$ are the <u>states</u>; and the elements of $\mathcal{I}$ are the <u>instructions</u>.

If M is infinite, the statement of Proposition III does not hold. Even if $\mathcal{S}$ satisfies condition (P2), and $B_x = B$ for all $x \in M$, the most we can say is that, given any state $S \in \mathcal{S}$, $\mathcal{S}$ consists of all states S' such that $\{x \in M: S(x) \neq S'(x)\}$ is finite. This is exactly the situation which is known in algebra as a <u>restricted product</u>. For example, if we are given an infinite number of groups G_x, then the restricted product of the G_x is the set of all elements of the cartesian product which have only a finite number of non-identity elements. The result is a subgroup of the

"complete product," i.e., the cartesian product with multiplication performed by multiplying co-ordinates. We give a general definition of restricted products before passing to the generalization of Proposition XII.

DEFINITION E. Let M be an arbitrary index set, and for each $x \in M$ let B_x be a non-empty set and b_x an element of B_x. The restricted product of the B_x, relative to the b_x, is the set of all elements $z \in \prod_{x \in M} B_x$ such that, if z_x is the co-ordinate of z in $x \in M$, then $\{x \in M: z_x \neq b_x\}$ is finite.

PROPOSITION XV. Let $(M, B, \mathcal{S}, \mathcal{Q})$ be a computer, let $S_0 \in \mathcal{S}$, and for each $x \in M$, let $B_x = \{b \in B: S(x) = b \text{ for some } S \in \mathcal{S}\}$. Then $\mathcal{S}$ is in fact the set of all maps $S: M \to B$ such that $S(x) \in B_x$ for all $x \in M$, and such that $\{x \in M: S_0(x) \neq S(x)\}$ is finite.

Proof: Let $S: M \to B$ be any map such that $S(x) \in B_x$ for all $x \in M$ and let $\{x \in M: S_0(x) \neq S(x)\}$ be a finite set $M' \subseteq M$. Since $S(x) \in B_x$, there exists, for each $x \in M'$, a map $S_x \in \mathcal{S}$ such that $S_x(x) = S(x)$. The proof is completed by applying condition (P1) repeatedly to S_0 and the (finite) collection of states S_x. Since the condition need only be applied a finite number of times, the state S will be in $\mathcal{S}$. Clearly, under the conditions (P1) and (P2), no other maps $S: M \to B$ can be in $\mathcal{S}$.

Thus any restricted product of sets B_x corresponds to the set of states of a computer, satisfying (P1) and (P2). Furthermore, we again gain no generality by considering a subset of $\mathcal{S}$. Thus we may give our alternative definition of a computer in full generality.

DEFINITION F. Let M be an arbitrary non-empty set, and for each $x \in M$, let B_x be a non-empty set and b_x a member of B_x. Let $\mathcal{S}$ be the restricted

product of the B_x, relative to the b_x. If $\mathcal{I}$ is a set of maps $I : \mathcal{S} \to \mathcal{S}$, then $(M, \mathcal{S}, \mathcal{I})$ is a computer. The index set M is the memory; the elements of $\mathcal{S}$ are the states; and the elements of $\mathcal{I}$ are the instructions.

It should be emphasized that the two general definitions of a computer (Definition C and Definition P) are not really different; the set of states $\mathcal{S}$ is being viewed in a different guise in each case. For Definition P, certain auxiliary concepts need to be redefined. If M' is a subset of M, we denote the natural projection of $\prod_{x \in M} B_x$ onto $\prod_{x \in M'} B_x$ by $Pr(M, M')$, and if $S \in \mathcal{S}$, we denote the element $Pr(M, M')(S)$ by $S \mid M'$. The element $S \mid \{x\}$, where $x \in M$, is a member of B_x which may be denoted by $S(x)$. With these new definitions of $S(x)$ and $S \mid M'$, such concepts as input and output region may again be defined, and correspond exactly to these notions under the model of Definition C.

It may happen, under the product model, that some of the sets B_x have cardinality 1 (for example, by selecting a subset $\mathcal{S}$, all of whose members are such that $S(x) = b$ for some $x \in M$, $b \in B$). The internal structure of such a computer is the same as that of the computer obtained by eliminating all such sets B_x. In particular, an element $x \in M$ for which B_x has cardinality 1 can never be contained in $IR(I)$ or $OR(I)$, for any instruction I.

3. Composition and Decomposition

In this chapter, we shall use computers under Definition C. One of the first things we notice about computer instructions, as they are formulated here, is that if two instructions are performed one after the other, as they are in a real computer, the result is their composition. Specifically, if $I_1 : \mathcal{S} \to \mathcal{S}$ and $I_2 : \mathcal{S} \to \mathcal{S}$ are instructions on a computer, then $I_1 \circ I_2 = J$ and $I_2 \circ I_1 = J'$ are maps from $\mathcal{S}$ into $\mathcal{S}$, and as such have their own input and output regions. What relation do these regions have with the regions $IR(I_1)$, $OR(I_1)$, $IR(I_2)$, and $OR(I_2)$? The following proposition gives the answer.

PROPOSITION V. Let $(M, B, \mathcal{S}, \mathcal{O})$ be a computer, $I_1, I_2 \in \mathcal{S}$, and let $J : \mathcal{S} \to \mathcal{S}$ be defined by $J(S) = I_2(I_1(S))$. Then $IR(J) \subseteq IR(I_1) \cup IR(I_2)$ and $OR(I_1) - OR(I_2) \subseteq OR(J) \subseteq OR(I_1) \cup OR(J_2)$. If $IR(I_2) \cap OR(I_1) = \emptyset$, then $IR(I_2) \subseteq IR(J) \subseteq IR(I_1) \cup IR(I_2)$ and $OR(J) = OR(I_1) \cup OR(I_2)$.

Proof: To carry through calculations like this, it is helpful to have the following lemmatic facts.

(a) The input property $IP(I)$ is preserved under the taking of supersets, as well as intersections; if M' possesses $IP(I)$, and $M'' \supseteq M'$, then M'' possesses $IP(I)$. The input region $IR(I)$ is the intersection of all subsets of M which possess $IP(I)$. Therefore, a subset possesses $IP(I)$ if and only if it contains $IR(I)$

(b) If M' possesses $IP(I)$, then $S_1 \mid M' = S_2 \mid M'$ implies $I(S_1) \mid M' = I(S_2) \mid M'$. For let $S_1 \mid M' = S_2 \mid M'$ and let $y \in M'$. If $y \in OR(I)$, then $I(S_1)(y) = I(S_2)(y)$ by Propositions I and II. If $y \notin OR(I)$, then

$I(S_1)(y) = S_1(y) = S_2(y) = I(S_2)(y)$, by definition of OR(I) and by hypothesis. The lemma is thereby proved.

(c) If $IR(I_2) \cap OR(I_1) = \emptyset$, then for each $S \in \mathcal{S}$ we have $I_2(S) | OR(I_2) = J(S) | OR(I_2)$. To see this, let $x \in IR(I_2)$; we have $S(x) = I_1(S)(x)$, since $x \notin OR(I_1)$, and this shows that $S | IR(I_2) = I_1(S) | IR(I_2)$. By Propositions I and II, $I_2(S) | OR(I_2) = I_2(I_1(S)) | OR(I_2) = J(S) | OR(I_2)$.

We proceed to the calculations. Let $x \notin OR(I_1)$, $x \notin OR(I_2)$. Then $S(x) = I_1(S)(x)$ and $S(x) = I_2(S)(x)$ for <u>all</u> states $S \in \mathcal{S}$, so $S(x) = I_1(S)(x) = I_2(I_1(S))(x) = J(S)(x)$. Therefore $x \notin OR(J)$, which shows that $OR(J) \subseteq OR(I_1) \cup OR(I_2)$. Also, if $x \in OR(I)$ but $x \notin OR(I_2)$, so that there exists a state $S \in \mathcal{S}$ with $S(x) \neq I_1(S)(x)$ but $I_1(S)(x) = I_2(I_1(S))(x) = J(S)(x)$, then $S(x) \neq J(S)(x)$, so that $x \in OR(J)$; this shows that $OR(I_1) - OR(I_2) \subseteq OR(J)$.

Now let $x \notin IR(I_1)$, $x \notin IR(I_2)$. Let S_1 and S_2 be any two states of $\mathcal{S}$ such that $S_1(z) = S_2(z)$, $z \neq x$; i.e., if $M_1 = M - \{x\}$, then $S_1 | M_1 = S_2 | M_1$. Since M_1 possesses $IP(I_1)$, we have $I_1(S_1) | M_1 = I_1(S_2) | M_1$, and also $I_1(S_1) | OR(I_1) = I_1(S_2) | OR(I_1)$. Since $M_1 \cup OR(I_1)$ possesses $IP(I_2)$, we have $I_2(I_1(S_1)) | M_1 \cup OR(I_1) = I_2(I_1(S_2)) | M_1 \cup OR(I_1)$, and also $I_2(I_1(S_1)) | OR(I_2) = I_2(I_1(S_2)) | OR(I_2)$; that is, $J(S_1) | M_1 \cup OR(I_1) \cup OR(I_2) = J(S_2) | M_1 \cup OR(I_1) \cup OR(I_2)$, and since $OR(J) \subseteq OR(I_1) \cup OR(I_2)$, we have $J(S_1) | OR(J) = J(S_2) | OR(J)$. Thus $x \notin IR(J)$, and $IR(J) \subseteq IR(I_1) \cup IR(I_2)$.

If $IR(I_2) \cap OR(I_1) = \emptyset$, then let $x \in OR(I_1) \cup OR(I_2)$. If $x \in OR(I_2)$, then $x \in OR(J)$ by the above. If $x \in OR(I_2)$, then let S be a state such that $S(x) \neq I_2(S)(x)$. By (c) above, we have $I_2(S)(x) = J(S)(x)$, so that $S(x) \neq J(S)(x)$ and $x \in OR(J)$. Thus $OR(I_1) \cup OR(I_2) = OR(J)$, since the inequalities hold in both directions.

To show that $IR(I_2) \subseteq IR(J)$, under the same hypothesis, we may show, by (a) above, that $IR(J)$ possesses $IP(I_2)$. Let $S_1, S_2 \in \mathcal{S}$ be such that $S_1\ IR(J) = S_2\ IR(J)$; then $J(S_1) \mid OR(J) = J(S_2) \mid OR(J)$. Since $OR(I_2) \subseteq OR(J)$, we have $J(S_1) \mid OR(I_2) = J(S_2) \mid OR(I_2)$. Applying (c) above, we see that $I_2(S_1) \mid OR(I_2) = I_2(S_2) \mid OR(I_2)$. Thus $IR(J)$ possesses $IP(I_2)$, completing the proof of Proposition 7.

The condition $IR(I_2) \cap OR(I_1) = \emptyset$, under which stronger statements can be made about input and output regions, is usually less interesting than its opposite. For example, if we perform a "clear and add X" followed by a "store in Y", we are using two instructions in which the input region of the second overlaps (in fact, coincides with) the output region of the first.

Under even stronger conditions, such as when $IR(I_2) \cap OR(I_1) = \emptyset$ and $OR(I_1) \cap OR(I_2) = \emptyset$, we can show that $IR(J) = IR(I_1) \cup IR(I_2)$. In fact, a more general statement holds, with a weaker conclusion: If $OR(I_1) \cap OR(I_2) = \emptyset$, and $OR(I_1) \cup OR(I_2) = OR(J)$, then $IR(I_1) \subseteq IR(J)$. To show this, we have only to show, as before, that $IR(J)$ possesses $IP(I_1)$. Let $S_1, S_2 \in \mathcal{S}$ be such that $S_1 \mid IR(J) = S_2 \mid IR(J)$, and let $x \notin OR(I_1)$; we must show that $I_1(S_1)(x) = I_1(S_2)(x)$. Since $x \in OR(I_2)$, we have $I_1(S_1)(x) = I_2(I_1(S_1))(x) = J(S_1)(x)$, and also $I_1(S_2)(x) = J(S_2)(x)$. Since $x \in OR(J)$, we have $J(S_1)(x) = J(S_2)(x)$. Therefore $I(S_1)(x) = I(S_2)(x)$. The statement made above, that $IR(I_2) \cap OR(I_1) = \emptyset$ and $OR(I_1) \cap OR(I_2) = \emptyset$ implies $IR(J) = IR(I_1) \cup IR(I_2)$, now follows from all of the preceding.

If $J': \mathcal{S} \to \mathcal{S}$ is defined by $J'(S) = I_1(I_2(S))$, then $J = J'$ (i.e., I_1 and I_2 will commute) if $OR(I_1) \cap (IR(I_2) \cup OR(I_2)) = \emptyset$ and $OR(I_2) \cap (IR(I_1) \cup OR(I_1)) = \emptyset$.

Under what conditions can we <u>decompose</u> an instruction $I: \mathcal{S} \to \mathcal{S}$ into the product of simpler maps from $\mathcal{S}$ into $\mathcal{S}$ (regardless of whether these are instructions on $\mathcal{S}$)? It should be apparent by now that the "simplest" instructions, in our sense, are those which have small input and output regions. It turns out that, if $IR(I) \cap OR(I) = \emptyset$, the instruction I can be written as the product of instructions having output region $\{x\}$, for $x \in OR(I)$. Even when $IR(I) \cap OR(I) = \emptyset$, we can "split off" the elements of $OR(I) - IR(I)$. Thus we reduce to the case $IR(I) \supseteq OR(I)$.

PROPOSITION VI. Let $x \in OR(I) - IR(I)$. Then I may be written as $I(S) = I_2(I_1(S))$, where $IR(I_1) \subseteq IR(I)$, $IR(I_2) \subseteq IR(I)$, $OR(I_1) = \{x\}$, and $OR(I_2) = OR(I) - \{x\}$.

Proof: Define $I_1(S)(x) = I(S)(x)$, $I_1(S)(z) = S(z)$, $z \neq x$; and define $I_2(S)(x) = S(x)$, $I_2(S)(z) = I(S)(z)$, $z \neq x$. Then $I_2(I_1(S))(x) = I_1(S)(x) = I(S)(x)$, and, for $z \neq x$, $I_2(I_1(S))(z) = I(I_1(S))(z)$. By lemma (b) in Proposition V, since $M-\{x\} \supseteq IR(I)$ and $I_1(S) \mid M-\{x\} = S \mid M-\{x\}$, we have $I(I_1(S)) \mid M-\{x\} = I(S) \mid M-\{x\}$. Thus $I(S) = I_2(I_1(S))$. It is clear from the definitions that $OR(I_1) \subseteq \{x\}$ and $OR(I_2) \subseteq OR(I)-\{x\}$. By Proposition V, $OR(I) \subseteq OR(I_1) \cup OR(I_2)$; intersecting both sides of this statement first with $\{x\}$ and then with $M-\{x\}$, we obtain $\{x\} \subseteq OR(I_1)$ and $OR(I)-\{x\} \subseteq OR(I_2)$, so that $OR(I_1) = \{x\}$ and $OR(I_2) = OR(I) - \{x\}$. If $S_1, S_2 \in \mathcal{S}$ are such that $S_1 \mid IR(I) = S_2 \mid IR(I)$, then $I(S_1) \mid OR(I) = I(S_2) \mid OR(I)$, so that $I_1(S_1)(x) = I_1(S_2)(x)$, and $I_2(S_1) \mid OR(I) - \{x\} = I_2(S_2) \mid OR(I) - \{x\}$. Hence $IR(I)$ satisfies both input properties $IP(I_1)$ and $IP(I_2)$, and $IR(I_1) \subseteq IR(I)$, $IR(I_2) \subseteq IR(I)$.

6. Products and Restructuring

In this chapter, we shall use computers as in Definition F. We would like to answer the questions: Can we have a product of two computers? Can we have a subcomputer of a computer? It turnes out that there is another question, related to these two, but of greater interest than either of them.

Since the set of states $\mathcal{S}$ of a computer is a product over an index set M, it would seem that two computers can be combined by taking a product over both index sets. If $\mathcal{S}_1$ and $\mathcal{S}_2$ are the sets of states of the two computers, and $\mathcal{S}$ is the set of states of the product, then $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$; and if $I_1 : \mathcal{S}_1 \to \mathcal{S}_1$ and $I_2 : \mathcal{S}_2 \to \mathcal{S}_2$, then we may define, in a natural way, $I_1 \times I_2 : \mathcal{S}_1 \times \mathcal{S}_2 \to \mathcal{S}_1 \times \mathcal{S}_2$.

DEFINITION G. Let $(M_1, \mathcal{S}_1, \mathcal{I}_1)$ and $(M_2, \mathcal{S}_2, \mathcal{I}_2)$ be two computers in the sense of Definition F; that is, for each element of M_1 and of M_2 we have defined a set B_x and an element b_x, and $\mathcal{S}_1$ ($\mathcal{S}_2$) is the restricted product, over M_1 (over M_2), of the B_x relative to the b_x. Let $\mathcal{S}$ be the restricted product of the B_x relative to the b_x over $M_1 \cup M_2$. Then $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$. If $I_1 \in \mathcal{I}_1$ and $I_2 \in \mathcal{I}_2$, we define the map $I_1 \times I_2 : \mathcal{S} \to \mathcal{S}$ by $(I_1 \times I_2)(S_1, S_2) = (I_1(S_1), I_2(S_2))$, and we denote the set of all such $I_1 \times I_2$, for $I_1 \in \mathcal{I}_1$, $I_2 \in \mathcal{I}_2$, by $\mathcal{I}_1 \times \mathcal{I}_2$. Then the computer $(M_1 \cup M_2, \mathcal{S}, \mathcal{I}_1 \times \mathcal{I}_2)$ is the <u>product</u> of the two computers $(M_1, \mathcal{S}_1, \mathcal{I}_1)$ and $(M_2, \mathcal{S}_2, \mathcal{I}_2)$.

If $I = I_1 \times I_2$, then the input and output regions of I are the unions, respectively, of the input and output regions of I_1 and I_2. Computers which are products, in this form, almost never occur in the real world. This is so even though our definition, despite its length, is the most natural definition we can make of a product computer. Several situations with real computers--for example, the addition of extra memory to an already existing computer--almost correspond to the taking of a product. It may be argued that the addition of extra memory always involves some complexity of a purely technical nature. But there is another perfectly good reason, and this is that product instructions are in a sense incomplete. Data can never be moved, under a product instruction, from the memory M_1 to the memory M_2, or vice versa, despite the fact that the input and output regions of the product may intersect both M_1 and M_2. We say that M_1 cannot "affect" M_2. This is one of the bases for our introduction of affected regions in the next chapter.

If M is the product of M_1 and M_2, then M_1 and M_2 are certainly "subcomputers" of M. As a matter of fact, this kind of subcomputer is the only kind that makes sense, unless we change our definition of product (for example, to cover enlargement of the sets B_x).

DEFINITION H. Let $(M, \mathscr{S}, \mathscr{I})$ be a computer in the sense of Definition F, and let M' be a subset of M. Let $\mathscr{S}'$ be the restricted product, over the index set M', of the B_x relative to the b_x (defined by M and $\mathscr{S}$). Suppose that for each instruction $I \in \mathscr{I}$ there exists a map $I': \mathscr{S}' \to \mathscr{S}'$, such that $I'(S \mid M') = (I(S)) \mid M'$ for each $S \in \mathscr{S}$. Let $\mathscr{I}'$ be the collection of all such $\mathscr{I}$. Then $(M', \mathscr{S}', \mathscr{I}')$ is a <u>subcomputer</u> of $(M, \mathscr{S}, \mathscr{I})$.

In particular, if $(M, \mathcal{S}, \mathcal{L})$ is the product of $(M_1, \mathcal{S}_1, \mathcal{L}_1)$ and $(M_2, \mathcal{S}_2, \mathcal{L}_2)$, then $(M_1, \mathcal{S}'_1, \mathcal{L}_1)$ and $(M_2, \mathcal{S}_2, \mathcal{L}_2)$ are, in this sense, subcomputers of $(M, \mathcal{S}, \mathcal{L})$. The question may be raised as to whether there is a condition on the input and output regions of the instructions of which would be necessary and sufficient for subcomputers of a certain form to exist. There is a condition, but it is on the affected regions, which are defined in Chapter 7.

A more interesting construction is the <u>restructuring</u> of a computer. Let $(M, \mathcal{S}, \mathcal{L})$ be a computer and let $\mathcal{D}$ be a decomposition of M, that is, a class of disjoint non-empty subsets of M whose union is M. For each $D \in \mathcal{D}$, let B_D be the restricted product of the B_x, relative to the b_x, over D as an index set, and let b_D be the element of this restricted product whose co-ordinate on the component B_x is b_x. Let $\mathcal{S}'$ be the restricted product of the B_D relative to the b_D, over D as an index set. Then there is a canonical one-to-one correspondence between $\mathcal{S}$ and $\mathcal{S}'$; to the element of $\mathcal{S}$ whose co-ordinate at $x \in M$ is a_x corresponds the element of $\mathcal{S}'$ whose co-ordinate at $D \in \mathcal{D}$, where $x \in D$, is a member of B_D whose co-ordinate at x is a_x. Thus it is possible to speak of $\mathcal{L}$ as a set of maps $L: \mathcal{S}' \to \mathcal{S}'$.

DEFINITION J. Let $(M, \mathcal{S}, \mathcal{L})$ be a computer, and let $\mathcal{D}$ be an arbitrary decomposition of M as above. Then $\mathcal{D}$ is called a <u>memory structure</u> for $(M, \mathcal{S}, \mathcal{L})$. The process of passing from $(M, \mathcal{S}, \mathcal{L})$ to $(\mathcal{D}, \mathcal{L}', \mathcal{L})$, as above, is called <u>restructuring</u> $(M, \mathcal{S}, \mathcal{L})$.

If $I \in \mathcal{I}$, then I has input and output regions as an instruction of $(M, \mathcal{L}, \mathcal{I})$ and as an instruction of $(\mathcal{D}, \mathcal{L}', \mathcal{I})$. The regions in $\mathcal{D}$ are found by including any member of $\mathcal{D}$ which contains a member of the corresponding regions in M.

Restructuring is a much commoner process than taking products. In the first place, the restructured computer has the same states as the original computer, and therefore restructuring is not a physical process (such as a change in hardware) so much as a logical reorientation of the way we view the computer. And it is a process in which computer programmers engage quite often. If a block of data in a binary computer is alphanumeric, for example, we think of the block as broken up into six-bit characters, which is equivalent to taking a memory structure in which the six-bit groups are sets occurring in the decomposition $\mathcal{D}$. Over these sets the B_D have order 64. Again, we can take a memory structure in which computer words are members of the decomposition. Over a computer word D, the set B_D will have cardinality 2^{36} (or 2^b, where there are b bits in a word). If the word is floating point, and we ignore the fact that decimals are given to only a finite accuracy, we can think of B_D as being the set of real numbers, thus bringing in an "idealized" computer.

The concept of restructuring also answers a question we may have had about the property (P1). Given a 4-tuple $(M, B, \mathcal{L}, \mathcal{I})$, as in Definition C, in which $\mathcal{L}$ does not satisfy (P1), but "almost" does so, and at the same time does satisfy (P2). How many of the properties of the set of states of a computer does $\mathcal{L}$ "almost" preserve? Putting this question on a rigorous basis, let us look at all subsets M' of M, relative to which (P1) is always satisfied. By the following theorem, this set always consists of the

members of a decomposition $\mathcal{D}$ and their unions. The concept of restructuring makes sense even without (*1), and so we may restructure our "almost-computer" by the decomposition $\mathcal{D}$ --and the result will be a computer.

PROPOSITION VII. Let M and B be arbitrary sets, let $\mathcal{S}$ be a set of maps $S: M \to B$ which satisfies (P2) (as in Definition C), and let $\mathcal{L}$ be a set of maps $L: \mathcal{S} \to \mathcal{S}$. Let $\mathcal{P}$ be a class of subsets of M, as follows: $P \in \mathcal{P}$ if and only if, given $S_1, S_2 \in \mathcal{S}$, there exists $S_3 \in \mathcal{S}$, with $S_3(x) = S_1(x)$ for $x \in P$ and $S_3(x) = S_2(x)$ for $x \notin P$. Then $\mathcal{P}$ consists of the members of a decomposition $\mathcal{D}$ of M, and their unions. If $(M, \mathcal{S}, \mathcal{L})$ is restructured under $\mathcal{D}$, then $(\mathcal{D}, \mathcal{S}', \mathcal{L}')$, as in Definition J, is a computer.

Proof: To show that $\mathcal{P}$ consists of the members of $\mathcal{D}$ and their unions, it suffices to show that $\mathcal{P}$ is closed under the formation of complements and arbitrary intersections. For let $x \in M$, and let D_x be the intersection of all members of $\mathcal{P}$ containing x; there exist such members, in fact, M itself is one of them. $D_x \in \mathcal{P}$ and $x \in D_x$. If D_y and D_z are not identical, say $D_y - D_z \neq \emptyset$, there exists $x \in D_y$, $x \notin D_z$; then $D_y \cap (M - D_z)$ is in $\mathcal{P}$, contains y, and is contained in D_y, and is therefore equal to D_y, so that $D_y \cap D_z = \emptyset$. Hence the D_x are either disjoint or identical. By De Morgan's Laws, $\mathcal{P}$ is closed under the formation of arbitrary unions, so that every union of sets D_x is in $\mathcal{P}$; if $P \in \mathcal{P}$ and $x \in P$, then $D_x \subseteq P$, so that P is a union of sets D_x. Hence $\mathcal{P}$ consists of the members of the decomposition $\mathcal{D}$ into the sets D_x, and their unions.

That $\mathcal{P}$ is closed under the formation of complements is obvious. If $S_1, S_2 \in \mathcal{S}$, and contains S_3: $S_3 \mid P = S_1 \mid P$, $S_3 \mid M-P = S_2 \mid M-P$, and S_4: $S_4 \mid Q = S_1 \mid Q$, $S_4 \mid M-Q = S_2 \mid M-Q$, then $\mathcal{S}$ contains S_5: $S_5 \mid Q = S_3 \mid Q$, $S_5 \mid M-Q = S_2 \mid M-Q$, from which it follows that $S_5 \mid P \cap Q = S_1 \mid P \cap Q$, $S_5 \mid M-(P \cap Q) = S_2 \mid M-(P \cap Q)$. Hence $\mathcal{P}$ is closed under finite intersections.

Now let A be an arbitrary index set, and for each $a \in A$ let $P_a \in \mathcal{P}$; we show that $\bigcap_{a \in A} P_a \in \mathcal{P}$. If $S_1, S_2 \in \mathcal{L}$, we let $N = \{x \in M: S_1(x) \neq S_2(x)\}$; since $\mathcal{L}$ satisfied property (P2), N is finite. For each $a \in A$, let $N_a = N \cap P_a$; since all N_a are finite, there is a finite subset $A' \subseteq A$ such that $\bigcap_{a \in A'} N_a = \bigcap_{a \in A} N_a$. If S_3 is such that $S_3 \mid N' = S_1 \mid N'$, $S_3 \mid M-N' = S_2 \mid M-N'$, where $N' = \bigcap_{a \in A} N_a$, then trivially $S_3 \mid P_a - N_a = S_1 \mid P_a - N_a = S_2 \mid P_a - N_a$ for each $a \in A$, so that $S_3 \mid N'' = S_1 \mid N''$, $S_3 \mid M-N'' = S_2 \mid M-N''$, where $N'' = \bigcap_{a \in A} P_a$. Thus $\mathcal{P}$ is closed under the formation of arbitrary intersections, and, by the preceding paragraph, consists of the members of a decomposition of M and their unions.

To prove the last statement of Proposition VII, it is first necessary to redefine restructuring in terms of Definition C of a computer. If $(M, B, \mathcal{L}, \mathcal{D})$ is a computer under Definition C (with the possible exception of property (P1)), and $\mathcal{D}$ is a decomposition of M, then we may define the restructured computer $(\mathcal{D}, B', \mathcal{L}', \mathcal{L})$ as follows (again possibly without (P1)). The set of all distinct $S \mid M'$, for $M' \subseteq M$, will be denoted by $\mathcal{L} \mid M$. Then B' is the union of all $\mathcal{L} \mid D_x$, where $D_x \in \mathcal{D}$ and $x \in D_x$, and $\mathcal{L}'$ is the set of all maps $S: \mathcal{D} \to B'$ such that $S(D_x) \in \mathcal{L} \mid D_x$ for all $D_x \in \mathcal{D}$. Now, if $\mathcal{L}$ satisfies (P2), but not necessarily (P1), and if $\mathcal{P}$ and $\mathcal{D}$ are constructed as above, then the restructured computer $(\mathcal{D}, B', \mathcal{L}', \mathcal{L})$ will be such that $\mathcal{L}'$ satisfies (P1) relative to $\mathcal{D}$. For let $S_1', S_2' \in \mathcal{L}'$ and $\mathcal{D}^0 \subseteq \mathcal{D}$; we are to verify that $S_3' \in \mathcal{L}'$, where $S_3'(D)$, for $D \in \mathcal{D}^0$, and $S_3'(D) = S_2'(D)$, for $D \notin \mathcal{D}^0$. If $S_1, S_2 \in \mathcal{L}$ correspond to $S_1', S_2' \in \mathcal{L}'$, and M' is the union of all $D \in \mathcal{D}^0$, then, by construction of $\mathcal{D}^0$, $M' \in \mathcal{P}$,

and there exists a state $S_3 \in \mathcal{S}$ with $S_3(x) = S_1(x)$, $x \in M'$, and $S_3(x) = S_2(x)$, $x \notin M'$. This state S_3 corresponds to $S_3' \in \mathcal{S}'$, completing the proof of Proposition VII.

7. Affected Regions

Affected and affecting regions, as we define them here, are a substructure on the input and output regions of an instruction. According to our definitions, the input region OR(I) is the set of all elements of memory which can be affected by I; the input region IR(I) is the set of all such elements which can affect OR(I). Given a subset of IR(I), what elements can affect it? We give precise definitions of these concepts.

DEFINITION K. Let $(M, B, \mathcal{L}, \mathcal{M})$ be a computer and let I be an instruction on I. Then

$$AR(M', I) = \{x \in OR(I): \exists\ S_1, S_2 \in \mathcal{L} : S_1(z) \text{ for } z \in IR(I), z \notin M', \text{ and } I(S_1)(x) \neq I(S_2)(x)\}$$

for each set $M' \subseteq IR(I)$, and

$$RA(M'', I) = \{x \in IR(I): AR(\{x\}, I) \cap M' \neq \emptyset\}$$

for each set $M'' \subseteq OR(I)$. We call $AR(M', I)$ an <u>affected region</u>, or the <u>region affected by</u> M' <u>under</u> I, and we call $RA(M'', I)$ an <u>affecting region</u> or the <u>region affecting</u> M'' <u>under</u> I, or the <u>region which affects</u> M' <u>under</u> I.

Every non-null subset of IR(I) affects some non-null subset of OR(I), otherwise there would exist $L \subseteq IR(I)$ such that $S_1 \mid IR(I)-L = S_2 \mid IR(I)-L$ implies $I(S_1) \mid OR(I) = I(S_2) \mid OR(I)$, and the set IR(I)-L would possess IP(I), contrary to the statement that IR(I) is the intersection of all subsets of M possessing IP(I). Also, $AR(\emptyset, I) = \emptyset$ and $RA(\emptyset, I) = \emptyset$; the second statement is trivial, and the first is just a restatement of Propositions I and II. It follows directly from the definition that, if $M' \subseteq M''$, then $AR(M', I) \subseteq AR(M'', I)$, and thus $AR(A \cap B, I) \subseteq AR(A, I) \cap$

$AR(B, I)$ and $AR(A \cup B, I) \supseteq AR(A, I) \cup AR(B, I)$. In fact, $AR(A \cup B, I) = AR(A, I) \cup AR(B, I)$; to show the inequality in the opposite direction, let $S_1, S_2 \in \mathcal{S}$ be such that $S_1 \mid IR(I)-(A \cup B) = S_2 \mid IR(I)-(A \cup B)$. Let $S_3 \in \mathcal{S}$ be defined by $S_3 \mid A = S_1 \mid A$, $S_3 \mid M-A = S_2 \mid M-A$. Then $S_3 \mid IR(I)-A = S_2 \mid IR(I)-A$, and, since $IR(I)-(A \cup B) \cup (A-B) = IR(I)-B$, $S_3 \mid IR(I)-B = S_1 \mid IR(I)-B$. Thus $I(S_1) \mid OR(I)-AR(B, I) = I(S_3) \mid OR(I)-AR(B, I)$ and $I(S_2) \mid OR(I)-AR(A, I) = I(S_3) \mid OR(I)-AR(A, I)$, and therefore $I(S_1) \mid OR(I)-(AR(A, I) \cup AR(B, I)) = I(S_2) \mid OR(I)-(AR(A, I) \cup AR(B, I))$. This shows that a member of $AR(A \cup B, I)$ cannot lie outside $AR(A, I) \cup AR(B, I)$, i.e., $AR(A \cup B, I) \subseteq AR(A, I) \cup AR(B, I)$.

It therefore follows that all the affected regions of an instruction I are determined by the affected regions of single elements $x \in IR(I)$. These regions can be single elements of OR(I), or they can be the entire output region. In the instruction (ADD Y) on a computer with an accumulator, the input region is $Y \cup AC$, and the output region is AC; the affected region of each bit position of Y (or of AC) is the corresponding bit position of AC, together with all the bit positions to its left (provided there is no provision for overflow). If there is end-around carry, the affected region of every bit position of Y or AC is the entire AC.

On the other hand, some elements of OR(I) may not be affected at all; i.e., we may have $AR(IR(I), I) \neq OR(I)$. This may happen, in particular, for an instruction in which $IR(I) = \emptyset$, that is, a constant instruction (such as "store zero"). In such an instruction $S \mid OR(I)$ is a constant, i.e., independent of S. Every instruction may be written as an instruction for

which AR(IR(I), I) = OR(I), followed by a constant instruction.

We may now formalize the statement we made in the last chapter about the product of two computers.

PROPOSITION VIII. Let $(M_1, \mathcal{A}_1, \mathcal{I}_1)$ and $(M_2, \mathcal{A}_2, \mathcal{I}_2)$ be computers, and let $(M, \mathcal{A}, \mathcal{I})$ be their product. Let $I = I_1 \times I_2$ be an instruction of $\mathcal{I}$, where $I_1 \in \mathcal{I}_1$ and $I_2 \in \mathcal{I}_2$. Then $IR(I) = IR(I_1) \cup IR(I_2)$ and $OR(I) = OR(I_1) \cup OR(I_2)$. Furthermore, $AR(IR(I_1), I) = OR(I_1)$ and $AR(IR(I_2), I) = OR(I_2)$. Thus, under a product instruction, no member of either memory can affect the other.

The proof is by direct computation.

As an illustration, suppose we take M_1 to be the core memory and the registers of a real computer, $\mathcal{A}_1$ to be the set of all maps from M_1 to $\{0, 1\}$, and $\mathcal{I}$ to be the set of all instructions in this computer that can be defined without reference to a location counter (arithmetic, logical, shift, floating point instructions, etc.) Let us take M_2 to consist of one element, called a <u>location counter</u>; $\mathcal{A}_2$ consists of all maps from M_2 to $\{0, 1, \ldots, c-1\}$, where c is the size of core, and $\mathcal{I}$ consists of one instruction I which increases the counter by 1: $I(S)(M_2) = S(M_2)+1$. If we take the product of these two computers, we get nothing essentially better than what we had to start with: each instruction increases the value of the location counter by 1, and that is all. We obtain an improvement when we let M_1 affect M_2: letting the location counter affect the contents of core (by decoding the instruction stored at the given location). We obtain an even further improvement when we let M_2 affect M_1. An instruction

which is such that the contents of core affect the location counter is known as a conditional jump (or transfer, or branch).

We obtain another proposition on subcomputers.

PROPOSITION IX. Let $(M, \mathcal{S}, \mathcal{I})$ be a computer, and let M' be a sub-set of M. Let $\mathcal{S}'$ be the restricted product of the B_x relative to the b_x for $x \in M'$ as in Definition H. Then there exists a subcomputer $(M', \mathcal{S}', \mathcal{I}')$ of $(M, \mathcal{S}, \mathcal{I})$ if and only if, for each $I \in \mathcal{I}$ and each $x \in IR(I)$, $AR(\{x\}, I) \cap M' = \emptyset$ if $x \notin M'$.

Thus M' is the memory of a subcomputer if and only if M' is never affected by M-M' (although M' can affect M-M').

Proof: Suppose the condition holds. Let $I \in \mathcal{I}$ and define I': $\mathcal{S}' \to \mathcal{S}'$ as follows. Let $S \mid M' \in \mathcal{S}'$, let $S_1 \in \mathcal{S}$ be such that $S_1 \mid M' = S \mid M'$ (e.g. $S_1 = S$) and defined $I'(S \mid M') = (I(S_1)) \mid M'$. If $S_2 \in \mathcal{S}$ is another state with $S_2 \mid M' = S \mid M'$, then, since no element of M' is in $AR(\{x\}, I)$ for $x \notin M'$, we have $(I(S_1)) \mid M' = (I(S_2)) \mid M'$; thus the instruction I, as above, is well defined. If $\mathcal{I}$ is the class of all such I', then $(M', \mathcal{S}', \mathcal{I}')$ is a subcomputer of $(M, \mathcal{S}, \mathcal{I})$. Conversely, if $y \in M' \cap AR(\{x\}, I)$, where $x \notin M'$, and $S_1, S_2 \in \mathcal{S}$ are such that $S_1 \mid M - \{x\} = S_2 \mid M - \{x\}$ (so that in particular $S_1 \mid M' = S_2 \mid M'$) and $I(S_1)(y) \neq I(S_2)(y)$, then no instruction I' can be defined on M' such that $I'(S \mid M') = (I(S)) \mid M'$ for each $S \in \mathcal{S}$.

8. A Test for the Validity of an Algorithm

Let us consider two instructions, I_1 and I_2. If $A \subseteq IR(I_1)$, we may calculate $AR(A, I_1) = A'$, and then $AR(A', I_2) = A''$. What relation does A'' have to $AR(A, I_1I_2)$? The trouble is that these regions may not all be defined; for instance, A' may not be contained in $IR(I_2)$. For this and other reasons, we will need a definition of affected region which does not depend on IR(I) and OR(I). Fortunately, this definition has a simple relation to the preceding one.

DEFINITION L. Let $(M, B, \mathscr{A}, \mathscr{L})$ be a computer and let I be an instruction of $\mathscr{L}$. Then

$$AR_2(M', I) = \{x \in M: \exists\ S_1, S_2 \in \mathscr{A},\ S_1(z) = S_2(z) \text{ for } z \notin M' \text{ and } I(S_1)(x) \neq I(S_2)(x)\}$$

for each subset $M' \subseteq M$. We call $AR_2(M', I)$ the <u>second affected region</u> of M' under I.

Now let I_1 and I_2 be two instructions and let I be their product: $I(S) = I_2(I_1(S))$. We verify the relation

$$AR_2(M', I) \subseteq AR_2(AR_2(M', I_1), I_2)$$

for each subset $M' \subseteq M$. It is easier to do this if we first characterize $AR_2(M', I)$ in a different way. We recall that IR(I) is the intersection of all subsets N of M which possess the predicate IP(I): if $S_1 \mid N = S_2 \mid N$, then $I(S_1) \mid OR(I) = I(S_2) \mid OR(I)$. Similarly, we may characterize $AR_2(M', I)$: if $S_1 \mid M\text{-}M' = S_2 \mid M\text{-}M'$, then $I(S_1) \mid M\text{-}N = I(S_2) \mid M\text{-}N$. For x is not in this intersection, if and only if there exists such a set N to which x

does not belong, i.e., $S_1 \mid M\text{-}M' = S_2 \mid M\text{-}M'$ implies $I(S_1)(x) = I(S_2)(x)$ (since $x \in M\text{-}M'$): that is, if and only if $x \notin AR_2(M', I)$.

It remains to show that $AR_2(M', I)$ itself possesses $AP_2(M', I)$; this is left as an exercise along the lines of Propositions I and II, using properties (P1) and (P2) of $\mathcal{S}$. It follows from this that a set $N \subseteq M$ possesses $AP_2(M', I)$ if and only if it contains $AR_2(M', I)$, since $AP_2(M', I)$ is clearly preserved under the taking of supersets. Now the verification of our equation is straightforward. If $S_1 \mid M\text{-}M' = S_2 \mid M\text{-}M'$, then $I_1(S_1) \mid M\text{-}AR_2(M', I_1) = I_1(S_2) \mid M\text{-}AR_2(M', I_1)$, so that $I_2(I_1(S_1)) \mid M\text{-}AR_2(AR_2(M', I_1), I_2)$ possesses $AP_2(M', I)$, and thus $AR_2(M', I) \subseteq AR_2(AR_2(M', I_1), I_2)$.

We now give the relation between $AR(M', I)$ and $AR_2(M', I)$ when $M' \subseteq IR(I)$. In order to do this we must first introduce the unit component of a computer.

DEFINITION M. Let $(M, \mathcal{L}, \mathcal{S})$ be a computer as in Definition F, and let $Z \subseteq M$ be the set of all elements $x \in M$ such that B_x has exactly one element. Then Z is the <u>unit component</u> of $(M, \mathcal{L}, \mathcal{S})$.

(In the same way, we could speak of the binary component, the decimal component, etc.)

PROPOSITION X. $AR_2(M', I) = AR(M, I) \cup (M' - OR(I)\text{-}Z)$, where Z is the unit component of $(M, \mathcal{L}, \mathcal{S})$.

Proof: Let $S_1, S_2 \in \mathcal{L}$ be such that $S_1 \mid M\text{-}M' = S_2 \mid M\text{-}M'$. Then $I(S_1) \mid M\text{-}M'\text{-}OR(I) = S_1 \mid M\text{-}M'\text{-}OR(I) = S_2 \mid M\text{-}M'\text{-}OR(I) = I(S_2) \mid M\text{-}M'\text{-}OR(I)$; also $I(S_1) \mid OR(I)\text{-}AR(M', I) = I(S_2) \mid OR(I)\text{-}AR(M', I)$ since $S_1 \mid IR(I)\text{-}M' = S_2 \mid IR(I)\text{-}M'$; finally, $I(S_1) \mid Z = I(S_2) \mid Z$. This shows that $(A \cup OR(I)) \cap (OR(I)\text{-}AR(M', I)) \cap (M\text{-}Z)$ possesses $AP_2(M', I)$, and since $AR(M', I) \subseteq$

OR(I), this is equal to $AR(M', I) \cup (M' - OR(I) - Z)$. Thus $AR_2(M', I) \subseteq AR(M', I) \cup (M' - OR(I) - Z)$. Conversely, let $x \in AR_2(M', I)$; then for every two states $S_1, S_2 \in \mathcal{S}$ with $S_1 \mid M-M' = S_2 \mid M-M'$, we have $I(S_1)(x) = I(S_2)(x)$. This means that $x \in AR(M', I)$; also if we choose $S_1 \mid M-M' = S_2 \mid M-M'$, $S_1(x) \neq S_2(x)$, as is possible for $x \in M'-Z$, and if in addition $x \notin OR(I)$, we have $I(S_1)(x) = S_1(x)$, $I(S_2)(x) = S_2(x)$, so that $I(S_1)(x) \neq I(S_2)(x)$, and $x \in AR_2(M', I)$. Therefore $AR(M', I) \cup (M' - OR(I)-Z) \subseteq AR_2(M', I)$. This completes the proof.

As a corollary, we may determine that second affected regions have the same connective properties as ordinary affected regions. For instance, if $M' \subseteq M''$, then $AR_2(M', I) \subseteq AR_2(M'', I)$; also $AR_2(M' \cup M'', I) = AR_2(M', I) \cup AR_2(M'', I)$.

These facts may be applied to computer algorithms, i.e., programs. Let us consider a program which has no jumps or conditional jumps, so that it can be thought of as just a sequence of instructions. Such a program might be one to determine the value of x according to the quadratic formula. The coefficients A, B, and C, the solutions ROOT1 and ROOT2, the accumulator AC, and the temporary register TEMP are all subsets of the memory M of the computer. Now the composition of all the instructions in the sequence is itself a map $I: \mathcal{S} \to \mathcal{S}$ (which may or may not be an instruction on the computer), and as such has its own input, output, and affected regions. If the instruction I determines the two values ROOT1 and ROOT2 from the coefficients A, B, and C, then ROOT1 and ROOT2 should be affected by A, B, and C, and by nothing else. By the use of the formula for second affected regions, and Proposition X, we may derive a mechanical test for this condition. It will never absolutely guarantee that I

actually computes ROOT1 and ROOT2 correctly. It will, however, eliminate errors of the type which cause the resulting (incorrect) algorithm to have the wrong affected regions. In an incorrect algorithm, for example, ROOT1 might not be affected by A at all. This will happen, in particular, when one of the partial results has been overwritten--or, in programmers' slang, "clobbered"--at some stage of the algorithm.

A diagram of this test, for the special case of the quadratic formula, is given in Figure 1. (The procedure could also, of course, be programmed.) The various relevant subsets of M are listed at the left; the steps in the algorithm are listed at the top. Each subset at each stage is represented by a dot. Each dot is connected by lines to all the dots at the next stage representing the second affected region of the subset given by that dot, and, to only those dots. A dot in the left-hand row can affect a dot in the right-hand row, only if there is a connected forward path from one to the other. Note that in Figure 1 there is, in fact, a connected forward path from each of A, B, C to each of ROOT1, ROOT2. Figure 2 shows what happens when two of the instructions (in this case I4 and I5) are interchanged. In this case there is no connected path from C to ROOT1 or ROOT2, and we know immediately that the resulting algorithm must be incorrect. The algorithm proceeds as follows:

		$IR(I_i)$	$OR(I_i)$
I_1:	Bring the value of A to AC. (Speaking in formal terms, $I_1(S) \mid AC = S \mid A$.)	A	AC
I_2:	Multiply the value of C by the contents of AC.	C∪AC	AC
I_3:	Multiply the value of AC by 4.	AC	AC
I_4:	Store AC in the temporary cell TEMP.	AC	TEMP
I_5:	Bring the value of B to AC.	B	AC

		IR(I_i)	OP(I_i)
I_6:	Multiply the value of B by the contents of AC.	B∪AC	AC
I_7:	Subtract the value in TEMP from the contents of AC.	TEMP ∪ AC	AC
I_8:	Take the square roots of the value in AC and transmit them to AC and TEMP.		
I_9:	Subtract the value of B from the contents of AC.	B∪AC	AC
I_{10}:	Divide the value in AC by the value in A.	A∪AC	AC
I_{11}:	Divide the value in AC by 2.	AC	AC
I_{12}:	Store the value of AC as ROOT1	AC	ROOT1
I_{13}:	Bring the value of the cell TEMP to the AC.	TEMP	AC
I_{14}:	Subtract the value of B from the contents of AC.	TEMP ∪ AC	AC
I_{15}:	Divide the value in AC by the value in A.	A∪AC	AC
I_{16}:	Divide the value in AC by 2.	AC	AC
I_{17}:	Store the value of AC as ROOT2	AC	ROOT2

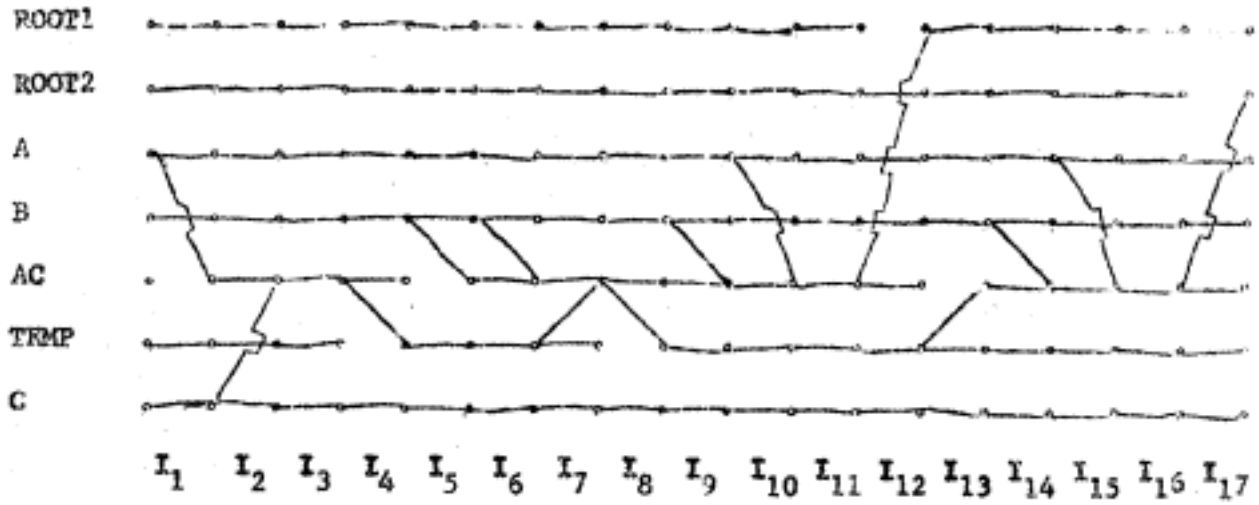

FIGURE 1

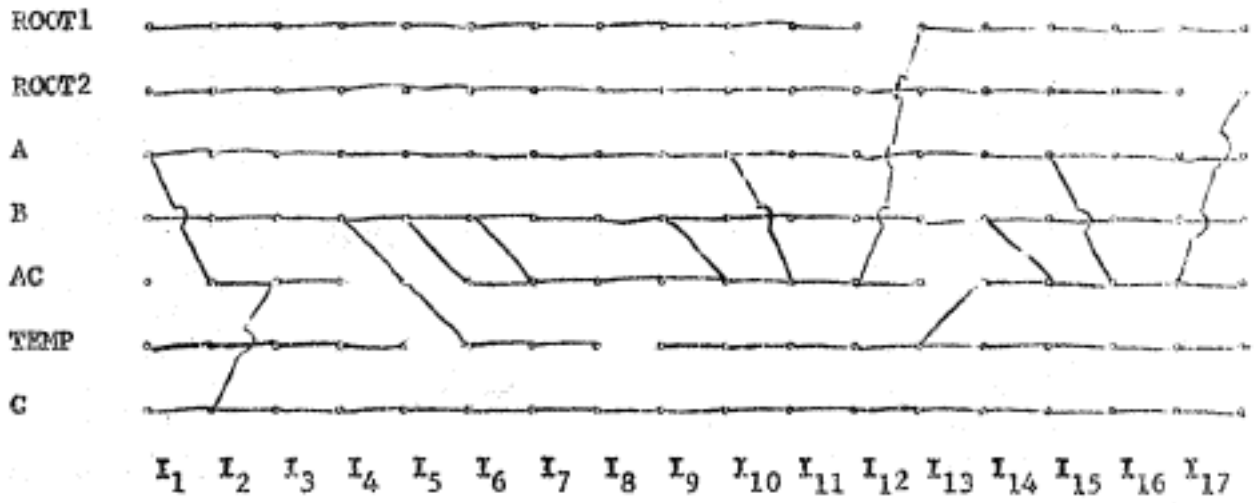

FIGURE 2

We may also verify that AR(M', I) is the intersection of all subsets $Q \subseteq OR(I)$ which possess the property AP(M', I): if $S_1 \mid IR(I)\text{-}M' = S_2 \mid IR(I)\text{-}M'$, then $I(S_1) \mid OR(I)\text{-}Q = I(S_2) \mid OR(I)\text{-}Q$. The intersection of two sets possessing AP(M', I) possesses AP(M', I) by virtue of property (P1), and the intersection of an arbitrary number of sets possessing AP(M', I) possesses AP(M', I) by virtue of property (P2). Thus a set possesses AP(M', I) if andonly if it contains AR(M', I). The entire subject of input, output, and affected regions can, in fact, be introduced by means of properties such as these, rather than directly as in Definitions B and K.

9. Transmission

The instructions defined below move data from one part of a computer memory to another in an unchanged fashion.

DEFINITION N. Let $(M, B, \mathscr{L}, \mathscr{L})$ be a computer, and let $G: M \to M$ be any map. The map $I: \mathscr{L} \to \mathscr{L}$, defined by $I(S)(x) = S(G(x))$, is the transmission instruction induced by G.

The following types of instructions are transmission instructions, induced by various maps G: move; load; store; block transfer; circular shift; sign extending shift; swap or exchange. A move instruction involving two single elements $x_1, x_2 \in M$, which moves the data in x_1 to x_2, is induced by a map G such that $G(x_2) = x_1$, $G(z) = z$ for $z \neq x_2$. A move instruction involving all the bits of two d-bit words, $x_1, \ldots, x_d$ and $y_1, \ldots, y_d$, which moves the data in the x_i to the y_i, is induced by a map G such that $G(Y_i) = x_i$, $1 \leq i \leq d$, and $G(z) = z$ for $z \neq y_i$. The same is true of a block transfer. "Load" and "store" are synonyms for "move," with a register involved as the output region for a "load," or as the input region for a "store." In all these cases $IR(I) \cap OR(I) = \emptyset$ (except possibly for the block transfer). A circular shift of a d-bit register $x_0, \ldots, x_{d-1}$, by k bits to the left, $k \leq d$, is induced by a map G such that $G(x_i) = x_j$, where $j = i+k \pmod d$. A sign extending shift of this register by k bits to the right is induced by a map G such that $G(x_i) = x_j$, where $j = \max(0, i-k)$. An exchange of two elements x_1 and x_2 is induced by a map G which permutes x_1 and x_2; the same idea may be extended to the case of two registers. In these last three cases $IR(I) = OR(I)$.

If the transmission instruction I is induced by the map G, then $IR(I) = \{G(x) \in M: G(x) \neq x\}$; $OR(I) = \{x \in M: G(x) \neq x\}$; and $AR_2(M', I) = G^{-1}(M')$. This is a further example of the fact that second affected regions are often easier to calculate than ordinary affected regions.

10. Input-Output Devices and Machine Theory

Although the set M in a computer is referred to as the "memory," it is well to remember that, strictly speaking, M is just an abstract set, which may not be a computer memory. In fact, we will construct, in the next chapter, computers with "memories" that have nothing to do with hardware. This point is further underscored by the fact that, when we are discussing input and output devices, the most effective model seems to involve including the entire (infinite) input and/or output sequence as part of the set M. The computer, of course, cannot make use of such an infinite "memory", other than to move it forward by one or more positions.

A <u>linear input device</u> can be constructed from a subset $M' \subseteq M$ which corresponds to the positive integers. An <u>input instruction</u> is an instruction I, with $I(S)(x) = S(x+k)$, for $x \in M'$, $k > 0$. Ordinarily, the set $\{1, \ldots, k\} \in M'$ will affect other subsets of M, but no other subset of M' will affect $M-M'$. A <u>linear output device</u> can also be constructed from M' as above; an <u>output instruction</u> is an instruction I with $I(S)(x) = S(x-k)$, for $x \in M' - \{1, \ldots, k\}$, in which $M-M'$ may affect $\{1, \ldots, k\}$, but M' does not affect $M-M'$. If the subset M' has both input and output instructions, it may be called an <u>infinite push-down store</u>. A <u>linear input-output device</u> can be constructed from a subset $M' \subseteq M$ which corresponds to the integers. On such a device, we might have the following types of instructions:

Forward read -- $I(S)(x) = S(x-k)$; $\{0, \ldots, k-1\}$ affects $M-M'$,

Forward write -- $I(S)(x) = S(x-k)$, except for $0 \leq x \leq k-1$;

$M\text{-}M'$ affects $\{0, \ldots, k-1\}$; M' does not affect $M\text{-}M'$.

Forward position -- $I(S)(x) = S(x-k)$; M' does not affect $M\text{-}M'$.

Backward read -- $I(S)(x) = S(x+k)$; $\{0, \ldots, k-1\}$ affects $M\text{-}M'$.

Backward write -- $I(S)(x) = S(x+k)$, except for $0 \leq x \leq k-1$;

$M\text{-}M'$ affects $\{0, \ldots, k-1\}$; M' does not affect $M\text{-}M'$.

Backward position -- $I(S)(x) = S(x-k)$; M' does not affect $M\text{-}M'$.

It may be left as an exercise to construct a computer which performs the functions of a "machine" or "automaton" as variously defined, e.g.: sequential machine; generalized sequential machine; Turing machine; Rabin-Scott two-tape automaton; push-down automaton; Fleck automaton.

The theory of sequential machines is, by now, quite extensive; at the same time, it is common knowledge that the theory of sequential machines is of little help to the computer programmer. The reason is not, as some people would believe, that the theory is not general enough; the reason is that the theory is too general. In a sequential machine the states form an arbitrary set. But in a computer with 32,768 thirty-six-bit words, there are $2^{1,179,648}$ states of the core memory alone. It is perfectly true that a computer can be thought of, in certain contexts, as a sequential machine. There are functions that determine what outputs are to be produced, and what new states entered, on the application of an input. The enumeration of such functions would be, not only completely impractical, but pointless, because what actually happens in a computer, depends on what instruction is being interpreted at the moment, and what its input, and output, and affected regions are.

11. Programs Written in Languages

Does the theory of computers, as developed here, apply to other objects than real computers? Are there objects which are computers, in the sense here defined, which have instructions with easily manageable input and output regions, but which do not involve "hardware"? There is at least one object of this nature which seems to be important--a program written in an algorithmic language such as ALGOL.

Let us consider such a program, and let V denote the set of all distinct variables in the program. (We are ignoring, for the moment, the fact that distinct variables may have the same name--if, for example, they are local to two different sub-blocks. Such variables are considered here to be distinct. We are also assuming that no block in the program calls itself, and in general that there is no need for recursive procedures of any kind--a restriction which will be removed later.) If the program contains arrays, V contains one element for each member of the array, according to its dimension; if the dimension is not specified, it is assumed, for the moment, to be infinite. Let R, Z, and C denote the reals, integers, and complex numbers, respectively, and let P denote the set of all statements of the program. We now construct a computer $(M, \mathcal{A}, \mathcal{L})$, as in Definition F, as follows: $M = V \cup \{L\}$, where L is a single object called the location; for each $x \in V$, $B_x = R$, Z or C, according as x is declared to be real, integer, or complex respectively, while $B_L = P$. Each statement of the program is now an instruction. For example:

(a) An arithmetic statement, such as A = X+Y-Z, is an instruction with output region $\{A, L\}$, defined by I(S)(A) = S(X) + S(Y)-S(Z), while I(S)(L) is the statement immediately following this one.

(b) A conditional statement, such as IF (X+Y-Z=0), is an instruction with output region $\{L\}$. The condition is evaluated on the state S; in this case, it is determined whether S(X) + S(Y) - S(Z) is actually equal to 0. If it is not, I(S)(L) is the next statement; if it is, I(S)(L) is the statement following the next statement.

(c) A go to statement is an instruction with output region $\{L\}$. The value of I(S)(L) is the (labelled) statement referred to.

If some of the blocks in the program do call themselves, or each other, so that certain of the variables need to be kept at several recursive levels, then each of these variables is replaced, in the set M, by an infinite push-down store as defined in the preceding section. (Or, for the sake of simplicity, every variable of the program, whether it needs to not, may be represented in M by such a store.) If other types of variables are allowed, such a matrices, then B_x may be altered to be a set of matrices or other objects. If switch variables are allowed, then $B_x = P$ for a switch variable. For each subroutine in the program with n parameters, we introduce into M a set of n elements $p_1, \ldots, p_n$, with $B_{p_i} = M$. When the subroutine is called, the elements p_i are filled with their corresponding values; i.e., the statement CALL XYZ(P1, P2, ..., PN) is an instruction with $I(S)(p_1) = P_1$, etc. When the parameter P1 is referred to in the subprogram, its value is S(S(P1)).

In this situation, there is one "universal" instruction, namely the instruction which executes whichever statement is contained in L. If S(L) is to be thought of as an instruction, this universal instruction is given by $I(S)(x) = S(L)(S)(x)$. This situation also holds in a real digital computer; the universal instruction is the instruction obtained by depressing the "step" key.

It is clear that the entire mechanism of input, output, and affected regions is just as applicable to these computers as to real computers.

12. Existence of Instructions

Can the input, output, affected regions of an instruction be taken arbitrarily? For computers with finite memory, the answer is yes, subject to the restrictions we have already mentioned. No input or output region can contain an element $x \in M$ such that B_x has exactly one element. The input region of an instruction can be void, but the output region cannot, unless the instruction is the identity. The affected regions of subsets of IR(I) are determined by the affected regions of one-element subsets of IR(I). Other than this, however, there are no restrictions for finite memory and no restrictions on input and output regions for infinite memory. Affected regions in a computer with infinite memory must satisfy certain other conditions. We shall give necessary and sufficient conditions, in the case of countable memory, for a set of affected regions to correspond to the affected regions of an actual instruction.

PROPOSITION XI. Let $(M, \mathcal{B}, \mathcal{L}, \mathcal{I})$ be a computer and let P and Q be subsets of M. Then there exists a map $I: \mathcal{L} \to \mathcal{L}$ with IR(I) = P and OR(I) = Q if and only if:

(1) $P \cap Z = \emptyset$ and $Q \cap Z = \emptyset$, where Z is the unit component.

(2) $Q \neq \emptyset$, unless $P = \emptyset$.

Proof: If $z \in Z$, then we can never have $S(z) \neq I(S)(z)$, so $z \notin OR(I)$ for any possible instruction I. Also, if $S_1(y) = S_2(y)$ for $y \neq z$, then, since $S_1(z) = S_2(z)$, we have $S_1 = S_2$, so we can never have $I(S_1)(y) \neq I(S_2)(y)$; hence $z \notin IR(I)$. If $Q = \emptyset$, we have $S(x) = I(S)(x)$ for all $x \in M$, so that $I(S) = S$ and I is the identity (and $IR(I) = \emptyset$).

Now let P and Q be chosen according to the conditions (1) and (2). We assume $Q \neq \emptyset$, as otherwise the identity instruction satisfies the conclusion of the theorem. First we consider the case in which Q consists of one element y. Since $y \notin Z$, there exist two states S_1 and S_2 such that $S_1(y) \neq S_2(y)$. Let us define $I: \mathcal{S} \to \mathcal{S}$ as follows: $I(S)(z) = S(z)$, $z \neq y$; $I(S)(y) = S_2(y)$ if $S \mid P = S_1 \mid P$; $I(S)(y) = S_1(y)$ otherwise. By property (P1), I(S) is again in $\mathcal{S}$. It is clear that $OR(I) \subseteq \{y\}$, and since $I(S_1)(y) = S_2(y) \neq S_1(y)$, we have $y \in OR(I)$, so that $OR(I) = \{y\}$. If $S_1' \mid P = S_2' \mid P = S_1 \mid P$, then either $S_1' \mid P = S_2' \mid P = S_1 \mid P$, so that $I(S_1')(y) = I(S_2')(y) = S_1(y)$; therefore P satisfies the input property IP(I), and $IR(I) \subseteq P$. On the other hand, if $x \in P$, then since $x \notin Z$, there exists $S_3 \in \mathcal{S}$ with $S_3(x) \neq S_1(x)$, $S_3(z) = S_1(z)$ for $z \neq x$; then $I(S_1)(y) \neq I(S_3)(y)$, so that $x \in IR(I)$. Therefore $IR(I) = P$.

Now let Q be arbitrary and let $y \in Q$. Construct an instruction I as above, with $IR(I) = P$, $OR(I) = \{y\}$, and let I' be a constant instruction on $Q' = Q - \{y\}$: $I'(S)(x) = S_1(x)$, for $x \in Q'$, and $I'(S)(x) = S(x)$, for $x \notin Q'$. We have $IR(I') = \emptyset$, $OR(I') = Q'$. Because of our known facts about the input and output regions of a composition (Chapter 5), we obtain immediately that $IR(I'') = P$ and $OR(I'') = Q$ where $I''(S) = I'(I(S))$. This completes the proof of Proposition XI.

PROPOSITION XII. Let $(M, B, \mathcal{L}, \mathcal{P})$ be a computer, and let P and Q be subsets of M, satisfying the hypotheses of Proposition XI. For each $x \in P$, let Q_x be a non-empty subset of Q, and let $Q'' = \bigcup_{x \in P} Q_x$ be finite. Then there exists a map $I: \mathcal{S} \to \mathcal{S}$ such that $IR(I) = P$, $OR(I) = Q$, and

$AR(\{x\}, I) = Q_x$ for each $x \in P$.

Proof: Let $S_0 \in \mathcal{S}$, let $Q'' = Q - Q'$, and for each $x \in P \cup Q$ let $S_x \in \mathcal{S}$ be such that $S_x(x) = b_x \neq S_0(x)$, $S_x(z)$ for $z \neq x$. For each state $S \in \mathcal{S}$ and each $y \in Q$ let $n(S, y)$ be the cardinality of $\{x \in P: y \in Q_x \text{ and } S(x) \neq S_0(x)\}$. By property (P2), $n(S, y)$ is always finite. Let $I(S)$ be defined thus:

$I(S)(y) = S_0(y)$	$y \in Q''$
$I(S)(y) = S_0(y)$	$y \in Q'$ and $n(S, y)$ is odd
$I(S)(y) = b_y$	$y \in Q'$ and $n(S, y)$ is even
$I(S)(y) = S(y)$	$y \in M-Q'$

Since Q' is finite, this choice yields a member of $\mathcal{I}$. We prove six assertions: (1) $OR(I) \subseteq Q$; (2) $OR(I) \supseteq Q$; (3) $IR(I) \subseteq P$; (4) $IR(I) \supseteq P$; (5) $AR(\{x\}, I) \subseteq Q_x$; and (6) $AR(\{x\}, I) \supseteq Q_x$.

(1) is clear; if $y \in M-Q'$, then, by definition of I, $y \notin OR(I)$.

(2) Let $y \in Q$. If $y \notin Q''$, then $S_y(y) \neq I(S_y)(y)$. If $y \in Q'$, then $y \in Q_x$ for some x. We have $n(S_x, y) = 1$ because $\{x \in P: y \in Q_x \text{ and } S_x(x) \neq S_0(x)\} = \{x\}$. Therefore $I(S_x)(y) = S_0(y) \neq S_x(y)$. In either case $y \in OR(I)$, and $OR(I) \supseteq Q$.

(3) If $S_1 \mid P = S_2 \mid P$, then it is clear, by the definition of $n(S, y)$, that $n(S_1, y) = n(S_2, y)$ for each $y \in Q'$. Therefore $I(S_1) \mid Q' = I(S_2) \mid Q'$; also $I(S_1) \mid Q'' = S_0 \mid Q'' = I(S_2) \mid Q''$. Thus $I(S_1) \mid Q = I(S_2) \mid Q$, which means that P satisfies the input property IP(I), and $IR(I) \subseteq P$.

(4) Let $x \in P$ and let $y \in Q_x$. Clearly $n(S_x, y) = 1$, while $n(S_0, y) = 0$. Therefore $I(S_x)(y) \neq I(S_0)(y)$, and $x \in IR(I)$. Thus $IR(I) \supseteq P$.

(5) For convenience, let $N(S, y) = \{x \in P: y \notin Q_x \text{ and } S(x) \neq S_0(x)\}$, for any state $S \in \mathcal{S}$ and element $y \in Q$. Let $S_1, S_2 \in \mathcal{S}$ be such that $S_1 \mid M-\{x\} = S_2 \mid M-\{x\}$ and let $y = Q'-Q_x$. We claim that $N(S_1, y) = N(S_2, y)$. For if $x' \in N(S_1, y)$, then $y \in Q_{x'}$, so that $x' \neq x$ and $S_2(x') = S_1(x') \neq S_0(x')$, so that $x' \in N(S_2, y)$, and vice versa. Therefore, $I(S_1) \mid Q'-Q_x = I(S_2) \mid Q'-Q_x$. Since we always have $I(S_1) \mid (M-Q) \cup Q'' = I(S_2) \mid (M-Q) \cup Q''$, therefore $I(S_1) \mid M-Q_x = I(S_2) \mid M-C_x$. Therefore Q_x possesses the property $AP(\{x\}, I)$. Thus $AR(\{x\}, I) \supseteq Q_x$.

(6) Let $x \in P$ and $y \in Q_x$. As in (4) above, $I(S_x)(y) \neq I(S_0)(y)$. This shows that $y \in AR(\{x\}, I)$. Thus $AR(\{x\}, I) \subseteq Q_x$.

This completes the proof of Proposition XII.

As a consequence of this theorem, we obtain the fact that for a machine with finite memory M, affected regions of single points of IR(I) can be completely arbitrary subsets of OR(I), provided that IR(I) and OR(I) have themselves been chosen properly.

If the condition that $\bigcup_{x \in P} Q_x$ is finite be removed from Proposition XII, the conclusion does not hold. There are two distinct types of counter-examples:

(1) Let us call an element $x \in M$ an <u>inversion point</u> of an instruction $I \in \mathcal{I}$ if $x \in IR(I)$, $x \in OR(I)$, B_x has exactly two elements, and $RA(\{x\}, I) = \{x\}$. Then any instruction can have only a finite number of inversion points. This statement is a generalization of the answer to the following question: Can an instruction be constructed on a

binary machine such that each element of M affects itself and only itself? For an infinite set M, the answer is no, because for such an instruction we must have $S(x) \neq I(S)(x)$ for all $S \in \mathcal{S}$ and all $x \in M$, contradicting property (P2). In fact, $S(x) \neq I(S)(x)$, for all $S \in \mathcal{S}$, holds for any inversion point x (and hence there can only be a finite number of inversion points). To see this, let x be an inversion point, and let $S_1, S_2 \in \mathcal{S}$ be such that $S_1(x) = S_2(x)$. Then $I(S_1)(x) = I(S_2)(x)$, since otherwise there would be an element $z \neq x$ which affects x, contradicting the statement $RA(\{x\}, I) = \{x\}$. Thus $I(S)(x)$ is an element of B_x which is dependent only on $S(x)$, which is also an element of B_x. There are only four possibilities for such a function. Identifying B_x for the moment with $\{0, 1\}$, these possibilities are: $I(S)(x) \equiv 0$ or $I(S)(x) \equiv 1$, which is impossible, since then x is not affected by any element of M; $I(S)(x) \equiv S(x)$, which is impossible, since then $x \notin OR(I)$; and $I(S)(x) \equiv 1 - S(x)$, the only remaining possibility, which verifies the statement that $S(x) \neq I(S)(x)$ for all $S \in \mathcal{S}$.

(2) The region affecting an infinite subset $Q_0 \subseteq \bigcup_{x \in P} Q_x$ must either be itself infinite, or contain an element p for which B_p is infinite. To see this, let $P = RA(Q_0, I)$, and suppose that P is finite and B_p is finite for each $p \in P$. Then $\mathcal{S}|P$, the set of all states of P, is finite. Let $S_1|P, S_2|P, \ldots, S_n|P$ be the set of all distinct states of P, for some states $S_1, \ldots, S_n \in \mathcal{S}$. For each S_i let $N_i = \{x \in M: I(S_i)(x) \neq I(S_1)(x)\}$; then $N' = \bigcup_{i=1}^{n} N_i$ is finite,

and therefore $Q_0 - N'$ is non-empty. Let $y \in Q_0 - N'$; we show that, for any two states S_1', $S_2' \in \mathcal{S}$, we have $I(S_1')(y) = I(S_2')(y)$, i.e., $y \notin AR(IR(I), I)$, contrary to hypothesis. To this end, let us assume that $S_1' \mid P = S_i \mid P$, $S_2' \mid P = S_j \mid P$. Then, by definition of $R_1(Q_0, I)$, we have $I(S_1')(y) = I(S_i)(y)$ and $I(S_2')(y) = I(S_j)(y)$; since $y \notin N_i$ for each i, we have $I(S_i)(y) = I(S_1)(y) = I(S_j)(y)$. Therefore $I(S_1')(y) = I(S_2')(y)$, as claimed.

It is remarkable that the two necessary conditions (1) and (2) above, taken together, are sufficient as well, provided that the memory M is countable. In fact, we may prove the following.

PROPOSITION XIII. Let $(M, B, \mathcal{S}, \mathcal{O})$ be a computer, with M countable. Let P and Q be subsets of M, satisfying the hypotheses of Proposition XI. For each $x \in P$, let Q_x be a non-empty subset of Q. Furthermore, let the following two conditions be satisfied:

(1) There are only a finite number of inversion points $x \in M$, i.e., elements such that:

(a) $x \in Q_x$;

(b) $x \notin Q_z$, for $z \neq x$;

(c) B_x has exactly two elements.

(2) For each infinite subset $Q_0 \subseteq Q'$, the set $A(Q_0) = \{x \in P : Q_x \cap Q_0 \neq \emptyset\}$ is either itself infinite, or contains an element p such that B_p is infinite.

Then there exists a map $I: \mathcal{S} \to \mathcal{S}$ with $IR(I) = P$, $OR(I) = Q$, and $AR(\{x\}, I) = Q_x$ for each $x \in P$.

BIBLIOGRAPHY

(1) Ginsburg, S. An Introduction to Mathematical Machine Theory. Addison-Wesley, 1962.