AN ABSTRACT OF THE THESIS OF

RUDOLPH JOSEPH FRANK for the DOCTOR OF PHILOSOPHY
(Name of Student) (Degree)

in Electrical and Electronics Engineering presented on 9-3-71
(Major) (Date)

Title: A FEASIBILITY STUDY ON THE USE OF ARITHMETIC-MEMORY REGISTERS IN THE DESIGN OF DIGITAL COMPUTER SYSTEMS

Abstract approved: Redacted for privacy
Professor Louis N. Stone

The concept of combining arithmetic and memory capability on a single semiconductor chip has become practical from a system's viewpoint through the decreased cost of semiconductor memories and high circuit densities achieved through large scale integration. This paper describes a model for studying the feasibility of such systems.

An arithmetic-memory register is defined as the basic hardware unit. A model consisting of instruction states and transition states is developed. The model is then applied to both past and contemporary computer structures. A general purpose machine is formulated from a set of arithmetic-memory registers. The feasibility of this structure is studied with respect to both performance and implementation.

The utilization of arithmetic-memory registers is also shown to be applicable to special-purpose systems. A system designed to compute power spectra is described. The state model proved to be a useful technique in the design of the system. Cost estimates and measures of performance were significant factors influencing the feasibility of the system. The structure was also found suitable for real-time application.

A Feasibility Study on the Use of Arithmetic-Memory Registers in the Design of Digital Computer Systems

by

Rudolph Joseph Frank

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

June 1972

APPROVED:

Redacted for privacy

Professor of Department of Electrical and Electronics Engineering

in charge of major

Redacted for privacy

Head of Department of Electrical and Electronics Engineering

Redacted for privacy

Dean of Graduate School

Date thesis is presented 9-3-71

Typed by Illa W. Atwood for Rudolph Joseph Frank

ACKNOWLEDGEMENT

The author would like to thank Professor Louis N. Stone for his guidance and direction in the preparation of this thesis. The author also wishes to thank Oregon State University for their support of this work. And lastly, the writer wishes to especially thank his wife, Donna, and his parents, Mr. and Mrs. Joseph Frank, for their constant encouragement.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

LIST OF FIGURES

LIST OF FIGURES (cont.)

LIST OF FIGURES (cont.)

LIST OF TABLES

LIST OF SYMBOLS

A_i	Terms associated with ith stage of fast-Fourier transform.
B	Upper bound for the number of possible interconnections for rectangular arrays.
C	Cost constant for arithmetic-memory registers.
CPU	Central processing unit.
D_n	Execution time to compute n-point spectral estimates.
f	Frequency in hertzs.
G	Number of signals analyzed by conventional system to match price of special-purpose system.
i	Designation for register identification.
I/O	Input and output.
j	Index for discrete frequency.
m	Number of operations in instruction set.
n	Number of arithmetic-memory registers.
PE	Processor element.
P(f)	Power spectral density.
P(i)	Designation for input to register i.
$R(\tau)$	Autocorrelation function.
R_n	Reduction ratio for system execution times for n-point transforms.
t	Time or time step.
T	Signal period.
x(t)	Time varying signal.

X	Cost per hour for computing time on conventional system.
X(f)	Fourier transform.
V	Multiplication execution time.
Δt	Sampling interval.
τ	Delay time.
ξ	No-op command for instruction state and blank input for transition state.
*	Multiplication command for instruction state.

A FEASIBILITY STUDY ON THE USE OF ARITHMETIC-MEMORY REGISTERS IN THE DESIGN OF DIGITAL COMPUTER SYSTEMS

I. INTRODUCTION

The speed of both past and contemporary computing systems has been limited primarily by the interactions between the processor and memory, and secondarily, by the pure arithmetic operations performed by the processor. To combat this limitation, designers of computer systems have relied on the following two methods:

1. Reducing the cycle time associated with the memory of the machine.
2. Increasing the number of processors associated with the machine.

Early attempts to increase computing speeds by these two methods were curtailed by the high cost of random access core memory and the increased complexity in the control of additional processors. However, recent advances in solid state technology have revealed the potential of semiconductor memories (Moore, 1971). The advantage in speed is clearly evident when comparing the 1 μsec. cycle time of standard 20 mil cores to the 400 nsec. cycle time of presently available semiconductor memories.

Projections indicate that semiconductor memories will also provide a cost advantage for both small and large capacity memory systems. Medium scale integration (MSI) introduced the capacity to include significant amounts of logic in the design of these memory systems. Large scale integration (LSI) holds even greater promises of combining memory and logic at reasonable cost since a larger number of logic elements can be squeezed on a chip.

The method of increasing computing speed by additional processors involves an interdependence between the hardware and software capabilities. From a hardware standpoint the potential of producing a chip capable of storing information and performing a limited amount of logic is certainly evident. The interconnection of several such chips would be analogous to the interconnection of several processors. Identical internal structure within a chip would enhance the utilization of LSI techniques.

From a software standpoint the utilization of more than one processor creates additional overhead in system performance. A case study on a multi-processing system undertaken by Rosenfeld (1969) indicated that the performance of a system does not necessarily increase linearly with the number of processors. Certain cases were cited where the performance decreased due to the additional overhead. Burnett, Koczela and Hokom (1967) studied the utilization of distributed processors in various applications and illustrated the difficulties encountered in the allocation of tasks for independent processors. The problem of determining which tasks do and which do not require preprocessing is a major cause of poor system performance.

This study formulates a model composed of units having both arithmetic and memory capabilities. The characteristic which differentiates this model from others rests in the concept that parallel processing is extended to its limit. Each unit has a memory element which acts as a single storage location together with an arithmetic element which executes preset instructions. The interconnection of these registers is universal, meaning that any unit can communicate with any other unit. Several classic machine structures are formulated by means of the model and the feasibility of special-purpose applications are also investigated.

II. BACKGROUND AND HISTORICAL DEVELOPMENT

Conventional Computer System

Conventional computer systems are composed of five basic units as illustrated in Figure 1. The central processing unit (CPU) controls the interactions of the various units through control lines and interupt routines. The memory unit is used only to store information. The arithmetic and logic unit performs the actual arithmetic or logic

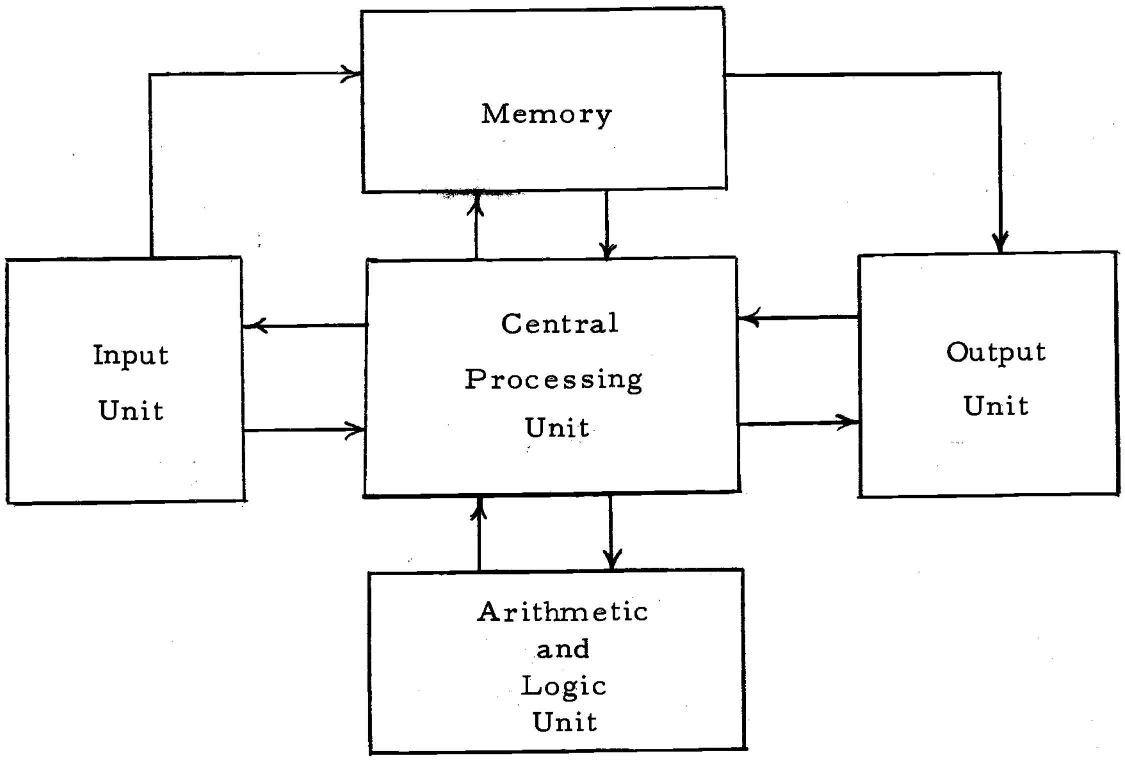


Figure 1. The five basic elements of a conventional computer system.

operations on the desired information. The input unit is used to transfer information into the computer memory, and the output unit is used to transfer information from the computer to the user.

The operation of the conventional type system is usually characterized by the following sequence of operations:

1. Sequential loading of memory with program and data.
2. Sequential computation of program instructions.
   - A. Access of memory for instruction.
   - B. Access of memory for data.
   - C. Execution of instruction on data.
   - D. Access of memory to store results.
3. Sequential unloading of computed results from memory.

When a user's program is to be processed, the CPU transfers control to a subroutine called the loader. This subroutine is resident in the memory and its function is to store the user's program and data into memory by means of some input device. During the loading process, the address of the first instruction is stored in the program counter. The function of the program counter is to keep account of the next instruction to be performed. The program instructions are loaded sequentially until a terminating instruction is detected, this initiates the loading of the program data. At the end of the loading routine, the complete user's program and data will be stored sequentially in the memory.

The actual execution of the user's program is initiated by a command from the CPU to fetch the instruction whose location is specified by the contents of the program counter. This instruction is transferred from memory into the program register. The function of the program register is to store the current instruction being executed. Once the instruction transfer is completed, the program counter is incremented to the address of the next instruction.

The contents of the program register is then decoded by the CPU to determine the nature of the instruction to be performed. The instruction usually consists of an operation code plus an operand. Since the arithmetic unit has the task of actually performing the operation, the data or information whose address is specified by the operand must be transferred from memory into the accumulator of the arithmetic unit. The accumulator is the heart of the arithmetic unit in the sense that nearly all instructions are performed with reference to its contents.

Once the current instruction and data have been accessed and transferred from memory, the accumulator ? performs the execution of the instruction on the data. The results of this instruction are now resident in the accumulator. Therefore, if these results are not needed in the next instruction, the memory must be accessed once again in order to store the results. The acquisition of results from memory also necessitates an access of memory. The data must be unloaded in a sequential manner from the memory onto an output device. This cycle of accessing memory to fetch instructions, fetch data, store results and retrieve output is repeated until the terminating instruction ends the program.

The speed of conventional computer systems has increased with the technology. The transition from vacuum tubes to transitors and from discrete components to integrated circuits has increased the speed of the arithmetic unit. However, the time to access and transfer a word from memory to an accumulator (referred to as a memory cycle) is several orders of magnitude greater than the time to complete logic or arithmetic operations. Therefore nearly all gains in logic speed are overshadowed by the time allocated for memory transfers. This problem is compounded by the necessity of two memory cycles for nearly every instruction executed. One memory cycle for the instruction fetch and one memory cycle for the acquisition of the operand.

The Holland Machine

The Holland machine is a parallel network computer possessing an array of modules. Each module is provided with relatively independent sections. One section serves as a memory unit which consists of binary storage registers. The other section governs the general communication functions which link each module with its neighbors. The network of modules is rectangular in shape and fixed in size. The original model proposed by Holland (1959) had a limited amount of circuitry associated with each module so that special instructions could be executed.

The normal operation of the Holland network is to perform instructions in time-steps. At each time-step a module may be either active or passive. An active module normally interprets the contents of its storage register as an instruction and proceeds to execute it. If a given module is active at time-step t, then at time-step t+1, its communication section selects one of its four neighbors as its successor. This leaves the original module in the inactive status and one of its neighbors in the active status. In this manner a line of successors is established in the system since each instruction required a module, and therefore a sequence of instructions requires a sequence of modules.

This sequence of successors corresponds to a given subprogram of the computer. Since more than one module can be active at the same time, the computer can execute several subprograms simultaneously. However, limiting conditions do exist:

1. A module may not belong to more than four paths.
2. For a given time-step a module may be active in only one path.

These restrictions do limit the number of subprograms which a Holland machine can simultaneously execute.

Modified Holland Machine

A parallel machine proposed by Comfort (1963) has features quite similar to the Holland machine. Its organization is structured around a rectangular array of modules, and its instruction sequencing is governed by the communicating capability of the modules with their neighbors. However, the array modules provide only data and instruction storage. The actual arithmetic and logic computations are all performed by accumulators called A-boxes. The A-boxes are connected to the modules through a switch as illustrated in Figure 2.

An instruction sequence must start with an A-box. The line of successors through the module array is then determined. During instruction execution, the instruction in an active module is sent back through the line of predecessor modules to the original A-box. This sequence is continued until the instruction located within the terminating module is completed.

The SOLOMON System

The SOLOMON (Simultaneous Operation Linked Ordinal Modular Network) proposed by Slotnick, Borch, and McReynolds (1962) was a parallel network computer under a central control unit. The system

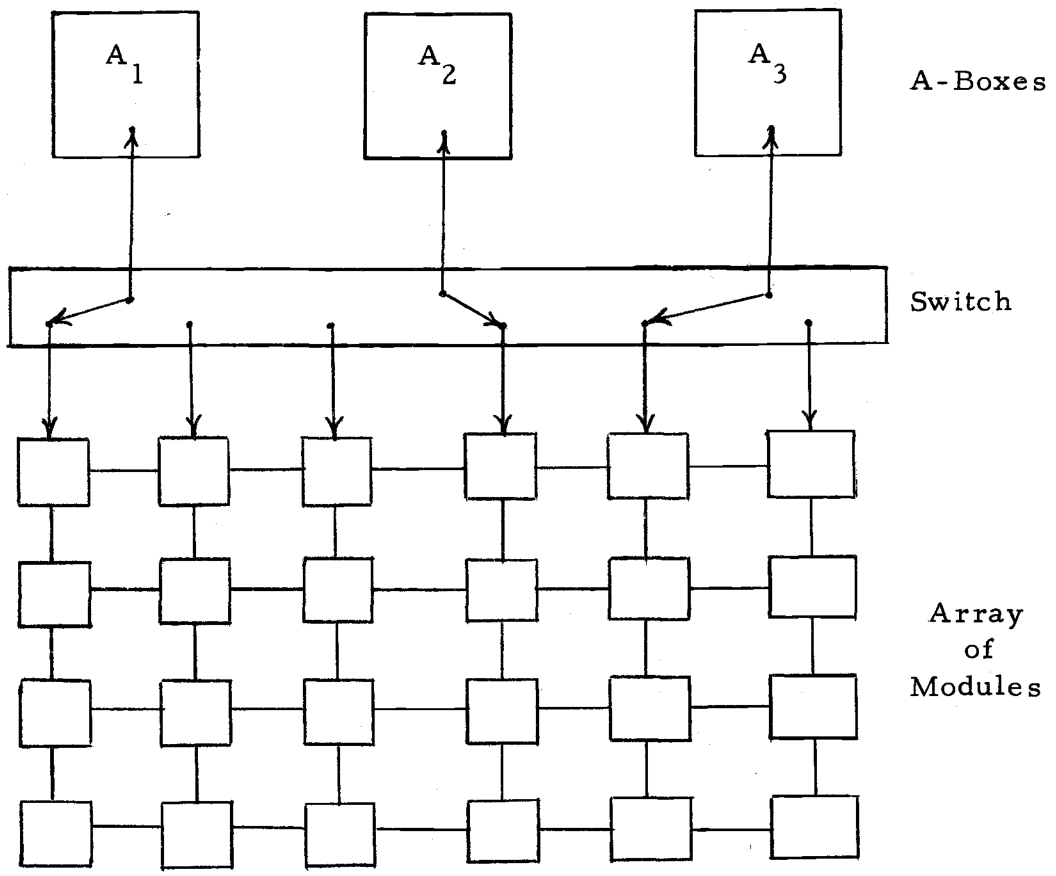


Figure 2. Parallel network computer with A-boxes and 4x6 array of modules.

consisted of many identical processing elements (PE's) in a matrix or mesh arrangement. The interconnections and programming of these elements were under the supervision of the central control. Figure 3 illustrates how the processing element array for a 4x4 network is structured.

The central control unit contains program storage in the form of large capacity random access memory. The stored instructions

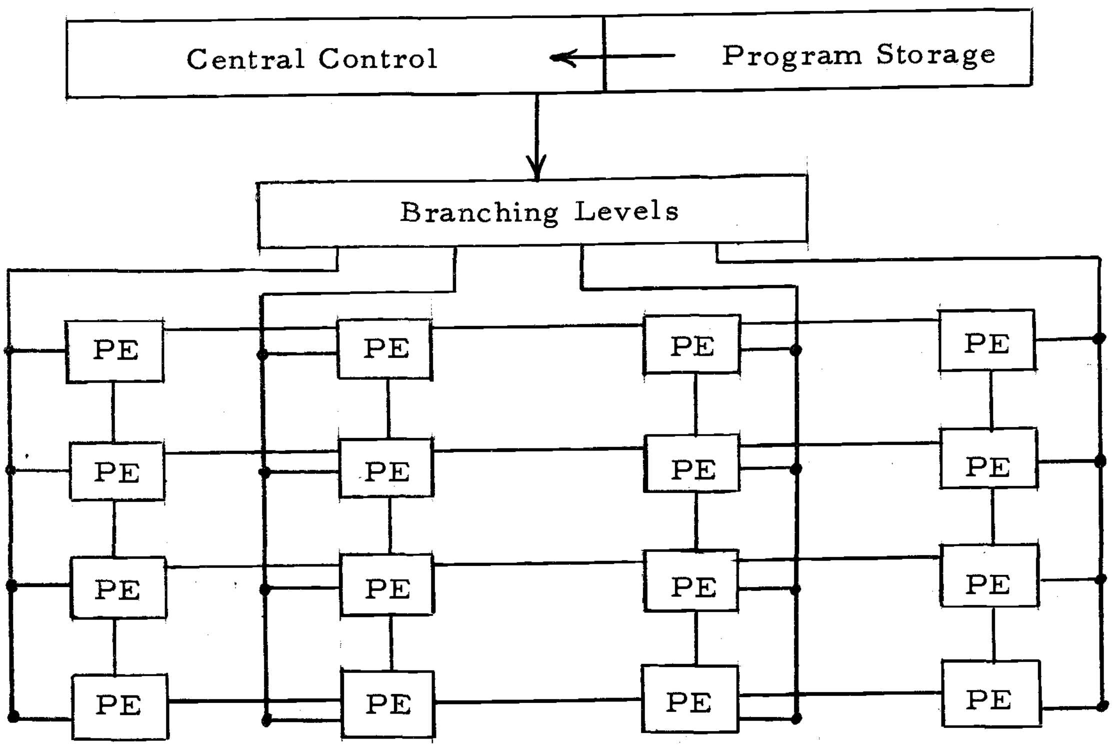


Figure 3. Processing element array for a 4x4 network SOLOMON system.

are retrieved and interpreted by the control unit and execution is performed within the processor array. The flow of the control information is illustrated in Figure 3 through the lines extending from the branching level. A given instruction is interpreted and translated into a sequence of signals which are transmitted from central control to the processing elements.

Each processing element can transmit and receive data serially from its four nearest neighbors. These elements basically consist of two parts:

4096 24bit-words

Error

1. A memory unit which provides 4096 bits of core storage.
2. An arithmetic and logic unit which provides the capability for arithmetic operations and direction of program flow.

A flow of information occurs between processing elements during network operation through this communication with neighboring elements.

The parallel operation of the SOLOMON system is shown in its capability of performing the same instruction on operands stored in the same memory location of each processing element. These instructions, however, may all be different and the system would still have the capability of parallel operation from central control. A limited amount of local control at the individual processing element level was obtained by permitting each element to enable or disable the execution of the central control instructions according to local tests. In other words, the direction of an instruction stream through the PE array could be altered by the logic portion of a single processing element in the array.

Studies with the original SOLOMON indicated that such a parallel approach was applicable to a variety of important computational areas. These areas included solutions of linear systems, the calculation of inverses of eigenvalues of matrices, correlation and autocorrelation, and numerical solution of systems of ordinary and partial differential equations.

ILLIAC IV

The ILLIAC IV computing system proposed by Barnes et al. (1968) consists of 256 processing elements arranged in four SOLOMON-type arrays of 64 processors each. The individual processors have a 240 nsec. add time and 400 nsec. multiply time for 64 bit operands. Each processor is also provided with 2048 words of 240 nsec. cycle time thin-film memory.

Each processor array possesses a common control unit which decodes the instructions and generates control signals for all processing elements in the array. Constants and operands used in common by all the processors are fetched and stored locally by the central control and broadcasted to the processors in conjunction with the instruction using them. The system can also be united by forming two arrays of 128 processors each or a single array of 256 processors. This capability is quite useful for matching problem dimensions with machine dimensions.

Figure 4 illustrates the system organization of ILLIAC IV. A large disk system is directly coupled to the arrays to provide back-up memory. The system program resides in a general-purpose B6500 computer which supervises program loading and array configuration changes. An I/O switch provides a real-time data linkage for the ILLIAC IV.

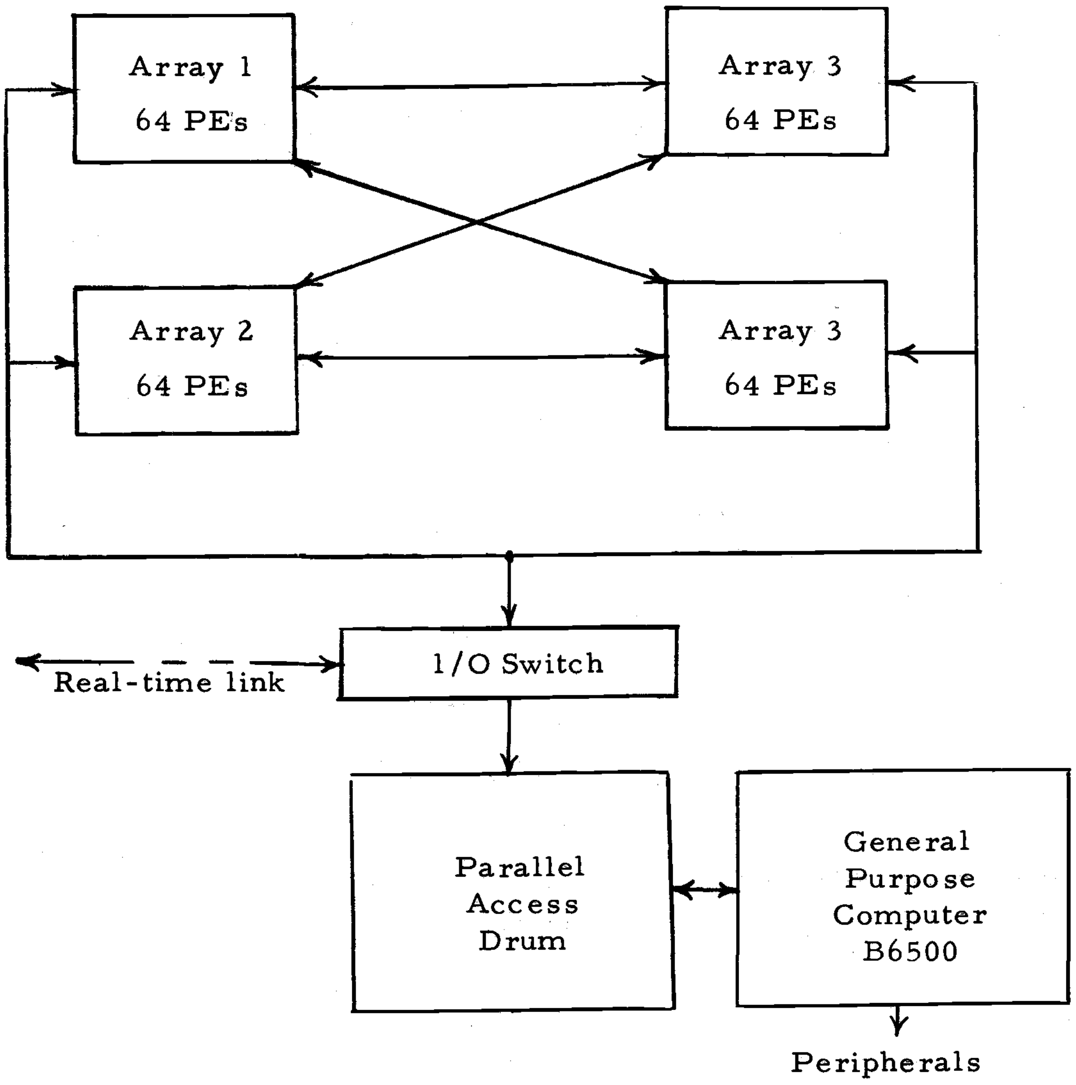


Figure 4. ILLIAC IV system organization.

MARIA

The multiple arithmetic iterative array computer (MARIA) proposed by Watson, Hansalik, and Emerson (1964) used a unique systems concept to achieve an increase in computational ability. The operands or data for specific problems were grouped into blocks where each block could be considered an operand for a block of instructions. For example, a simple block operation would be to add block A to block B. Since the iterative array of the arithmetic section was a three dimensional array of cells, the additions could be performed in a parallel fashion on a two dimensional plane within this array. This system is especially efficient for problems involving matrix manipulations.

Multiprocessing Systems

The SOLOMON and Holland machines are classic structures in the area of multiprocessing systems. The systems are similar in that both utilize a rectangular processing array. However the two systems differ in that the SOLOMON system employs a central control and the Holland system emphasizes local control at the processor level. The tremendous programming problems associated with both machines caused the development of many modified models.

Most multiprocessing systems including the ILLIAC IV and MARIA are designed upon the rectangular or mesh array of processors. This limitation of directional communcation among the processors is a primary cause of the intolerable programming problems encountered in such machines. An investigation of other processor structures may lead to fewer programming difficulties and more efficient machine performance.

III. DEVELOPMENT OF THE MODEL

To investigate the structure of a computer system, the limitations and characteristics of the arithmetic and memory capabilities must be defined. The following definitions will be the framework for a model which will be used to investigate computer structures.

Definition 3.1 A word is a set of n binary bits of information.

Definition 3.2 A memory element is a device capable of storing m words of information.

Definition 3.3 An arithmetic element is a device capable of operating on one or two input words to produce a single output word through a given instruction set.

Definition 3.4 An instruction set is a group of operations which can be executed by the logic or hardware contained in the arithmetic element. Figure 5 illustrates a typical instruction set consisting of 16 commands.

Definition 3.5 An arithmetic-memory register is a device consisting of both an arithmetic element and a memory element.

Independent Elements

The internal connection of an arithmetic-memory register can be implemented in many ways. Figure 6 illustrates a register in

Binary Coding	Instruction
Arithmetic	
0000	Add
0001	Subtract
0010	Multiply
0011	Divide
Branching	
0100	Unconditional Jump
0101	Jump if Equal
0110	Jump if Not Equal
0111	Jump if Less Than
Loading	
1001	Load from Register
1010	Load from Device
Shifting	
1011	Shift (right or left one bit)
Logical	
1100	Logical And
1101	Logical Or
1110	Exclusive Or
1111	Complimentation

Figure 5. A typical instruction set consisting of 16 binary coded instructions.

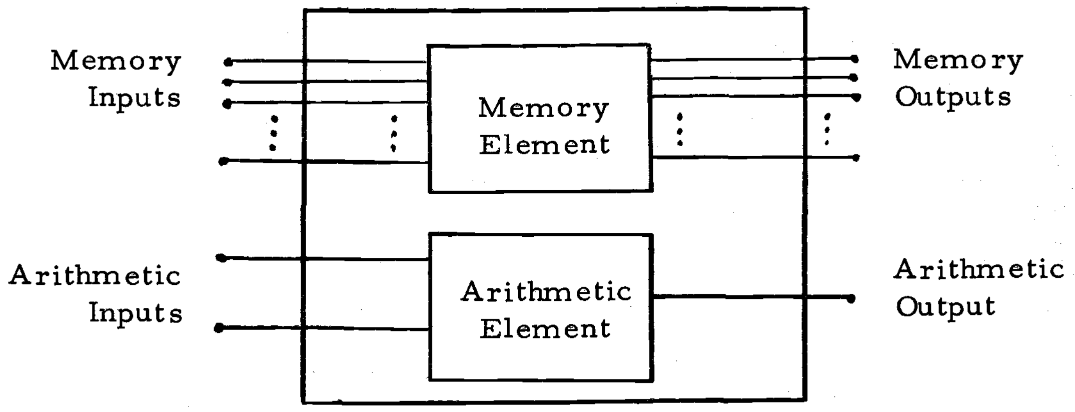


Figure 6. An arithmetic-memory register with independent elements.

which both inputs to the arithmetic element are free of connections to the memory element. In theory, an independent internal connection between the elements would make it possible to perform an instruction on two words residing in separate registers which are distinct from the operating register.

In applications where memory protection is needed, such a design would free the arithmetic element associated with any memory protected register. However this internal connection is not consistent with the concept of an arithmetic-memory register because the arithmetic element and memory element are independent and therefore the necessity of a common structure is eliminated. In short, this implementation is identical to the present concept of a memory module separate from the processing module.

Dependent Elements

Another structure for the connection between the elements of an arithmetic-memory register is to specify that one input to the arithmetic element is a word contained in its associated memory element. Figure 7 illustrates such a structure.

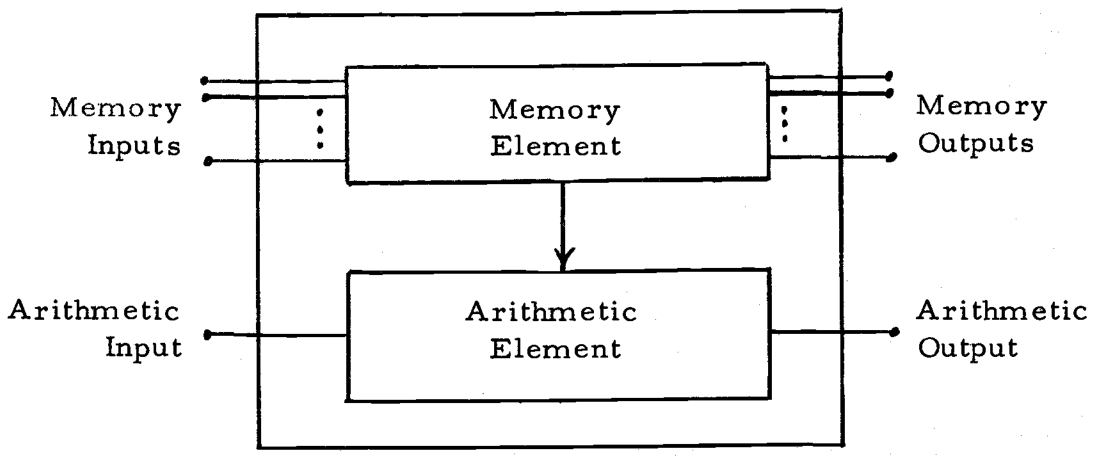


Figure 7. An arithmetic-memory register with dependent elements.

The advantage of this structure is that for single input instructions there is no transfer of information between registers. The arithmetic element has the needed information resident within the memory of the register. Examples of such instructions include the test instructions for greater than, equal to, or less than zero. The shift instruction and logical complementation are also appropriate examples.

A difficulty may occur in determining which word of the associated memory element is the desired input. For this problem an addressing scheme must be devised for each arithmetic-memory register. A control mechanism must also be developed whereby any word of the memory element is capable of being connected to the arithmetic element. Each register therefore must have a control section for selecting the proper address and transmitting its contents to the arithmetic element. This capability introduces a memory cycle time and some form of local programming for each register. A system consisting of a number of these arithmetic-memory registers would perform in the same fashion as an identical number of conventional single accumulator computers linked together under a common structure.

The above case can be altered to a design in which a specific word of the memory element is dedicated as the input to the arithmetic element. This internal structure for the dependent elements of the arithmetic-memory register would be less complicated. However, some means of transferring words within the memory element would be required so that the needed information would be stored at the dedicated location prior to instruction execution.

In any event the latter case is less complicated and demands a simpler internal structure for the arithmetic-memory register. Unlike the independent element configuration, this structure is consistent with the concept of an arithmetic-memory register and it requires only one external input to the arithmetic element.

Size of the Memory Element

If it is assumed that a fixed location or a dedicated word among the n words of the memory element is to be designated as one input to the arithmetic element. Then the utility of the other n - 1 storage locations associated with the memory element is in question.

The residual storage can be utilized in the following ways:

1. For larger problems, instruction coding storage may need to be divided among the residual storage associated with the memory elements of the individual registers. However, in most cases the instruction code and operand addresses are used only for decoding and information transfers respectively. Therefore only trivial manipulations or indexing is performed on the coding and operands of these instructions and there is no real necessity to link this coding with an element containing more powerful arithmetic capabilities.

   This utilization would also tend to destroy program unity and introduce addressing problems since the program would be scattered among the arithmetic-memory registers of the system.

2. The thought of using the residual n - 1 locations for data

storage also deserves consideration. One reasonable method discussed previously was to transfer the contents of one of the residual locations into the dedicated location whenever necessary. This utilization has two drawbacks. It introduces a memory cycle for the location of the resident input to the arithmetic element, and if more than one word within the memory element is needed for instruction execution it hinders the speed of parallel processing.

If it is assumed that simplicity of internal structure and facility in the area of parallel processing are the prime priorities, the most beneficial structure for the concept of arithmetic-memory registers appears to be the utilization of memory elements consisting of only one word.

Structure of an Arithmetic-Memory Register

The final model for a single arithmetic-memory register consists of a memory element and an arithmetic element as illustrated in Figure 8. The arithmetic element has two inputs, a resident input and an external input. The memory element is used as a storage location for a single word of information. This location also acts as the resident input of the arithmetic element. For instructions utilizing only the resident information of the memory element, no external inputs are needed. For instructions performed on two words residing

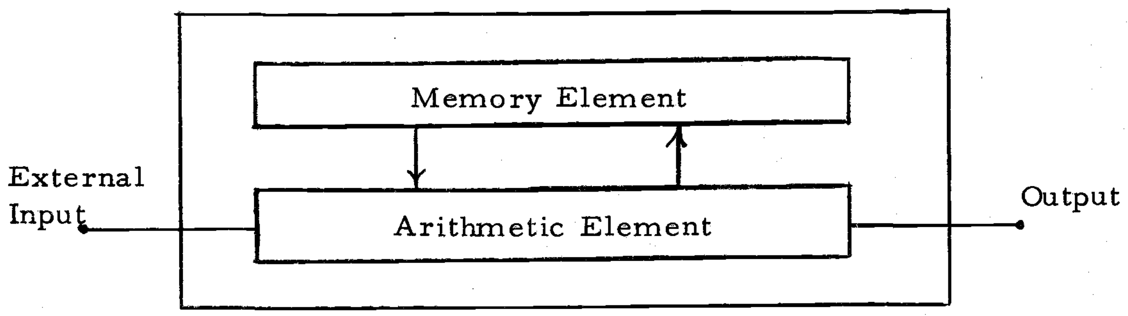


Figure 8. Final model for the structure of an arithmetic-memory register.

in different registers, one of the registers must be selected to perform the operation and the external input to its arithmetic element must come from the memory element of the other register. The result will be stored in the memory element of the operating register.

The final model for an arithmetic-memory register will therefore have one input and one output. To store initial information into the memory element, the load instruction is performed. This instruction will load one word into the memory element through the external input of the arithmetic element. Once the memory element is loaded, it acts as a buffer to the arithmetic element. The external input also has a buffer in the arithmetic element.

To illustrate the utilization of these registers, assume the logical "and" and "or" operations are to be executed on the variables A and B in registers 1 and 2 respectively. Initially the load instruction would be used to store the binary representation of the values A and B into the memory elements of registers 1 and 2. The "and" instruction, associated with an operand designating that the external input is from register 2, is performed by register 1. Simultaneously an "or" instruction, associated with an operand designating that the external input is from register 1, is performed by register 2. The resulting "and" and "or" outputs will therefore be resident in the memory elements of registers 1 and 2 respectively.

Transition States and Instruction States

A computing system composed entirely of arithmetic-memory registers can be described in terms of transition states and instruction states.

Definition 3.6 A transition state for a set of n arithmetic-memory registers is defined as a column vector of n elements representing the interconnection of the registers. The first element of the column vector identifies the external input to the first register. The second element of the vector identifies the external input of the second register, and so forth, until the nth element of the vector identifies the external input of the nth register.

If a binary addressing system is assumed for a computer composed of eight arithmetic-memory registers as illustrated in Figure 9, three binary bits will be required for input addressing. Figure 10

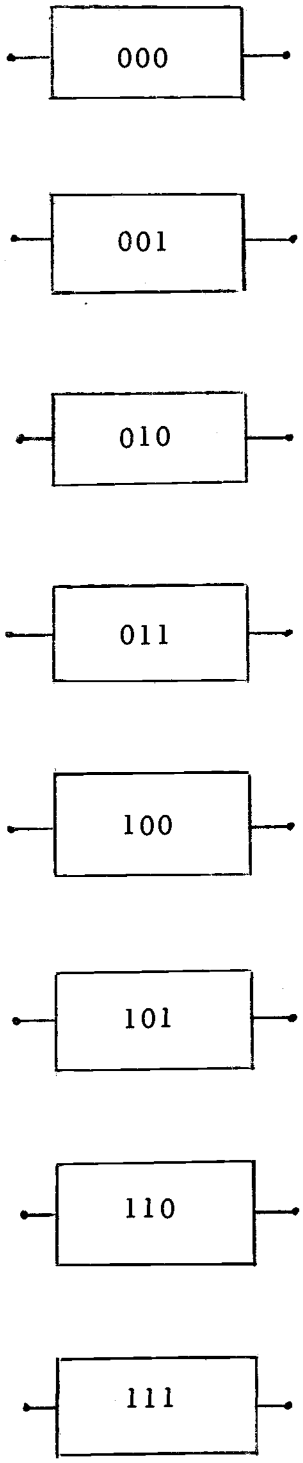


Figure 9. Set of eight arithmetic memory registers.

(111)

(000)

(001)

(010)

(011)

(100)

(101)

(110)

Figure 10. A typical transition state.

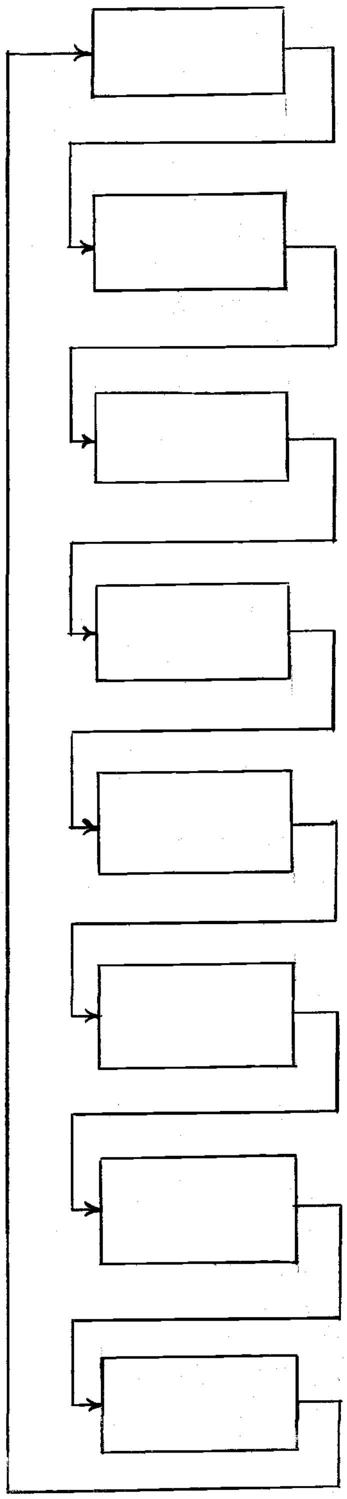


Figure 11. The associated interconnections for the transition state.

illustrates a typical transition state for this system and Figure 11 illustrates the corresponding interconnection of the registers.

This example simply defines the external input to register (n) to be from the memory element of register (n-1) with the output appearing in the memory element of register (n). This transition state can be utilized for any calculations requiring this particular interconnection between registers.

To completely describe the structure of a computing system, an instruction state must also be defined in order to specify the type of operation each register is to perform.

Definition 3.7 An instruction state for a set of n arithmetic-memory registers is defined as a column vector of n elements representing the particular instruction each arithmetic-memory register is to execute. The first element of the column vector identifies the instruction to be executed by the first register. The second element of the vector identifies the instruction to be executed by the second register, and so forth, until the nth element of the vector identifies the instruction to be executed by the nth register.

For a binary operating code of n bits there can be 2^n possible instructions. The 16 instructions in Figure 5 are represented by a four bit binary code varying from (0000) representing the add instruction to (1111) representing the logical complimentation instruction. Figure 12 illustrates a typical instruction state for eight registers using these instructions. The first two registers will perform additions, the second two registers will perform subtractions, the third two registers will perform loading operations, and the last two registers will perform logical complementations.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 12. A typical instruction state for eight arithmetic-memory registers using the instruction set illustrated in Figure 5.

Parallel Cycles

Any parallel operation of the complete set of arithmetic-memory registers is specified by both the transition state and the instruction state. The transition state and instruction state vectors contain an identical number of elements. Each transition state must have an associated instruction state, and each instruction state must have an associated transition state. This restriction does not mean that a particular state cannot possess a null element. The occurrence of a null element within a state simply indicates that the associated register will not be used when the other operations specified by that state are executed.

Definition 3.8 A parallel cycle for a set of n arithmetic-memory registers is defined as any pair of states consisting of one transition state and one instruction state each of whose column vectors contain n elements.

Theorem 3.1 If there are n arithmetic-memory registers associated with a given machine, then there are n! different transition states associated with that machine.

Proof

1. The input to the first register can be obtained from any one of the n registers, therefore, there are n possible choices for its input.
2. Given that the first register's input is selected, the second register has n-1 possible choices for its input.
3. If an arbitrary register such as the pth register is selected, it will have n-(p-1) possible choices for its input.
4. The last or nth register has $n-(n-1) = 1$ possible input.
5. The total number of possible transition states is therefore equal to $(n)(n-1)(n-2)\ldots.(1) = n!$.

Theorem 3.2 If there are n arithmetic-memory registers associated with a given machine and there are m instructions in its instruction set, then there are m^n different instruction states associated with that machine.

Proof

1. The first register can perform one of m instructions.
2. The second register can likewise perform any one of the m instructions.
3. The nth register has m possibilities for its instruction also.
4. The total number of possible instruction states is therefore equal to $(m)(m)\ldots.(m) = \prod_{i=1}^{n} m = m^n$.

Theorem 3.3 If there are n arithmetic-memory registers associated with a given machine and there are m instructions contained in its instruction set, then there are $(n!m^n)$ different parallel cycles associated with that machine.

Proof

1. There are $n!$ different transition states associated with the machine as stated in Theorem 3.1.
2. There are m^n different instruction states associated with the machine as stated in Theorem 3.2.
3. Therefore since each parallel cycle is composed of an instruction state and a transition state, there are $(n!)(m^n)$ possible parallel cycles associated with the machine.

Table 1 is a tabulation of the number of possible parallel cycles in machines with a varied number of arithmetic-memory registers and different sized instruction sets.

Table 1. A tabulation of the number of parallel cycles for a machine with n arithmetic-memory registers and m operations in its instruction set.

	m = 1	m = 2	m = 4
n = 0	1	1	1
1	1	2	4
2	2	8	32
3	6	48	384
4	24	384	6×10^{3}
5	120	3.8×10^{3}	1.2×10^{5}
6	720	4.5×10^{4}	2.9×10^{6}
8	4.0×10^{4}	1.0×10^{7}	2.6×10^{9}
10	3.6×10^{6}	3.6×10^{9}	3.6×10^{12}
20	2.4×10^{18}	2.4×10^{24}	2.4×10^{30}
50	3.0×10^{64}	3.0×10^{79}	3.6×10^{94}
100	9.0×10^{157}	1.0×10^{188}	1.0×10^{218}

n = number of arithmetic-memory registers

m = number of operations in the instruction set

IV. APPLICATION OF THE MODEL

Model Applied to Conventional Computer System

The conventional computer system of Figure 1 can be modeled by transition states and instruction states. The transition state vector and instruction state vector for the conventional system have the same number of elements as the system to be modeled has words of memory. Assume a 1K memory which would correspond to a transition state vector of 1000 elements and an instruction state vector of 1000 elements. Since the conventional system requires that all instructions be executed through a single accumulator, only a single element among the 1000 elements of the instruction state is capable of performing an instruction. The other 999 elements of the instruction state can theoretically perform only no-op commands since every instruction must be performed through the one arithmetic element. If there are m operations in the instruction set, then there are m different instructions which that arithmetic element is capable of executing. Therefore only m different instruction states are associated with the system.

In a similar manner the transition state vector has 1000 elements of which only one is capable of receiving an input address. The number of different transition states is 1000 since the single input can originate from any one of the 1000 memory locations. The product of the number of different transition states and instruction states gives the number of possible parallel cycles. Therefore for this system there are 1000 times m possible parallel cycles.

All conventional systems can be modeled in this manner. The number of possible parallel cycles is equal to the product of the memory size and the number of operations in the instruction set. The limitations of the conventional system are brought out in the model by the fact that only one of the elements of the transition state and instruction state vectors may be utilized. This demonstrates the "bottleneck effect" that can take place in a computing system with a single accumulator and a tremendous amount of memory capability.

Model Applied to SOLOMON and Holland Machines

The application of the transition state and instruction state model to the SOLOMON machine and Holland machine is discussed simultaneously because both machines have neighbor-to-neighbor communication within their processor arrays. The number of elements in the instruction state and transition state is equal to the number of total processors contained in their respective arrays. It is the structure of these arrays that pose the limiting conditions on the parallelism of the system.

The SOLOMON and Holland processor arrays are rectangular in shape. This means that each processor has communication with each of its four adjacent neighbors. This structure is reflected in the positioning of the elements in the transition state vector. Assume that the processors within the rectangular array are ordered in a binary representation such that adjacent processors in the same row differ in at most one bit position, and adjacent processors in the same column differ by at most one bit position. Such a binary representation of these processors is illustrated in Figure 13.

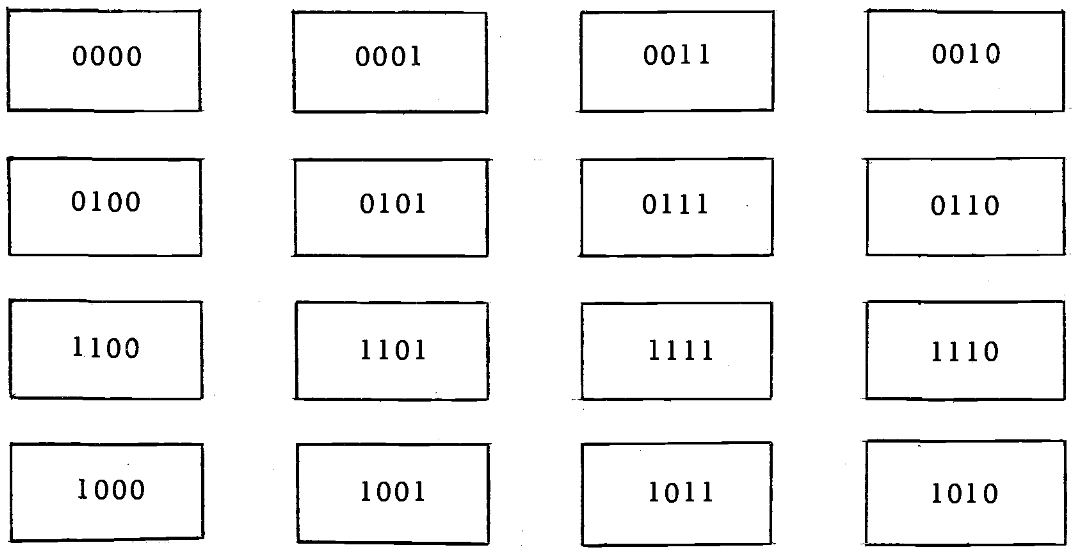


Figure 13. Binary representation of processors in a rectangular array.

The utility of this representation is exhibited in the ease in which neighboring processors can be specified. All neighbors will differ in at most one bit position of their binary representations. For instance, the neighbors of processor (0111) are processors (0011), (0110), (1111) and (0101).

Let a neighboring group for a particular processor be defined as all possible communicating elements with that processor. Each neighboring group will consist of the processor itself plus its four neighbors. For a set of n processors there are n distinct neighboring groups. Each processor will belong to five different neighboring groups. For the case of Figure 13 with n = 16, the 16 neighboring groups are listed in Table 2. The five neighboring groups associated with processor (0000) are 1, 2, 3, 5 and 9.

Table 2. Neighboring groups for the processor array illustrated in Figure 13.

Neighboring Group	Neighboring Processors				
1	0000	0001	0100	0010	1000
2	0001	0011	0101	0000	1001
3	0010	0110	0011	1010	0000
4	0011	0010	0111	0001	1011
5	0100	0000	0101	1100	0110
.					
.					
.					
9	1000	1100	0000	1001	1010
.					
.					
.					
16	1111	0111	1011	1101	1110

Within a neighboring group an initial processor has five possible inputs. The next processor has four possible inputs, and so forth, until the fifth processor of the neighboring group has only one possibility for its input. This makes a total of 5! possible input interconnections within a neighboring group. If this concept is extended to a square array of $n = s^2$ processors with n distinct neighboring groups, the fact that each processor belongs to five separate neighboring groups complicates the generalization. However, an upper bound (B) for the number of possible interconnections for a square array of $n = s^2$ processors is given by the following equation.

$$B = 5^n \qquad \text{Equation 4.1}$$

The concept of limiting the interconnections among processors to only those processors among a neighboring group reflects upon the structure of the transition state vector. Since the elements within the transition state vector are inputs to the arithmetic-memory registers, an upper bound for the number of different transition states for $n = s^2$ processors is identical to the number B given in Equation 4.1. The number of possible instruction states is equal to m^n, where m is the number of operations in the instruction set. Therefore the SOLOMON and Holland structures with $n = s^2$ processors have at most $(B)(m^n)$ possible parallel cycles.

General-Purpose Model

A general-purpose computer consisting of a set of n arithmetic-memory registers and an instruction set of m operations can be modeled in terms of instruction states and transition states. Let the interconnection between these registers be universal, meaning that the output of any register can be connected to the input of any register. Then such a machine structure would be capable of all possible n! transition states.

Table 3 lists the number of transition states for the conventional model, the rectangular model, and the general purpose model for variations in the number of transition state elements.

Table 3. The number of transition states for the conventional system, rectangular array, and general-purpose model for variations in the number of elements in the state.

State Elements	Conventional System	Rectangular Array	General Purpose Model
n = 1	1	1	1
4	4	6	24
9	9	1.9×10^{6}	3.6×10^{5}
16	16	1.5×10^{11}	2.1×10^{13}
25	25	3.0×10^{17}	1.5×10^{25}
36	36	1.5×10^{25}	3.7×10^{41}
49	49	1.8×10^{34}	6.1×10^{62}
64	64	5.4×10^{44}	1.3×10^{89}
81	81	4.1×10^{56}	5.8×10^{120}
100	100	7.9×10^{69}	9.0×10^{157}

The utilization of the number of possible transition states as a measure of the parallel capability of a machine should not be confused with the efficiency in which such machines perform tasks. In other words, the general-purpose model with 4 arithmetic memory registers can not perform 6 times faster or more efficiently than a conventional system with a single accumulator and 4 words of memory. The fact that there are 6 times as many transition states simply indicates that there are 6 times as many ways to manipulate the information. The efficiency of performance for such a system will depend predominately on the application and the programming of the machine. In Chapter 5 the performance and practicality of the general-purpose model will be discussed.

Special-Purpose Model

The concept of arithmetic-memory registers can be extended to special-purpose computers by allowing only a selected number of transition states and limiting the number of operations contained in the instruction set. The allowable number of transition states and the particular interconnections selected will depend upon the application. Clearly it would not be practical to examine all $n!$ possible transition states. However, some examination of the application of certain transition states together with selected instruction sets is of interest.

The "identity" transition state for a set of n arithmetic-memory registers is defined in Equation 4.2.

$$P(i) = i \qquad 0 \leq i \leq n \qquad \text{Equation 4.2}$$

where i is the designation for the register number, and $P(i)$ is the designation for the input to register i.

The "identity" transition is the interconnection of the registers such that each register's output is tied back to its own external input. In reality the memory element contains the result of the last instruction and its contents are merely transferred into the external input of the arithmetic element. Figure 14 illustrates the interconnection of the registers for this transition.

The "identity" transition is obviously useful for calculations and

$$\begin{tikzcd}
0 \arrow[r] & 0 \\
1 \arrow[r] & 1 \\
2 \arrow[r] & 2 \\
3 \arrow[r] & 3 \\
\vdots & \vdots \\
n \arrow[r] & n
\end{tikzcd}$$

Figure 14. Interconnection of registers for the "identity" transition.

logical operations utilizing only the contents of the register's memory element. Operations such as shifting and complementation can be executed using this transition. If the multiply instruction is used, the square of the memory contents can be obtained. In binary representation, shifting each register's contents right or left n positions is equivalent to dividing or multiplying the contents of each memory element by 2^n.

The "half-cross" transition for a set of n arithmetic-memory registers is defined in Equation 4.3.

$$\begin{aligned} P(i) &= i + n/2 \qquad 1 < n/2 \\ &= i - n/2 \qquad 1 \geq n/2 \end{aligned} \qquad \text{Equation 4.3}$$

The interconnection of the registers for this transition is illustrated in Figure 15. This transition is useful if a set of calculated

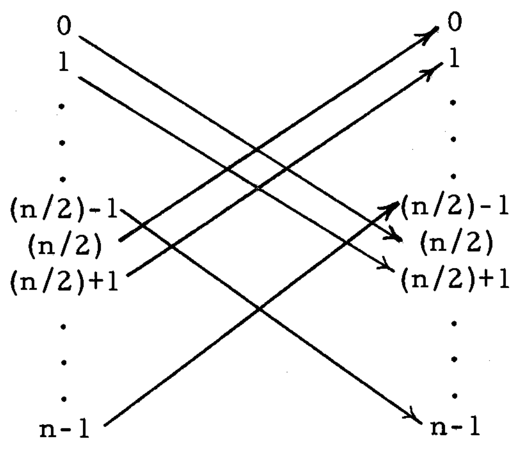


Figure 15. Interconnection of registers for the "half cross" transition.

values needs to be stored for use in preceding parallel cycles. If the first half or upper half registers contain the appropriate values, the "half-cross" transition could then be used with the load instruction to store the values of the upper half registers into the lower half registers. This infers that the instruction set contains no-op instructions for the lower half registers so that their contents are not transferred into the upper registers.

If the above load instruction for the "half-cross" transition is followed by a sequence of m multiply instructions, the lower half registers would contain the contents of the upper half registers raised to the mth power.

The "pairing" transition for a set of n arithmetic-memory registers is defined in Equation 4.4.

$$P(i) = \begin{cases} i + 1 & \text{if } i \text{ is even} \\ i - 1 & \text{if } i \text{ is odd} \end{cases} \qquad \text{Equation 4.4}$$

The interconnection of this transition state is illustrated in Figure 16. This transition is useful in adding or multiplying a set of numbers. Since a single arithmetic-memory register is limited to operating on only two words at one time, it is sometimes convenient to operate on adjacent words. Redundant values can also be obtained using this transition by utilizing the same instruction for the pair of registers.

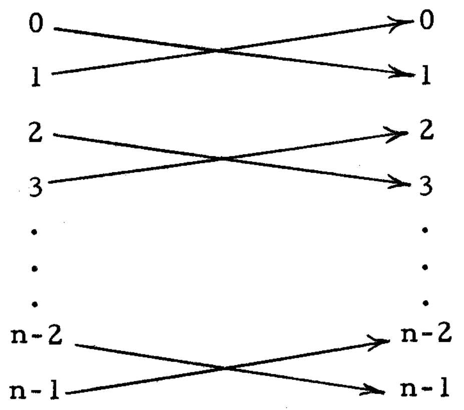


Figure 16. Interconnection of the registers for the "pairing" transition.

The "shift one" transition for a set of n arithmetic-memory registers is defined in Equation 4.5.

$$
\begin{aligned}
P(i) &= i + 1 \qquad & i \neq 0 \\
&= n - 1 \qquad & i = 0
\end{aligned}
\qquad \text{Equation 4.5}
$$

The interconnection for this transition is illustrated in Figure 17. This transition can be used for any calculations requiring delays or shifts in a set of numbers. A typical application would be in the area of correlation calculations.

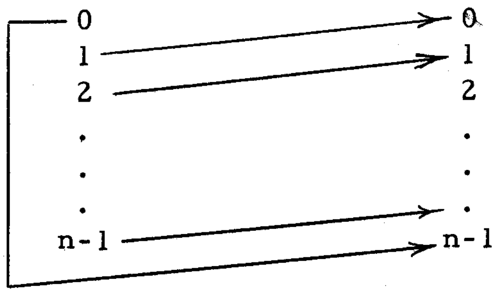


Figure 17. Interconnection of the registers for the "shift one" transition.

The "evens up and odds down" transition for a set of n arithmetic-memory registers is defined in Equation 4.6.

$$P(i) = \begin{cases} i/2 & \text{if } i \text{ is even} \\ (n+i-1)/2 & \text{if } i \text{ is odd} \end{cases} \qquad \text{Equation 4.6}$$

The interconnection for this transition is illustrated in Figure 18. This transition can be used after the "pairing" transition to place results back into adjacent registers for further applications of the "pairing" transition.

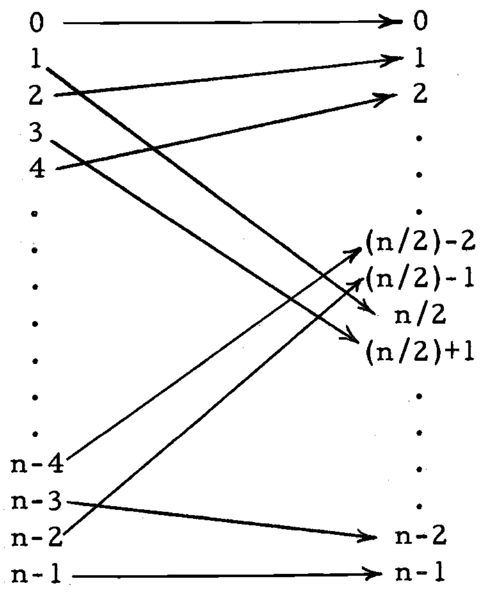


Figure 18. Interconnection of the registers for the "evens up and odds down" transition.

The "perfect shuffle" transition for a set of n arithmetic-memory registers is defined in Equation 4.7.

$$P(i) = \begin{cases} 2i & i \leq (n/2)-1 \\ 2i+1-n & i \geq (n/2) \end{cases} \qquad \text{Equation 4.7}$$

The interconnection for this transition is illustrated in Figure 19. This transition was initially discovered by Pease (1968) and later applied by Stone (1971) in the field of transform theory.

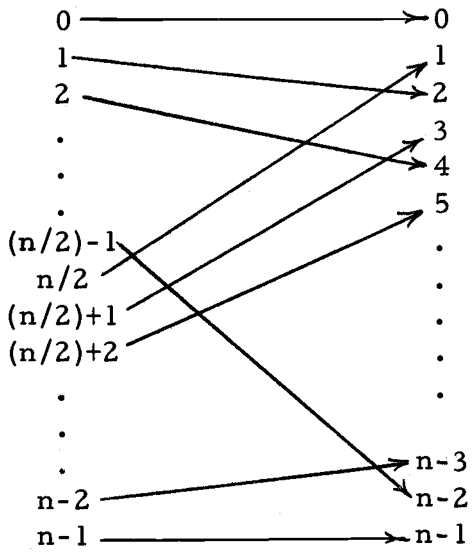


Figure 19. Interconnection of the registers for the "perfect shuffle" transition.

V. GENERAL-PURPOSE SYSTEM

A general-purpose computing system is a machine which has the capability of performing a wide range of tasks. The efficiency of these systems is usually quite moderate in comparison to special-purpose machines designed for a specific purpose. Traditionally the moderate performance of general-purpose systems has not been questioned from a hardware-software viewpoint. Software algorithms have been developed to perform tasks in an optimal fashion, and hardware technology has made tremendous developments; however, advances in both areas have been independent for the most part. The following section is an examination of the general-purpose system for a hardware-software viewpoint.

Hardware-Software Interdependence

Let a general-purpose computer system consisting of n arithmetic-memory registers be structured in such a way that the output of any register can be connected to the input of any other register. In other words, there exist a universal set of interconnections between the registers.

This structure represents a hardware development in that each word of memory is associated with a dedicated processor. The conventional approach is for a group of words to share a single processor.

The software development rests in the ability of the programmer to utilize this processor-word pair as efficiently as possible.

The hardware-software interdependence is demonstrated by the fact that the programmer is solely responsible for the selection of the interconnections that are going to ultimately solve his problem. In most multiprocessing systems the selection of a particular processor to perform a given task is under system's control. The utilization of the transition state and instruction state model for a general-purpose system composed of arithmetic-memory registers allows the programmer to control processor operations.

Intuitively, since parallel operations must be independent, the element controlling the selection of processors must have some means of determining independent operations. Because the programmer determines the dependence and independence of tasks by the very nature of his program, he must have primary control over processor selection if parallel processing is to extend to its limit.

Example

Assume that a conventional system is to perform the task of computing Z and W by means of the following equations.

$$Z = (B+C) * (D+F)$$

$$W = (B*C) + (D*F)$$

The normal sequence of instructions for the conventional system would be as follows:

Program Z = (B+C) * (D+F)

	Instruction		Operation
1.	LDA	B	Loads B into the accumulator
2.	ADA	C	Adds C to B
3.	STA	X	Stores (B+C) into X
4.	LDA	D	Loads D into the accumulator
5.	ADA	F	Adds F to D
6.	M	X	Multiplies (F+D) and (B+C)
7.	STA	Z	Stores the result in Z

Program W = (B*C) + (D*F)

8.	LDA	B	Loads B into accumulator
9.	M	C	Multiplies B and C
10.	STA	X	Stores (B*C) in X
11.	LDA	D	Loads D into the accumulator
12.	M	F	Multiplies D and F
13.	ADA	X	Adds (B*C) to (D*F)
14.	STA	W	Stores the result in W

This program requires 14 memory cycles to fetch the instructions plus 14 memory cycles to fetch the operands. This corresponds to a total of 28 memory cycles.

If the system were composed of four arithmetic-memory registers with the values B, C, D and F residing in registers 00, 01, 10 and 11 respectively, the transition and instruction states illustrated in Figure 20 can be utilized to calculate Z and W.

Register

$$\begin{array}{l} B \longrightarrow 00 \\ C \longrightarrow 01 \\ D \longrightarrow 10 \\ F \longrightarrow 11 \end{array}$$

Transition States

$$\begin{array}{cc} (1) & (2) \\ \begin{bmatrix} 01 \\ 00 \\ 11 \\ 10 \end{bmatrix} & \begin{bmatrix} 10 \\ 11 \\ \xi \\ \xi \end{bmatrix} \end{array}$$

Instruction States

$$\begin{array}{cc} (1) & (2) \\ \begin{bmatrix} + \\ * \\ + \\ * \end{bmatrix} & \begin{bmatrix} * \\ + \\ \xi \\ \xi \end{bmatrix} \end{array}$$

Figure 20. Transition and instruction states for computing $Z = (B+C) * (F+D)$ and $W = (B*C) + (F*D)$.

The task requires 2 parallel cycles, the first cycle computes the values (B+C) in register 00, (C*B) in register 01, (D+F) in register 10, and (F*D) in register 11. The second cycle computes the values (B+C) * (D+F) in register 00 and (C*B) + (F*D) in register 11. The ξ notation corresponds to a no-op command in the instruction state and a blank input in the transition state. It can be shown that such a system is optimal for the given task.

Feasibility of the System

For systems limited to operations involving two inputs, such a general-purpose system would extend parallel processing to its limit. However, if such a system were implemented for n arithmetic-memory registers, some technique for handling the n! distinct transition states would have to be developed.

A straight forward wired-in approach would provide each register with dedicated inputs from the other n-1 registers. Practical values of word size range from 8 to 64 input bit positions for each register. In order that the proper input word be selected for each register, a decoding network would be required to distinguish between the n! different transition states. For a system consisting of 100 registers, such a network would have to distinguish between approximately 10^{158} different transition states so that the proper interconnections between the registers could be provided. The implementation of the decoding network together with the hardware required to control such a tremendous number of interconnections would make this structure unfeasible.

It should be noted that the cost of the registers has not affected the feasibility of this system. The pitfall for such a structure rests in the implementation of the control and interconnections between the registers.

VI. SPECIAL-PURPOSE SYSTEM

Digital Spectral Analysis

Spectral estimates derived from finite length time functions involve a considerable amount of computational effort and permit a relatively high degree of parallelism in their computations. By utilizing these characteristics, special-purpose digital computers can calculate these estimates at much greater speeds than conventional general-purpose machines. The following discussion concerns the application of arithmetic-memory registers in the design of a special-purpose computer for the calculation of power spectra.

Basic Procedure

The Fourier transform X(f) of a signal x(t) in the time domain represents the distribution of the signal strength in the frequency domain. The continuous Fourier transform is defined by Equation 6.1.

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-i2\pi ft} dt, \qquad \text{Equation 6.1}$$

where $i = \sqrt{-1}$, $-\infty < t < \infty$, $-\infty < f < \infty$.

The power spectral density P(f) is defined in terms of the transform in Equation 6.2.

$$P(f) = \lim_{T \to \infty} 1/T \, | X(f) |^2, \qquad \text{Equation 6.2}$$

where T is the period of the signal x(t).

There are two procedures which yield the factor $| X(f) |^2$. These procedures are illustrated in Figure 21. The first procedure involves computing the autocorrelation function R(T) directly using Equation 6.3, and then Fourier transforming the autocorrelation function to obtain the power spectral density.

$$R(\tau) = \lim_{T \to \infty} \frac{1}{T} \int_{-T/2}^{T/2} x(t) \; x(t-\tau) \, dt, \qquad \text{Equation 6.3}$$

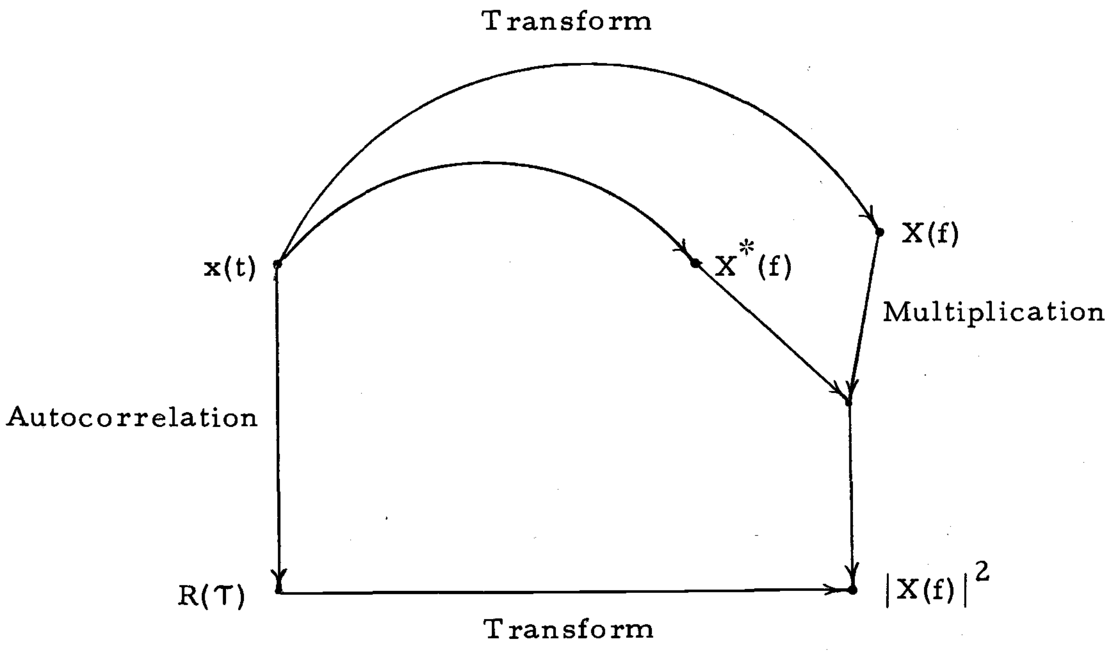


Figure 21. Two procedures for calculating the power density spectrum.

The second procedure involves computing the transform $X(f)$ plus its complex conjugate $X^*(f)$ and performing an additional multiplication to obtain the $|X(f)|^2$ term. If desired the autocorrelation term can also be obtained by taking the inverse transform of $|X(f)|^2$.

Discrete Versus Continuous

The continuous transform is defined in Equation 6.1. However, if the time varying function $x(t)$ is sampled every Δt seconds such that n discrete samples are available, then the discrete Fourier transform and autocorrelation function are given by Equations 6.4 and 6.5 respectively.

$$X(j) = \frac{1}{n} \sum_{k=0}^{n-1} x(k)\, e^{-(2\pi ijk)/n}, \qquad \text{Equation 6.4}$$

where $j = 0, 1, \ldots, n-1$ and $n = T/\Delta t$.

$$R(v) = \frac{1}{n} \sum_{k=v}^{n-1} x(k)\; x(k+v), \qquad \text{Equation 6.5}$$

where $v = 0, 1, \ldots, n-1$.

If the expression $W_n = e^{-2\pi i/n}$ is substituted into Equation 6.4, then the discrete Fourier transform takes the form of Equation 6.6.

$$X(j) = \frac{1}{n} \sum_{k=0}^{n-1} x(k)\; W_n^{-jk} \qquad \text{Equation 6.6}$$

The fast Fourier transform algorithm derived by Cooley and Tukey (1965) requires that the number of sample points be a power of 2, such that $n = 2^m$. The following notation can be utilized to derive an algorithm for computing Equation 6.6.

Assume $j = 0, 1, \ldots, n-1$ and $k = 0, 1, \ldots, n-1$.

Let $j_{m-1}, j_{m-2}, \ldots, j_0 = 0, 1$ and $k_{m-1}, k_{m-2}, \ldots, k_0 = 0, 1$,

then $$j = j_{m-1}2^{m-1} + j_{m-2}2^{m-2} + \ldots + j_1 2 + j_0$$

and $$k = k_{m-1}2^{m-1} + k_{m-2}2^{m-2} + \ldots + k_1 2 + k_0.$$

Using this representation for j and k, Equation 6.6 becomes

$$X(j) = \frac{1}{n} \sum_{k_0=0}^{1} \sum_{k_1=0}^{1} \cdots \sum_{k_{m-1}=0}^{1} x(k_{m-1}, k_{m-2}, \ldots, k_0) \qquad \text{Equation 6.7}$$

$$W_n^{(j_{m-1}2^{m-1}+j_{m-2}2^{m-2}+\ldots+j_0)(k_{m-1}2^{m-1}+k_{m-2}2^{m-2}+\ldots+k_0)}$$

The above equation can be computed from the following set of equations.

$$A_0(k_{m-1}, k_{m-2}, \ldots, k_0) = x(k_{m-1}, k_{m-2}, \ldots, k_0)$$

$$A_1(j_0, k_{m-2}, \ldots\ldots, k_0) = \sum_{k_{m-1}=0}^{1} A_0(k_{m-1}, k_{m-2}, \ldots, k_0) W_n^{j_0 k_{m-1} 2^{m-1}}$$

$$A_2(j_0, j_1, k_{m-3}, \ldots, k_0) = \sum_{k_{m-2}=0}^{1} A_1(j_0, k_{m-2}, \ldots, k_0) W_n^{(j_1 2 + j_0) k_{m-2} 2^{m-2}}$$

$$\vdots$$

$$A_p(j_0, j_1, \ldots, j_{p-1}, k_{m-p-1}, k_{m-p-2}, \ldots, k_0) =$$

$$\sum_{k_{m-p}=0}^{1} A_{p-1}(j_0, \ldots, j_{p-2}, k_{m-p}, \ldots, k_0)\, W_n^{(j_{p-1} 2^{p-1} + \ldots + j_0) k_{m-p} 2^{m-p}}$$

$$\vdots$$

$$A_m(j_0, j_1, \ldots, j_{m-1}) = \sum_{k_0=0}^{1} A_{m-1}(j_0, j_1, \ldots, j_{m-2}, k_0) W_n^{(j_{m-1} 2^{m-1} + \ldots + j_0) k_0}$$

$$X(j_{m-1}, j_{m-2}, \ldots, j_0) = A_m(j_0, j_1, \ldots, j_{m-1})$$

The fast Fourier Transform algorithm applied to a series of n points, where $n = 2^m$, requires m iterations of the algorithm for evaluating the $A_0, A_1, \ldots, A_m$ terms. This procedure is followed by a bit reversal on the last A_m term to arrange the transform components in the proper order. Figure 22 illustrates the use of the algorithm for the case where $n = 8$. Each node is a sum of two terms and the number adjacent to a path is the power of W_n which is to be multiplied with the associated term prior to summation. For instance, $A_1(000) = A_0(000)\, W^0 + A_0(100)\, W^0$ and $A_2(101) = A_1(101)\, W^0 + A_1(111)\, W^2$.

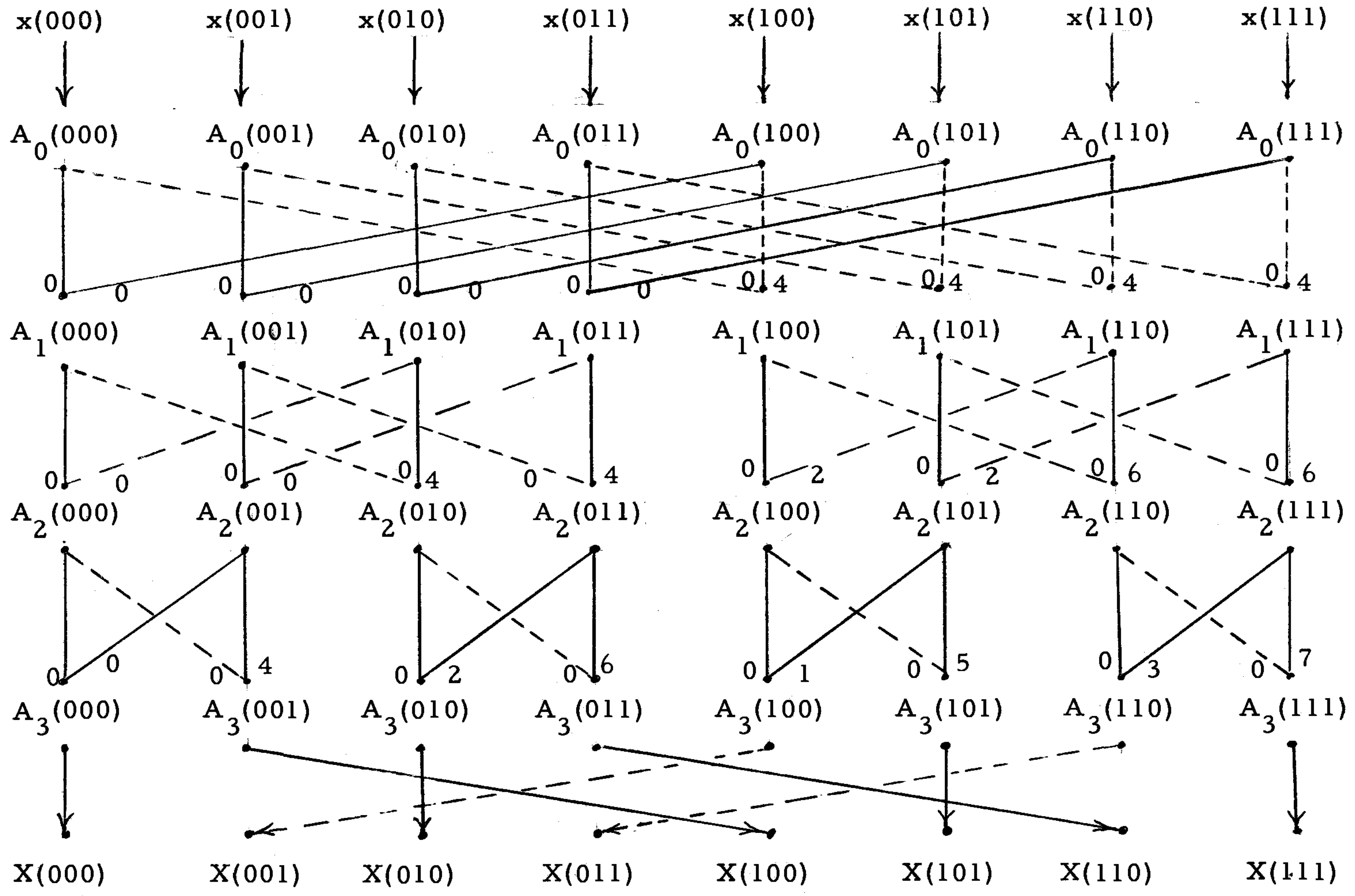


Figure 22. Flow diagram of the fast-Fourier transform algorithm for the case N = 8.

Application to Arithmetic-Memory Registers

In order to calculate the estimates of a power spectrum, a special-purpose system composed of arithmetic-memory registers must have the following capabilities:

1. An addition routine for calculating the sum of n numbers.
2. An autocorrelation routine.
3. A fast-Fourier transform routine.

These three routines will require certain interconnections between the arithmetic-memory registers. In the case of the addition and autocorrelation routines the necessary interconnections are easily determined. However, in the case of the fast-Fourier transform routine, the necessary interconnections between the registers are not readily apparent. It will be shown that a set of workable interconnections for this routine can be determined through an examination of the required transition states.

Addition Routine

The "identity," "pairing," and "evens up and odds down" transitions discussed in Chapter 4 can be combined to form a routine for the addition of n numbers. A flow diagram for this routine is illustrated in Figure 23 for the case where $n=8$. During the time step 1, the "identity" transition is used to load the registers from the

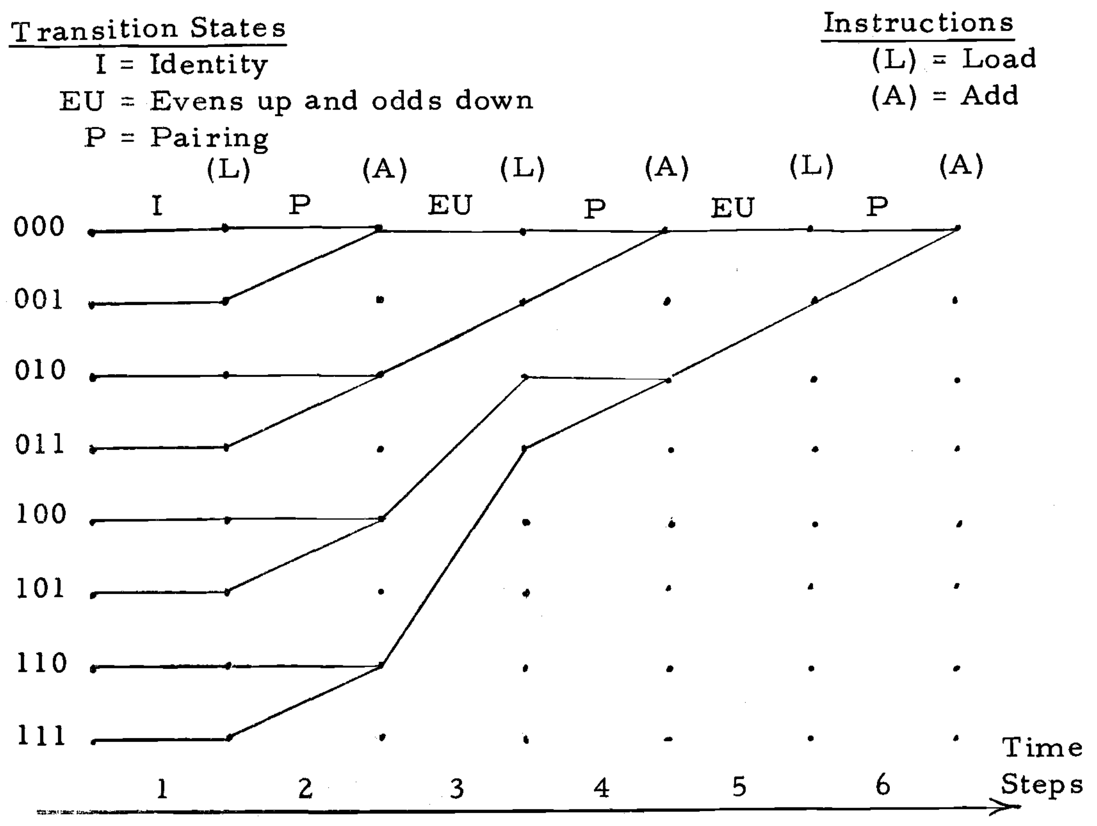


Figure 23. Flow diagram for the add routine.

input device. During time-step 2, the "pairing" transition is used to add adjacent pairs. The third time-step is used to load the resulting pair summations into the upper registers through the "evens up and odds down" transition. The latter two transitions are then alternately applied using the add and load instructions until the result of the summation is contained in the top register.

If the addition routine were generalized for n numbers, it would require 2 ($\log_2 n$) parallel cycles for the summation. This is twice the lower bound ($\log_2 n$) for the number of parallel cycles necessary to complete the summation. The performance of this special-purpose machine is indeed inferior to the previously mentioned general-purpose machine which is capable of achieving the lower bound for the summation calculation. However, it should be noted that only three different transition states were required for the special-purpose system while the general-purpose system requires $\log_2 n$ different transition states for the calculation.

Autocorrelation Routine

The "identity", "half-cross" and "shift one" transitions can be combined with the addition routine to calculate the autocorrelation function. This routine requires twice as many registers as points to be calculated because the original signal values are needed for preceding calculations and therefore must be stored in registers separate from the operating registers. A flow diagram for this routine for the case of $n = 8$ is illustrated in Figure 24. The sequence of transitions and commands which are performed during the various time-steps are explained as follows:

Time Step 1. The eight signal samples are loaded into the upper eight registers by means of the "identity" transition.

2. The "half-cross" transition is used to load the values into the eight lower registers.

3. The "half-cross" transition is again utilized to multiply the upper and lower registers.

4 to 10. The addition routine is utilized on the upper half registers yielding the first autocorrelation value corresponding to a shift of zero.

11. The upper registers are loaded once again with the original values by means of the "half-cross" transition.

12. The "shift one" transition is used to shift the values in the proper order.

13. The "half cross" transition is used to form the appropriate products.

14 to 20. The addition routine is used to sum the products for the second term of the autocorrelation function corresponding to a shift of one.

20 to 80. This cycle is repeated until the autocorrelation term corresponding to a shift of eight is computed.

If the autocorrelation routine were generalized for n numbers, it would require $n(3+2\log_2 n)$ parallel cycles for the calculation. The multiplication operation would be required in n parallel cycles and the add routine would account for $n(2\log_2 n)$ of these cycles. Five different transition states would be required for the routine.

<u>Transitions</u>
I = Identity
HC = Half-cross
SO = Shift One

<u>Instructions</u>
(L) = Load instruction
(M) = Multiply instruction

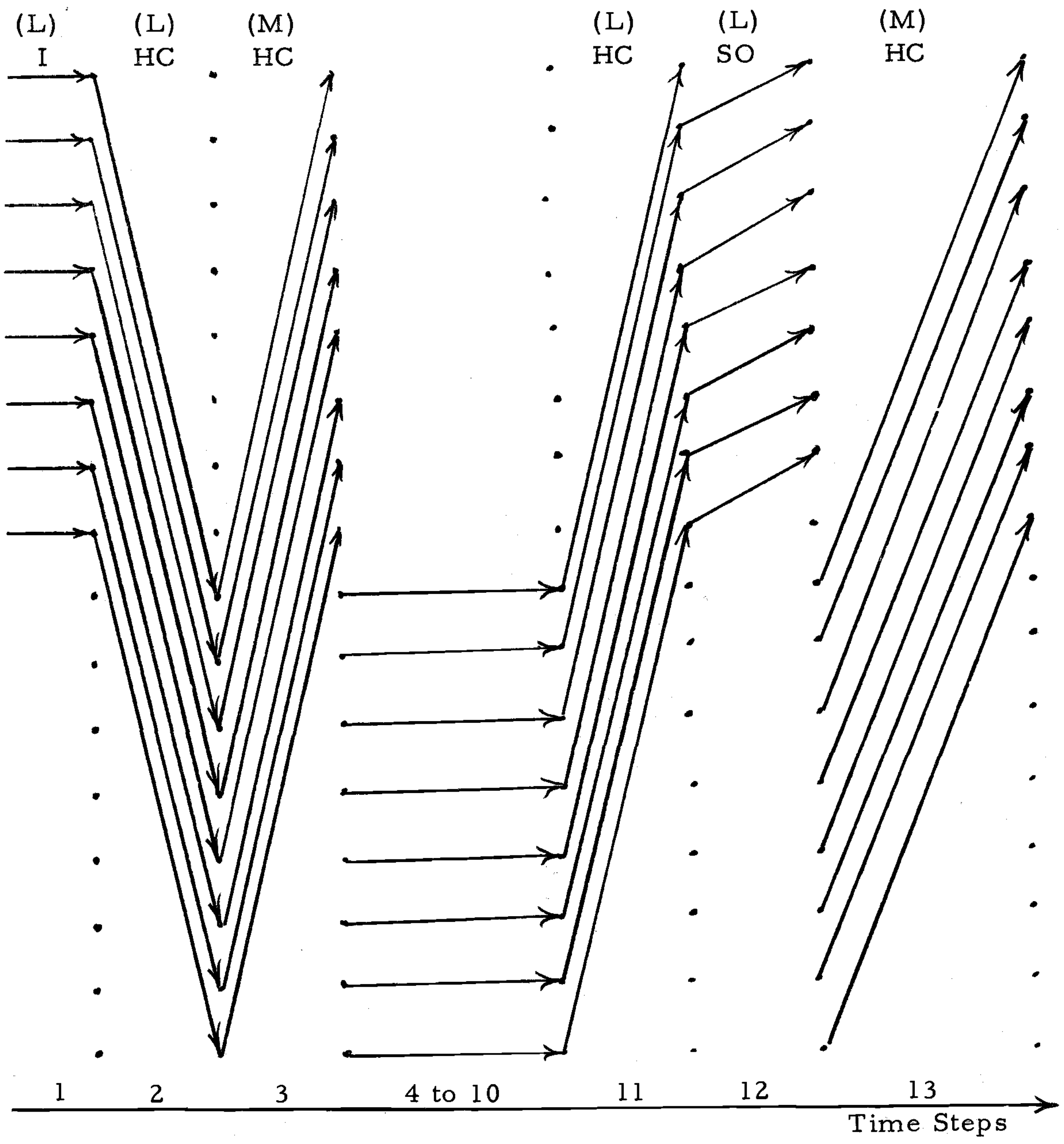


Figure 24. Flow diagram for calculating autocorrelation function.

Fast-Fourier Transform Routine

The arithmetic-memory register interconnections required to calculate the fast-Fourier transform are not readily apparent. However, it will be shown that the utilization of the transition state and instruction state model will lead to a workable set of interconnections for a system of arithmetic-memory registers.

The transition states necessary for the implementation of the algorithm for an eight-point transform are given in Figure 25.

Register	Transition State 1	Transition State 2	Transition State 3
000	100	010	001
001	101	011	000
010	110	000	011
011	111	001	010
100	000	110	101
101	001	111	100
110	010	100	111
111	011	101	110

Figure 25. The required transition states for an eight-point fast-Fourier transform.

Transition state 1 is required to compute the A_1 terms appearing in the flow diagram of Figure 22. Transition states 2 and 3 are required for the A_2 and A_3 terms respectively.

Let $(X_3X_2X_1)$ be the binary representation for a given register, where X_1, X_2 and X_3 are binary variables.

Let $(Y_3Y_2Y_1)$ be the binary representation for the input to register $(X_3X_2X_1)$.

For transition state 1,

$$\left.\begin{array}{lll} Y_3 = 0 & \text{if} & X_3 = 1 \\ Y_3 = 1 & \text{if} & X_3 = 0 \end{array}\right. \text{, thus } Y_3 = \overline{X}_3,$$

$$\left.\begin{array}{lll} Y_2 = 0 & \text{if} & X_2 = 0 \\ Y_2 = 1 & \text{if} & X_2 = 1 \end{array}\right. \text{, thus } Y_2 = X_2,$$

$$\left.\begin{array}{lll} Y_1 = 0 & \text{if} & X_1 = 0 \\ Y_1 = 1 & \text{if} & X_1 = 1 \end{array}\right. \text{, thus } Y_1 = X_1.$$

Therefore $\quad (Y_3Y_2Y_1) = (\overline{X}_3X_2X_1).$

In other words, transition state 1 is merely the interconnection of the registers such that the most significant bit is complemented in the binary representation of each register. By inspection of Figure 15 in Chapter 4, it can be shown that transition state 1 is identical to the "half-cross" transition for eight registers.

In a similar manner, transition state 2 can be represented by the following equation,

$$(Y_3Y_2Y_1) = (X_3\overline{X}_2X_1).$$

Transition state 3 can be represented by the equation,

$$(Y_3 Y_2 Y_1) = (X_3 X_2 \overline{X}_1).$$

If the analysis were generalized for the case $n = 2^m$ points, the binary representation for each register would be of the following form

$$(X_m X_{m-1} \ldots X_1).$$

The fast-Fourier transform algorithm would require m distinct transition states defined by the following equations.

Transition State	Equation
1.	$(Y_m Y_{m-1} \ldots Y_1) = (\overline{X}_m X_{m-1} \ldots X_1)$
2.	$(Y_m Y_{m-1} \ldots Y_1) = (X_m \overline{X}_{m-1} \ldots X_1)$
.	
.	
.	
m.	$(Y_m Y_{m-1} \ldots Y_1) = (X_m X_{m-1} \ldots \overline{X}_1)$

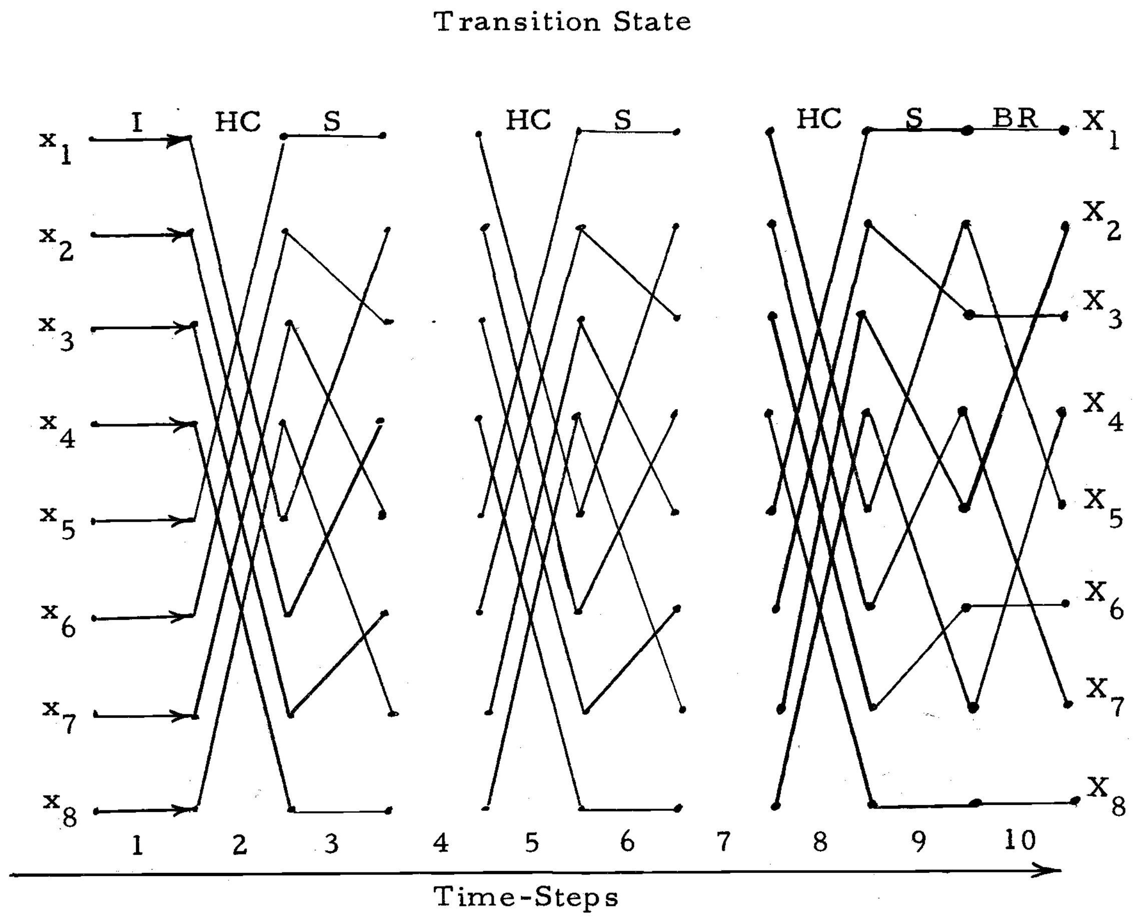
For large values of n, the number of transitions required may become impractical. Therefore some method of manipulating the transition states in order to reduce the required number of interconnections would be desirable. One common characteristic among the m required transition states is the presence of the complementation operation. If the transition state which complements the most significant bit position is combined with a transition state which shifts the binary representation one bit position to the right, all m transition states could be attained. This results in a significant reduction in the number of required interconnections.

By inspection of Figure 19 in Chapter 4, it can be shown that the "perfect shuffle" transition is identical to the transition required to shift the binary representation one bit position to the right.

The fast-Fourier transform can be computed using the "half-cross" and "perfect shuffle" transitions. Figure 26 illustrates the complete operation for an eight-point transform. The flow diagram for this operation is explained as follows:

Time step 1. The time domain samples are loaded into the registers. These values correspond to the A_0 terms.

2. The "half-cross" transition is used to calculate the A_1 terms.
3. The "perfect shuffle" is used to shift the A_1 terms one bit position to the right.
4. The weights W^{jk} which are resident in other registers are multiplied with the A_1 terms.
5. The "half-cross" transition is utilized to calculate the A_2 terms.
6. The "perfect shuffle" arranges the A_2 terms in the proper order.
7. Weights W^{jk} are multiplied with the A_2 terms.



Transition States

I =	Identity
HC =	Half-Cross
S =	Perfect Shuffle[1]
BR =	Binary Reversal

Figure 26. Flow diagram for the calculation of the fast-Fourier transform.

[1]It should be noted that the "perfect shuffle" transition was originally derived by Pease (1968) and developed by Stone (1971). The technique of utilizing the transition state to derive a workable set of interconnections is the point stressed by the author. This technique is considerably simpler and more intuitive than the complex technique used by Pease.

8. The "half-cross" transition is used to calculate the A_3 terms.
9. The "perfect shuffle" arranges the A_3 terms in the proper order.
10. The transition to arrange the A_3 terms in reverse order is utilized to align the spectral estimates.

The transitions and instructions used to derive the weights are given in Appendix A. If the given fast-Fourier transform were generalized for $n = 2^m$ points, there would be a total of $3 \log_2 n$ parallel cycles required for the transform. The multiplication instruction would be utilized in $\log_2 n$ cycles, the add instruction would be utilized in $\log_2 n$ cycles, and the remaining $\log_2 n$ cycles would involve the load command.

System Performance

The performance of the special-purpose system can be compared with a conventional computer system utilized to calculate the same spectral estimates. Figure 27 gives the type of instructions required and the number of times these instructions must be executed for the general case of $n = 2^m$ points. Using these figures the performance of the special-purpose system can be estimated with respect to the conventional system.

		Conventional System	Special-Purpose System
Autocorrelation Function	Multiply	n^2	n
	Add	n^2	$n(\log_2 n)$
	Load	n^2	$2n + n(\log_2 n)$
	Store	n^2	
Fast-Fourier Transform	Multiply	$n(\log_2 n)$	$\log_2 n$
	Add	$n(\log_2 n)$	$\log_2 n$
	Load	$n(\log_2 n)$	$\log_2 n$
	Store	$n(\log_2 n)$	0
Total	Multiply	$n^2 + n(\log_2 n)$	
	Add	$n^2 + n(\log_2 n)$	
	Load	$n^2 + n(\log_2 n)$	$2n + (n+1)\log_2 n$
	Store	$n^2 + n(\log_2 n)$	0

Figure 27. Type and number of instructions used in the calculation of spectral estimates.

Since the execution of the multiplication command requires the greatest amount of execution time, its frequency of occurrence will have a significant effect on the performance of the system. The conventional system requires n times as many multiplications as the special purpose system. For even relatively small values of n, the corresponding reduction in the number of multiplications required of the special-purpose system could represent a significant increase in speed.

The special-purpose system also requires approximately n^2 fewer additions and $n(n-2)$ fewer load instructions. Another factor affecting the relative speeds in which these system operate is the fact that the transition states of the arithmetic-memory registers are completely wired interconnections. This means that access times in the order of 100 nsec. are appropriate. The conventional system requires a memory cycle in the μsec. range for each instruction executed. Therefore the special-purpose system from its very structure offers an order of magnitude increase in speed.

The true performance of the special-purpose system can be illustrated by comparing the times required for computing the spectral estimates for various length data records. Typical instruction execution times taken from an EAI 680/640 hybrid computing system which is frequently utilized for spectral analysis are as follows:

Load	3.3 μs
Store	3.3 μs
Add	3.3 μs
Multiply	18.5 μs

The total execution times (D_n) required to compute n-point spectral estimates are listed in Table 4 for the conventional and special-purpose systems assuming the above instruction execution

Table 4. Tabulation of execution times D_n for the conventional and special-purpose systems.

	Conventional System	Special-Purpose System
	D_n (μsec.)	D_n (μsec.)
n = 2	168	120
4	675	293
8	2,440	660
16	9,000	1,470
32	33,500	3,265
64	126,000	7,260
128	485,000	16,000
256	1,900,000	34,900
512	7,500,000	76,800
1,024	29,500,000	167,600
2,048	119,000,000	212,000

D_n = Time required to compute n-point spectral estimates.

times. A record containing 1024 data points would require approximately 29.5 seconds on the conventional system as compared to less than one second execution time on the special-purpose system.

Figure 28 is a plot of the reduction ratio for the execution time associated with the special-purpose system relative to the conventional system.

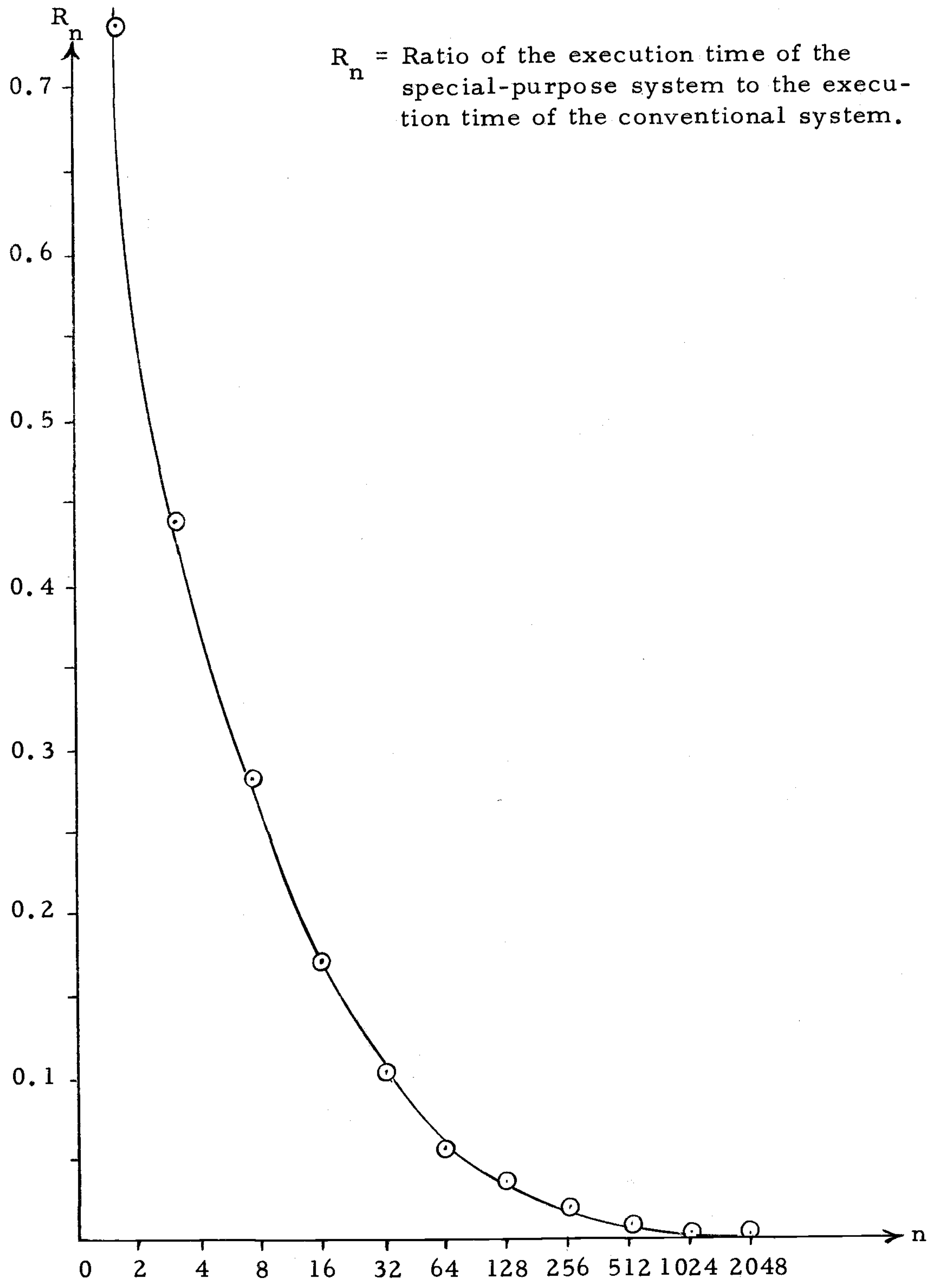


Figure 28. Plot of the reduction ratio R_n for the execution times given in Table 4.

Cost and Execution Time

Assume the cost of the special-purpose system is directly proportional to the number of registers which it contains. Let C denote the cost constant. For analyzing n points, the system requires 2n arithmetic memory registers. Therefore the total cost for the system is approximately equal to 2nC.

For the conventional system assume that computing cost is set at X dollars per hour. Then the conventional system would have to operate for (2nC)/X hours in order to match the cost of the special-purpose system. Let G designate the number of signals which can be analyzed by the conventional system before the cost of the special-purpose system is matched. The number G can be expressed by the relationship

$$G = (3600)\ (2nC)/(X)\ (D_n)$$

where D_n is the time required to analyze n-point estimates for the conventional system. A plot of G versus C is illustrated in Figure 29 for variations in X.

Assume a worst case condition where the conventional system's time is priced at \$100/hr. and the arithmetic-memory registers are priced at \$500 each. Under such conditions the conventional system would have to analyze 1,250,000 signals consisting of 1024 points each to match the approximate cost of the special-purpose system.

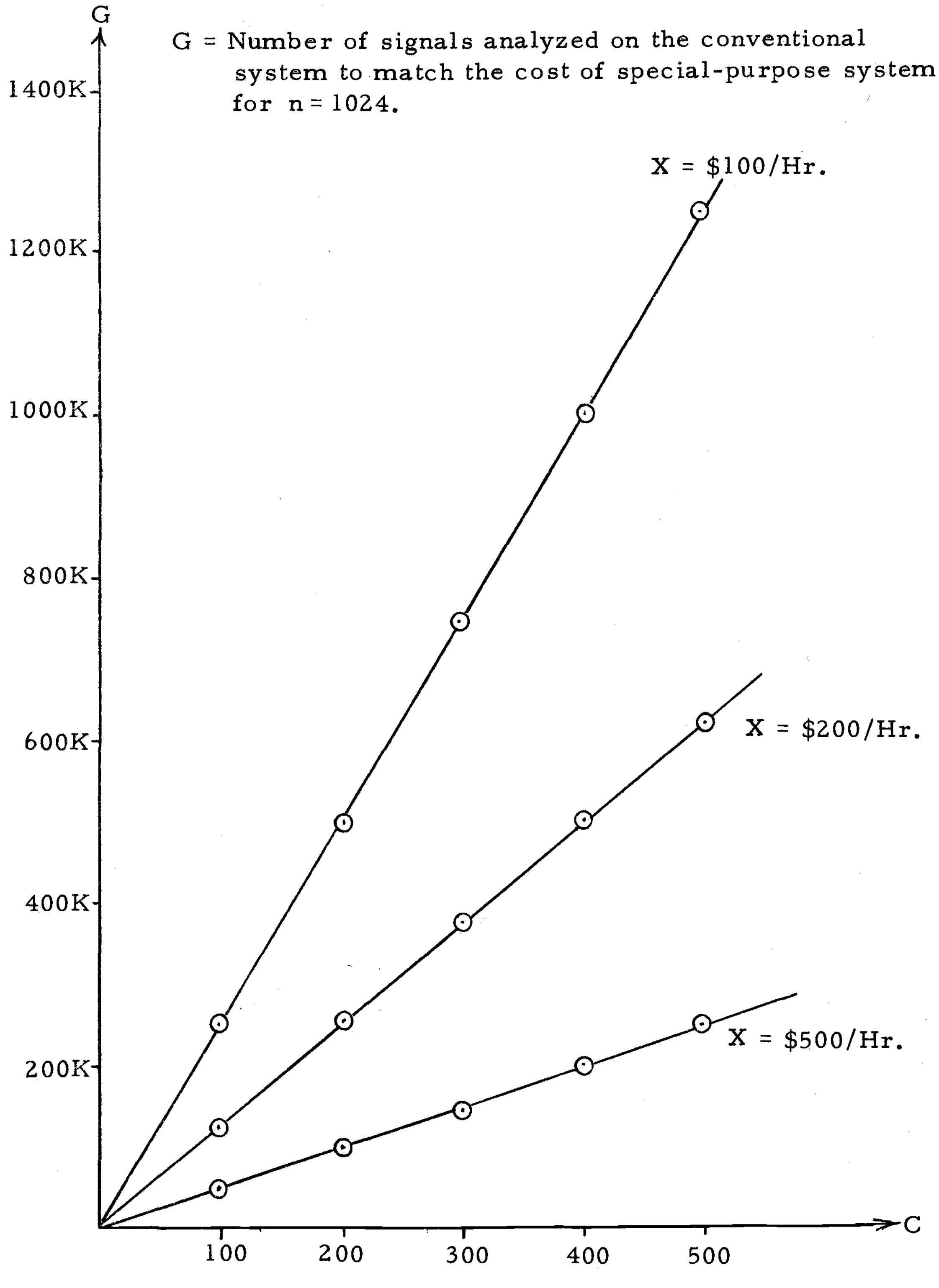


Figure 29. Plot of G versus C for n = 1024 and variations in X.

The special-purpose system can complete a set of spectral estimates for a 1024 point signal in 167 milliseconds. This means that the special-purpose system can match its cost in approximately 210,000 seconds or 60 hours of continuous operation.

A similar worst case analysis for n = 128 points assuming the same instruction execution times requires approximately 9,600,000 signals analyzed on the conventional system to match the cost of the dedicated system. This corresponds to 110,000 seconds or approximately 30 hours of continuous operation.

Real-Time Application

The function of the special-purpose system previously described is to compute spectral estimates from discrete samples of time-varying signals. If this system is to operate in real time, the capability of computing the complete set of estimates for a given signal within the signal time period (T) would be advantageous. Such a machine organization would allow spectral estimates to be computed continuously.

The system initiates operation upon the collection of a complete set of samples for a given signal. During the time required for computing spectral estimates for this set of samples, the next set is loaded into the system. Assuming the time necessary for computing the initial set of estimates is less than the signal period of the latter set, the system can operate continuously in real time.

Let $n = 2^m$ be the number of points sampled in the signal time period T. Assuming the execution time (V) of the multiplication instructions is dominant, the spectral estimates require $V(n+\log_2 n)$ seconds for completion. This means that such a system can support sampling rates up to $n/(V)(n+\log_2 n)$.

Figure 30 illustrates the sampling rates which can be supported by the special-purpose system for variations in the number of samples and time required for multiplication.

The supportable sampling rate increases as the number of samples increases. However, an upper bound of $1/V$ exists for large values of n. This upper bound is expected since each sample point requires at least one multiplication in the calculation of a spectral estimate.

The utilization of a conventional computer system for real-time applications in the area of spectral analysis is quite limited. To illustrate this point, assume a data record of 1024 points is to be analyzed. For Table 4 the conventional system requires 29.5 seconds for completing the analysis. This corresponds to supporting sampling rates up to 36 samples per second. The Sampling Theorem states that the sampling rate must be at least twice the highest signal frequency in order to prevent aliasing. Thus the conventional system would be impractical for signals containing

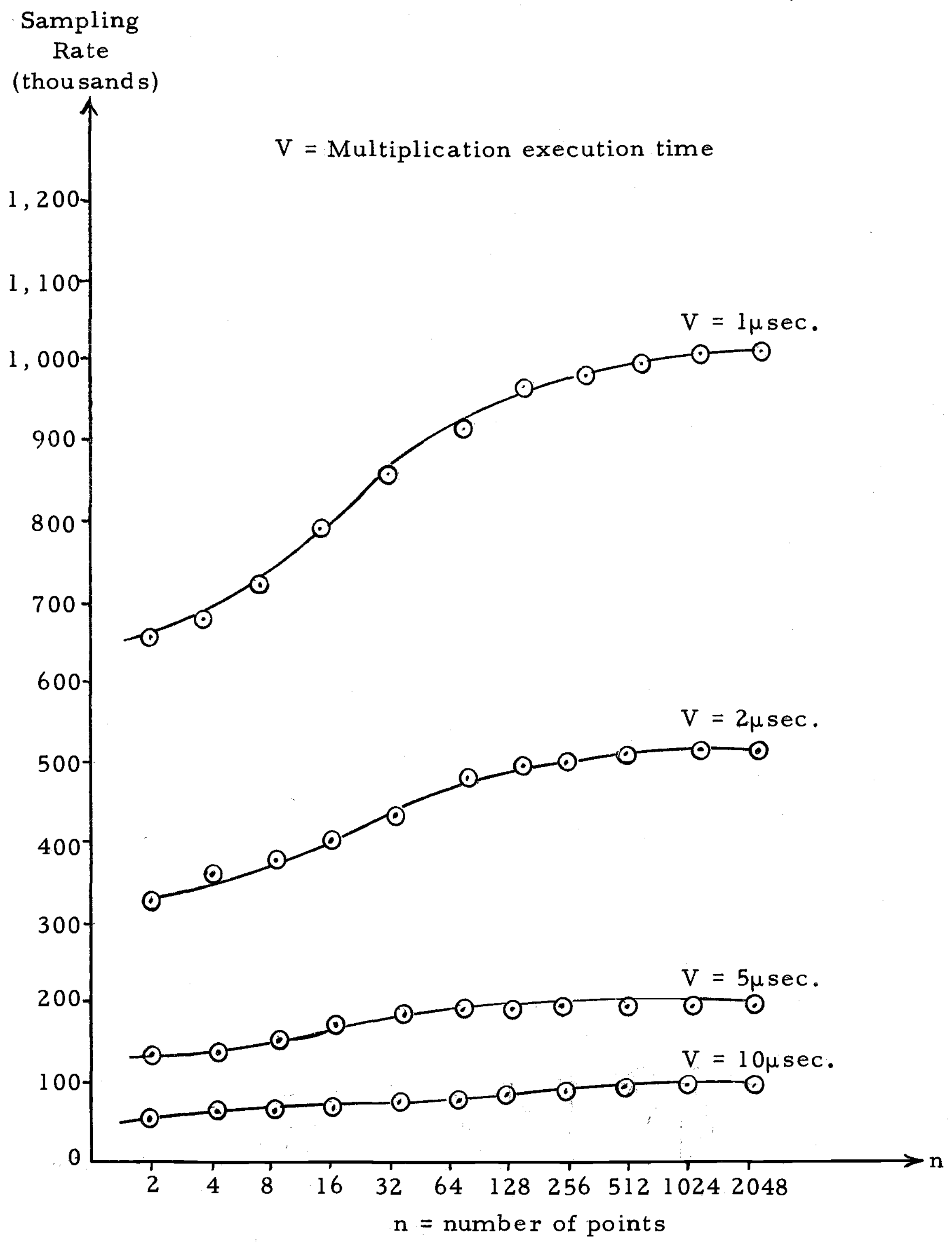


Figure 30. Plot of the supportable sampling rate for the special-purpose system for corresponding variations in the number of points and multiplication time.

frequencies above 18 Hz. Besides the limitation on frequency, the conventional system also places a second restriction that the signal period be greater than 29.5 seconds for continuous operation.

The special-purpose system for data records containing 1024 points is capable of supporting signal periods greater than 168 milliseconds as found in Table 4. This corresponds to supporting sampling rates up to 6150 samples per second. Thus such a system is practical for signals containing frequencies up to approximately 3 KHz. As shown in Figure 30, if 1 μsec. multiplication is available, the special-purpose system will support sampling rates up to 500K samples per second and signals with frequency components up to 250 KHz.

The high supportable sampling rates and speed in computing spectral estimates is especially desirable in applications characterized by burst or pulse signals. Typical signals may last only a few milliseconds and repeat every second. The dead time between burst signals is usually unpredictable. The requirements on analysis are not as severe for these signals since the sum of the signal period and dead time is available for computing estimates before the next signal occurs. But even with the additional dead time, the conventional system would not be practical for these signals since a small signal period is characterized by high frequencies and large data records.

VII. CONCLUSIONS

The concept of an arithmetic-memory register has been examined in this paper. From this concept a model composed of instruction states and transition states was developed. It was shown that such a model could be applied to both past and contemporary computing systems.

The number of possible parallel cycles was used to measure the parallel capability of a machine. The general-purpose machine composed of arithmetic-memory registers was shown to be superior to both conventional systems and systems structured upon rectangular arrays. However, the implementation of such a general-purpose system was found unfeasible due to the tremendous number of interconnections required.

The concept of arithmetic-memory registers together with the state model was shown to be applicable to special-purpose systems. A special-purpose system for computing power spectra was examined. The application of the transition state model proved to be a straightforward but useful technique in the determination of a workable set of interconnections between the registers. The special-purpose system exhibited the following characteristics:

1. Required fewer loading, addition, and multiplication instructions as compared with the conventional system.

2. For data records characterized by a large number of samples, the system's speed in computing spectral estimates was several orders of magnitude faster than the conventional system.
3. The cost estimates for the special-purpose system were very competitive when compared to similar costs estimated for the conventional system.
4. The special-purpose system was suitable for real-time applications and supported sampling rates as high as 500,000 samples per second for a 1 μsec. multiplication execution time.

Through the use of a model consisting of instruction states and transition states, the concept of utilizing arithmetic-memory registers in the design of digital computing systems has proven unfeasible in the case of general-purpose systems, and feasible in the case of a special-purpose system. The special-purpose system used in this study is by no means unique. Other dedicated systems such as in the areas of matrix manipulation, solving systems of differential equations, and the general area of real-time applications have characteristics which complement such a structure and requirements which demand it.

BIBLIOGRAPHY

1. Barnes, B. H., et al. The ILLIAC IV computer. IEEE Transactions on Computers C-17:746-757. 1968.

2. Bergland, G. D. A guided tour of the fast-Fourier transform. IEEE Spectrum. 6:41-52. 1969.

3. Bingham, C., M. D. Godfrey and J. W. Tukey. Modern techniques of power spectrum estimation. IEEE Transactions on Audio and Electroacoustics. AU-15:56=66. 1967.

4. Blackman, R. B. and J. W. Tukey. The measurement of power spectra from the point of view of communications engineering. New York, Dover, 1958. 190 p.

5. Brigham, E. O. and R. E. Morrow. The fast-Fourier transform. IEEE Spectrum. 4:63-70. 1967.

6. Burnett, G. J., L. J. Koczela and R. A. Hokum. A distributed processing system for general purpose computing. In: Proceedings of the Fall Joint Computer Conference of the American Federation of Information Processing Societies, Anaheim, 1967. Vol. 31. Washington, D. C., Thompson, 1967. p. 757-768.

7. Comfort, W. T. A modified Holland machine. In: Proceeding of the Fall Joint Computer Conference of the American Federation of Information Processing Societies, Las Vegas, 1963. Vol. 24. Baltimore, Spartan, 1963. p. 481-488.

8. Cooley, J. S. and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation. 19:297-301. 1965.

9. Gentleman, W. M. and G. Sande. Fast-Fourier transforms--for fun and profit. In: Proceedings of the Fall Joint Computer Conference of the American Federation of Information Processing Societies, San Francisco, 1966. Vol. 29. Washington, D. C., Spartan, 1966. p. 563-578.

10. Gold, B., et al. The FDP, a fast programmable signal processor. IEEE Transactions on Computers. C-20:33-38. 1971.

11. Holland, J. W. A universal computer capable of executing an arbitrary number of sub-programs simultaneously. In: Proceeding of the Eastern Joint Computer Conference, Boston, 1959. Vol. 16. p. 108-113.

12. Kuck, D. J. ILLIAC IV software and applications programing. IEEE Transactions on Computers. C-17:758-770. 1968.

13. Moore, G. E. Semiconductor RAMS--a status report. Computer 4:6-10. March, 1971.

14. Pease, M. C. An adaptation of the fast-Fourier transform for parallel processing. Journal of the Association for Computing Machinery. 15:252-263. 1968.

15. Richards, P. I. Computing reliable power spectra. IEEE Spectrum. 4:83-90. January, 1967.

16. Rosenfeld, J. L. A case study in programming for parallel processors. Communication of the Association for Computing Machinery. 12:645-655. 1969.

17. Slotnick, D. L., W. C. Borck and R. C. McReynolds. The SOLOMON computer. In: Proceedings of the Fall Joint Computer Conference of the American Federation of Information Processing Societies, 1962. Vol. 22. Washington, D. C., Spartan, 1962. p. 97-107.

18. Stockham, T. G. High-speed convolution and correlation. In: Proceedings of the Spring Joint Computer Conference of the American Federation of Information Processing Societies, 1966. Vol. 28. Washington, D. C., Spartan, 1966. p. 229-233.

19. Stone, H. S. Parallel processing with the perfect shuffle. IEEE Transactions on Computers. C-20:153-623. 1971.

20. Watson, G. A., W. E. Hansalik and H. B. Emerson. Multiple arithmetic iterative array computer. In: National Electronics Conference, Chicago, 1964. Vol. 20. Chicago, 1964. p. 669-674.

APPENDIX

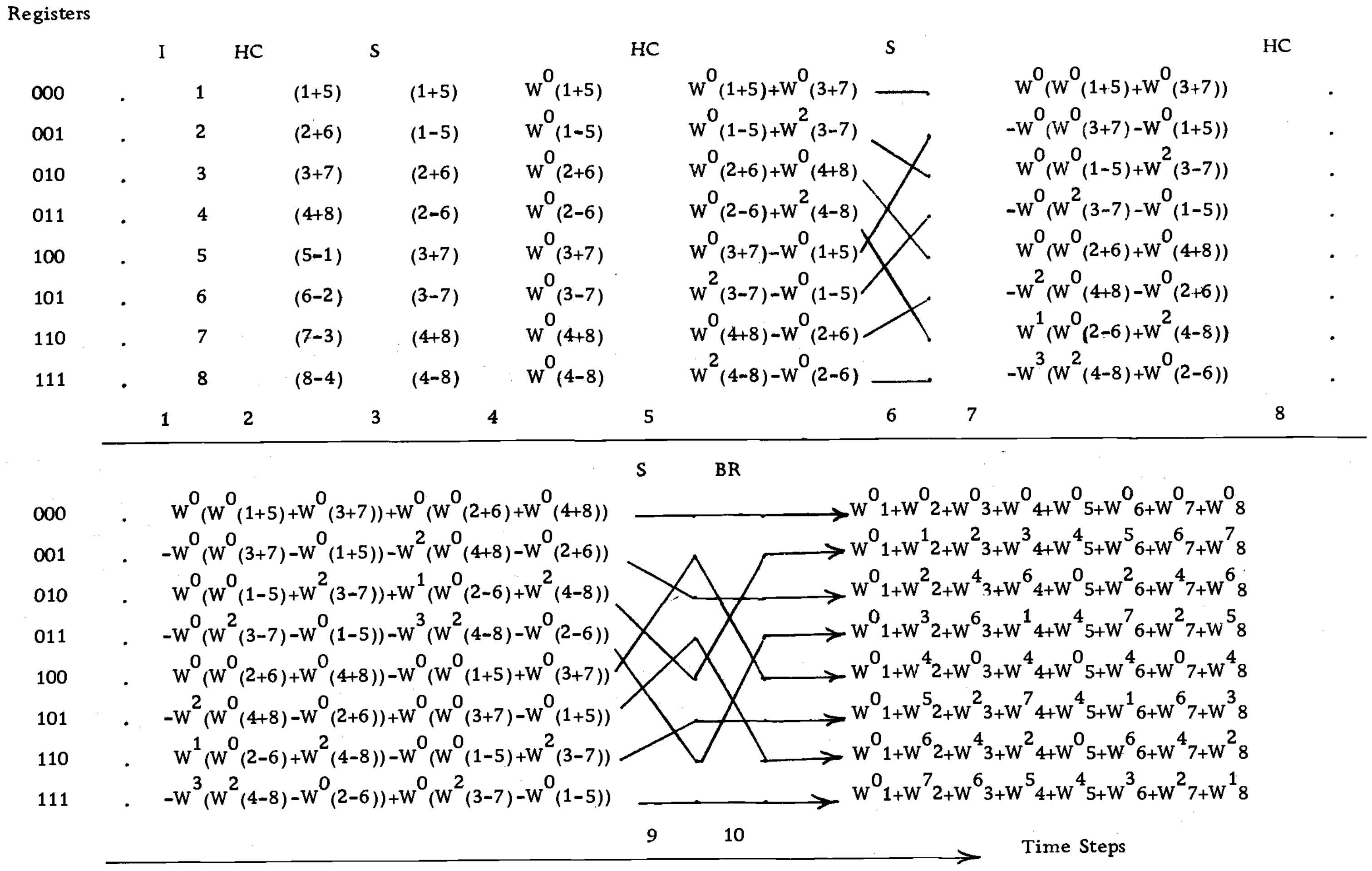
APPENDIX A

Transition States and Instruction States Used to Formulate the Weighting Coefficients

The flow diagram for the fast-Fourier transform is illustrated in Figure 22. The required interconnections between the arithmetic-memory registers are illustrated in Figure 26. The contents of the various registers at the termination of each time-step is illustrated in Figure 31.

Instruction State for "Half-Cross" Transition

The contents of the registers following the "half-cross" transitions of time-steps 2, 5 and 8 were attained using the instruction state consisting of the addition operation for the upper half registers and the subtraction operation for the lower half registers. Since the second time step computes terms using W^0 in the upper half register and W^4 in the lower half register, the relationship $W^4 = -W^0$ is utilized in the lower registers. The fifth time-step utilizes the relationships $W^4 = -W^0$ and $W^6 = -W^2$. Therefore the subtraction operation performed by the lower half registers reduces the number of required weights by means of the general relationship $W^i = -W^{i \pm n/2}$.



Registers

		I	HC	S		HC	S		HC
000	.	1	(1+5)	(1+5)	$W^0(1+5)$	$W^0(1+5)+W^0(3+7)$		$W^0(W^0(1+5)+W^0(3+7))$	.
001	.	2	(2+6)	(1-5)	$W^0(1-5)$	$W^0(1-5)+W^2(3-7)$		$-W^0(W^0(3+7)-W^0(1+5))$	.
010	.	3	(3+7)	(2+6)	$W^0(2+6)$	$W^0(2+6)+W^0(4+8)$		$W^0(W^0(1-5)+W^2(3-7))$	.
011	.	4	(4+8)	(2-6)	$W^0(2-6)$	$W^0(2-6)+W^2(4-8)$		$-W^0(W^2(3-7)-W^0(1-5))$	.
100	.	5	(5-1)	(3+7)	$W^0(3+7)$	$W^0(3+7)-W^0(1+5)$		$W^0(W^0(2+6)+W^0(4+8))$	.
101	.	6	(6-2)	(3-7)	$W^0(3-7)$	$W^2(3-7)-W^0(1-5)$		$-W^2(W^0(4+8)-W^0(2+6))$	.
110	.	7	(7-3)	(4+8)	$W^0(4+8)$	$W^0(4+8)-W^0(2+6)$		$W^1(W^0(2-6)+W^2(4-8))$	.
111	.	8	(8-4)	(4-8)	$W^0(4-8)$	$W^2(4-8)-W^0(2-6)$		$-W^3(W^2(4-8)+W^0(2-6))$	.
	1	2	3	4	5		6	7	8

			S	BR	
000	.	$W^0(W^0(1+5)+W^0(3+7))+W^0(W^0(2+6)+W^0(4+8))$			$W^0 1+W^0 2+W^0 3+W^0 4+W^0 5+W^0 6+W^0 7+W^0 8$
001	.	$-W^0(W^0(3+7)-W^0(1+5))-W^2(W^0(4+8)-W^0(2+6))$			$W^0 1+W^1 2+W^2 3+W^3 4+W^4 5+W^5 6+W^6 7+W^7 8$
010	.	$W^0(W^0(1-5)+W^2(3-7))+W^1(W^0(2-6)+W^2(4-8))$			$W^0 1+W^2 2+W^4 3+W^6 4+W^0 5+W^2 6+W^4 7+W^6 8$
011	.	$-W^0(W^2(3-7)-W^0(1-5))-W^3(W^2(4-8)-W^0(2-6))$			$W^0 1+W^3 2+W^6 3+W^1 4+W^4 5+W^7 6+W^2 7+W^5 8$
100	.	$W^0(W^0(2+6)+W^0(4+8))-W^0(W^0(1+5)+W^0(3+7))$			$W^0 1+W^4 2+W^0 3+W^4 4+W^0 5+W^4 6+W^0 7+W^4 8$
101	.	$-W^2(W^0(4+8)-W^0(2+6))+W^0(W^0(3+7)-W^0(1+5))$			$W^0 1+W^5 2+W^2 3+W^7 4+W^4 5+W^1 6+W^6 7+W^3 8$
110	.	$W^1(W^0(2-6)+W^2(4-8))-W^0(W^0(1-5)+W^2(3-7))$			$W^0 1+W^6 2+W^4 3+W^2 4+W^0 5+W^6 6+W^4 7+W^2 8$
111	.	$-W^3(W^2(4-8)-W^0(2-6))+W^0(W^2(3-7)-W^0(1-5))$			$W^0 1+W^7 2+W^6 3+W^5 4+W^4 5+W^3 6+W^2 7+W^1 8$
			9	10	

Time Steps

Figure 31. Contents of registers at the termination of each time-step for the case n=8.

Transition State			Instruction State
HC	=	Half Cross	(upper registers add and lower registers substract)
S	=	Perfect Shuffle	(change sign of odd numbered registers)
I	=	Identity	(load instruction for all registers)
BR	=	Binary Reversal	(load instruction for all registers)

"Perfect Shuffle" Transition

As illustrated in Figure 31, the "perfect shuffle" transition is utilized in time-steps 3, 6 and 9. During these transitions the odd numbered registers are required to make a sign change during the shuffle. This change is clearly indicated between time-steps 7 and 8 by the negative sign preceding the odd terms. The sign change associated with time-step 3 is indicated by a reversal in the order of the terms appearing in odd numbered registers. This sign reversal is needed because the input is inherently subtracted from the contents of the register and the exact reverse is desired.

Weighting Coefficients

Since the autocorrelation function required 2n registers for analyzing n-point data records while the transform requires only n registers for computing its estimates, the remaining n registers used to compute the autocorrelation function can be utilized to store the weighting coefficients.

The weighting coefficients are applied during time-steps 1, 4 and 7. The weights applied during time-step 1 are the W^0 weights corresponding to unity. The weights applied during time-step 4 are W^0 and W^2; and for the general case these will be W^0 and $W^{n/4}$. Time-step 7 requires W^0, W^2, W^1 and W^3; and in the general case

W^0, $W^{n/4}$, $W^{n/4-n/8}$ and $W^{n/4+n/8}$. A system for generating these weights for n registers need only operate on the lower half registers since the upper half registers are all multiplied by the weight $W^0 = 1$.

Assume the set of n/2 registers contain the weights W^0. During time-step 1, the lower half of these registers is multiplied by the term $W^{n/4}$. The "evens up and odds down" transition is used during time-step 2 to rearrange the terms. A multiplication by $W^{n/8}$ on the lower half is accomplished during time-step 3. The above cycle is repeated until a multiplication by $W^{n/n} = W^1$ is completed. Figures 32 and 33 illustrate that the proper weights will be generated for the cases n = 8 and n = 16 respectively.

Figure 31 can be used to verify the fast-Fourier transform algorithm. With the term $W^0 = 1$, the first term will indeed correspond to the DC component since it is the average value of the signal. The other seven terms correspond multiples of the fundamental frequency.

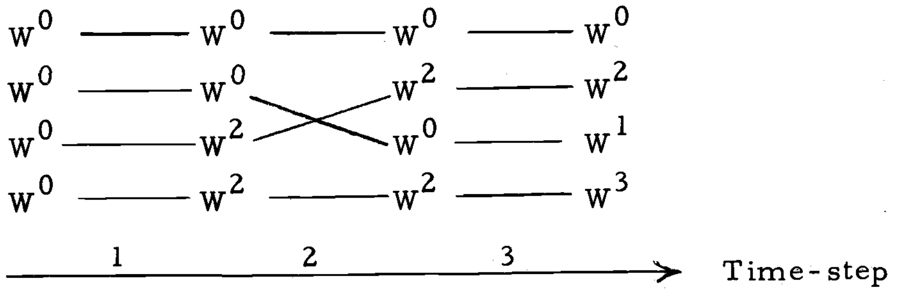
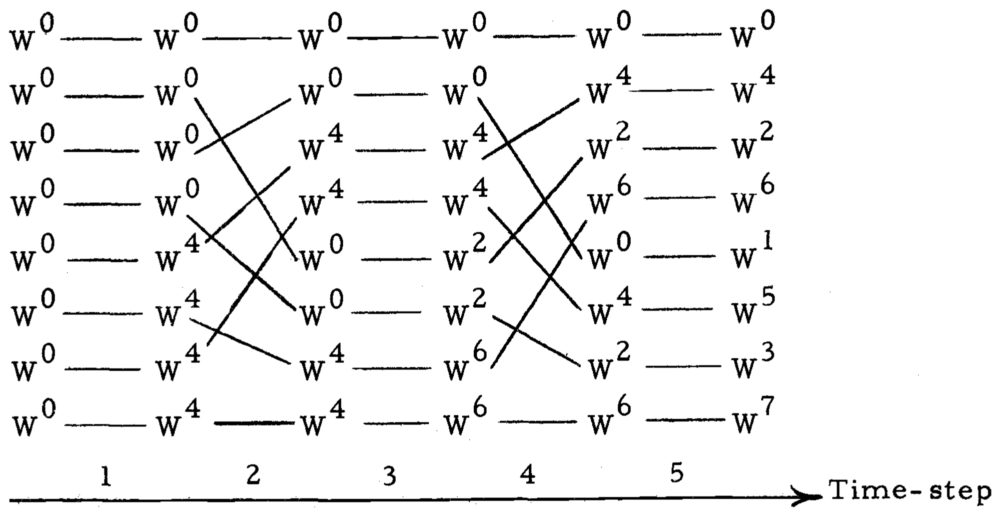


Figure 32. Weights for n = 8.



W^0 W^0 W^0 W^0 W^0 W^0
W^0 W^0 W^0 W^0 W^4 W^4
W^0 W^0 W^4 W^4 W^2 W^2
W^0 W^0 W^4 W^4 W^6 W^6
W^0 W^4 W^0 W^2 W^0 W^1
W^0 W^4 W^0 W^2 W^4 W^5
W^0 W^4 W^4 W^6 W^2 W^3
W^0 W^4 W^4 W^6 W^6 W^7
1
2
3
4
5
Time-step

Figure 33. Weights for n = 16.