T H E U N I V E R S I T Y O F M I C H I G A N

COLLEGE OF ENGINEERING
Department of Electrical Engineering
Information Systems Laboratory

Technical Note

PHYSICAL AND LOGICAL DESIGN OF A HIGHLY PARALLEL COMPUTER

Jon S. Squire
Sandra M. Palais

ORA Project 04794

under contract with:

UNITED STATES AIR FORCE
AERONAUTICAL SYSTEMS DIVISION
CONTRACT NO. AF 33(657)-7391
WRIGHT-PATTERSON AIR FORCE BASE, OHIO

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

October 1962

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

An organization for a parallel processing computer is proposed, and its capabilities and limitations are discussed. Simultaneous random access is accomplished by logical circuitry which does step-by-step connection of paths from operands to arithmetic units. Many priority problems which arise from parallel processing are eliminated by a logical structure with processing units as the vertices of an n-dimensional cube and interconnections between processing units along the edges of the n-cube. The remaining priority problems are solvable with conventional logic.

1. INTRODUCTION

There are a number of features each programmer would like to have in a large-scale digital computer. The desires are as numerous as the programmers and often serve opposite purposes. Thus, one of the foremost problems in developing an improved computer organization is to recognize the fundamental requirements of the programmer. It can be safely said that, in general, programmers desire computers to be large, fast, versatile, and easy to program.

We are presenting a machine organization for a general-purpose computer that could be superior to existing computers for some problems and would extend the range of problems solvable on computers.

2. OBJECTIVES

Our primary objective will be to increase the amount of computation that can be done in a given time by means of a new organization rather than faster components. To this end, our organization provides for simultaneous execution of many instructions, each by a separate processing unit. The machine's ability to do parallel processing leads to the desire for providing unlimited interaction among processing units, i.e., each processor not only has circuitry that interprets an instruction and causes control action, but also is able to communicate with every other processor and to operate on any data. In a parallel computer these abilities are needed for the efficient running of large programs and for programs well suited to parallel computation. But the

organization must also provide for a partitioning of the machine so that a number of small programs could run simultaneously without interaction. This is usually referred to as "interprogram protection" and would need to be under program control to be completely versatile. Since processing units can interact directly with each other there is no need for a central control. In fact, any distinguished or superior processing unit would unreasonably complicate programming.

The organization should allow the size of the machine to be flexible, a variety of I-O devices to be provided, and a powerful set of operations to be available for the benefit of the programmer. In particular, there should be complete flexibility with respect to the number of instructions and number of data; the only restriction being that their sum does not exceed the storage capacity of the machine. Further, it would be convenient to the programmer and conservative of storage if one instruction could cause an operation to be performed at a number of locations simultaneously. For example, a single instruction could cause one number to be added to the contents of many memory locations.

In addition, the hardware should be able to accept an arbitrary number of instructions for execution at any given program step. If all instructions can not be processed simultaneously, then the computer should process them in groups. When all instructions for a program step have been completed, the next program step should begin executing all instructions designated as successors by the instructions of the immediately proceeding program step.

And finally, in the light of previous requirements, it would be unrea-

sonable to cause any computation bottleneck due to a shortage of arithmetic units or delay while accessing data. Therefore, every memory location should be directly accessible by an arithmetic unit. The computer, being equally limited by computation and memory access, could employ lookahead efficiently, thus enabling the greatest overall computation speed.

These objectives form a basis from which a very powerful computer organization can be developed. We realize, of course, that there are other objectives which might be added or substituted to meet other criteria. Our choice of objectives is based on the fact that a machine organization fulfilling these objectives could substantially reduce average computation time for some problems as compared to a computer with a single processor constructed from similar components.

As might be expected, any computer fulfilling these objectives would require many times the number of components in existing computers. Potentially inexpensive components and useful construction techniques are presented in the last section of this report. A brief look at cost versus problem-solution time indicates such a machine would be uneconomical in the next year or so. Yet, technological improvements that reduce the cost of logical components without necessarily increasing their speed, plus reasonable development of parallel programming techniques, could make such a machine economically competitive in the near future.

3. ORGANIZATION

The following computer organization meets the objectives outlined above

and has some novel features for logical design and physical construction.

The computer consists of many identical modules, blocks of logical circuits, imbedded in a passive connecting network. Each module contains a basic arithmetic unit, storage for one word of data or one instruction, and some control circuitry. There is a central timing and synchronization but no other common memory or control units. This is sufficient to form a complete general-purpose computer minus input-output equipment.

Our main consideration is the description of a module, of module connection, and of program execution within this computer. Input-output devices are to be connected directly to modules, with, at most, one per module. In this way, arbitrarily many I-O devices can be operating simultaneously without slowing down computation in other modules.

Since we are considering a highly parallel computer we wish to allow arbitrarily many instructions to be executed simultaneously. Rather than having instruction counters hold the locations of the next instructions to be executed, an additional bit position, the execution bit, is appended to each memory location. At the time when execution is to begin, the contents of each memory location having an execution bit equal to 1 is executed as an instruction. A 0 then replaces the 1 in the execution bit of those locations from which instructions were just executed. Thus an instruction specifies its successors, if any, by setting the execution bit to 1 in the memory locations of the instructions to execute next. To avoid the priority problems of assigning instructions to be executed to processing units, each memory location has an instruction processor directly connected to it.

The instruction processor operates, or is active, only when the execute bit is 1 and the execute signal is received from the central timing circuits. The function of the instruction processor is to route operands to an arithmetic unit with information as to what operation is to be performed.

To avoid the problem of assigning arithmetic units to active instruction processors, each memory location has its own arithmetic unit. The memory register itself serves as the accumulator, the register that contains the operand which is to be used and then replaced by the result of an arithmetic operation. Thus, by using the arithmetic unit at the location where the result is to be stored, there will never be a time when an instruction processor must wait for an arithmetic unit. An attempt by two instructions to store information in the same location at the same time is considered a programming error.

An additional feature of having many arithmetic units is to allow one instruction to specify that an operation is to take place at many locations simultaneously. The addressing of many locations by a single instruction is accomplished by indirect addressing.

As might be expected, in this machine the data accessing for arithmetic operations is considerably different from conventional computers. No fetching of instructions is required, thus the data accessing circuits can be simplified and computation speed is increased. Even with this simplification, far too much circuitry would be required if each instruction processor needed the ability to access every memory location directly. To cut down the number of components and yet keep flexibility of accessing the following or-

ganization is used:

For each memory location there is a module containing the memory register, instruction processor, arithmetic unit, and what we will call path-connecting circuitry. Each module has direct connection, by wire without gates, to a few other modules. For one module, say X, to gain access to a module not directly connected, say Y, the destination (address of Y) is gated onto the wire that directly connects X to a module closer to Y. As soon as a path has been completed from X to Y, X has access to the memory register and arithmetic unit in the Y module. In this way every module can have access to every other module while having a direct connection to only a few modules.

One of the most significant factors in the design of such a machine is the logical organization of path segments (directly connected modules). The two extremes of path segment organization are: (1) every module connected to just two other modules (geometrically the modules could be placed in a line with wire connecting adjacent points on the line and the two end points), and (2) every module connected to every other module (geometrically, k modules would form a k-1 dimensional simplex).

Neither of these seems acceptable for the general-purpose computer being considered by this paper. The line of modules uses less components than any other but relatively few accesses could be made simultaneously, e.g., several short paths could isolate many modules from others to which access is required.

Let the machine under consideration have 2^n modules. Now, as a compro-

mise between the number of components and expected number of simultaneous accesses, let each module have a direct connection to n other modules. Thus the number of direct connections is a function of the size of the machine. For a 32,768-word machine each module would be connected to 15 other modules. The logical organization of these connections would be to have the modules as the vertices of a 15-dimensional cube with the edges of the cube being the direct connections between modules. Since each module can be represented by a unique 15-bit number, the direct connections correspond to a wire from each module to the 15 other modules whose numbers differ in one bit position, i.e., unit Hamming distance.

There is a physical construction whereby the wires for the direct connections can be laid out in n layers for a machine with 2^n modules. The modules are laid out in a two-dimensional square array as shown below. Each layer contains exactly one connection for each module and no connections cross within a layer. The layers could be made by deposition or printed circuit techniques. Only n/2 masks would be required and each mask would have 2^{n-1} lines on it formed from $2^{n/2}$ repetitions of a $2^{\frac{n}{2}-1}$ line pattern, e.g., for a 4096-module machine n = 12; thus each mask would have 2048 lines formed from 64 copies of a 32-line pattern.

The mask, layers, and composite view of a machine with 16 modules are given below:

There are several interesting measures of accessibility which depend on the logical organization. First, the maximum length, in number of segments, that any minimal path may be is n in a machine with 2^n modules, i.e., maxi-

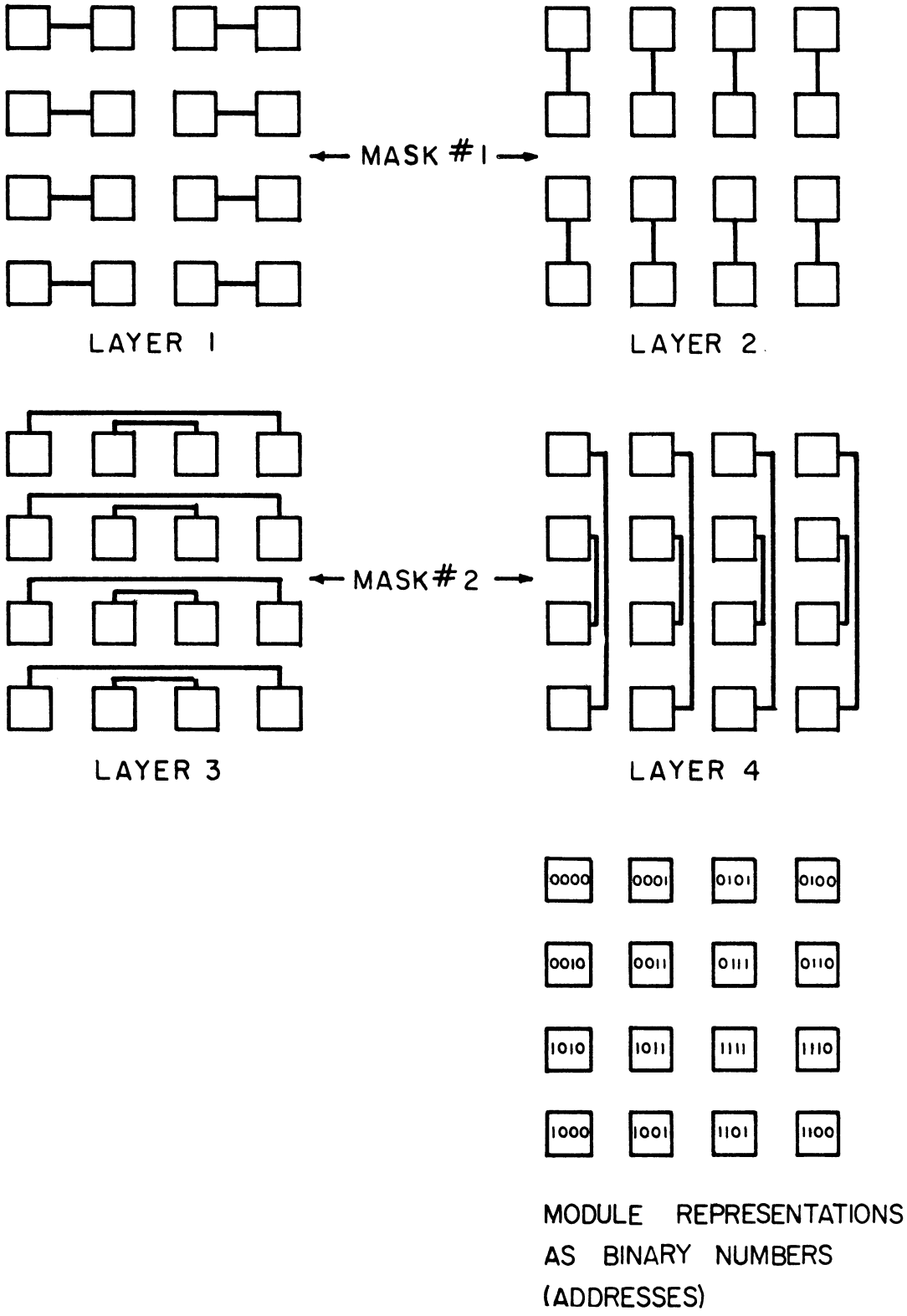

Fig. 1. Detail of path segment wiring.

mum Hamming distance between two n-bit numbers is n. Second, the number of different paths between two modules differing in k bits is k! i.e., all permutations in the order of reducing the Hamming distance by 1 each step for k steps. Finally, statistically the expected number of simultaneous accesses that could be made in a 4096-module machine is over 300, assuming random storage assignment of data and instructions. Of course, instructions and data are not randomly assigned storage. Considering the timing factor on accessibility that each module is directly connected to only a few others, it is not difficult to see that clever programming could yield many more simultaneous accesses than the random case, while intentionally poor programming could yield many less.

Due to the inherent limitation on parallel accessing as the number of paths increases, it seems advisable to remove all path connections when the access has been completed. In this way each step in the execution of a program starts with an uncluttered machine.

Actually, by allowing the machine to have some paths still connected when the next execution step begins, there can be a path-connecting lookahead which could, in general, speed up computation more than no lookahead and an uncluttered machine. Preliminary logical design of a module indicates that clever logical circuit design could make the average path-connecting time about the same as the longest arithmetic operation time. Thus the average time to perform an operation becomes equal to the time required by the slowest operation, but there is no accessing time required during a sequence of execution cycles.

To allow simultaneous path connecting from an active instruction to the

first operand (also arithmetic unit), to the second operand, and to the succeeding instruction, three independent path-connecting circuits are provided.

There are no index registers (relative addressing) as exist in conventional computers. This is necessary due to the unconventional scheme of accessing. In place of index registers, operations are provided so that one instruction can do arithemtic directly on the address part of another instruction, i.e., the address part of every instruction is essentially an index register.

To further supplement addressing an indirect address can be specified. When a path has been connected from an instruction to some module, say X, and if the instruction specified indirect addressing, the path is extended according to the address in the memory register of X. The address in X may also be designated as indirect. Since the memory register of X is large enough to hold several addresses, each address position is interpreted and each can start an extension of the path into X. In this way one instruction with an indirect address can refer to a memory location with several indirect addresses, each of which can refer to other locations, etc. Thus, one instruction can control the arithmetic units of many randomly placed modules simultaneously.

A more detailed description of this machine's operation is given in the next section which is essentially a programming manual. A more detailed description of the logic follows that section.

4. INSTRUCTION CODE

The programming of an iterative circuit computer must be flexible enough

to justify having a highly parallel computer rather than a number of single-processor computers. Since it is possible that hundreds of instructions could be executing simultaneously and these instructions could be using the same data, the hardware must provide some basic synchronization of instructions. Therefore, in order to simplify programming, the computer execution cycle proceeds as follows: a number of instructions are being executed simultaneously; each specifies locations of instructions to be executed next; when all instructions have completed execution all of the "next" instructions start executing simultaneously; and so on.

Thus the execution of individual instructions is asynchronous, but the execution of sequences is synchronous. Even if the programmer specifies more instructions than the machine can execute simultaneously, the hardware is set up to process all of them in several bunches. Then, when all have been executed, the "next" instructions are started.

INSTRUCTION FORMAT

Every instruction has the same basic format: an operation code and three addresses.

The first address, α, may designate the location of one operand for arithmetic and logical operations. The result of the arithmetic and logical operations replaces the contents of α. With conditional transfer operations, α may be used to specify the location of the next instruction.

The second address, β, may designate the location of the second operand for arithmetic and logical operations, i.e., multiplier, divisor, etc. For some conditional transfer instructions, β is the location tested for the con-

dition. With shift instructions, β is the shift count rather than an address.

The third address, γ, designates the location of the next instruction. For some conditional transfer operations, the condition determines whether α or γ specifies the next instruction.

The number of bits and relative positions of an instruction are shown in the following figure: (n might range from 10 to 20 depending on the number of locations, 2^n, in the computer).

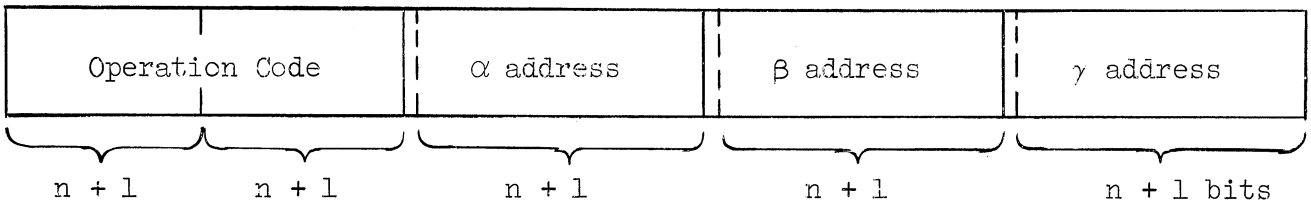

Fig. 2. Instruction format.

The word length is $5(n+1)$ bits. If the leftmost bit of α, β, or γ is 1, then that address is indirect. The remaining n bits specify the location to be used.

The three-address scheme allows flexible arithmetic and control instructions to aid parallel programming and spatial program organization.

EXECUTION BITS

There are three more bits at each location to control execution instructions, called e_1, e_2, e_3. The e_2 bit of a location is set by any instruction referring to that location as a successor by a γ address, i.e., the contents of the location where the e_2 bit is set will be active, or execute as an instruction, during the next execution cycle. The e_3 bit of a location is set if the contents of the location are to be inactive during the next execution

cycle. The e_3 bit is set rather than the e_2 bit depending on the operation code.

At the beginning of each execution cycle, the e_1 bit is set if the e_2 bit was set and the e_3 bit was not set during the previous instruction cycle. If both e_2 and e_3 were set (by different instructions), the e_1 bit is not set. Once the e_1 bit is computed, both e_2 and e_3 are reset.

These three bits influence execution in the following way: once the e_1 bit has been determined at every location, the instructions in all these locations become active. Those instructions which successfully completed paths for α, β, and γ accesses reset their e_1 bit and perform their operations. Some of these instructions will be setting e_2 and e_3 bits of other locations. While operations are being performed by these instructions, the others with e_1 bit set but paths not completed try again to connect their α, β, and γ paths. This process repeats itself, possibly requiring a number of attempts for some instruction to complete its path. The execution cycle terminates when all e_1 bits have been reset. At this time, the next execution cycle begins with the computations of e_1 bits as specified by the instructions of the preceding execution cycle setting the e_2 and e_3 bits.

The hardware has been designed so that a large number of instructions can simultaneously have their paths connected. After an instruction has completed its operation, its paths are removed (disconnected). A priority scheme has been developed which allows any number of paths to be forming simultaneously, and which also guarantees that at least one path will be connected on each attempt. Therefore, in the worst possible case, the time re-

quired to execute a group of instructions activated during a given execution cycle will never exceed the time required to execute them sequentially.

INTERPROGRAM PROTECTION

To provide isolation of instructions and data, an additional bit is required at each memory location. If this 'isolation' bit is set in some module, the hardware will not allow a path to be built through the module, but the module may still be a path termination. The ability to be a termination is necessary in order to allow for the resetting of the isolation bit.

To isolate a program, those locations containing instructions on data which form a spatial boundary must have their isolation bits sets. If all programs in the machine have their boundary isolation bits set, there will be a barrier that prevents any program from accessing any other program.[1]

INDIRECT ADDRESSING

The contents of a location referred to as an indirect address are interpreted as shown in the figure below. There are five possible addresses and any combination may be used. An unused address is recognized by the fact that it is all zero. Any combination of addresses may be specified as indirect by setting the leftmost bit of these addresses. The computer timing imposes a limit of 40, at most, on the length of a sequence of dependent indirect addresses.

[1]The isolation bit can be set or reset by the LOAD instruction with the appropriate modification of the operation code.

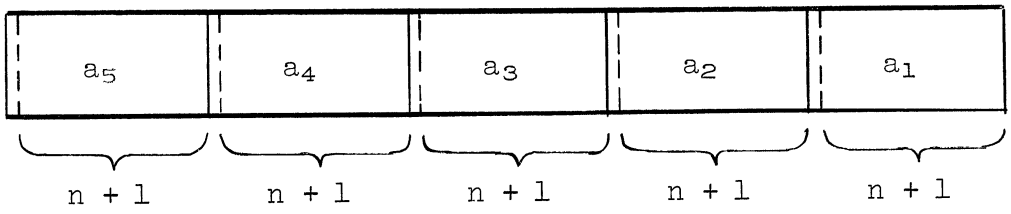

Fig. 3. Location referred to by indirect address.

Through the use of indirect addressing it is possible to have one instruction perform its operation on the contents of many locations simultaneously. This is done by having α be an indirect address. The contents of the location that α refers to can have up to five more indirect addresses, each of which can refer to five more, etc. Thus, a tree structure of paths is connected from the instruction to many modules. Upon execution of the instruction, the operation code followed by the second operand is sent down the tree and all the terminal modules perform the operation simultaneously.

Similarly, γ can specify one successor directly, or many successors, through indirect addressing.

ARITHMETIC OPERATIONS

The four basic arithmetic operations — addition, subtraction, multiplication, and division — are available. The normal mode of full-word arithmetic is floating point. The mantissa is shifted to make the characteristic zero whenever no loss of accuracy occurs. In this way the programmer has the benefits of high speed when working with integers, and full accuracy by automatic scaling of non-integers. The format for full-word numbers is given below. The magnitude of a number is the mantissa (binary point to right of low-order bit) times two raised to the power of the characteristic.

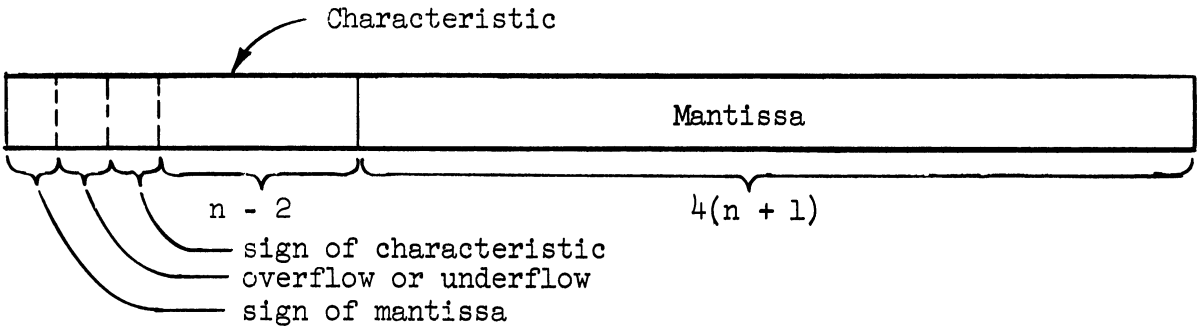

Fig. 4. Format for full-word number.

Each number has an overflow-underflow bit which is set if at any time the magnitude exceeds $2^{[2^{n-2} - 1]}\left(2^{[4(n+1)]} - 1\right)$ or is less than $2^{-[2^{n-2} - 1]}$. When the overflow bit is set, the remaining bits of the number are reset, made zero. Any succeeding arithmetic operation on a number with its overflow bit set results in another number which also has its overflow bit set. Normal arithmetic is performed on numbers even if their overflow bits are set. Overflow or underflow can occur only on a full-word addition, subtraction, multiplication, or division. There is also a conditional transfer instruction capable of testing the overflow bit.

BYTE MODIFICATION

To facilitate relative addressing and instruction composition, the arithmetic operations add and subtract, as well as load, complement, and, or, exclusive or, and some conditional transfer instructions can refer to any of the five n+1 bit bytes. For designating which bytes are to be operated on there are five bits in the operation code which can be used to modify these operations.

0 0 0 0 0	specifies full-word arithmetic. Fig. 3.
0 0 0 0 1	specifies the operation is to be performed on the low-order n+1 bits (a_1) of the operands.
.	
.	
.	
1 0 1 1 0	specifies the operation is to be performed on the a_5, a_3 and a_2 bytes.
.	
.	
.	
1 1 1 1 1	specifies the operation is to be performed on all five n+1 bit bytes.

All bytes are assumed positive, a negative result remains in the byte as its 1's complement. There is no carry from one byte to the next. Carry-out of the high-order position is lost.

These partial word operations may be used directly on instructions or indirect address words as if the locations were index registers.

TRANSFER INSTRUCTIONS

Since each instruction has a γ address to specify the location of its successor, only conditional transfer instructions are assigned specific operation codes. There are two basic forms of conditional transfer instructions: BRANCH and PROCEED.

The BRANCH operation tests the contents of the location specified by the β address. If the condition (specified in the three low-order bits of the operation code) is met, a signal is sent to the modules designated by the α address. Otherwise, the signal is sent as usual to the modules designated by the γ address.

The PROCEED operation compares the contents of the locations specified by α and β. The comparisons such as $=$, $\neq$, $>$, etc. are specified by the three

low-order bits of the operation code. If the comparison is not true, the instruction is treated as if no successor were specified. Otherwise, the successor specified by the γ address is treated in the normal way.

INHIBIT MODIFICATION

The use of the word 'signaled' rather than 'transferred to' is necessary because in this machine many instructions can be executing simultaneously. It is possible for an instruction to specify itself as its successor for incrementing or counting purposes. Another instruction could be testing for a desired value. When this value is reached there must be a way to stop the instruction which is transferring to itself. The ability to stop an instruction from executing during the next execution cycle is called the inhibit modification. Every instruction has a bit in its operation code which if 1 causes the signal to the successor to set the e_3 bit (described on page 12). If the inhibit modification bit is 0, the signal goes to the e_2 bit of the successor. In either case, the effect of a signal applies only to the next execution cycle.

Any instruction can be a local HALT instruction by having an all-zero γ address. i.e. no successor.

INPUT-OUTPUT INSTRUCTION

The α address of the input-output instruction refers to a module which is directly connected to a particular I-O device of the desired type. The memory register of this module may contain control information for the I-O device. The location of information which is entering or leaving the com-

puter is specified by the β address of the I-O instruction.

Each I-O device has its own simple buffer between itself and the main computer. For a magnetic tape unit, the buffer may be core storage which holds several blocks of information. As long as there is information available on reading or space available on writing, the main computer uses only a normal length of execution cycle for an I-O operation. If the buffer is empty or full, execution is held up until the I-O operation is completed. Backspace, rewind, skip file, etc. are determined by the information in the location connected to the I-O device. These require only a normal length execution cycle unless the queue of commands exceeds the buffer capacity, in which case further execution in the main computer must wait.

With this type of I-O, the programmmer should give control information as early as possible and do information I-O at the last possible moment.

OPERATION CODES

Arithmetic

1.	ADD α, β, γ	The contents of α are replaced by $\alpha+\beta$. (Byte or full word)
2.	SUBTRACT α, β, γ	The contents of α are replaced by $\alpha - \beta$ or $\beta - \alpha$, depending on the high-order bit of the operation code being 0 or 1 respectively. (Byte or full word)
3.	MULTIPLY α, β, γ	The contents of α are replaced by $\alpha \cdot \beta$. (Full word only)
4.	DIVIDE α, β, γ	The contents of α are replaced by α/β or β/α, depending on the high-order bit of the operation code being 0 or 1 respectively. (Full word only)

Logical - Byte modification applies to 5 thru 9

5. LOAD α, β, γ — The contents of α are replaced by the contents of β.

6. AND α, β, γ — The contents of 2 are replaced by the bit wise AND of α with β.

7. OR α, β, γ — The contents of α are replaced by the bit wise OR of α with β.

8. EXCLUSIVE OR α, β, γ — The contents of α are replaced by the bit wise EXCLUSIVE OR, ring sum, of α with β.

9. COMPLEMENT α, γ — The contents of α are bit wise complemented.

Shifting - Full word only

10. SHIFT α, β, γ — β is not an address. β is the number of bit positions the contents of α are to be shifted. The first and second bits of the operation code being 1 and 0 respectively determine left or right and end around or linear. Vacated positions on linear shifts are filled with zeros. A shift instruction with $\beta = 0$ is a NO OPERATION that requires 1 execution cycle and can specify a successor.

11. SCALE α, β, γ — The contents of α are treated as a floating point number. The low-order n-1 bits of β are treated as a sign and magnitude of a characteristic. β is not an address. If the first bit of the operation code is 1, then the mantissa in α is shifted so as to make the characteristic equal to β. If the first bit of the operation code is 0, then β is added to the characteristic in α and the mantissa in α is shifted accordingly. The second bit of the operation code being 1 or 0 specifies rounding or truncation respectively.

Transfer

12. BRANCH if $\mathcal{R}(\beta)$ α, β, γ — If $\mathcal{R}(\beta)$ is true a signal is sent to location α, otherwise the signal is sent to γ. $\mathcal{R}(\beta)$ may be any of the following:
a) $\beta = 0$ (Byte or full word)
b) β negative

c) β has overflow bit set

13. PROCEED if $\alpha \mathcal{R} \beta$ α, β, γ

If $\alpha \mathcal{R} \beta$ is true a signal is sent to location γ, otherwise no signal is sent. $\mathcal{R}$ may be any of the following:
a) $\alpha > \beta$
b) $\alpha \geq \beta$
c) $\alpha = \beta$
d) $\alpha \leq \beta$
e) $\alpha < \beta$
f) $\alpha \neq \beta$
All relations above may apply to byte or full word.
g) the βth bit of α is a 1
h) the βth bit of α is a 0
If α refers to more than one location through indirect addressing, the logical OR of the α's will be used to test the relation.

Other - Full word only

14. INPUT-OUTPUT α, β, γ

α is the module which controls the I-O device. The memory register of α contains the command for the I-O device while β specifies the location into which information is read, or out of which information is written.

15. SENSE PANEL α, γ

The contents of the display panel is the address of the last module where an unresolvable programming error was detected. e.g., trying to execute an undefined opeation code. The contents of the display panel replace the β address position of location α.

16. SET ISOLATION α, γ

The isolation bit is set to 1 at location α. α may still be referred to by other instructions but no access can be made which would use a path through α.

17. RESET ISOLATION α, γ

The isolation bit is reset to 0.

18. ERROR MODE γ

Depending on the first two bits of the operation code being 1 or 0, this instruction sets the mode of operation to:

	continue or stop executing instructions of type 1 thru 18 and activate or inhibit ERROR START instructions respectively. The mode remains set until changed.
19. ERROR START γ	If there is an error and the computer is in ERROR START activate mode, all instructions with this operation code become active during the next execution cycle.

This concludes the description of instructions available in the hardware of the computer. Because many operations have bits which further qualify them, an assembly language distinguishing the various operations would be useful to programmers. The operations are meant to be convenient to the general-purpose programmer. Many special instructions, symbol and list manipulation, etc., have purposely been omitted to keep the amount of hardware to a minimum. This should cause no loss of speed since special instructions can be achieved by clever programming using simultaneous application of those instructions given above.

5. PHYSICAL AND LOGICAL DESIGN

Now comes the problem of determining how much circuitry would be required by a machine as described in the earlier sections of this report. The most accurate way to determine the required number of components is to do a complete logical design. Even then, the cleverness of the logical designer and the choice of component types could affect the result by a factor of two or three. Considering the time involved to do a detailed logical design and considering that we are far from the cleverest logical designers available, the following approach was taken: The part of this machine not found in con-

ventional computers, the path-connecting circuitry, was designed in some detail. The logical circuitry for arithmetic operations, timing pulse generations, etc., was not designed. Instead, their requirements are given with estimates for the number of components required based on current technology.

We will first consider the somewhat conventional hardware that must be in each module. Even here the logical design would not really be conventional. Where, at most, hundreds of computers of a given type may have been built, we are talking of building thousands of modules for a single machine. Although a module is versatile when embedded in an I.C.C., it is far from being a complete computer. Thus, due to the greater importance of economical design and lesser requirements, a greater effort could be justified for a fully integrated, clever, logical design. A number of trial modules could be built, tested and perfected with the goal being low costing mass-produced modules.

There are several components which could be used in the construction of modules. For example, RTL circuits can be produced fairly inexpensively in quantity using low-speed, low-power transistors. The RTL circuits which are currently being manufactured by deposition techniques have a density of about 100 transistors and 400 resistors per square inch. Circuits such as these used in an I.C.C. have the advantage of less noise pick-up since the physical size of a module can be small and the connecting leads between modules can be correspondingly short. A second component potentially useful for module construction is all-magnetic logic. Again, this is not the fastest possible logical component but it is reliable and potentially can be manufactured by automated equipment. Multi-aperture cores and other types of all-

magnetic logic require fairly close tolerances, thus careful design. Moreover, this type of design is well suited to modules which have relatively few external connections to other modules. A final example of a potentially inexpensive and fast component is the cryotron. Again, automated production may be possible, and making a large number of identical modules should reduce considerably the cost per module.

At first glance, everyone considers an I.C.C. impractical, even with inexpensive construction, since it could have thousands of modules which seem to be simplified versions of processing units in conventional computers. Although an I.C.C. requires many times the number of components in conventional computers, one cannot expect to get simultaneous accessing, simultaneous arithmetic, and simultaneous instruction processing without more components. To show that a module is far less than the processing unit in conventional computers we will list all the circuitry that is <u>not</u> in a module but is in conventional processing units.

First, there are several obvious registers that are not required in a module. There is no sense (storage) register or address register since there is no store to access. There is no instruction register or instruction counter since execution, not instructions, moves from module to module. The one-word store in the module corresponds to a conventional accumulator. By having numbers in the integer form with scale factors, the multiply and divide operations can be performed in a single-length accumulator.

The next major block of circuitry, not within modules, is for timing. There could be one timing unit which would make all the required sequences

of control pulses available to all modules. By having more specific timing sequences available than in conventional computers, the amount of logic in a module can be greatly decreased. The central timing unit becomes correspondingly larger, but the component saving in one module multiplied by the number of modules should be far greater.

Considering that instructions are all the same basic format and are relatively stationary, a scheme exists for having various bits in the memory register of a module directly control gates when the module is executing an instruction. This would eliminate most of the instruction decoding circuitry existing in conventional machines.

There would have to be a basic adder and the arithmetic control logic in every module. Here, a decision between serial and parallel arithmetic would have to be made based on the differences in speed and cost.

The remaining circuitry in a module is the path-connecting logic. In place of drivers, cores, and sense amplifiers, the path-connecting logic closes gates in various modules, forming a path to access information in other modules. To give an idea of how much circuitry is required for path connecting, a fairly complete logical design of this follows:

The basic segments from which paths are formed are conductors from the periphery of one module to the periphery of another. Fig. 5a shows the top view of an I.C.C. with the modules appearing as squares. Fig. 5b shows how each module passes through a number of thin layers on which the path segments (the conductors) are placed (by printed circuit or deposition techniques). The addresses α, β, and γ are shown here to have completely isolated path

structures. Correspondingly three times as much path-connecting circuitry would be required in each module. The choice of separate layers for each address can stem from the fact that this is well over three times as fast from computation standpoint and yet requires less than three times the circuitry since each layer can be specialized to its particular address bit positions.

Before describing the logical circuitry for path connecting, we will explain the function the circuitry must perform. Basically, the problem can be stated as follows: There is a binary number in the memory register of some module. This binary number refers to another module. The circuitry must close gates to form a path between these two modules. The path must allow information to flow in both directions. A path need not be a single wire; it could be physically a bunch of wires for transmission by bytes or in parallel, and there could be two separate circuits for transmission in each direction. For convenience of explanation and simplification of logical circuitry, a scheme with one wire for each direction will be used. See Fig. 6. The connections to the module labelled P_1 through P_6 are physically n wires (where there are 2^n modules in the computer). Information can flow in P_1 out P_2 , in P_3 out P_4 , and in P_5 out P_6 without affecting the operation of this module. This module may initiate paths along a wire of P_2 , P_4 , and P_6 as determined by the α, β, and γ addresses. Other modules may access this module by having their paths terminate in the P_1 , P_3, or P_5 lines of this module. When this module is used as the α address of some instruction, the operation code of the instruction enters before the β operand. The operand code enters via some wire of P_3 and is placed in the arithmetic control ope-

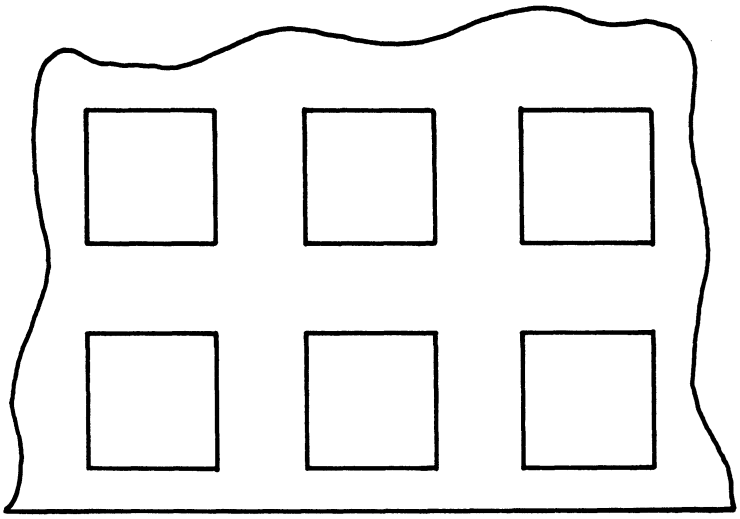

Fig. 5a. Top view of the I.C.C.

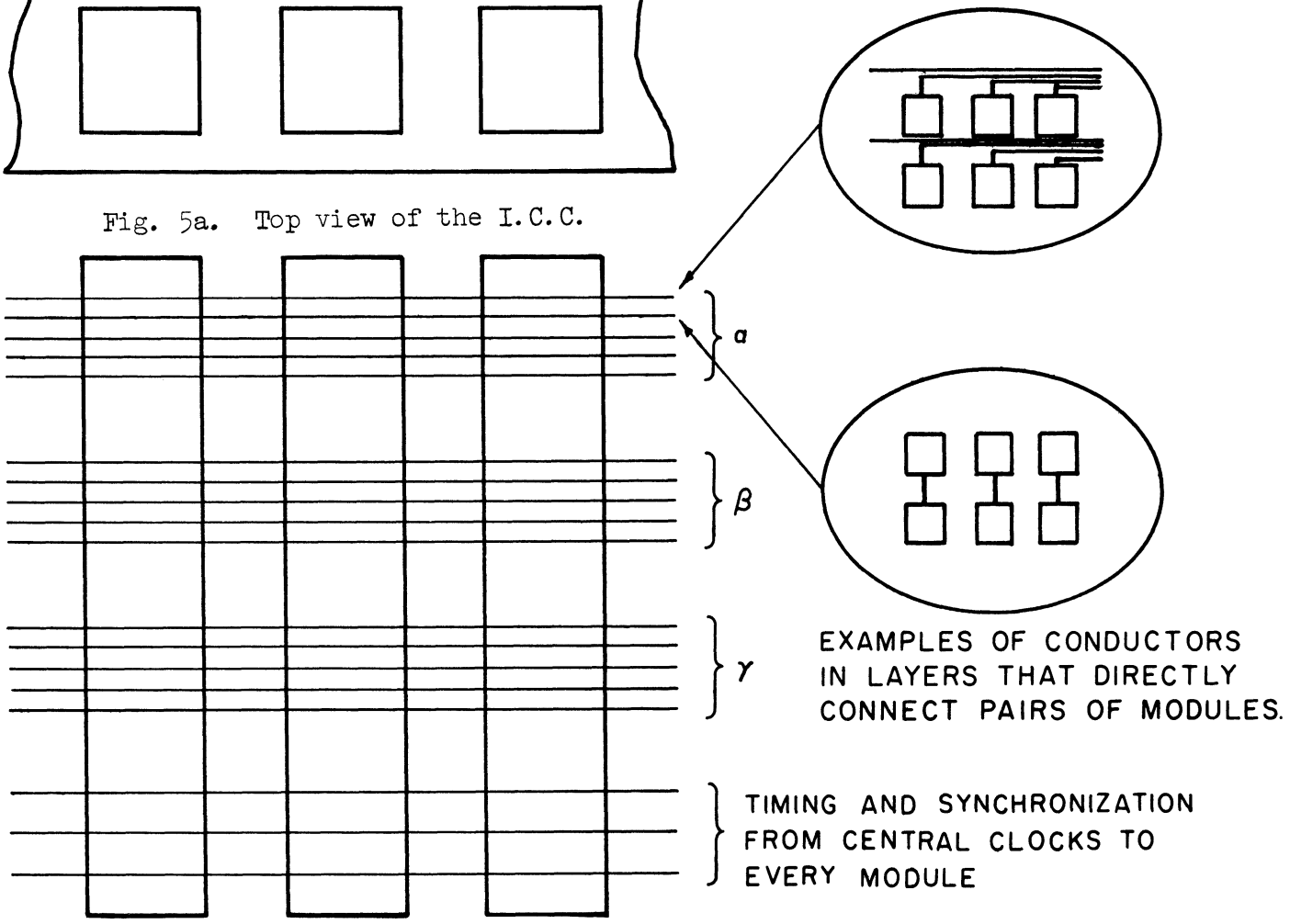

Fig. 5b. Side view of the I.C.C.

ration register.

The control of arithmetic operations in this module comes from the operation register and not from the memory register. No addresses need be sent to the module acting as an arithmetic unit since the module containing the active instruction is doing the required switching to set up the operand accesses.

Figure 7 shows the significant information flow when an instruction executes. In this example, the contents of the memory register of module Y are being added to the memory registers of modules X_1 and X_2. The ADD instruction is in module R, and indirect addresses are in module X.

Execution proceeds as follows:

Step 0 — This e_1 bit (described on page 12 of this report) is set assuming an activate signal was sent to module R over one of its P_5 paths. The e_2 and e_3 bits in R are reset.

Step 1a — A path is connected from R to X. (The prime indicates that X is an indirect address.) Then two paths connect from X to X_1 and X_2 respectively.

1b — A path is connected from R to Y (second operand).

1c — A path is connected from R to S (next instruction).

Step 2a — The e_2 bit in module S is set. Removal of the path between R and S begins at S.

2b — The operation code from the memory register of module R is sent to the operation registers of X_1 and X_2 via X.

Step 3 — Module R controls the gating of the contents of the memory register of Y to the path into X. Module X controls the gating from its P_1 input to the two lines to X_1 and X_2. X_1 and X_2 set their gates to send the contents of their memory registers and the incoming path from X to their adders respectively.

Step 4a — X_1 and X_2 gate the resultant sum back to their respective memory registers.

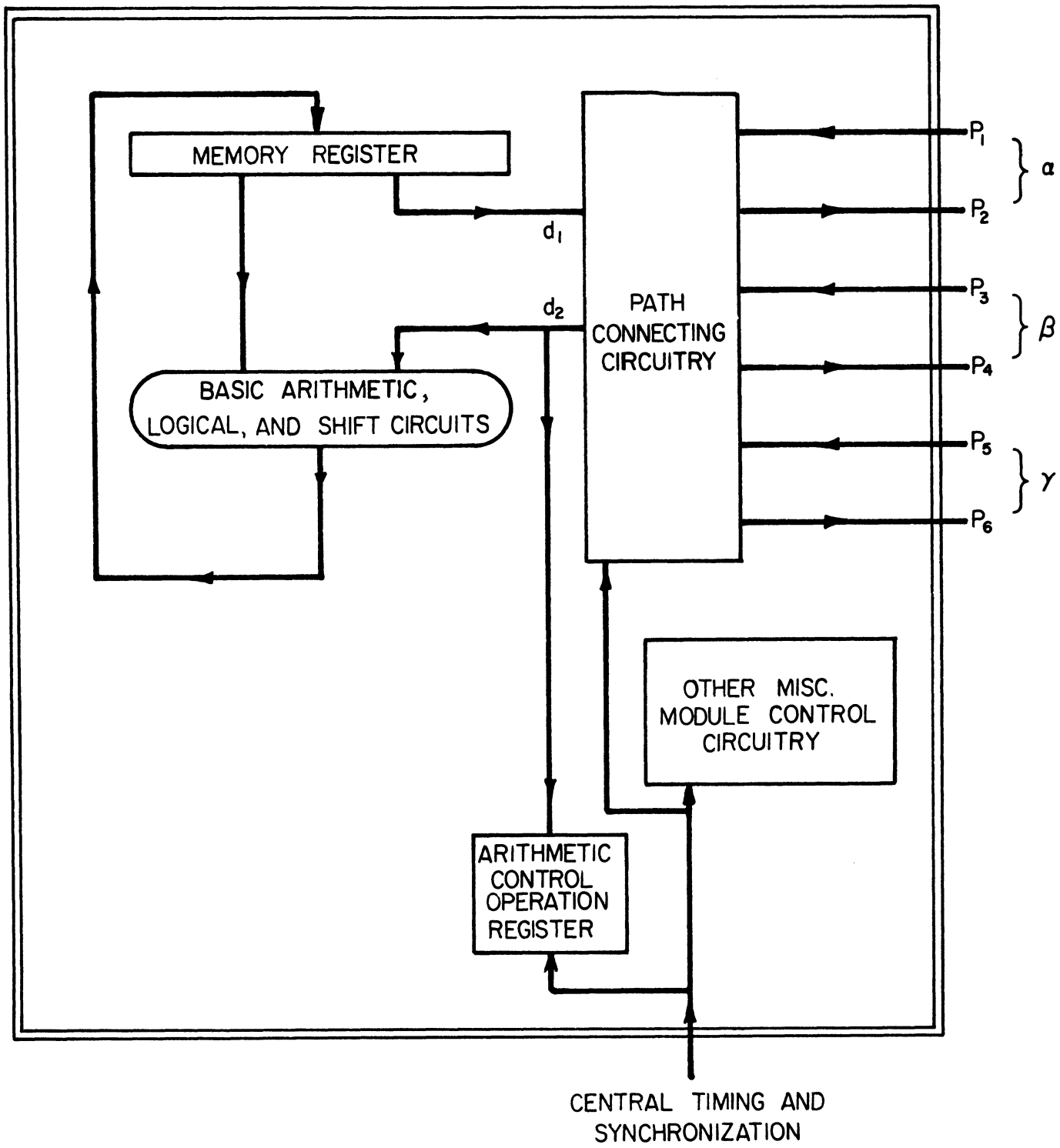

Fig. 6. Function and flow block diagram of module.

4b Removal of the paths into X_1 and X_2 is begun at X_1 and X_2 respectively.

4c Removal of the path from R to Y is begun at Y.

Step 5 When all paths have been removed to R, the e_1 bit of R is reset.

Step 6 When all e_1 bits are reset the central synchronization emits a signal to all modules which compute the new e_1 bits and Step 0 begins again.

This completes the description of Fig. 7 involving the overall path structure. We will now concentrate on one type of path, say α. (For convenience of construction, all three types of paths, α, β, and γ would probably be the same logic, or all three could be operating simultaneously in the same circuitry if some restrictions were placed on programming.)

The decision procedure for connecting a path that must be performed in each module requires two pieces of information. Each module must know its own binary representation as an address, called 'HERE.' (This can be wired into the layers shown in Fig. 1. thus allowing all modules to be identical and interchangeable.) Also, each module must know which of its accessible n path segments are busy. (This we will call the 'BUSY' register.)

The n bit address of the termination of a path can come from n+5 places, i.e., n from the n path segments connected to this module plus 5 from the 5 byte positions of the memory register. For instructions, only the three low-order bytes are addresses which could initiate paths but an indirect address can cause all 5 bytes to initiate paths.

Suppose an n bit address has reached a module. This is the address of the termination of a path. By taking the bit-wise exclusive or of this n bit

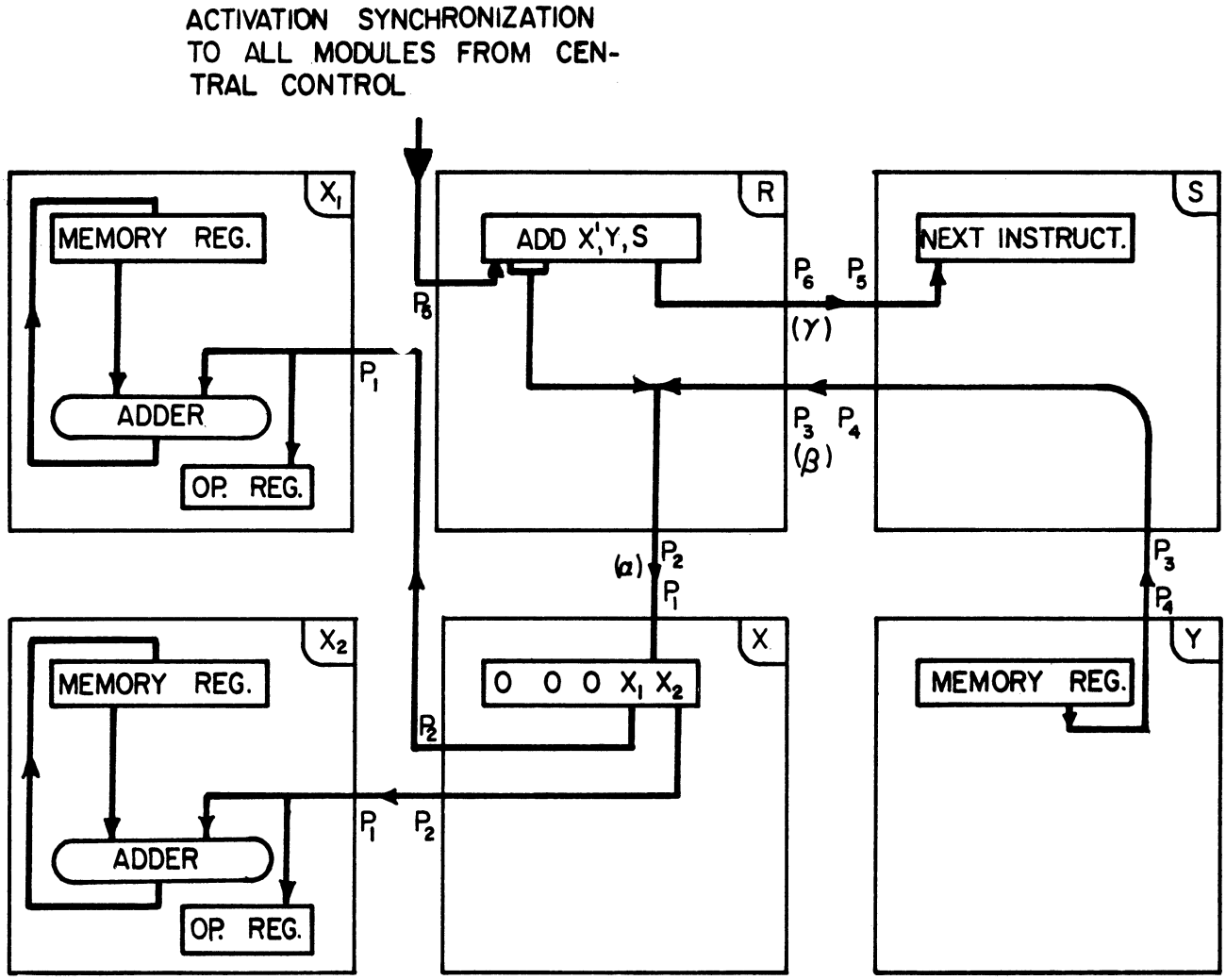

Fig. 7. Information flow during execution.

address with the n bit representation 'HERE', those positions of the result which are 1 denote the possible path segments which can serve as extensions for the path. This is just a reduction of 1 in the Hamming distance since each neighbor of a module differs from it in exactly one bit position. We will establish the convention that the lowest 1 bit resulting from the exclusive or will be tried first as an extension of the path. It may be that the desired segment is already being used by another path, in which case the BUSY register has a 1 in that position. To eliminate busy segments from potentially useful segments, the complement of BUSY is anded to the result of the previous exclusive or. This result is retained in a 'GOING' register.

To connect from the n possible incoming path segments to the n possible outgoing segments an **n x n** switching matrix is used. Five more inputs are appended to the switching matrix to allow for path initiation, and a diagonal pair of wires allow for path termination at a module.

The operations of path connecting are staggered such that all modules with an even number of 1's in their addresses, 'HERE', extend (or remove) their paths one segment during alternate times with modules having an odd number of 1's in their addresses. In this way, priority problems are avoided which involve two adjacent modules trying to connect to their common segment. It is possible to have two modules connect a path to the same module at the same time and have the same destination for both paths. This priority decision is made by the circuitry just prior to setting the gates of the switching matrix. The circuit for this two-dimensional priority selector is shown in Fig. 9. The composite of the logic just described is shown in Fig. 8. The

one-directional segments are shown as B_1, B_2, B_3, ..., B_n grouped under the P_1 designation. The one-directional outputs are grouped under the P_2 designation. For a path to pass through a module two inputs will be connected to two outputs with reversed subscripts, thus forming a piece of a two-directional path.

If a path cannot be extended due to complete blockage by other paths, a NO-GO signal is sent back towards the origination of the path. Upon receipt of a NO-GO signal, a module selects the next (higher order) potentially useful segment from the 'GO' register.

To further explain the logic involved, an example of the progression of a path connection is given in Fig. 10. Here we have a machine with n equal to 4. Only 8 of the 16 total modules are shown and only the values of 'HERE', 'DESTINATION', 'BUSY' and 'GO' are shown in boxes. The path-segment connections B_1, B_2, B_3, B_4 each correspond to a pair P_1 and P_2 shown in Fig. 8.

We will concern ourselves with the path originating at module 0001 with the destination 1010. We assume two other paths, indicated by ..... and ----, are already present. The path connecting proceeds in two phases. During phase A, the modules with an odd number of 1's in their address perform the logic to compute the contents of their 'GO' registers, and the modules with an even number of 1's in their address transmit the 'destination' over the path segment specified by the lowest 1 bit in their 'GO' registers. During phase B, the roles of the two sets of modules are reversed. To get the phasing started there is no transmission between modules on the first step and no logic on the last step of path connecting. Thus we have for Fig. 8:

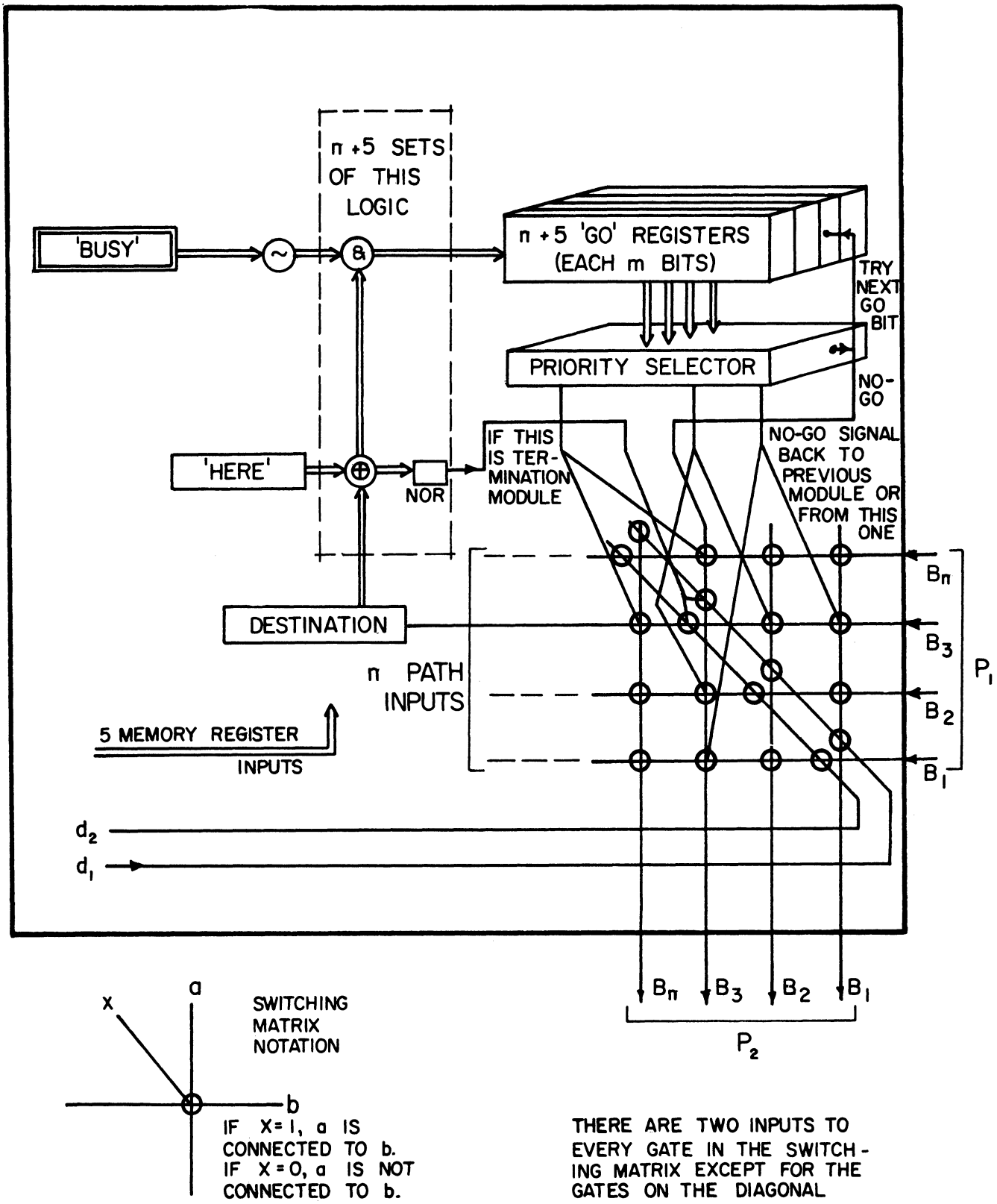

Fig. 8. Function and flow diagram for path-connecting circuitry.

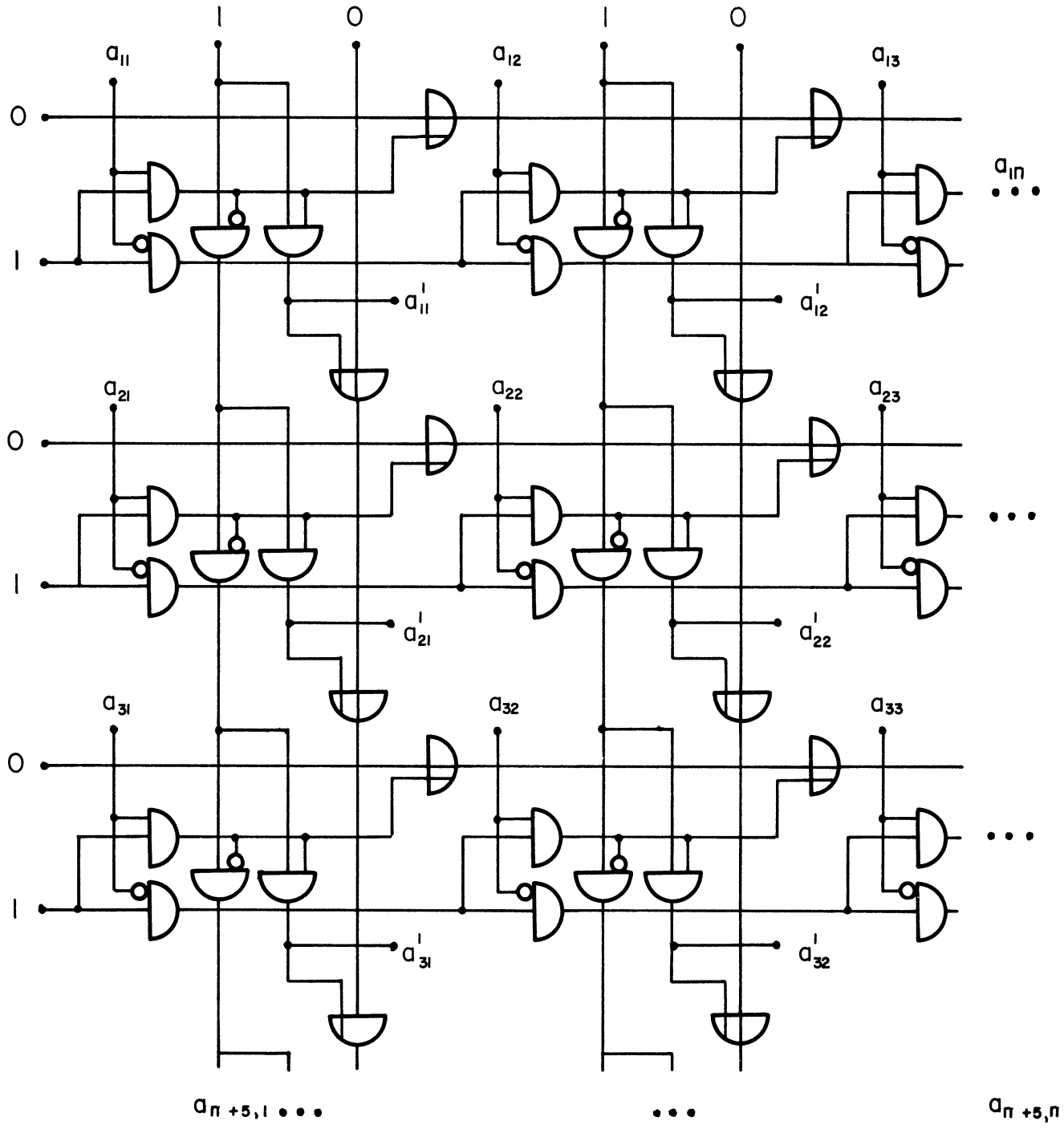

Fig. 9. Two-dimensional priority selector.

Step 1A

The module 0001 is initiating a path and therefore computes the contents of its GO register.

(HERE ⊕ DESTINATION)∧ ~ (BUSY) ,→ GO.

is (0001 ⊕ 1010) ∧ ~ (0000) = 1011

Other modules could simultaneously be initiating or extending paths but for simplicity of explanation only one path is being considered.

The lowest 1 in the GO register of 0001 determines which segment will become a part of the path. This segment connects to module 0000 which is one step closer to the destination than 0001.

Step 1B

The BUSY bit for the B_1 segment is set in both 0001 and 0000. The destination is sent along the segment ⟨1⟩ to 0000.

The GO register of module 0000 is set to (0000 ⊕ 1010)∧ ~ (0111) = 1000. The second and third BUSY bits had been previously set when the path coming in B_3 and going out B_2 was built.

The lowest 1 in the GO resister of 0000 specifies that the B_4 segment, ⟨2⟩ is to become a part of the path.

Step 2A

The 4th BUSY bit is set in 0000 and 1000 and the destination is sent from 0000 to 1000.

the GO register of module 1000 is set to (1000 ⊕ 1010)∧ ~ (1010) = 0000.

Step 2B

Since the GO register is all zero a NO-GO signal is sent back along ⟨2⟩. The 4th BUSY bits in 1000 and 0000 are reset.

Upon receipt of the NO-GO signal the module 0000 sets the lowest 1 in its GO register to zero. (In this case making it all zero.)

Step 3A

Since the GO register is all zero a NO-GO signal is sent back along ⟨1⟩ . The first BUSY bits of 0000 and 0001 are set to zero.

Upon receipt of the NO-GO signal, 0001 sets the lowest 1 in its GO register to zero.

Step 3B

The lowest 1 in the GO register of 0001 is now in position 2, thus a segment ⟨3⟩ is added to the path. The 2nd BUSY bits are set in 0001 and 0011 and the destination is sent from 0001 to

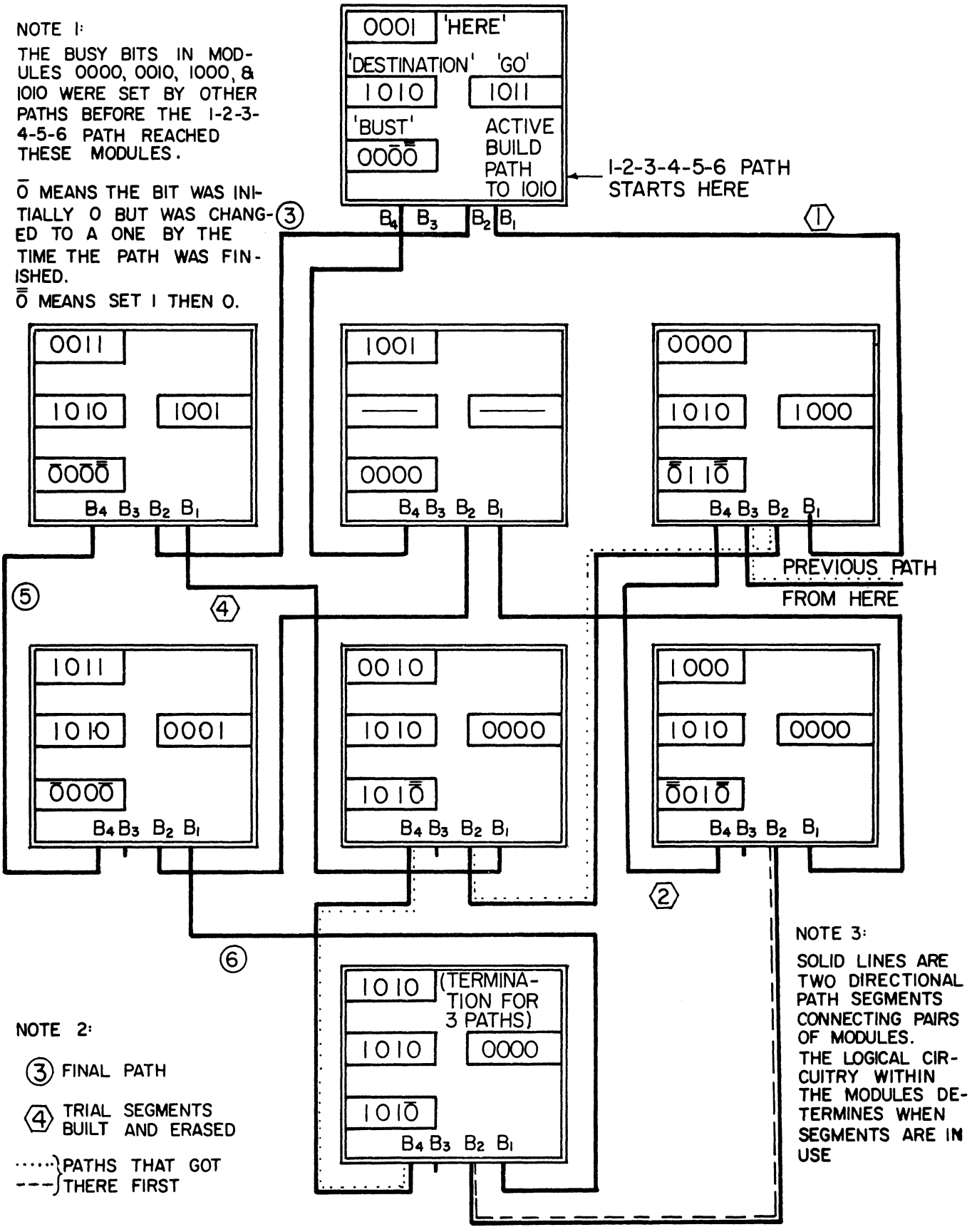

Fig. 10. Progression of a path connection.

0011.

The GO register of 0011 is set to $(0011 \oplus 1010)_{\wedge} \sim (0010) =$ 1001.

Step 4A — The segment ⟨4⟩ is added.

The GO register of 0010 is set to $(0010 \oplus 1010)_{\wedge} \sim (1011) =$ 0000.

Step 4B — A NO-GO signal is sent back along ⟨4⟩

The lowest GO bit is set to 0 in 0011.

Step 5A — The segment (5) is added.

The GO register of 1011 is set to $(1011 \oplus 1010)_{\wedge} \sim (1000) =$ 0001.

Step 5B — The segment (6) is added.

The termination is detected since $(1010 \oplus 1010) = 0000$. Execution using this path can now take place.

The logic and transmission properties could be designed to perform one phase per basic clock time. Thus the example given above would require 10 basic clock times to complete the path. Lookahead could be accomplished by having paths connecting by successors while the arithmetic operations of the predecessors are being performed. It seems that the average path-connecting time will require about the same time as an average arithmetic operation.

6. CONCLUSION

The basic question of the economics of an I.C.C. is: How much is fast computation worth? We have yet to hear a concrete answer to this question, and indeed there will probably never be a simple answer. The consensus seems to be that a computer twice as fast in every respect is not worth twice the

cost. To determine how much speed is worth, the reliability, type of problem, qualifications of the programmers and numerous other factors must be considered. For a computer such as being described here, there is one further factor to consider. That is: How parallel is highly parallel? There are examples of problems that could be done on this machine in 1/1000 the time required by a conventional computer built from the same components. There are other examples where this machine could barely cut the computation time in half. We have no accurate measure of the average parallelism possible in this machine. Based on our experience in considering a few problems, we estimate that on the average between 10 and 100 instructions could be executing simultaneously on a medium-sized computer. This is to be contrasted with our educated guess of a cost 10 to 100 times that of a conventional machine.

A medium-sized I.C.C. is certainly within engineering feasibility. Perhaps the first machine of this type should be designed for a user with much computation suitable for parallel processing, i.e., on problems involving matrices, solving systems of equations, inverting matrices, finding eigenvalues; or in other specific problems such as solving boundary value differential equations by the relaxation method. In these and some other problems, hundreds of calculations could be made simultaneously. By specifically choosing the command structure and size of the machine for a few specific problems, an economically competitive computer could be built today.

The rather powerful machine described in detail in this report is tailored to a need not yet fully developed. Until some good programmers and numerical analysts have such a machine in their hands, it is difficult to pre-

dict how much potential a computer of this type will have. We are optimistic that the iterative circuit computer organization is one of the methods that will enable computers to do much more computation in a given time.

APPENDIX A

HARDWARE REQUIREMENTS FOR MACHINE AS DESCRIBED

The path-connecting logic and registers require the following hardware in each module:

Quantity	Bits of Storage	Logical Elements	Description
n+5	n		'GO' storage memory
1	5(n+1)+4		storage 'BUSY' storage
1	n		
3(n+5)		n input logic	&, ⊕, NOR logic
n^2		switching logic	Switching matrix*
n(n+5)		one stage priority	priority circuit

For a 4096-module machine n would be 12. The number of bits of storage would be (204 + 69 + 12) 4096 = 1,167,360. (This is a few less than the number of storage bits in conventional memory of 32k 36-bit words). There would also need to be about another 1.6 million simple logical elements.

We estimate approximately 400 logical elements for the arithmetic unit in addition to 26 bits of storage for the operation control register.

Another 100 logical elements would be needed for miscellaneous module control. These would be for computing execution bits, signaling successors, and routing operands to the appropriate paths, etc.

Assuming the central timing and synchronization to be less than 10% of the machine, the total number of storage bits and logical elements would be less than five million.

* For serial two-way transmission, multiply by the number of bits to be sent in parallel.

APPENDIX B

MATRIX INVERSION PROGRAM FOR AN I.C.C.

INSTRUCTION LOCATION	OPERATION	ADDRESSES α,β,γ
*		
*		THE NXN MATRIX IN THE 'A' REGION IS
*		INVERTED BY A GAUSS-JORDAN METHOD.
*		THE INVERTED MATRIX REPLACES THE
*		ORIGINAL CONTENTS OF THE 'A' REGION
*		
ENTRY	PROCEED=	0,0,SET1'(1)
*		
*		FIRST EXECUTION STEP OF A THREE-EXECUTION
*		STEP LOOP THAT WILL BE PERFORMED N TIMES.
*		
SET1'(1)...SET1'(N)	INDADR	LA'(1),E(1),Q'(1)...LA'(N),E(N),Q'(N)
SET1'(N+1)	INDADR	EXIT
*		
E(1)...E(N)	LOAD	AKK,A(1,1),F(1)...AKK,A(N,N),F(N)
*		
LA'(1)...LA'(N)	INDADR	L(1,1)...L(N,1),...,L(1,N)...L(N,N)
L(1,1)...L(1,N)	LOAD	B'(1),A(1,1),M(1,1)...B'(1),A(1,N),M(N,1)
L(N,1)...L(N,N)	...	B'(N),A(N,1),M(1,N)...B'(N),A(N,N),M(N,N)
B'(1)...B'(N)	INDADR	AT(1,1)...AT(1,N),...,AT(N,1)...AT(N,N)
*		
Q'(1)...Q'(N)	INDADR	U(1,2)...U(1,N),,U(I,1)...U(I,I-1),U(I,I+1)...
	ETC	U(I,N),,U(N,1)...U(N,N-1)
U(1,1)...U(1,N)	DIVIDE	A(1,1),A(1,1)...A(1,N),A(1,1),T(1)
U(N,1)...U(N,N)	...	A(N,1),A(N,N)...A(N,N),A(N,N),T(N)
*		
*		SECOND EXECUTION STEP
*		
F(1)...F(N)	DIVIDE	A(1,1),AKK,G(1)...A(N,N),AKK,G(N)
*		
M(1,1)...M(1,N)	MULTIPLY	C'(1),A(1,1),S(1,1)...C'(N),A(1,N),S(1,N)
M(N,1)...M(N,N)	...	C'(1),A(N,1),S(N,1)...C'(N),A(N,N),S(N,N)
C'(1)...C'(N)	INDADR	AT(1,1)...AT(N,1),...,AT(1,N)...AT(N,N)
*		
T(1)...T(N)	LOAD	V'(1),ZERO,Y(1)...V'(N),ZERO,Y(N)
V'(1)...V'(N)	INDADR	A(2,1)...A(N,1),,A(1,I)...A(I-1,I),A(I+1,I)...
	ETC	A(N,I),,A(1,N)...A(N-1,N)

INSTRUCTION LOCATION	OPERATION	ADDRESSES α,β,γ
*		
*		
*		THIRD EXECUTION STEP
*		
G(1)...G(N)	DIVIDE	A(1,1),AKK,SET1'(1+1)...A(N,N),AKK,SET1'(N+1)
*		
S(1,1)...S(1,N)	SUBTRACT	A(1,1),AT(1,1)...A(N,1),AT(N,1)
S(N,1)...S(N,N)	...	A(N,1),AT(N,1)...A(N,N),AT(N,N)
*		
*		ON THE ITH PASS THROUGH THE LOOP
*		THE ITH ROW OF SUBTRACT INSTRUCTIONS
*		IS INHIBITED.
*		
Y(1)...Y(N)	INHIBIT	-,-,Z'(1)...-,-,Z'(N)
*		
Z'(1)...Z'(N)	INDADR	S(1,1)...S(1,N),...,S(N,1)...S(N,N)
*		
*		END OF COMPUTATION LOOP
*		
*		STORAGE ASSIGNMENT
*		
A(1,1)...A(1,N)	DATA	
A(N,1)...A(N,N)	...	
*		
AT(1,1)...AT(1,N)	TMPSTR	
AT(N,1)...AT(N,N)	...	
*		
AKK	TMPSTR	
*		
ZERO	DEC	0
*		
	END	

To illustrate the Gauss-Jordan method the following ALGOL program is given.

```
procedure   INVERT (N,A); value N; integer N; real array A;
comment     The N by N matrix in the A region is inverted by a Gauss-Jordan
            method. The inverted matrix replaces the original contents
            of the A region;
begin       integer I,J,K; real AKK,AIK;
    for     K:=0 step 1 until N do begin
            AKK:=A[K,K]; A[K,K]:=1.;
            for   J:=0 step 1 until N do A[K,J]:=A[K,J]/AKK;
            for   I:=0 step 1 until N do begin
                  if   I≠K then begin
                       AIK:=A[I,K]; A[I,K]:=0.;
                       for   J:=0 step 1 until N do
                       A[I,J]:=A[I,J]-AIK X A[K,J];
                  end   skip reduction of the Kth row;
            end   Kth column finished and matrix to left reduced;
    end     all N columns finished;
EXIT: end INVERT
```

A few additional comments should enable the interested reader to understand the details of the I.C.C. program.

First, the sequence of instructions which form the "loop" are:

ENTRY —(SET1'(1))→ E(1)→F(1)→G(1) —(SET1'(2))→ E(2)→F(2)→ ... →F(n)→ G(N) —(SET1'(N+1))→ EXIT. Thus, the number of execution steps is 3N+1.

Next, the instructions are the 3-address type described in Part 4. The first address is one operand and the location of the result; the second address is the second operand; and the third address is the next instruction to be executed. The last address being omitted implies it is not used, while a "—" implies an intermediate address is not used. An address with a prime, ', is indirect. Indirect address words may refer to more than one other word indirectly, thus enabling one instruction address to refer to many locations.

Further, the "*" at the front of a line implies that the line is a comment.

Finally, the ... notation has the usual meaning: to generate all intermediate subscripts.