A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously*

JOHN HOLLAND†

Introduction

THIS PAPER describes a universal computer capable of simultaneously executing an arbitrary number of sub-programs, the number of such sub-programs varying as a function of time under program control or as directed by input to the computer. Three features of the computer are:

(1) The structure of the computer is a 2-dimensional modular (or iterative) network so that, if it were constructed, efficient use could be made of the high element density and "template" techniques now being considered in research on microminiature elements.

(2) Sub-programs can be spatially organized and can act simultaneously, thus facilitating the simulation or direct control of "highly-parallel" systems with many points or parts interacting simultaneously (e.g. magneto-hydrodynamic problems or pattern recognition).

(3) The computer's structure and behavior can, with simple generalizations, be formulated in a way that provides a formal basis for theoretical study of automata with changing structure (cf. the relation between Turing machines and computable numbers).

The computer presented here is one example of a broad class of universal computers which might be called universal iterative circuits. This class can be rigorously characterized and formally studied (the characterization will be published in another paper). The present formulation is intended as an abstract prototype which, if current component research is successful, could lead to a practical computer.

General Description

The computer can be considered to be composed of modules arranged in a 2-dimensional rectangular grid; the computer is homogeneous (or iterative) in the sense that each of the modules can be represented by the same fixed logical network. The modules are synchronously timed and time for the computer can be considered as occurring in discrete steps, $t=0, 1, 2, \ldots$.

Basically each module consists of a binary storage register together with associated circuitry and some auxiliary registers (see Fig. 1). At each time-step a module may be either active or inactive. An active module, in effect, interprets the number in its storage register as an instruction and proceeds to execute it. There is no restriction (other than the size of the computer) on the number of active modules at any given time. Ordinarily if a module $M(i,j)$ at coordinates (i,j) is active at time-step t, then at time-step $t+1, M(i,j)$ returns to inactive status and its successor, one of the four neighbors $M(i+1,j)$, $M(i,j+1)$, $M(i-1,j)$, or $M(i,j-1)$, becomes active. (The exceptions to this rule occur when the instruction in the storage register of the active module specifies a different course of action as, for example, when the instruction is the equivalent of a transfer instruction).

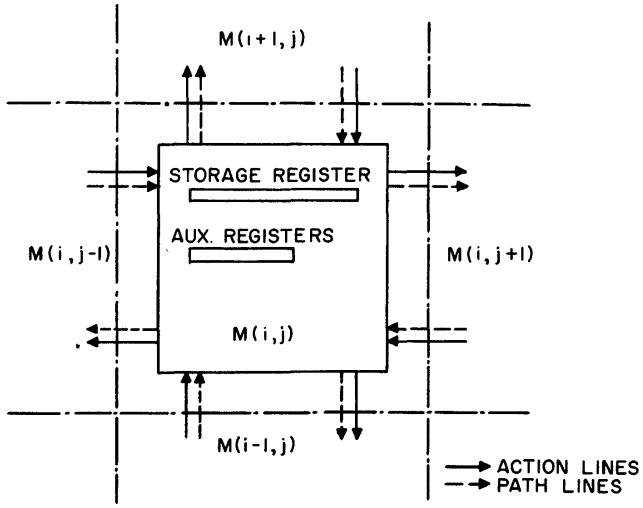

Fig. 1—A module.

The successor is specified by bits s_1, s_2 in $M(i,j)$'s storage register. If we define the line of successors of a given module as the module itself, its successor, the successor of the successor, etc., then a given sub-program in the computer will usually consist of the line of successors of some module. Since several modules can be active at the same time the computer can in fact execute several sub-programs at once. We have noted parenthetically that there are orders which control the course of action — there are also orders equivalent to store orders which can alter the number (and hence the instruction) in a storage

* Funds for this study were supplied to the Logic of Computers Group of the University of Michigan initially by the Air Force Office of Scientific Research under Contract No. AF18(603)-72 and later through a National Science Foundation grant (G4790).

† University of Michigan, Ann Arbor, Michigan.

register. Therefore the number of sub-programs being executed can be varied with time, and the variation can be controlled by one or more sub-programs.

The action of a module during each time-step can be divided into three successive phases:

(1) During phase one, the input phase, a module's storage register can be set to any number supplied by a source external to the computer.

(2) During phase two, an active module determines the location of the operand, the storage register upon which its instruction is to operate. This the module does by, in effect, opening a path (a sequence of gates) to the operand. Phase two is called the path-building phase.

(3) During phase three, the execution phase, the active module interprets and executes the operation coded in its storage register.

Path-Building

An active module determines the location of the storage register upon which its instruction is to operate by, in effect, opening a path to it. The path-building action depends upon two properties of modules:

First, by setting bit p in its storage register equal to 1, a module may be given special status which marks it as a point of origination for paths; the module is then called a P-module.

Secondly, each module has a neighbor, distinct from its successor, designated as its predecessor by bits q_1, q_2 in its storage register; the line of predecessors of a given module M_0 is then defined as the sequence of all modules $[M_0, M_1, \ldots, M_k, \ldots]$ such that, for each k, M_k is the predecessor of M_{k-1} and M_{k-1} is the successor of M_k (see Fig. 2). Note that the line of predecessors may in extreme cases by infinitely long or non-existent. The line of predecessors of an active module ordinarily serves to link it with a P-module (through a series of open gates). During the initial part of phase two the path specification bits $y_0, \ldots, y_n$ and d_1, d_2, in the storage register of an active module M_0, are gated down its line of predecessors to the nearest P-module (if any) along that line. The path specification bits are then used by the P-module to open a path to the operand (the storage register addressed by the active module).

Each path must originate at a P-module and only one path can originate at any given P-module. The path originating at a P-module is gated by means of a sequence of auxiliary registers called *-registers. Each module possesses 4 *-registers and if the module belongs to a path in direction (b_1, b_2) the appropriate *-register, (b_1,b_2)*, is turned on. When (b_1,b_2)* is on it gates lines (to be described) from the module $M(i,j)$ to its neighbor $M(i+b_1,j+b_2)$ permitting certain signals coming into $M(i,j)$ to be passed on to $M(i+b_1, j+b_2)$ and vice-versa. Since each *-register

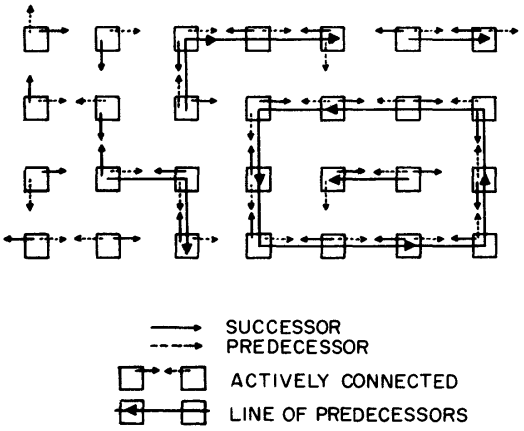

Fig. 2—Lines of predecessors.

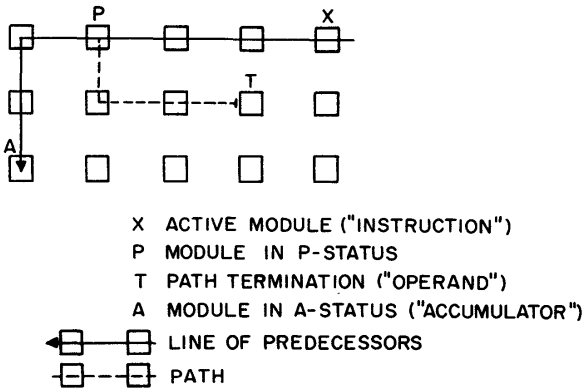

Fig. 3—Modules used in the execution of an instruction.

gates a separate set of lines, a module may (with certain exceptions) belong to as many as four paths. Once a *-register is turned on its stays on until turned off; thus a path, once marked, persists until erased.

The modules belonging to a given path can be separated into sub-sequences called segments. Each segment of the path is the result of the complete phase two action of a single active module. A segment consists of y modules extending parallel to one of the axes from some position (i,j) through positions $(i+b_1,j+b_2)$, $(i+2b_1,j+2b_2)$, $\ldots$, $(i+(y-1)b_1, j+(y-1)b_2)$, where $b_1 = \pm 1$ or 0 and $b_2 = \pm(1-|b_1|)$; the module at $(i+yb_1, j+yb_2)$ will be called the termination of the segment (note that the termination of the segment is *not* a member of the segment.) The segments are ordered so that the first segment constructed has as its initial module the P-module. The k^{th} segment constructed as part of the path has as its initial module the termination of the $k-1^{th}$ segment. If the path consists of n segments, the termination of the n^{th} segment (the last segment constructed) will be called the path termination (see Fig. 3).

As noted, the path specification bits $y_n, \ldots, y_0$ and d_1,d_2 are gated down the line of predecessors from the storage register of the active module to

the nearest P-module at the start of phase two. If $y_n = 0$, bits $y_{n-1}, \ldots, y_0$ and d_1,d_2 determine the length and direction, respectively, of the new segment. The total number of digits $y_{n-1}, \ldots, y_0$ equal to 1 gives the length of the segment — if j of the digits are equal to 1 then the segment will be j modules long. The digits d_1,d_2 turn on and set an auxiliary register, the direction register, in the initial module of the new segment. This gives the direction b_1,b_2 of the segment. The direction registers of the other modules belonging to the segment are all off, but each of the modules belonging to the segment (including the initial one) has its (b_1,b_2)* register turned on.

When $y_n = 1$, the final segment of the path originating at the P-module is erased. That is, the direction register in the initial module of the last segment is turned off and, as a consequence, all *-registers marking the last segment are turned off. If the path consists of a single segment or none at all the effect of $y_n = 1$ is to turn off the direction register in the P-module thereby making the P-module the termination of the path. That is, in this latter case, the path has no segments but it does have a termination — the P module itself (note that the *status* of the P-module is unchanged).

The following additional rules apply to paths:

(1) When a given module is the termination of several paths and direction register on-pulses arrive over more than one path at the same time, t, the result is no action — the direction register is not turned on and none of the paths is extended.

(2) Only one path can proceed through a module in which the direction register is on. Whenever the direction register of a given module M is turned on or given a new setting, any paths already running through that module will now have it as their termination. Furthermore, for each such path, the portion lying between M and the previous path termination is at once erased — the *-registers and direction registers marking that portion of the path are turned off.

(3) No P-module can belong to any part of a path other than its origin. If a path in the process of construction reaches a P-module then all construction ceases and the P-module becomes the termination of the path regardless of the value of digits $y_n,y_{n-1}, \ldots,y_0$. Further extension of the path will not be carried out unless the P-module's status is changed (its p bit set to zero).

Execution

Three modules play an important role during the execution phase of an active module: the active module itself holds the order code in bits i_1,i_2,i_3 of its storage register; the storage register of the nearest path termination contains the word to be operated on (the operand); finally there must be a module which serves as accumulator (see Fig. 3). In order to serve as an accumulator, the storage register of a module must first have bits (p,a) in it set to the value (0,1), giving the module special status — A-module status. (Note that this means a module in P-module status, $p = 1$, cannot be an A-module). If $M(i,j)$ is an active module then the first A-module along its line of predecessors serves as the accumulator. An A-module serves, in effect, to terminate a line of predecessors, since it can have no designated predecessor.

In the present formulation there are eight basic orders:

(1) The arithmetic operation ADD. Execution of ADD causes the number in the storage register at the nearest path termination (the operand) to be transferred to the nearest A-module and there added to whatever number is in the storage register of the A-module. (By using complements and iteration all the arithmetic operations, such as subtraction and multiplication, can be accomplished by means of this operation).

(2) The storage operation STORE. Execution of STORE causes the number in the storage register of the nearest A-module to be transferred to the storage register at the operand.

(3) The transfer operation TRANSFER ON MINUS. Execution of TRANSFER ON MINUS depends upon the number in the storage register in the nearest A-module. If $y_n = 0$ in this number then the active module, after completing phase two, becomes inactive and its successor becomes active. If $y_n = 1$ then the module at the nearest path termination, rather than the successor, becomes active.

(4) The index operation ITERATE SEGMENT. If $y_n = 0$ in the nearest A-module, execution of ITERATE SEGMENT (upon completion of phase two) reduces the number in the A-module by 1 and the active module remains active *without* causing its successor to become active. If $y_n = 1$, then execution of the order simply causes the successor to become active and the active module inactive at the completion of phase one. This operation provides a convenient means of building long paths in a given direction since, if N is the number in the nearest A-module, the path-building phase of the active module is iterated N times.

(5) SET REGISTERS causes the first 9 bits of the number in the nearest A-module to be used to set all 9 auxiliary registers at the nearest path termination, the j^{th} register being set on if the j^{th} bit is a one. It is important that the SET REGISTERS order can give the operand module active status by setting the appropriate auxiliary register. In this case the active module gives rise to two active modules on the next time-step, its successor and the operand module. By this means one sub-program can initiate activity in another.

(6) RECORD REGISTERS causes the state of the 9 auxiliary registers at the nearest path termination to be recorded in the first 9 bits of the nearest A-module (in the same order as used by the SET REGISTERS instruction).

(7) NO ORDER causes the execution phase to pass without the execution of an order.

(8) STOP causes the active module to become inactive without passing on the activity to its successor at the next time-step.

With the exception of the STOP, ITERATE SEGMENT, and TRANSFER orders, the active module becomes inactive and its successor becomes active at the conclusion of the execution of an order.

It is possible for a given active module to have *no* nearest P-module (or A-module) for any one of three reasons: (1) the module does not have a line of predecessors, (2) none of the modules along the line of predecessors is currently designated a P-module (or A-module), (3) there is no P-module along the line of predecessors *between* the active module and the nearest A-module. If there is no nearest P-module then there is neither path-building nor execution of instruction with respect to the active module (regardless of the content of its storage register). If there is no nearest A-module along the line of predecessors then the instruction of the active module is not executed although the path-building phase will be carried out (assuming a nearest P-module).

The following additional rules apply to active modules and their action with respect to P-modules and A-modules:

(1) If M_0 belongs to the line of predecessors of M_1, if the nearest P-module of M_0 is also the nearest P-module of M_1, and if M_0 and M_1 are both active, then the action of M_0 proceeds normally but M_1's action is as if it had *no* nearest P-module.

(2) If M_0 and M_1 are situated as in rule (1) except that they have the same nearest A-module, without sharing the same P-module, then the action of M_0 proceeds normally but M_1 acts as if it were executing a NO ORDER instruction.

(3) As mentioned earlier, a module can be given A-module status by setting the pair of bits (p,a) to the value (0,1). This turns on an auxiliary register in the module, the A-register. At the same time the bits of another auxiliary register pair, the $(D_1,D_2$-register, are set to match the bits s_1,s_2 in the module's storage register; i.e., when the A-register is on the (D_1,D_2)-register indicates the successor of the A-module.

Once a module is given A-module status it can be returned to normal status only in one of two restricted ways. The first way requires that a STORE order be executed by an active module which has the given A-module as its operand module (nearest path termination). Then, if bit a is 0 or bit p is 1 in the number being stored, the A-module reverts to normal status and the word in its storage register is that specified by the STORE instruction. Otherwise the A-module is unchanged, the STORE order not being executed. The other way of returning an A-module to normal status requires that the A-module receive external input during phase one. The above restrictions prevent the A-module from changing status when numbers are placed in its storage register during the normal course of its operation as an accumulator. During the time a module is an A-module the bits in its storage register are not interpreted in any way except as the digits of a binary number.

(4) A module in A-module status can become part of a path (or several paths) so long as it is not to be the initial module of a path segment. In this latter case the path-building action, which would make the A-module the initial module of a segment, is not carried out — the A-module remaining the termination of the path.

(5) A given module can be acted upon simultaneously by 2, 3, or even 4 STORE instructions if it is the termination of more than one path. Some provision must then be made to resolve conflicts when the numbers being stored are not identical. In the present formulation the conflict is resolved digit by digit: a 1 is stored at bit j in the storage register if and only if at least one of the incoming numbers has a 1 at position j.

(6) When a STORE instruction changes the word in the storage register of a module it is assumed that this change does not take place until the completion of phase three. Thus, for example, there is no conflict when the STORE instruction of an active module acts upon that module's own line of predecessors or, for that matter, upon the module itself.

(7) If an active module has an A- or P-module as successor then, at the next time-step, *the successor of the A- or P-module* becomes active, rather than the A- or P-module itself (unless, of course, the instruction just executed specifies otherwise).

INPUT

During phase one, the initial phase of each time-step, a module's storage register can be set to any arbitrarily chosen value and its auxiliary registers to any desired condition. The numbers and conditions thus supplied are the computer's input. Although the number in the storage register can be arbitrarily changed at the beginning of each time-step, it need not be; for many purposes the majority of modules will receive input only during the first few moments of time ("storing the program") or only at selected times $t_1,t_2,\ldots$ ("data input"). Of course, some modules may have a new number for input at each time-step; in this case the modules play a role similar to the inputs to a sequential circuit.

SUMMARY OF ORGANIZATION AND SYMBOLS

As noted in the general description of section 2,

each module consists of a storage register plus some auxiliary registers. The earlier discussions indicate that the auxiliary registers required are:

(1) the E-register, a one-bit register which is on if and only if the given module is active;

(2) the A-register, a one-bit register which is on if and only if the given module is an A-module;

(3) the D-register, a one-bit register which is on if and only if the given module is the initial module of a path segment;

(4) the (D_1,D_2)-register, a register, with two bits of storage, which indicates the direction (b_1,b_2) of a segment if and only if the D-register is on and which indicates the direction of the module's successor if and only if the A-register is on.

(5) the (b_1,b_2)*-registers; each is a one-bit register which is on if and only if the given module is a member of a path segment with direction (b_1,b_2).

For formal purposes we can symbolize the state of a given register, X, at coordinates (i,j) and time t by the predicate $X(i,j,t) = 1$ if the given register is on at time t.

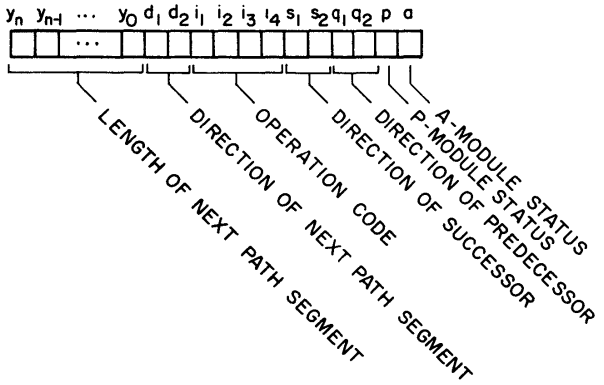

Fig. 4—Control of module by storage register.

The storage register of each module in the present formulation consists of $n+12$ bits (see Fig. 4) labelled in the following order:

bit number:
$n+12\ n+11 \ldots 12\ 11\ 10\ 9\ 8\ 7\ 6\ 5\ 4\ 3\ 2\ 1$
label: $y_n\ y_{n-1} \ldots y_0\ d_1\ d_2\ i_1\ i_2\ i_3\ s_1\ s_2\ q_1\ q_2\ p\ a$

The bits s_1,s_2 and q_1,q_2 designate the successor and predecessor, respectively, of the module. If bit p is 1 the module has P-module status. If the pair of bits (p,a) are set to the value (0,1) as the result of input or a STORE operation, the module has A-module status. During the path-building phase bits $y_n, \ldots, y_0$ and d_1,d_2 in an active module are interpreted as segment length and direction respectively. During the execution phase bits i_1,i_2,i_3 in an active module are interpreted as the operation to be performed. The word in the storage register of an A-module is treated strictly as a binary number with y_n being the sign bit and the other $n + 11$ bits being arranged as indicated with y_{n-1} being the high order bit and a being the low order bit.

Comment

A universal machine in which the programs have a spatial organization has several properties over and above those usually associated with Turing machines and their concrete counterparts. For example, cycles in the program can actually be stored as cycles (of successors) in the rectangular grid (see Fig. 5). This, in effect, provides *each* cyclic sub-program with an instruction address counter which counts modulo the number of instructions in the sub-program (cf. an index register which can be set to cycle modulo any base number). Furthermore, each sub-program can be allotted a certain area in the grid and this allows the spatial arrangement of the sub-programs to match, for example, the structural organization of a process which is being simulated — each subprogram in this case directly simulating one of the components of the process.

Efficient programming of certain types of problem will require techniques similar to those required for asynchronous operation. That is, when several sub-programs are operating simultaneously, each sub-program will from time to time require results from other sub-programs, however these results will not in general be available at just the time desired. In problems like this, usually arising in the control or simulation of "highly-parallel" systems with many points or parts interacting simultaneously, the programmer will employ many of the techniques of the logical designer.

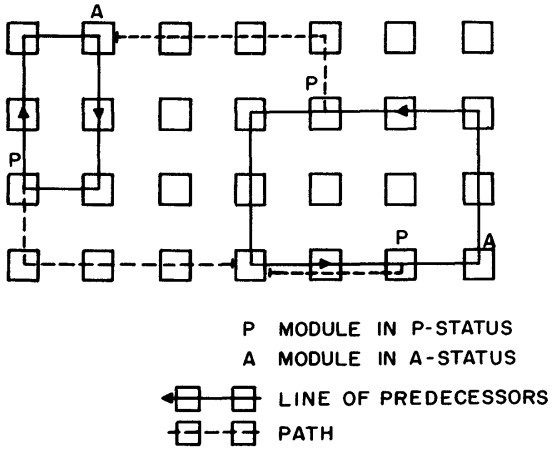

Fig. 5—Two interacting subroutines.

Problems such as the one just discussed emphasize the desirability of a computer formulation amenable to theoretical investigation. The present formulation is one example of a broad class of computers which can be rigorously characterized and investigated by abstract deductive techniques. Actually, the defini-

tion of this class of computers comes as part of an effort to provide a formal basis for the study of growing automata. By considering the rectangular grid to be infinite (or potentially infinite) in each of its dimensions (in analogy to the infinite tape of a Turing machine) many problems of automata theory can be expressed in a formal framework similar to that provided by Turing machines for problems of computability. Thus, for example, models of various processes can be stated as programs, or classes of programs, for the machine and investigated both directly and theoretically.

There are several variants of the formulation given here which yield computers which are either more flexible or have simpler modules. As a single instance, the path-building procedure could be altered to make branching paths possible; in this way the same sub-program could operate on several storage registers simultaneously.

A final word about concrete realization of such a computer: a partial rendering of the logical diagrams for a module in the described computer indicates that a module with a 40-bit storage register could be constructed with approximately 1000 basic elements. If this is actually the case and if switching is accomplished with micromodular densities, say 10^8 elements per cubic foot, then the basic portion of a computer with 100,000 modules should be realizable within a volume of a few cubic feet (exclusive of input-output equipment, power supply, etc.).

Discussion

M. Rubinoff: Would you expect this two-dimensional mondular structure to have important significance for character recognition and other aspects of "gestalt" or area vision?

Dr. Holland: I would hope so. This was part of the original intent in constructing the computer. The hope was that it could operate on all parts of the pattern at the same time. The computer is really a by-product of research on what we call growing automata.

P. Rosenblatt (Teleregister): Since any module may be in *P* status at some time, will not this "anywhere-to-anywhere" transmission result in extremely large cost of gating when the number of modules is large?

Dr. Holland: I think the point here is that any module may be in *P* status; however, gating is from module to module, and you open these gates in serial fashion as specified by the number in the storage register of the active module. Since each module will have a fixed number of gates controlling the path lines to its four neighbors, the number of such gates is directly proportional to the number of modules. Note that only one path can proceed from any one *P* module at any given time.

R. A. Kirsch (Nat'l Bureau of Standards): Why did you constrain the machine to the two-dimensional rather than the multi-dimensional?

Dr. Holland: This was for purposes of exposition. As I say, this is one example of a large class of machines and you can have any number of dimensions in general. If you get beyond three dimensions you are operating entirely in theory.

G. Richmond (Cornell Aero Lab.): Can paths be destroyed as well as created?

Dr. Holland: Yes, I didn't mention this in the talk but if you set the first bit in the storage register (labeled *Yn*) to one, in effect this says that, when the path-building phase of that module is carried out, the last segment of the path is to be erased. In other words, you pay no attention to the rest of the bits *Yn–i* through *yo*; you simply erase the last segment of the path.

L. R. Bowyer (Bell Labs.): Is only one module active at a time? How does a path get formed through non-active modules?

Dr. Holland: No, any number of modules may be active at the same time. In fact, by means of interlocking conditions you can allow modules along the same line of successors to be active. You recall there was a square red line of successors in the last slide; it is even possible for more than one module to be active in that red square. The number of interlocking and override conditions is surprisingly small. There are only 12 of them. The activity of a module has no direct effect upon paths passing through it. If a module is active it makes no difference as far as these paths are concerned. The path lines are separate sets of lines that pass through modules. The opening of gates in the path lines are caused by active modules (via a *P*-module) but the path doesn't have to pass through an active module.

G. J. Moss (Naval Ordnance Lab.): Could you elaborate on the process of reading data into the computer?

Dr. Holland: This is a part I haven't paid a great deal of attention to. I would say this, that in general one would like to have some device that could put data into any storage register of this modular arrangement at any time. This seems a little difficult from a practical viewpoint so it seems you would have to settle for some sort of scanning technique to put data in at particular times. In order to make effective use of this computer in some communication system or pattern recognition device you would certainly put data in at many modules at many times; perhaps at each time-step during the input phase. All data has to go in during the input phase.

W. Buchholz (IBM): What happens if one module tries to create a path that would have to cross an already existing path?

Dr. Holland: As long as it does this at a place which is different from the termination of a segment this is all right. As I mentioned during the talk, each *segment* of the path proceeds in a given direction (parallel to one of the axes). Modules can have as many as four paths running through them one to each of the four neighbors. These lines are independent except at points where the paths turn. In other words, at the termination of a segment where you add on a new piece you may have a right angle turn. At this point you need a crossover matrix from one path line to another. At this point an interlock would prevent a second path going through. If you tried to build a second path at this point nothing would happen.

T. Gilmer (ITT Federal): What is the function of the auxiliary storage? How do you deactivate a net once started?

Dr. Holland: I didn't go through this but I can give you an example. I mentioned that if you have the *a* bit in the storage register equal to one then the module is in *A*-module status. This means that during an ADD operation you might change the content of the storage register in the *A*-module and change the *a* bit to a *O*. Then the module would lose its *A*-module status. This you don't want. So one of the auxiliary registers keeps the module in an *A* module status so that changes in the contents of its storage register make no difference. Other auxiliary registers perform functions that are similar.

L. Clapp (Sylvania): Would Mr. Holland care to comment on debugging, diagnostic and preventive maintenance techniques of this computer?

Dr. Holland: The programming here would involve two kinds of techniques. It would take quite a bit of the best of programming techniques together with techniques of logical design. You may have circumstances very similar to asynchronous operation, if the second program has to wait for the first program to produce the data. A programmer of this machine has to be a good programmer but also has to be a good designer, too. In some cases, e.g. the solution of two-dimensional differential equations, since you repeat the same program over and over across the module, the debugging might be simpler than in a single-sequence computer because you have no logical effort to put into a scanning program — the program structure would match the mesh structure.