1987

A reconfigurable interface for systolic arrays /

Anthony William Seaman
Lehigh University

A Reconfigurable Interface For Systolic Arrays

by
Anthony William Seaman

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Electrical Engineering

Lehigh University
1987

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

Aug 27, 1987
(date)

Professor in Charge

D. L. Talhelm
EE Division Chairperson

L. J. Varnerin
CSEE Department Chairperson

Acknowledgements

I wish to express my gratitude to Dr. Meghanad D. Wagh for the invaluable assistance and insight he provided with this thesis. His guidance and technical contributions were essential to completing this work. Thanks also to the faculty, staff, and other graduate students of the CSEE department for their help during my research for this thesis.

Table of Contents

List of Figures

Abstract

Demands for faster and more powerful signal processing capabilities have resulted in the proliferation of reconfigurable bit-sequential systolic arrays. Data communication with these machines must be at rates commensurate with their operating speed in order to fully realize their computational potential.

The front end for the systolic array is typically a general purpose computer which must be interfaced with the array. This thesis is concerned with studying such an interface, identifying design problems, and proposing optimal solutions.

Architectural differences lead to a great disparity in speed and data formats between the general purpose host computer and the reconfigurable array. The difference in operating speeds and asynchronicity is overcome with the use of FIFOs and careful selection of local memory word sizes. Input data conditioning is performed in a series of subsystems, each designed with VLSI implementation as a consideration. Results generated by the simultaneous execution of multiple tasks in the array are routed through two subsystems. The first subsystem separates data by tasks and the second reformats data to match the host computer's requirements and pipelines these words for optimal throughput.

Chapter 1
Introduction

1.1 Systolic Arrays

1.1.1 An Overview

Exploding technological applications have resulted in demands for ever increasing processing power in digital hardware. Applications such as image processing require data processing at rates exceeding 10 Mbytes/sec. To meet these needs, many new architectural innovations have emerged in recent years. Among these, one of the most promising is the systolic array [1-4]. This architecture benefits from multiprocessing without sacrificing synchronization and the simplicity of data paths. Systolic Arrays are modular in nature, as the name implies, and each module communicates only with its nearest neighbor. This characteristic translates to a very efficient implementation in silicon.

In a normal processor, an execution cycle is always preceeded by an equally time-intensive instruction fetch cycle. The high throughput of a systolic array may be attributed to the fact that each processing element in it executes only a single instruction and, therefore, no potential processing time is spent performing an instruction fetch. However, this results in a very inflexible architecture which requires a separate array for each task to be executed. Such limitations have prompted researchers to investigate *reconfigurable* systolic arrays. Processing elements in such arrays are reconfigured, if needed, before the execution of a task.

A further improvement in systolic arrays comes from the use of bit-sequential processing elements. Such elements are smaller and easier to design.

Additionally, inter-processor communication is greatly simplified, since all data buses are one only bit wide. Both these factors contribute to more efficient use of VLSI area, making this a very attractive option in systolic array designs.

1.1.2 Current Technology

The Massively Parallel Processor (MPP), designed in 1983 by Goodyear Aerospace Corp. under contract from NASA Goddard Space Flight Center, is the first example of a bit-sequential systolic array [5,6]. The MPP is composed of 16,368 processing elements arranged in a square array. This SIMD (single instruction multiple data) machine performs image processing on data collected from satellites. The first commercially available reconfigurable systolic array, the Geometric Arithmetic Parallel Processor (GAPP), with bit-sequential modules, was marketed by NCR Corp. in 1984 [2,7]. This MIMD (multiple instruction multiple data) array is composed of 72 processing elements arranged in a 9x8 grid on a single VLSI chip. Because of the MIMD nature, the applications of this array are wide and varied. There is an ongoing effort at Lehigh University, and at other schools, to investigate various aspects of reconfigurable VLSI bit-sequential systolic arrays.

1.2 Function of Interface

The systolic arrays previously described typically have a general purpose host computer as a user interface. The data and instructions resident in the host computer must be communicated efficiently to the systolic array to take full advantage of its processing power. This thesis is concerned with the development of techniques necessary to accomplish this efficient transfer of information.

Due to the increasing popularity of bit-sequential systolic arrays, the need for such interfaces is now greater than ever. Design of these interfaces is complicated because of the differences in speed and data formats of the host and the array. It is not unusual for bit-sequential systolic arrays to have clocking rates in excess of 30-40 MHz, whereas baud rates of a typical host are limited to 19.2 kHz. Similarly, the data format of the host computer is a fixed size word which does not necessarily match the bit-slice format requirement of a systolic array. Further, this bit-slice format is dependent upon the particular application, in the case of a reconfigurable array, giving rise to additional complexity in the design of the interface.

NASA's MPP system uses a "staging memory" to effect data reformatting [5,6]. This staging memory consists of three blocks: the main stager; the input sub-stager; and the output sub-stager. The main stager communicates with the two sub-stagers and these, in turn, interface with the front-end computer and the array. The staging memory is programmed to manipulate data flowing through it by the staging memory manager, which receives from the front-end computer a description of the data and task to be performed.

Most of the memory is in the main stager, which is made up of 32 RAM banks (in a fully populated memory) arranged in 64-bit words. During one cycle of 1.6 μsec, each memory bank can receive and transmit one word for a transfer rate of 5 Mbytes/sec per bank (160 Mbytes/sec overall) for both the input and output.

The sub-stagers' memory is smaller and composed of 128 1-bit RAMs (since a bit slice for the MPP array is 128 bits), allowing data to be accessed and rearranged bit by bit. Cycle time for the sub-stagers is 100 nsec, during which 128 bits can be read and written. These bits are packed into 4-bit nibbles to reduce the wire count between the main stager and the sub-stagers. Data is transferred in 4-bit nibbles, 32 nibbles every 100 nsec. Incorporated in each sub-stager is a permutation network to rearrange the data before being stored in the memory.

1.3 Objectives of Thesis

This thesis is devoted to investigating the architectural features of a host-to-array interface. Design problems have been identified and solutions to them examined, with results of these studies presented in Chapter 2. Some of the more critical design considerations were found to be collecting data from the array; matching memory and array bandwidths; and reformatting data between host and the array. We will elaborate briefly on these points in the following paragraphs.

The array generates data in formats based upon its current configuration. It is the function of the data collection network to convert these different formats to the data format of the host. Our solution to this problem allows the array to execute more than one task simultaneously while ensuring that any results generated are collected without any timing conflicts (Section 2.6.4).

Since an array works at clock rates many times greater than the presently available memory elements, the array data storage and retrieval from this slow memory presents a bottleneck. In addition, the memory must service the needs of the host in transferring data to and from the array. The solution to this problem is to time multiplex RAM chips so that the effective memory bandwith satisfies the demands of both the array and host simultaneously. Details of this are presented in section 2.7.

Conversion of input data from the word parallel format of the host computer to the serial nature of the array must be performed by a circuit that is configurable to different formats as required by the array. The solution presented in section 2.3 suggests a two stage data manipulation: first stage performing a row-to-column permutation (if necessary) and the second stage implementing a generalized parallel-to-serial conversion.

1.4 Organization of Thesis

Chapter 2 of this thesis summarizes techniques which would be employed in the design of a general interface to be used between a host computer and a bit-sequential systolic array. Several key subsystems of this interface are examined in great detail and, wherever possible, designs which are optimum in terms of hardware minimization are presented. Pipelining is used extensively in order to achieve speeds compatible with the array.

A specific implementation of the interface previously described is presented in Chapter 3. It is used in an ongoing research project, in which the array's flexibility demands are not as great as a general bit-sequential systolic array. This has enabled us to present a simpler but complete design, and to discuss specifically the control of the interface through the use of a bit-slice microcontroller. Strict adherence to optimum subsystems detailed in Chapter 2 is relaxed in the design of some circuits of this interface due to the required use of commercially available parts.

Finally, Chapter 4 summarizes the results obtained in this work and points out areas that need further investigation.

Chapter 2
General Interface Design

2.1 Introduction

The principal goal of this thesis is to investigate efficient interfacing between a general purpose word-parallel host computer and a high speed application-specific bit-sequential systolic array. This chapter presents the design concepts of a universal interface that is adaptable to any size host and array. The design will be such that the circuit is flexible enough to work directly with various hosts and arrays.

2.2 Basic Design Considerations

The interface circuit between the host computer and the array of PEs consists of several smaller circuits, or blocks, interconnected to perform the required data staging functions. Essentially, what the interface needs to overcome is the mismatch in data rates of the array and the host and the fact that the host computer operates on data in a word-by-word manner and the array in a bit (or bit slice) sequential format.

A block diagram of the interface design is shown in figure 2-1. The Input and Output FIFOs and Conditioner & Memory blocks are used to bridge the gap in data rates between the host and the array. The Input and Output Stagers, together with the Parallel-to-Serial Converter and Switch Network, are responsible for manipulating data into the proper format of the unit receiving the data: word-by-word for the host or bit-sequential for the PE array.

There are many different ways for the interface to be configured depending on the task to be performed. A particular configuration is dictated by the host

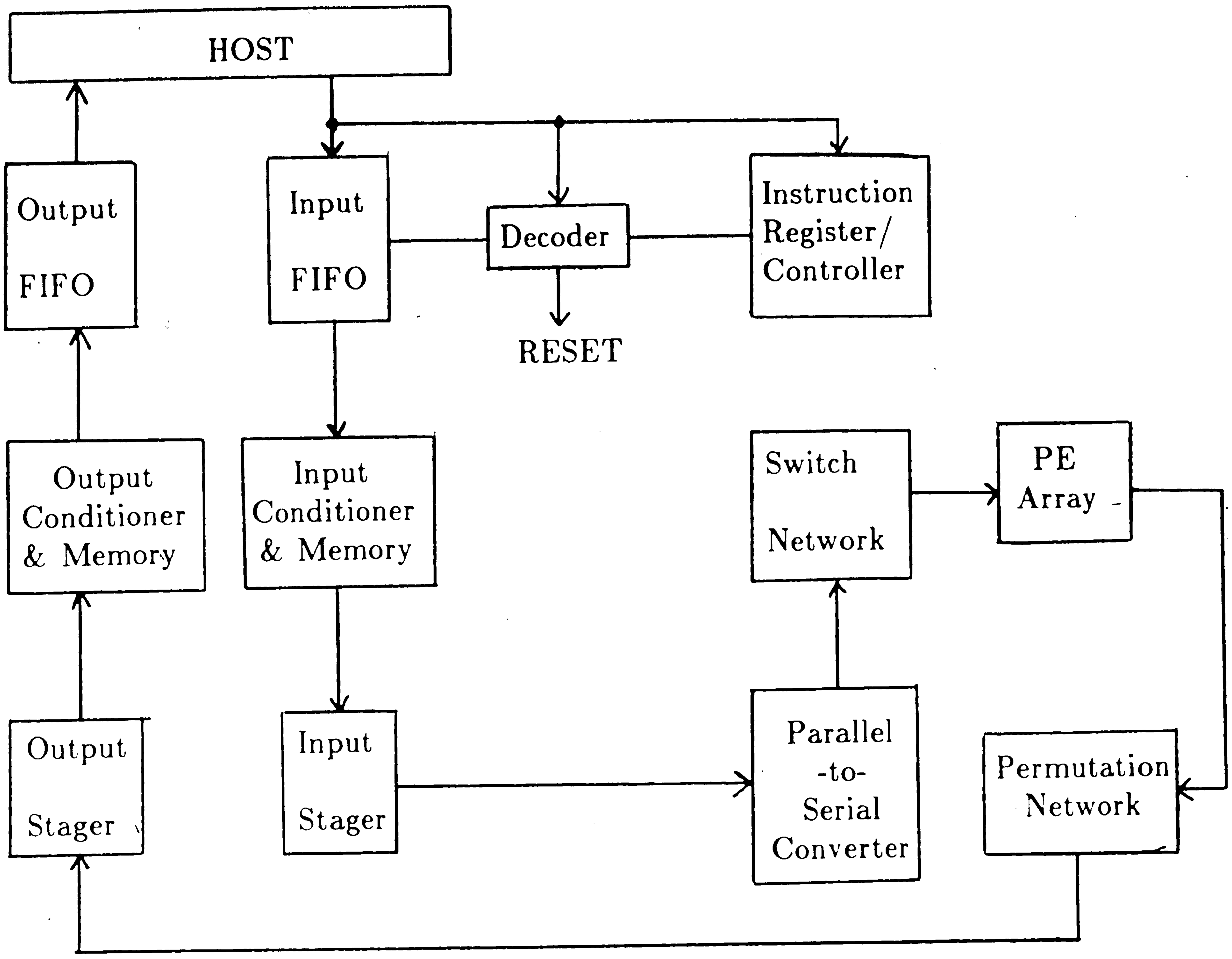
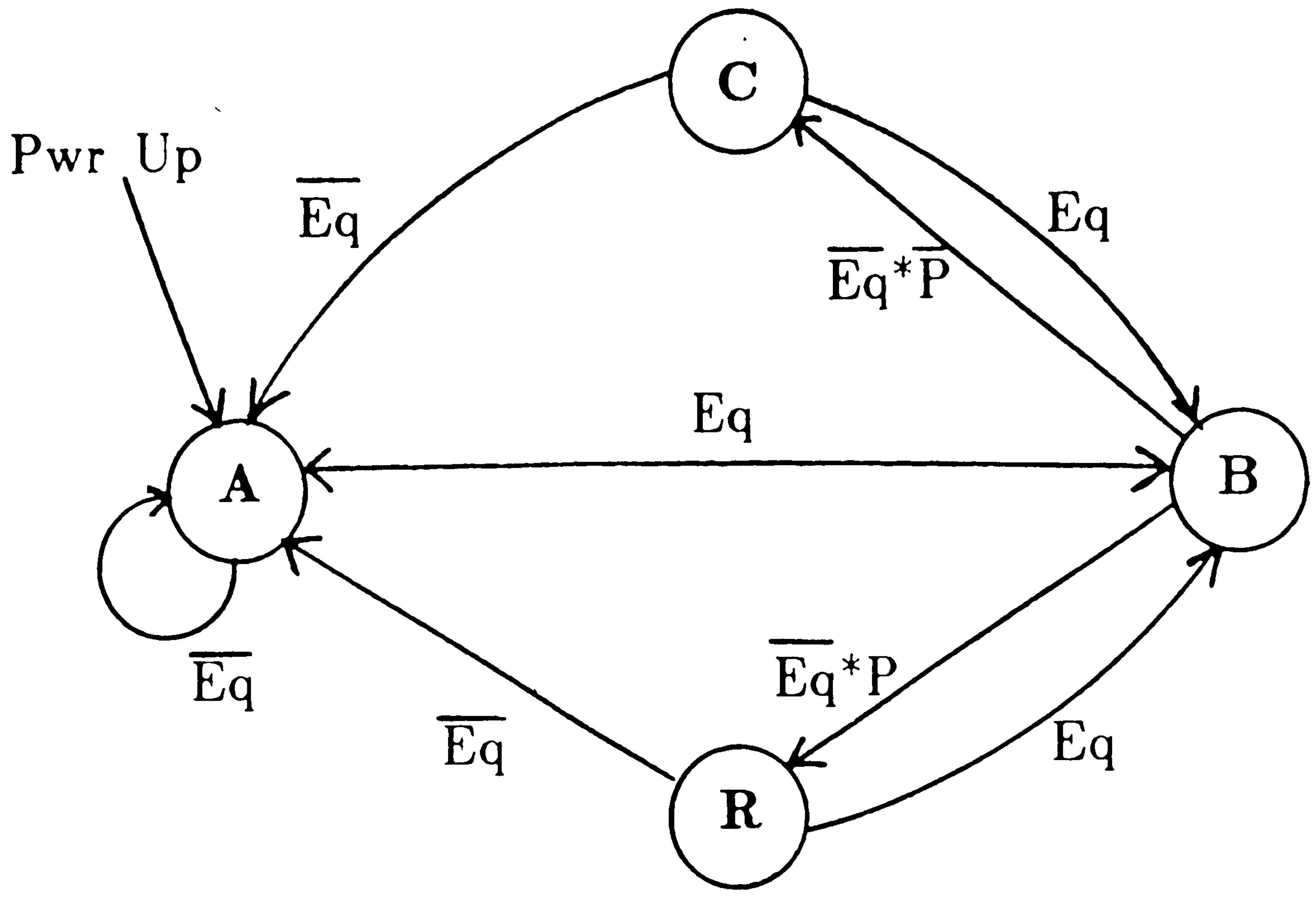


Figure 2-1: Interface Block Diagram

computer by means of instructions downloaded to the interface and interpreted appropriately by the Instruction Register/Controller block shown. Since both data and instructions will be on the same bus between the host and the interface, a Decoder circuit is required to decide whether the word on the bus is a data word or an instruction and to route the byte to the corresponding destination.

The Decoder works by assuming that any byte on the bus is data unless it is the all 0 byte. If it is the all 0 byte, then the next byte is an instruction unless the next byte is also all 0, in which case the data is all 0. Obviously, then, the all 0 byte may not be used as an instruction. The Decoder also responds to a RESET instruction to configure the interface to a pre-determined format. This insures that the host computer can always regain control of the interface. A state diagram of this machine is drawn in figure 2-2.



Eq: Byte matches all 0.
P: Panic, byte matches master reset instruction.

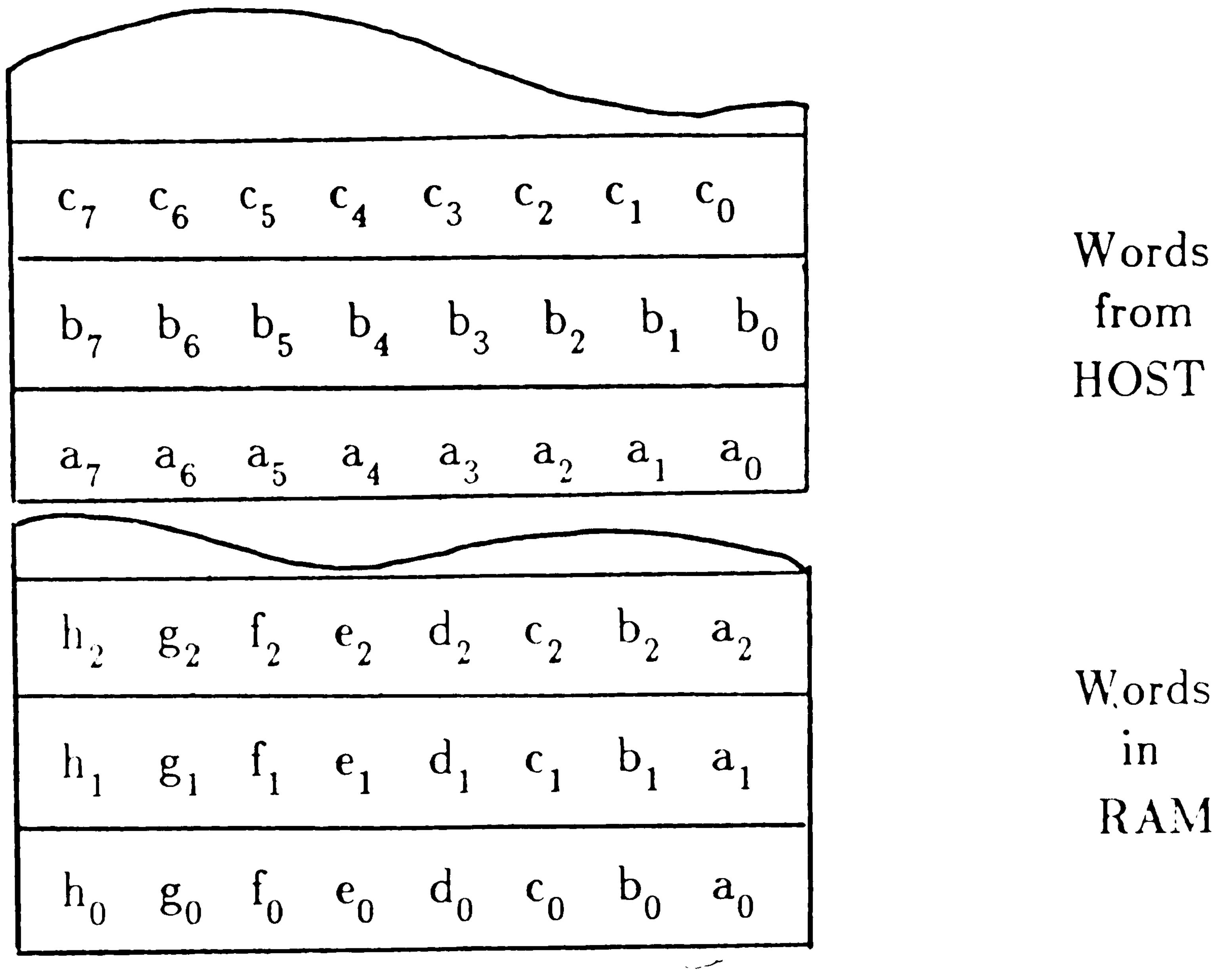
Enable Data FIFO in state A.
Enable Instruction Register in state C.
Enable Master Reset in state R.

Figure 2-2: Decoder State Machine

The Decoder and the Instruction Register/Controller will not be discussed in detail here, but the Instruction Register/Controller could well consist of a programmable microcontroller. A specific design of such a controller will be presented in the next chapter. The rest of the subsystems comprising the interface will now be examined in depth in the following sections.

2.3 Input Stager and RAM

The Input Stager, together with the RAM (within the Conditioner & Memory block, which will be discussed in section 2.7), allows the input data to be permuted from row to column form for up to N_A rows at a time (where N_A is the maximum number of distinct data lines to the array). That is, the first word stored in RAM may not be the first word of data sent by the host but, rather, the first bit of D ($D \leq N_A$) consecutive words; the second word in RAM is made up of the second bit of these D words; and so on. Figure 2-3 illustrates this function (showing only the first three words of the host and RAM) for N_A=D=8.



c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0

b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0

h_2 g_2 f_2 e_2 d_2 c_2 b_2 a_2

h_1 g_1 f_1 e_1 d_1 c_1 b_1 a_1

h_0 g_0 f_0 e_0 d_0 c_0 b_0 a_0

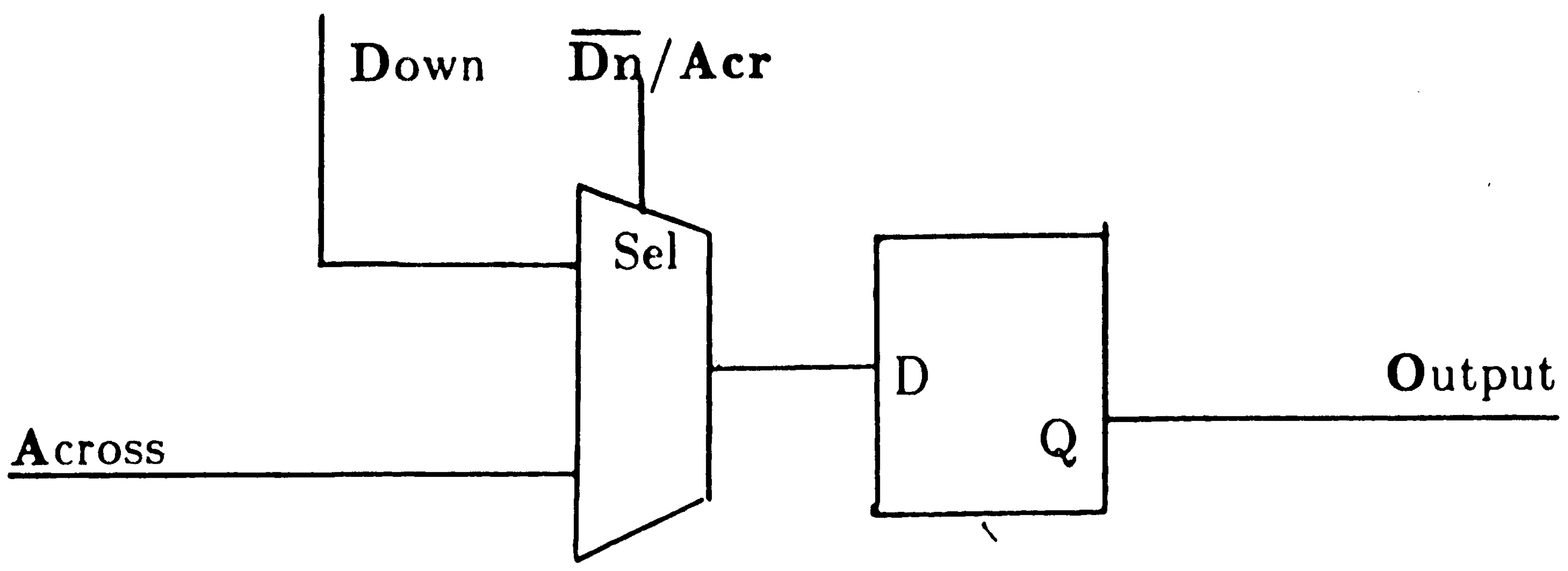
Figure 2-3: Row-to-Column Permuting

This configuration (first bits in first word, second bits next word, etc.) is necessary to execute bit slice oriented tasks in the array of PEs. If no row-to-column permuting stager existed, the first bits of each word would have to be accessed sequentially (only one memory word can be accessed in a given clock period). Since the Input Stager, once initially filled, has a throughput of unity, it represents a vast increase in speed over sequential access.

Figure 2-4 shows an Input Stager that would be used with an N-bit RAM. It consists of N^2 cells (arranged in an N by N square) each with a flip-flop and a 2:1 multiplexer as shown in figure 2-4. There are N 2:1 "collection multiplexers" that lead data to the Parallel-to-Serial Converter. Operation of this circuit is straight-forward: data enters the circuit from the top and propagates down one row per clock with permuted data being collected from the bottom row until all the data (up to N_A words) has entered the circuit. At this time, data flow changes direction (by changing the common select line of each 2:1 multiplexer) and will now enter from the left edge and propagate horizontally. Permuted data is now collected at the right edge of the circuit.

2.4 Switch Network

Data stored in the Input Memory is transferred to the array through a generalized Parallel-to-Serial Converter and the Switch Network. These two subsystems stage the data to the format required by the array. In addition, as explained in section 2.3, the Input Stager also participates in the data staging operation. Of these two blocks, the Switch Network is a conceptually simpler circuit. Its function is to route bits from any output line of the Parallel-to-Serial Converter to the appropriate row(s) of the array of PEs. This is accomplished by simply providing R N_A:1 multiplexers where R is the number of rows of the PE array (one multiplexer per row) and N_A is the number of output lines from the Parallel-to-Serial Converter of the next section. This configuration allows maximum flexibility: any bit can be routed to any row, including multiple destinations.



Input Stager Cell

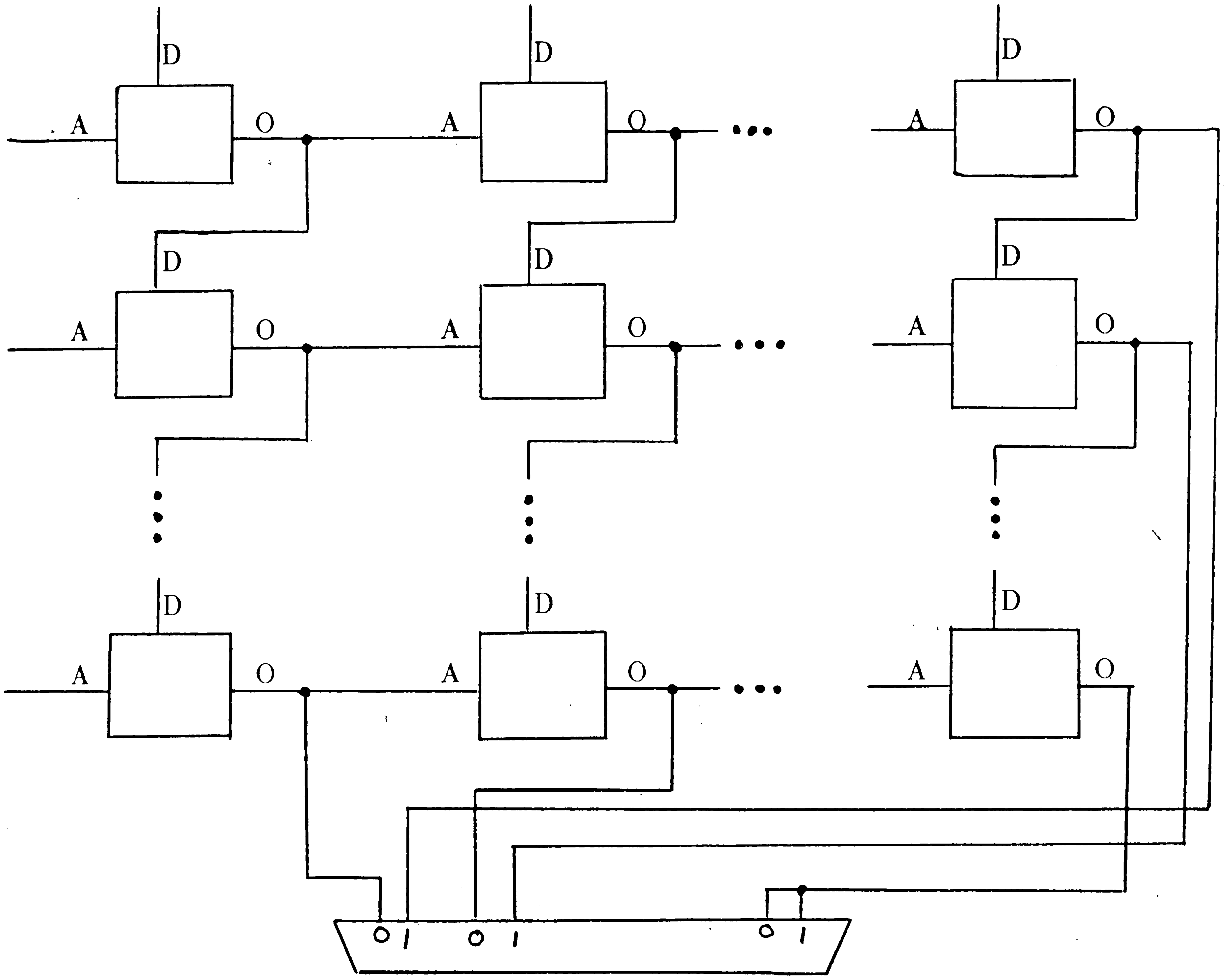


Figure 2-4: Input Stager Circuit

2.5 Generalized Parallel-to-Serial Converter

Data that is stored in the Input Memory in parallel fashion must be converted to some serial format before being presented to the array of PEs. Given that a word in memory is of the form (for N_A=8 bit word) $a_7a_6a_5a_4a_3a_2a_1a_0$, there are four possible combinations of serial output that will be treated. These forms are shown in figure 2-5.

a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 All bits to one row

a_6 a_4 a_2 a_0 Bits to
a_7 a_5 a_3 a_1 two rows

a_4 a_0
a_5 a_1 Bits to
a_6 a_2 four rows
a_7 a_3

a_0
a_1
a_2
a_3 Bits to
a_4 eight rows
a_5
a_6
a_7

Figure 2-5: Output Formats

These formats are achieved by reading the Input Memory word-by-word into a shift register and shifting the bits out. There are essentially two choices for implementing this process which will be referred to as:

1. Simple-Shift/Complex-Load and
2. Simple-Load/Complex-Shift.

These two approaches will now be discussed and compared in terms of their hardware complexity. The complexity of a boolean expression is equal to the number of its literals times the size of each. Thus the complexity of an n:1 multiplexer is

(# of terms)*(size of each term)

$= n*(\log_2 n + 1)$.

Thus, since a 4:1 multiplexer's expression is

Output = $\bar{s}_1\bar{s}_0x_0 + \bar{s}_1s_0x_1 + s_1\bar{s}_0x_2 + s_1s_0x_3$,

its complexity is 12.

The concept of simple-shift/complex-load is that, when in shift mode, bits proceed one cell at a time toward their output destinations. That is, if a bit is currently in cell c_i then it will next be in cell c_{i-1}. The loading of the register is arranged to achieve the desired output staging. Figure 2-6 shows an 8-bit simple-shift/complex-load shift register. Notice that the output is from different cells of the shift register depending on the format of the output (number of bits/clock). For instance, if the output is two bits per clock (even indices on one line, odd indices on another) then the output is from cells 0 (even indices) and 4 (odd indices). If the output format is four bits per clock then cells 0, 2, 4, and 6 generate the output.

The complexity of the simple-shift/complex-load form can be calculated as follows (for the example of N=8):

6) 4:1 multiplexers: $6*(4*(\log_2 4 + 1)) = 72$
1) 2:1 multiplexer: $1*(2*(\log_2 2 + 1)) = \underline{\ 4\ }$
76
8) D-type Flip Flops

A simple-load/complex-shift configuration (figure 2-7) is different in that the parallel data is always loaded into the same cell of the shift register: data

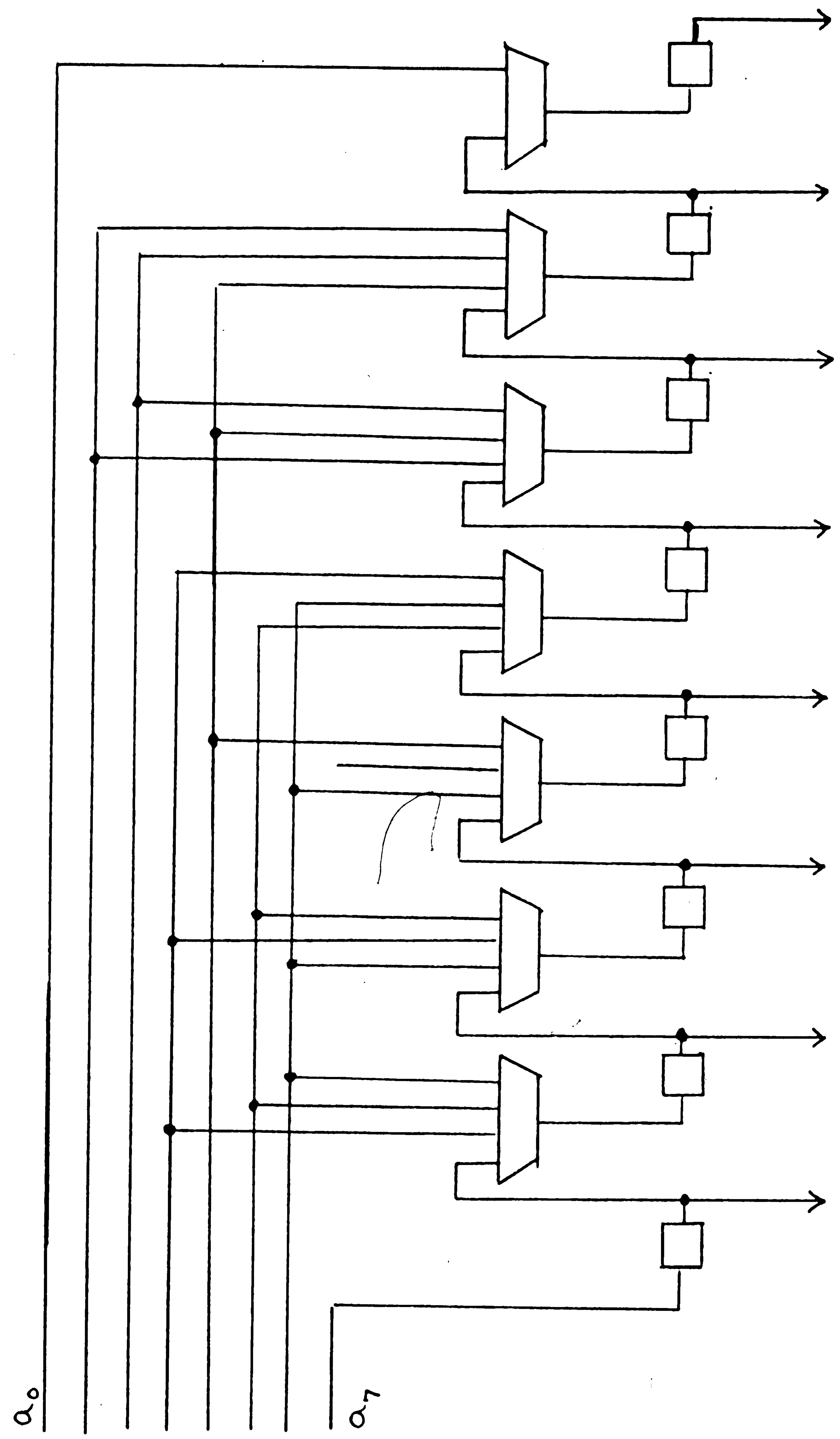


Figure 2-6: Simple-Shift/Complex-Load

bit a_i is loaded into cell c_i. However, cell c_i is not necessarily transferred into c_{i-1} when in the shift mode. For instance, if the output format is two bits per clock, then c_i is transferred to c_{i-2}. To generalize, if the format is l bits per clock then c_i is transferred to c_{i-l} at the clock edge. This parallel-to-serial conversion technique will generate the l output bits in the right-most l cells of the shift register. The complexity of an 8-bit simple-load/complex-shift converter is:

4)	4:1 multiplexers:	$4*(4*(\log_2 4 + 1))$	= 48
2)	3:1 multiplexers:	$2*(3*(\log_2 3 + 1))$	= 16
1)	2:1 multiplexers:	$1*(2*(\log_2 2 + 1))$	= 4
			68
8)	D-type Flip Flops		

Obviously, there is a reduction in complexity with the simple-load/complex-shift approach and this is our method of choice for parallel-to-serial conversion.

2.6 Accessing Results

2.6.1 Overview

Data results generated by the array are available one column (slice) at a time. This column is the right-most PE of each row of the array. Just as we assume that input data to the array will be 2^n bits per clock ($n \geq 0$) we will allow data output from the array only in groups that are powers of two. In other words, serial data that is to be grouped together as the same word in memory must be outputted one bit per clock (one row), two bits per clock (two rows), four bits per clock (four rows), etc. up to $N=2^n$ bits per clock where N is the "word size" of the array.

Figure 2-8 depicts how results are captured from the array of PEs. Since

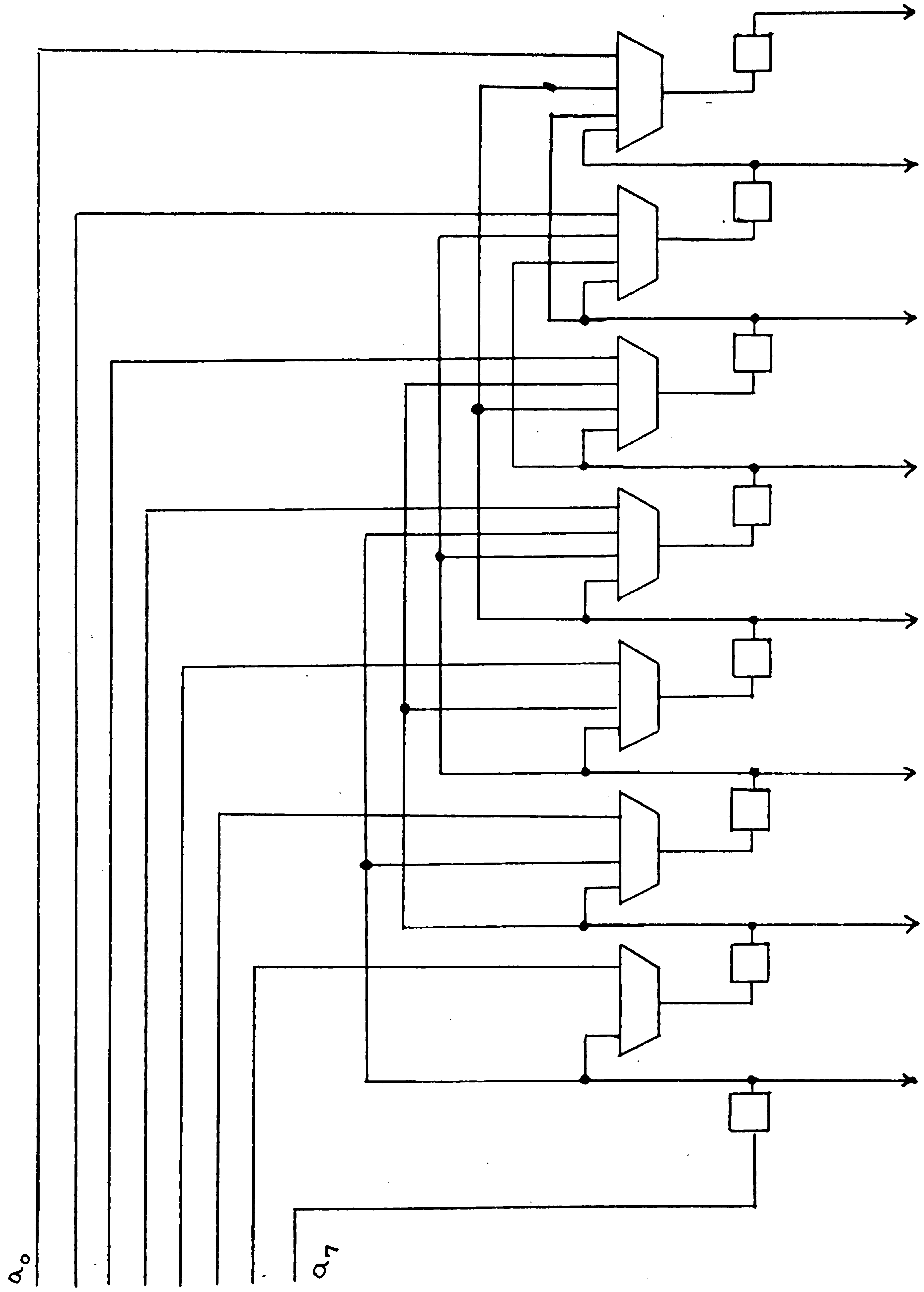


Figure 2-7: Simple-Load/Complex-Shift

the array is of row size M and the word size is N, at most N of the M rows may be accessed at any one time.

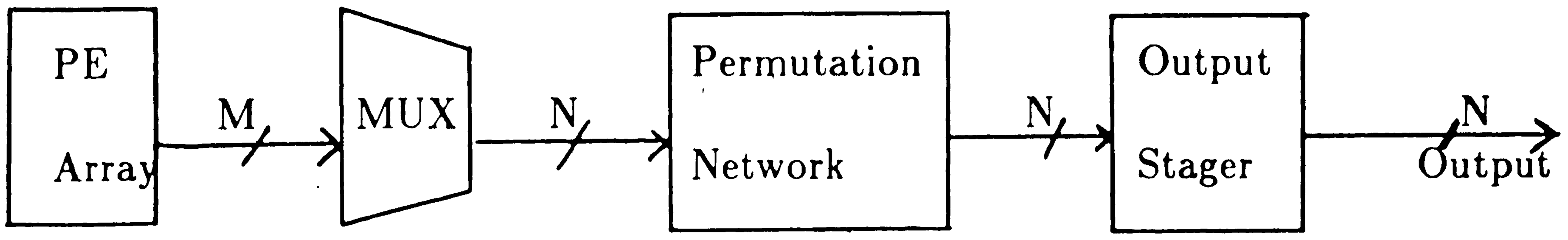


Figure 2-8: Accessing Results

M may be greater than N necessitating an M-to-N multiplexer to select which rows will be accessed (fewer than N rows may be active but this poses no problem).

The Permutation Network can re-order the N input lines ($r_0 r_1$ ... r_{N-1}) to all possible permutations at the output. The need for this network will be subsequently explained.

The Output Stager circuit takes as input the N lines of output from the Permutation Network (which is basically serial in nature) and outputs an N-bit word at a rate up to one word per clock (when there is active data on N lines). The Output Stager is modular in design, with each module consisting of a serial-to-parallel converter, N 2:1 multiplexers, and an N-bit latch. All components of the data access circuit will now be discussed in more depth.

2.6.2 Permutation Network

The Permutation Network consists of q butterfly networks where q is calculated from $2^q \geq N!$. For N=8, N!=40,320 which means that q=16 butterfly networks would be needed. This can be implemented as four stages of four networks each. Figure 2-9 shows an example of how the butterfly networks are implemented by a Permutation Network for N=4 (4!=24, q=5).

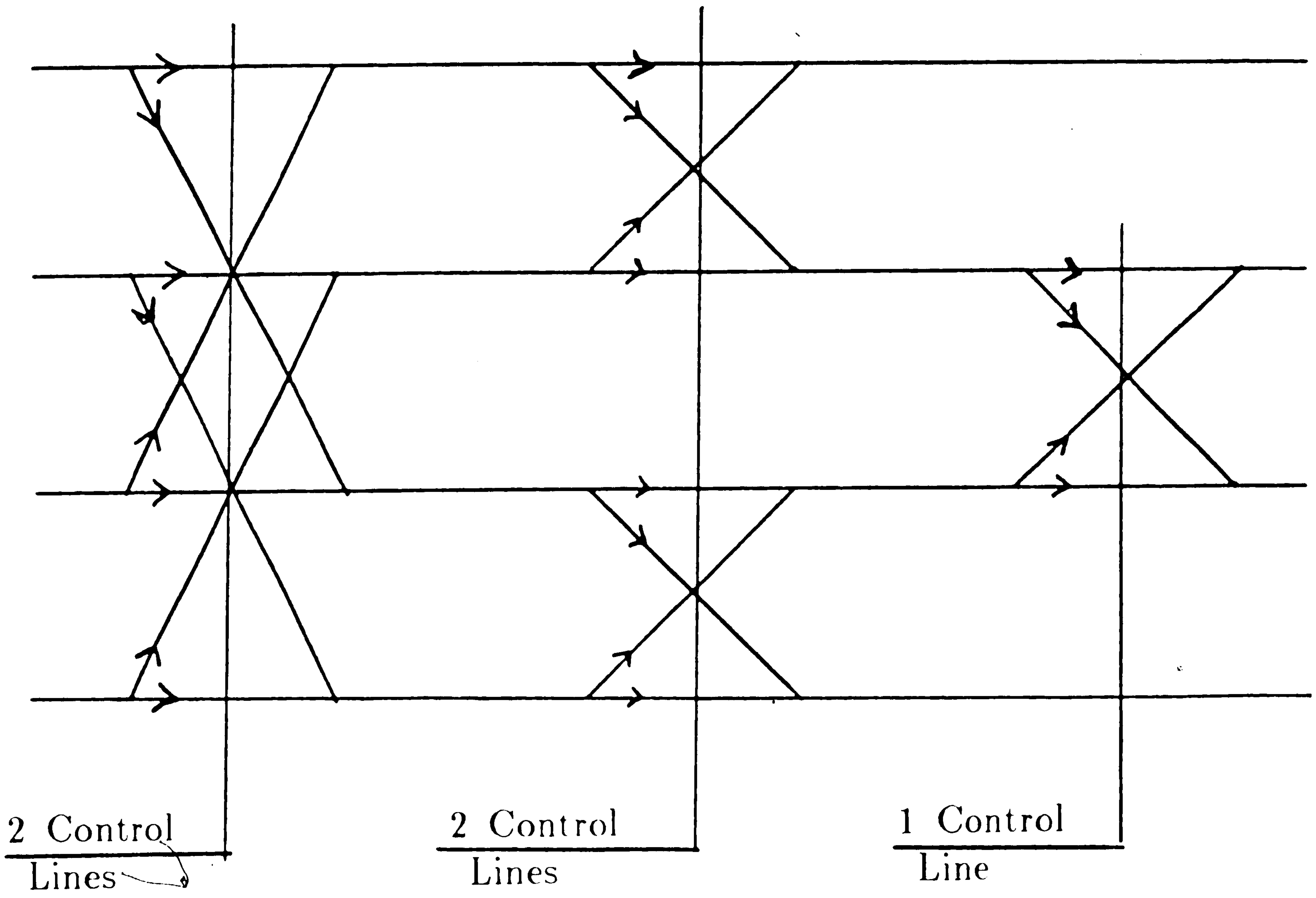


Figure 2-9: Permutation Network, N=4

Each butterfly network consists of two 2:1 multiplexers sharing a common select line (see figure 2-10). Each butterfly network has a unique control line so there are q control lines to the Permutation Network.

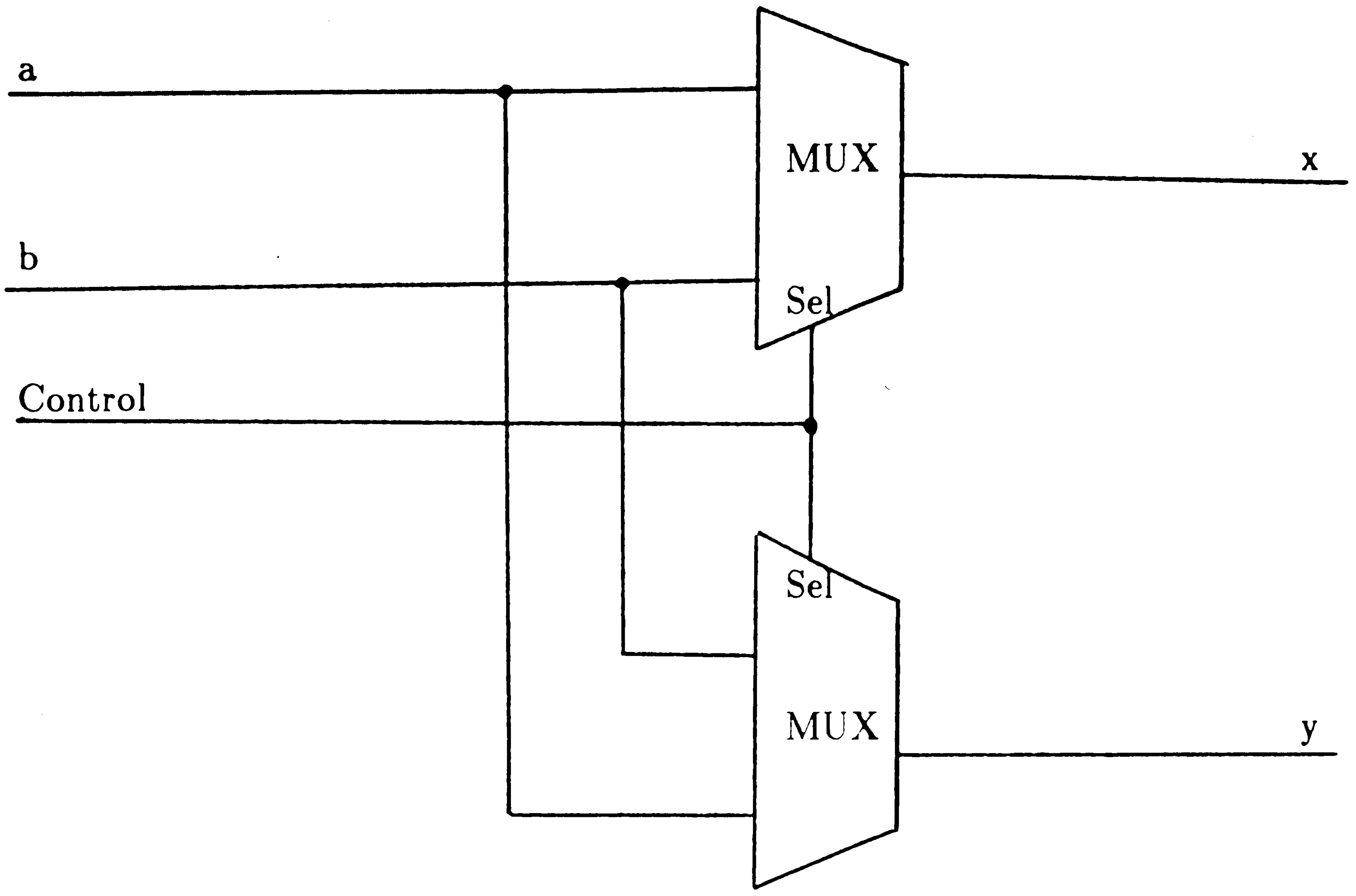


Figure 2-10: Butterfly Network

2.6.3 Output Stager

Figure 2-11 shows one of the modules used to construct the Output Stager circuit. There are $N=2^n$ modules making up the Output Stager circuit.

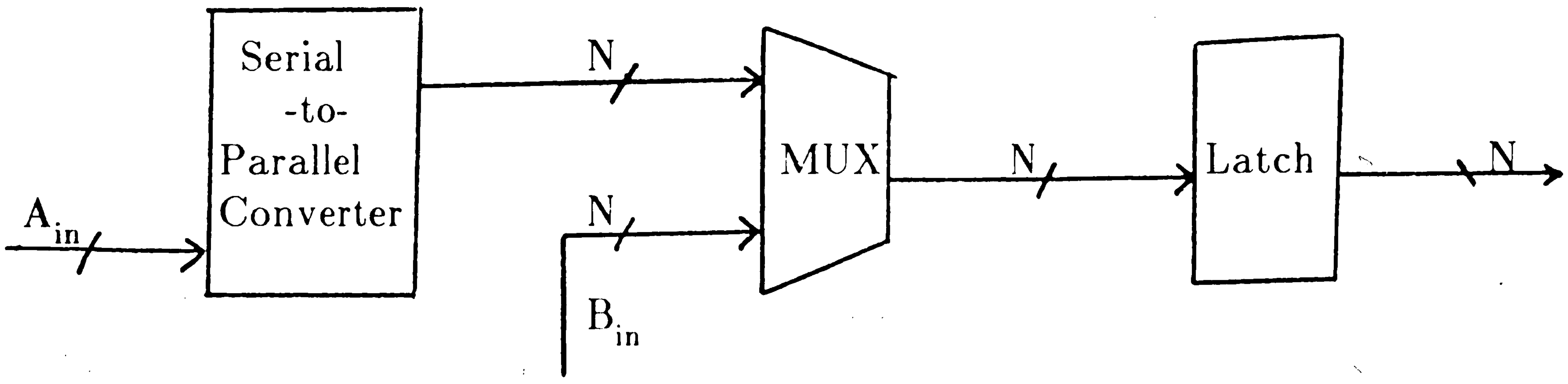


Figure 2-11: Output Stager Module

The basic operation is that data from the Permutation Network, which is primarily serial in nature, is clocked into a module's shift register. When the shift register fills up it is dumped into that module's latch. The data byte in the latch then propagates toward the output by being transferred to the next higher (lower index) module with each clock cycle until it reaches the top (output) module.

These modules are not strictly identical: the serial-to-parallel (shift register) converters are of different degrees of flexibility depending upon the shift register's position relative to the final output latch. This is done to reduce the total hardware complexity without affecting the system's flexibility or expandability. If desired, all the shift registers can be made identical. Modules are interconnected as shown in figure 2-12 with D_{out} of lower modules connected to B_{in} of the module directly above it. A_{in} lines to all modules are from the Permutation Network.

The Output Stager must be capable of handling output data generated by the array in formats analogous to those at the array input. That is, data belonging to one task (that which should be stored together) may be generated from only one row (one bit per clock), from two rows (two bits per clock), etc. up to 2^n rows (2^n bits per clock), where 2^n=N. In addition, the Output Stager should be capable of servicing more than one task being executed concurrently by the array, providing the total number of bits per clock generated by all the tasks does not exceed N.

Figure 2-13 shows the Output Stager for the case of N=2^n=8. When there is one task producing eight bits of output every clock, the top module (with 8 input lines) is the only module utilized and is dumped into $latch_0$ at

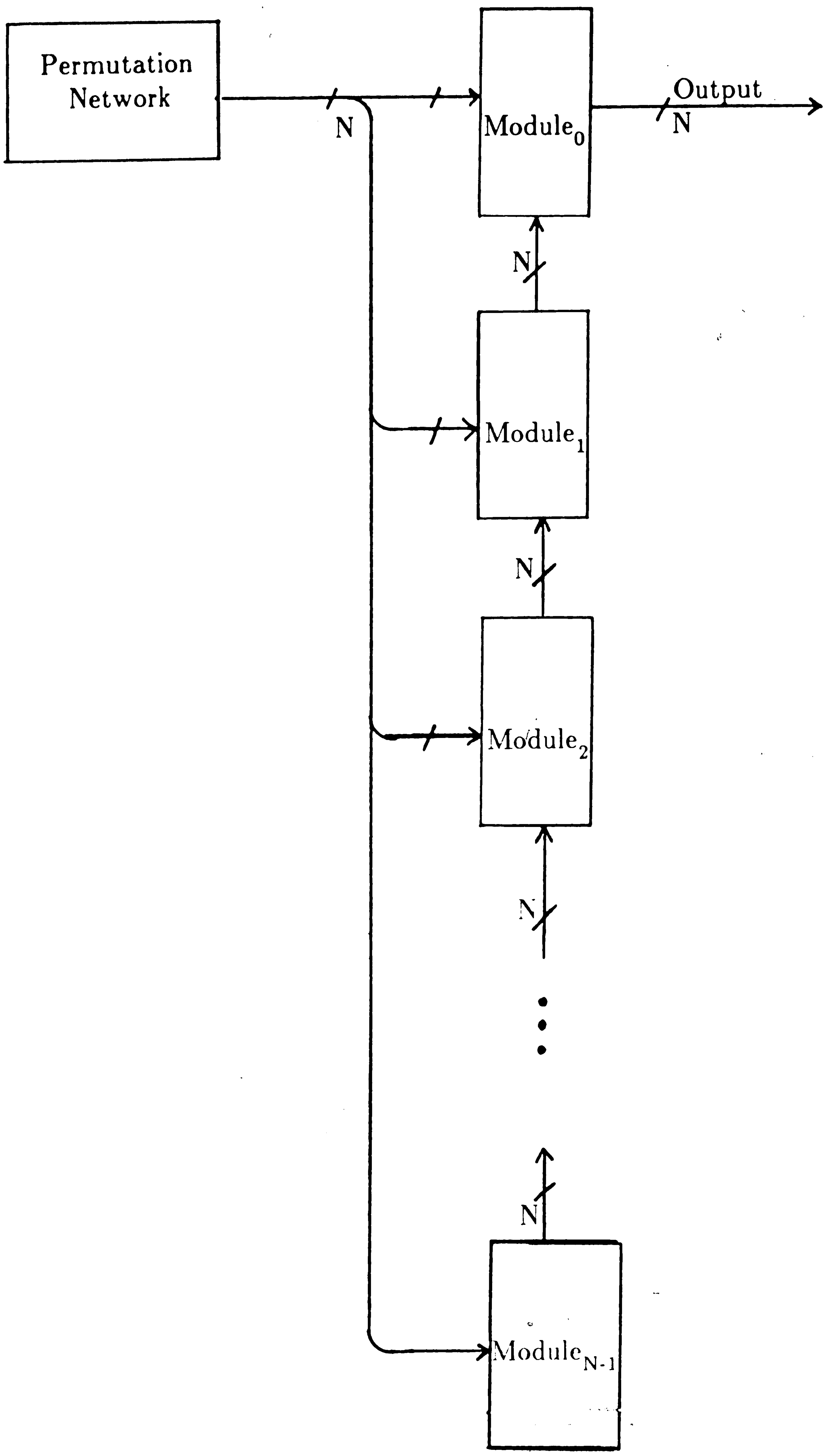


Figure 2-12: Output Stager

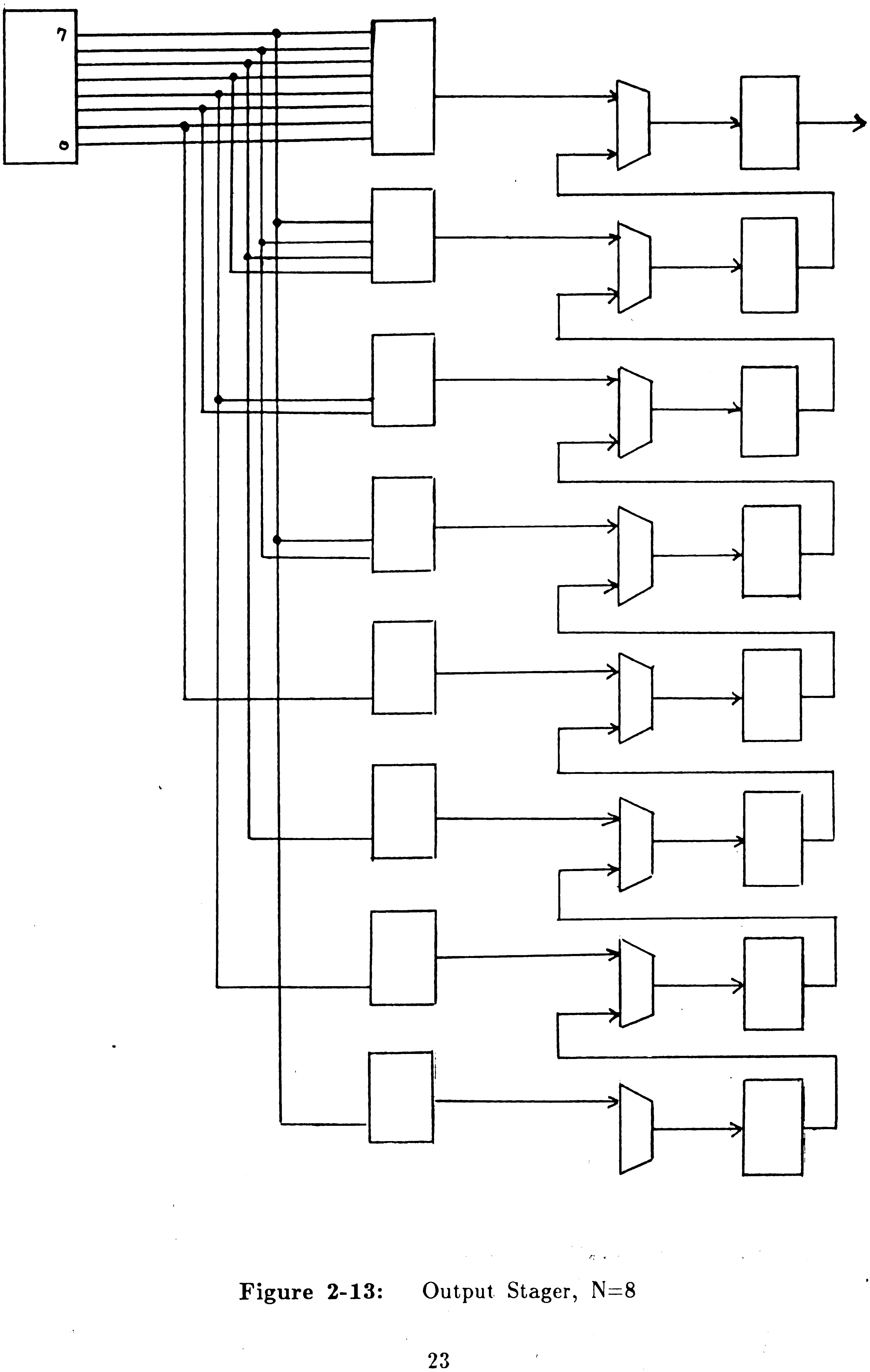


Figure 2-13: Output Stager, N=8

every clock. At the other extreme, if eight separate tasks are running concurrently and each is generating one bit of output per clock then eight modules are active and are being dumped into their respective latches every eight cycles.

The question that then arises is: "If the total number of bits per clock generated does not exceed N, can modules always be assigned such that, for any combination of tasks, all data from those tasks can be collected?" If so, then maximum efficiency is achieved. Since data in the modules' latches propagates upward one module at a time, there is the possibility of "data collision" if a data word from a lower module_{i+1} arrives at module_i at the same time that shift register_i is dumping collected data into latch_i.

Consider as an example a system with N=8 and two tasks being executed concurrently, each producing four bits of output per clock. Thus there are 8 bits of output per clock being generated by the two tasks and, since this does not exceed the word size (N=8, in this example), all data should be collectible. Since each task produces four bits per clock, each task will fill an 8-bit shift register every other clock. Therefore, each shift register will have to be dumped into its latch every other clock. Clearly, then, data from these two tasks should not be routed to modules that are separated by one module (module_0 and module_2 for example). If they are, then when the data originally collected in module_2 propagates to module_0 (2 clock cycles), the shift register of module_0 will be dumping data to its latch and data collision will result. This situation is remedied if adjacent modules (e.g., module_0 and module_1) are scheduled. Actually, data collision in this example will result any time the two tasks' outputs are scheduled for modules whose *distance* (difference in modules' indices) is a multiple of two and no collision will result otherwise.

Obviously, to achieve maximum efficiency (the ability to collect N bits of output per clock), proper assignment of tasks to modules must be employed. Following is an algorithm for scheduling such that, for any combination of concurrent tasks generating no more than N bits of total output per clock, an assignment of tasks to modules results such that all bits can be collected without data collision.

2.6.4 Task Scheduling

Define the weight of a task, w, to be the number of bits per clock generated by that task. We will allow w to take on only values that are powers of 2 (1, 2, 4, 8, etc.). The following algorithm is a means of scheduling tasks to the Output Stager modules in such a way that data collision, as previously described, will not occur.

Task Output Scheduling Algorithm

1. Choose S, the set of available indices, to be {0, 1, 2,... N-1}.
2. Let s be the minimum element of S and w the maximum element of W.
3. Assign task of weight w to module s.
4. Delete w from W and all elements j from S where $j = s + k\cdot(N/w)$.
5. If set W is not empty then return to step 2. Else the required output scheduling is complete.

Before presenting the proof of collision avoidance, the algorithm will first be illustrated with the following example. Consider a case where N=8 and the tasks to be scheduled are: task_1, producing four bits per clock; task_2, producing two bits per clock; task_3 and task_4, each producing one bit per clock.

Therefore, W={4, 2, 1, 1} and S={0, 1, 2, 3, 4, 5, 6, 7}. Tasks are then assigned to modules as follows:

1. The largest w is 4 and the smallest available index is 0. Hence assign task$_1$ to module$_0$. Eliminate indices 0, 2, 4, and 6 yielding W={2, 1, 1} and S={1, 3, 5, 7}.

2. Schedule task$_2$ to module$_1$ and eliminate indices 1 and 5. Now W={1, 1} and S={3, 7}.

3. Schedule task$_3$ to module$_3$ and eliminate index 3. This leaves W={1} and S={7}.

4. Schedule task$_4$ to module$_7$. This concludes the task scheduling.

Thus, the modules used for *collecting* data in this example are: module$_0$, module$_1$, module$_3$, and module$_7$. The remaining modules serve only as links for data word propagation. The reader can convince himself that there are no data collisions with this scheduling.

We now prove that the algorithm described above yields a task assignment with no collisions.

> **Theorem 1:** The Task Output Scheduling Algorithm described earlier gives a schedule to avoid collisions between outputs of tasks.

Proof: Let module$_i$ and module$_j$ be two arbitrary modules. Assume (without loss of generality) that $i < j$. Data words are collected in module$_j$ at times t which are multiples of (N/w_j). These words then propagate through the lower order modules and arrive at module$_i$ at times t given by

$$t = (j\text{-}i) \bmod (N/w_j).$$

However, module$_i$ collects its own data at times t' such that

$$(N/w_i) \mid t'.$$

In order to avoid conflicts between the *collected* data in module$_i$ and the *propagating* data from module$_j$, one must satisfy the inequality

$$(N/w_i)\cdot k \neq (j \;\; i) \bmod (N/w_j) \quad \text{for any } k$$

Since $i < j$, $w_j \mid w_i$, or $(N/w_i) \mid (N/w_j)$. Thus, to avoid collision, one should have

$$(N/w_i) \nmid (j-i)$$
$$\text{or } j \not\equiv i \bmod (N/w_i).$$

However, according to the scheduling algorithm, j is picked from the set of indices which has had indices of the type i mod (N/w_i) already removed. Hence, $j \not\equiv i$ mod (N/w_i) and collision between data from module_i and module_j is avoided. **Q.E.D.**

The proof of Theorem 1 indicates that there would be a collision between data outputs of module_i and module_j if, and only if,

$$(N/\max\{w_i, w_j\}) \mid (j-i).$$

This is avoided by proper association of the modules with data weights in the task assignment algorithm. The next obvious question is "Can all tasks' output be collected using the scheduling algorithm described?" The following theorem answers that question.

> **Theorem 2:** **(Algorithm Coverage)** The task scheduling algorithm presented earlier can schedule every available task provided that $(\sum_{w \in W} w) \leq N$.

Proof: The required result is proved by induction over the cardinality of set W. We show that S is non-empty as long as W is non-empty and, therefore, at any stage one can always associate the smallest $s \in S$ to the largest $w \in W$. In particular, we show that $(\sum_{w \in W} w) \leq |S|$ at any stage of the algorithm. This is true at the starting stage by assumption of the theorem. Suppose it is true at a particular stage and let Ω be the largest element in W at the time. According to the algorithm, assign to Ω the smallest index s available in S (step #3 of the algorithm). This then removes from S all integers of the form $\{s + k\cdot(N/\Omega)\}$ (step #4). But there can be only Ω such distinct integers (for k

- 0, 1, ... Ω-1) since each of them must be less than N. Thus $|S|$ decreases at most by Ω.

However, since Ω is now dropped from W, $\sum_{w \in W} w$ decreases by Ω at the next stage. Note that the assignment Ω to s is compatible with ***any*** future assignment of any remaining $j \in S$ to a remaining $\Omega' \in W$ because non-compatibility (i.e. collision) between module$_s$ of weight Ω and module$_j$ of weight Ω' would imply that

$$(j-s) \mid (N/\max\{\Omega,\Omega'\})$$

But, $\max\{\Omega,\Omega'\}$ is Ω and this would mean $j \equiv s \bmod (N/\Omega)$. Since j belongs to a set of indices obtained by dropping all indices congruent to s mod (N/Ω), the non-compatibility is avoided. **Q.E.D.**

We next consider the problem of minimizing the complexity of each module. In particular, we attempt to design modules with the minimum number of data input lines which will still support all possible task weight sets W. A non-minimal solution is to connect each output line from the Permutation Network to each module. This is undesirable since, as we will show later, the complexity of a module is dependent upon the number of its input lines. For clarity, the solution to the problem will first be stated, followed by a proof of its minimality.

The Output Stager consists of N=2^n modules labeled with indices 0 through N-1. Each of the modules accepts a certain number of output lines from the Permutation Network. Let M_i denote the set of indices of the lines entering module$_i$. These sets are chosen in the very specific manner indicated below for reasons that will become clear later.

Data Collection Network: Each M_i starts with the bit-reversed i and uses consecutive $N/2^{\lceil \log_2(i+1) \rceil}$ integers.

Thus, the first set, M_0, has N elements, the second set N/2 elements, the third and fourth sets N/4 elements, and so on until the final N/2 sets with one element each. For example, if N=8 then

		First	no. of
	Index	Elem	elems.
$M_0=\{0,1,2,3,4,5,6,7\}$	000	000	8
$M_1=\{4,5,6,7\}$	001	100	4
$M_2=\{2,3\}$	010	010	2
$M_3=\{6,7\}$	011	110	2
$M_4=\{1\}$	100	001	1
$M_5=\{5\}$	101	101	1
$M_6=\{3\}$	110	011	1
$M_7=\{7\}$	111	111	1

We now prove that the Data Collection Network presented here is sufficient *and* minimal.

Theorem 3: (Sufficiency of Data Collection Network) The Data Collection Network presented earlier is sufficient to collect data from any set W of tasks (of weights w) provided $(\sum_{w \in W} w) \leq N$.

Proof: In order to show the sufficiency of the data collection network we first prove that for any arbitrary weight distribution set, W, every module can be configured to collect data on distinct output lines from the Permutation Network (i.e., the proposed network does indeed provide distinct lines to distinct modules). If this is true, sufficiency follows because for any W with $(\sum_{w \in W} w) = N$, all tasks together generate N output bits in one clock (if $(\sum_{w \in W} w) < N$ then add pseudo operations of weight w to satisfy this relation). If each module picks up w distinct bits per clock then the network is sufficient for all the modules together to gather all the N distinct bits per clock. This implies sufficiency of the network.

Thus, to prove sufficiency:

Let $N=2^n$. Consider two arbitrary modules i and j of the Output Stager such that, expressed as binary strings,

$$i = [\underbrace{00\ldots0}_{n-p}\underbrace{a_{p-1}a_{p-2}\ldots a_1a_0}_{p}]$$

and

$$j = [\underbrace{00\ldots0}_{n-q}\underbrace{b_{q-1}b_{q-2}\ldots b_1b_0}_{q}].$$

It is now shown that if the Data Collection Network specified earlier is used then module_i and module_j collect distinct bits. Define α and β to be bit-reversed i and j, respectively, i.e.

$$r=a_0a_1\ldots a_{p-2}a_{p-1}, \qquad \alpha=r\cdot 2^{n-p} \tag{2.1}$$

$$s=b_0b_1\ldots b_{q-2}b_{q-1}, \qquad \beta=s\cdot 2^{n-q} \tag{2.2}$$

We now compute the elements of set M_i. Clearly it starts with $r\cdot 2^{n-p}$, (bit-reversed i) and has $N/2^{log_2(i+1)} = 2^n/2^p = 2^{n-p}$ elements. Hence,

$$M_i = \{r\cdot 2^{n-p}+k_i \mid 0 \le k_i < 2^{n-p} \}.$$

Similarly,

$$M_j = \{s\cdot 2^{n-q}+k_j \mid 0 \le k_j < 2^{n-q} \}.$$

Assume (without loss of generality) that $j > i$. Then $|M_i| \ge |M_j|$. The same data element picked up by module_i and module_j implies that an element in M_i is the same as an element in M_j. That is

$$r\cdot 2^{n-p} + k_i = s\cdot 2^{n-q} + k_j \qquad \text{for some } k_i \text{ and } k_j$$

Let $q = p + t$. Then the same equation can be rewritten as

$$s\cdot 2^{n-q} = r\cdot 2^{n-p} + k \qquad \text{where } k = k_i - k_j ,$$

or
$$s = r\cdot 2^{q-p} + k/2^{n-q}$$
$$= r\cdot 2^{t} + c \ .$$

In this last expression,

$$c = k/2^{n-q} < 2^{t},$$

since $k = k_i - k_j \leq k_i < (2^{n-p} = 2^{t})$

Thus the last t bit positions in the binary expansion of s would be determined by c and the remaining n-t by r. The n-bit string for s would then take the form:

$$s = [\overbrace{a_0a_1\cdots\quad a_{p-2}\ \underset{\substack{\uparrow\\ 2^{t}\text{ posn.}}}{a_{p-1}}}^{r}\overset{\substack{2^{t-1}\text{ posn.}\\ \downarrow}}{c_{t-1}}c_{t-2}\cdots\quad c_1c_0]$$

Comparing this string of s with the string in equation (2.2), one gets that, for collision, $a_0{=}b_0$, $a_1{=}\ b_1,\ldots\quad ,a_{p-1}{=}b_{p-1}$, and $c_{t-1}{=}b_{p+0}$, $c_{t-2}{=}b_{p+1},\ldots$ $c_1{=}b_{q-2}$, $c_0{=}b_{q-1}$.

Thus the binary string j may be expressed as

$$j = [00\ldots\quad 0c_0c_1\ldots\quad c_{t-2}c_{t-1}b_{p-1}\ldots\quad b_1b_0].$$

Comparing this with the string for i

$$i = [00\ldots\quad 0b_{p-1}b_{p-2}\ldots\quad b_1b_0]$$

one gets

$$(j-i) = (c_0c_1\ldots c_{t-2}c_{t-1})\cdot 2^{p}.$$

However, note that since $i < j$, $w_i \geq w_j$, and

$$N/\max\{w_i,w_j\} = 2^{n}/2^{n-p} = 2^{p}.$$

Thus, a collision implies:

$$(N/\max\{w_i,w_j\}) \mid (j-i),$$

or $$(N/w_i) \mid (j-i),$$

or $$j \equiv i \bmod (N/w_i).$$

However, according to the task scheduling algorithm presented earlier, j is chosen from a set of indices which have already had elements removed that are congruent to $i \bmod (N/w_i)$. This contradiction shows that elements selected from module$_i$ and module$_j$ are distinct, completing the proof of the sufficiency of the Data Collection Network. **Q.E.D.**

We will now discuss the measure of complexity alluded to in Theorem 3 which is related to the number of input lines to an Output Stager module. We stated earlier that module complexity increases as the number of its input data lines increases. The variable in the module complexity is all within the module's serial-to-parallel converter, since the multiplexer and latch size are determined by N, and is fixed and constant for all modules.

By way of example, we will show that module complexity increases as the number of input lines, l, increases. Figure 2-14 shows the data collection circuit (serial-to-parallel converter) for word size N=8 and number of input lines l=8. For this module, data collection can be one, two, four, or eight bits per clock. Therefore, every cell must be connected to some l_i (to collect eight bits per clock); the four right-most cells to a cell a distance of four to its left (to collect four bits per clock); the six right-most cells to a cell a distance of two to the left (to collect two bits per clock); and the right-most seven cells to the cell to its immediate left (to collect one bit per clock). Thus, for N=8, l=8 one has the following:

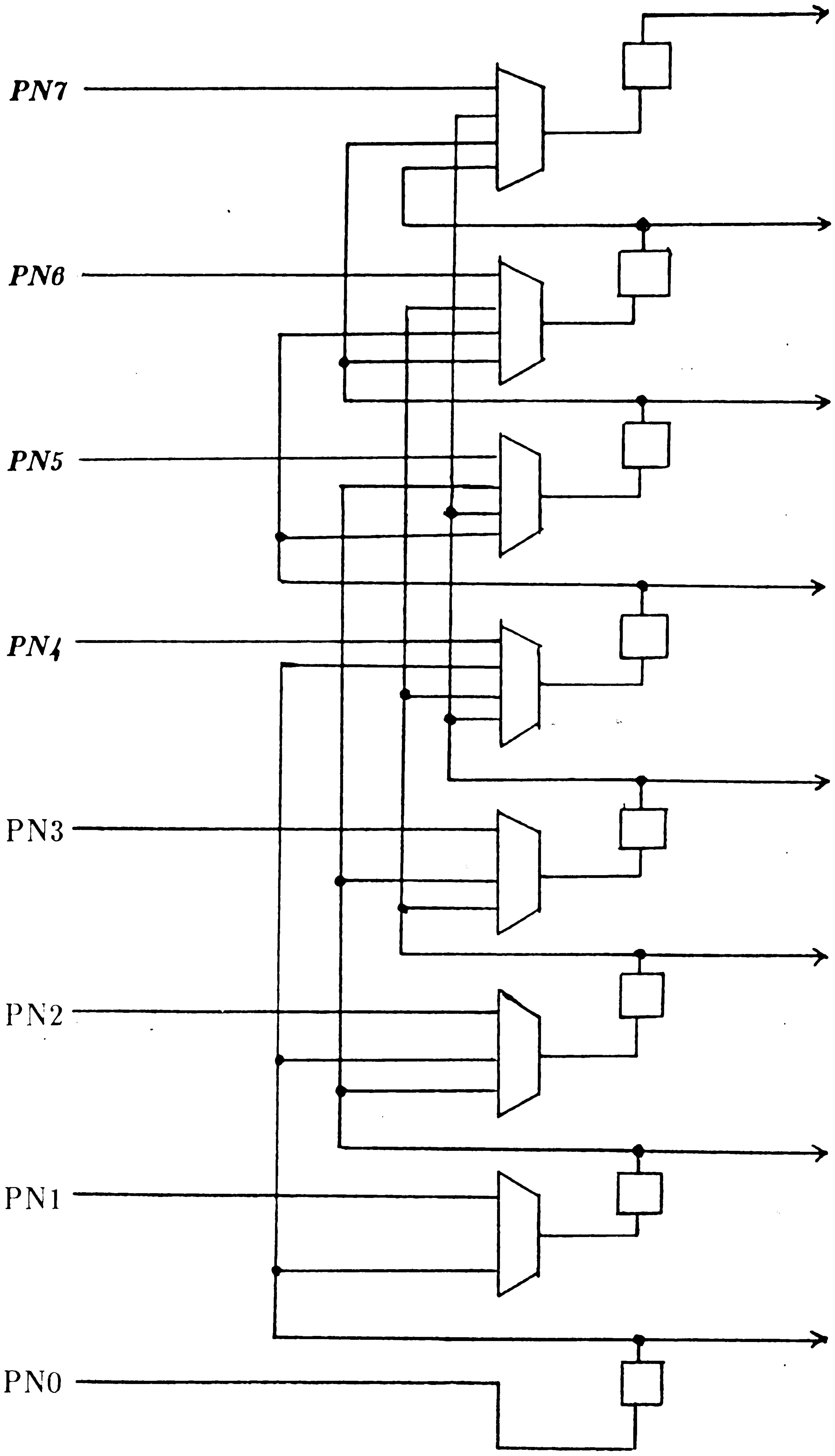


Figure 2-14: Data Collection Circuit: N=l=8

4) 4:1 multiplexer	$4 [4 (\log_2 4 + 1)]$	=	48
2) 3:1 multiplexer	$2 [3 (\log_2 3 + 1)]$	=	16
1) 2:1 multiplexer	$1 [2 (\log_2 2 + 1)]$	=	4
			68

for a total complexity of 68 (as defined for a multiplexer in section 2.5). If the number of lines is reduced to four, then the lines in bold italics in figure 2-14 are not necessary because one never collects eight bits per clock and the complexity is reduced to 51 as the right-most four multiplexers are reduced from 4:1 to 3:1 multiplexers. Similar reductions occur as more lines are eliminated.

Since we have demonstrated the desire to keep the number of input lines to a module to a minimum, we will show that the Output Stager circuit presented is indeed minimal. Notice that the Output Stager circuit consists of modules of the following type:

1	N-input module
1	N/2-input module
2	N/4-input modules
4	N/8-input modules
	⋮
N/2	1-input modules

for a total of N modules altogether. Clearly, at least one N-input module is necessary if a there is a task which produces N bits of output per clock. If there are two N/2-bit output tasks (total N bits per clock) then there must be at least two modules of size N/2 or greater. We use the one N-bit module and ***one*** N/2-bit module. In general, if there are v concurrent tasks producing N/v bits per clock then there must be at least v modules with $l \geq N/v$. For all cases, the Output Stager circuit has ***exactly*** v modules meeting the necessary requirements (the minimum number necessary), giving rise to the following theorem.

Theorem 4: (Minimality of Data Collection Network) The Data Collection Network, consisting of N Output Stager modules, is the minimal network capable of collecting data from any set W of tasks (of weight w) with $(\sum_{w \in W} w) \leq N$.

The need for a Permutation Network to re-order lines in any sequence should now be clear. This flexibility is necessary to route the various tasks to the appropriate Output Stager module for data collection. This allows a minimal Output Stager circuit at the expense of having a Permutation Network, resulting in an overall savings in hardware.

Finally we consider the control of the Output Stager modules. Note that the Permutation Network requires a separate control line for each butterfly multiplexer or $\lceil \log_2(N) \rceil$. On the other hand, the Output Stager requires N/2 distinct control lines for its N 2-word:1-word multiplexers (one in each module). This is because module_i and $\text{module}_{i+N/2}$ $(0 \leq i < N/2)$ will share the same control line. What this means is that both modules are either propagating a word from the next lower module or loading a collected word form its own serial-to-parallel converter. Using a single control line for two modules is permitted as long as no data collision occurs in $\text{module}_{i+N/2}$ as a result. Because of the Task Output Scheduling Algorithm there are three cases that must be considered:

1. neither module is collecting data from the array;
2. module_i is used for data collection but $\text{module}_{i+N/2}$ isn't;
3. module_i and $\text{module}_{i+N/2}$ are both used for collecting data.

Obviously there is no data collision in the first case since neither module is collecting data (the control line always selects data propagation).

In case 2, if module_i is assigned a task of weight w=1 then $\text{module}_{i+N/2}$

would not be eliminated from consideration for task scheduling and, if it is unused, then all modules with a higher index must also be unused because the Task Output Scheduling Algorithm schedules tasks to the lowest indexed module available. If no modules with index greater than i+N/2 are used there can be no data collision in $\text{module}_{i+N/2}$.

If module_i has been assigned a task with weight $w > 1$ then $\text{module}_{i+N/2}$ is not used for data collection but it will still be loading data from its serial-to-parallel converter whenever module_i is, since the modules use a common control line. If this loading in $\text{module}_{i+N/2}$ occurs when a word is attempting to propagate into $\text{module}_{i+N/2}$ then data collision exists. However, by the way tasks are assigned to modules, no valid data word can be propagating into $\text{module}_{i+N/2}$ at time t (when module_i and $\text{module}_{i+N/2}$ are being loaded) because this word would then arrive at module_i at time t+N/2 when it would be loading again. Since the Task Output Scheduling Algorithm precludes this collision, there could have been no valid data propagating into $\text{module}_{i+N/2}$ at time t.

For case 3 to exist both module_i and $\text{module}_{i+N/2}$ must each be collecting data from tasks of weight $w=1$ since, if module_i was collecting a task of weight $w > 1$ then $\text{module}_{i+N/2}$ would *not* be assigned a task according to the Task Output Scheduling Algorithm. Therefore, both modules load from their respective serial-to-parallel converters once every N clocks and there is no data collision.

Notice that, because of the Task Output Scheduling Algorithm, there exists no case 4 with module_i unused and $\text{module}_{i+N/2}$ collecting data since tasks are always assigned to lower indexed modules first and, if module_i had to be eliminated from scheduling consideration because of data collision with a lower indexed module, then $\text{module}_{i+N/2}$ necessarily had to be eliminated, too. Remember that, to prevent data collision, modules a distance of N/w (or a multiple thereof) from an assigned module cannot be assigned a task. Thus if module_i is a distance of $k(N/w)$ away from an assigned module then $\text{module}_{i+N/2}$ is a distance of $(k + w/2)N/w$ (a multiple of N/w) away, too.

Figure 2-15 shows an activity table for N=8 and five tasks of weight $w_1=4$, $w_2=w_3=w_4=w_5=1$. These tasks are assigned (by the Task Output Scheduling Algorithm) respectively to module_0, module_1, module_3, module_5, and module_7. This is an example in which module_2 and module_6 illustrate case 1; module_0 and module_4, case 2; and module_1 and module_5, case 3. In the figure, the presence of an L in a row means that data from the serial-to-parallel converter is entering the module and a P means that data word is propagating. Remember that when module_i is being loaded, so is module_{i+4} ($N/2 = 4$, in this example).

2.7 Memory Utilization

Overall interface efficiency can be further enhanced by selecting RAM memory size to be large enough to service incoming and outgoing data demands simultaneously. Consider, for example, the Input Conditioner & Memory block shown in figure 2-1 containing a *bank* of RAM chips which basically are written to by the host computer and read from by the array of PEs. It is true that the Input FIFO and Input Stager are between the host and the RAM, but these units operate at the same data rate as the host, so it is effectively the host that writes to the RAM bank (as far as data transfer rates are concerned, which is what interests us now). Similarly, the Parallel-to-Serial Converter is

Time Module	0	1	2	3	4	5	6	7	8
module$_0$	L_0	P_1	L_0	P_3	L_0	P_5	L_0	P_7	L_0
module$_1$	L_1		P_3		P_5		P_7		L_1
module$_2$		P_3		P_5		P_7			
module$_3$	L_3		P_5		P_7				L_3
module$_4$	L_4	P_5	L_4	P_7	L_4		L_4		L_4
module$_5$	L_5		P_7						L_5
module$_6$		P_7							
module$_7$	L_7								L_7

L: module being loaded from its serial-to-parallel converter
P: propagating word

Figure 2-15: Data Collection/Propagation

clocked with the same signal as the PE array (the Switch Network is purely combinatorial logic and, therefore, doesn't affect data transfer rates) so it is the PE array that governs the data transfer rate out of the RAM bank. Similar arguments hold for the Output Conditioner & Memory and data transfer rates so that the interface block diagram can be re-drawn, as far as the memories are concerned, as shown in figure 2-16.

Let's focus our attention on the Input Conditioner & Memory and then relate our results to the Output Conditioner & Memory. For the Input, it is likely that data will be read from the memory to the PE array at a much higher rate than will be written to the RAM by the host. Nonetheless, while the array is extracting data from the memory to execute the current task, it makes sense to allow data to be written to the memory from the host as well.

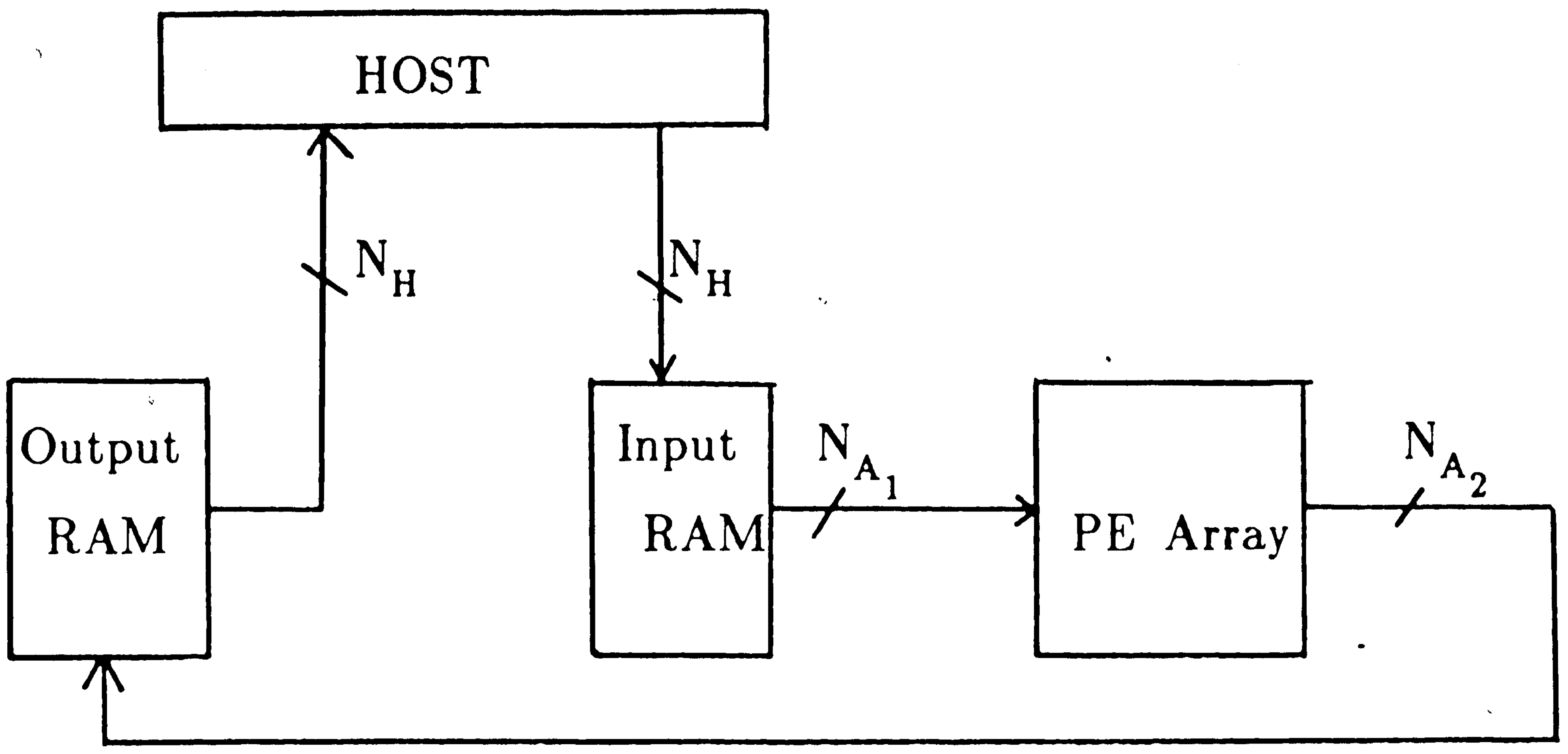


Figure 2-16: Interface Memory Block Diagram

This "simultaneous" read/write capability becomes even more advantageous as the data transfer demands of the host approach those of the array: the memory will be filling at a rate comparable to the rate it is emptying and there will be no "wait" time.

To effect this dual capability, the RAM bank must be of sufficient size (word size, width) to service the array with data as fast as the array needs it and still have idle time (as far as the array is concerned) to service the host's demands to write data to the memory. The following equation determines the number of RAM chips necessary to implement this dual capacity:

$$s_R = \lceil (T_R/N_R) \cdot (N_H/T_H + N_A/T_A) \rceil \qquad (2.3)$$

where

s_R is the number of RAM chips
T_R is the access time of the RAM chips
N_R is the word size of each RAM chip

T_H is the time between words from the host
N_H is the word size of the host (bus)

T_A is the period of the array clock
N_A is the "word size" of the array (the maximum number of lines requesting distinct bits).

This equation is simply stating that the number of RAM chips (s_R) times the data rate each chip is capable of must equal or exceed the data rate demands of the host and array combined. We now know how many RAM chips are needed to meet the demands of the system. What needs to be examined is how the reading and writing is to be scheduled, since the writing is done by "cycle stealing" from the reading and since N_H, N_A, and N_R may not all be equal. Figure 2-17 shows a block diagram of the scheme developed for this.

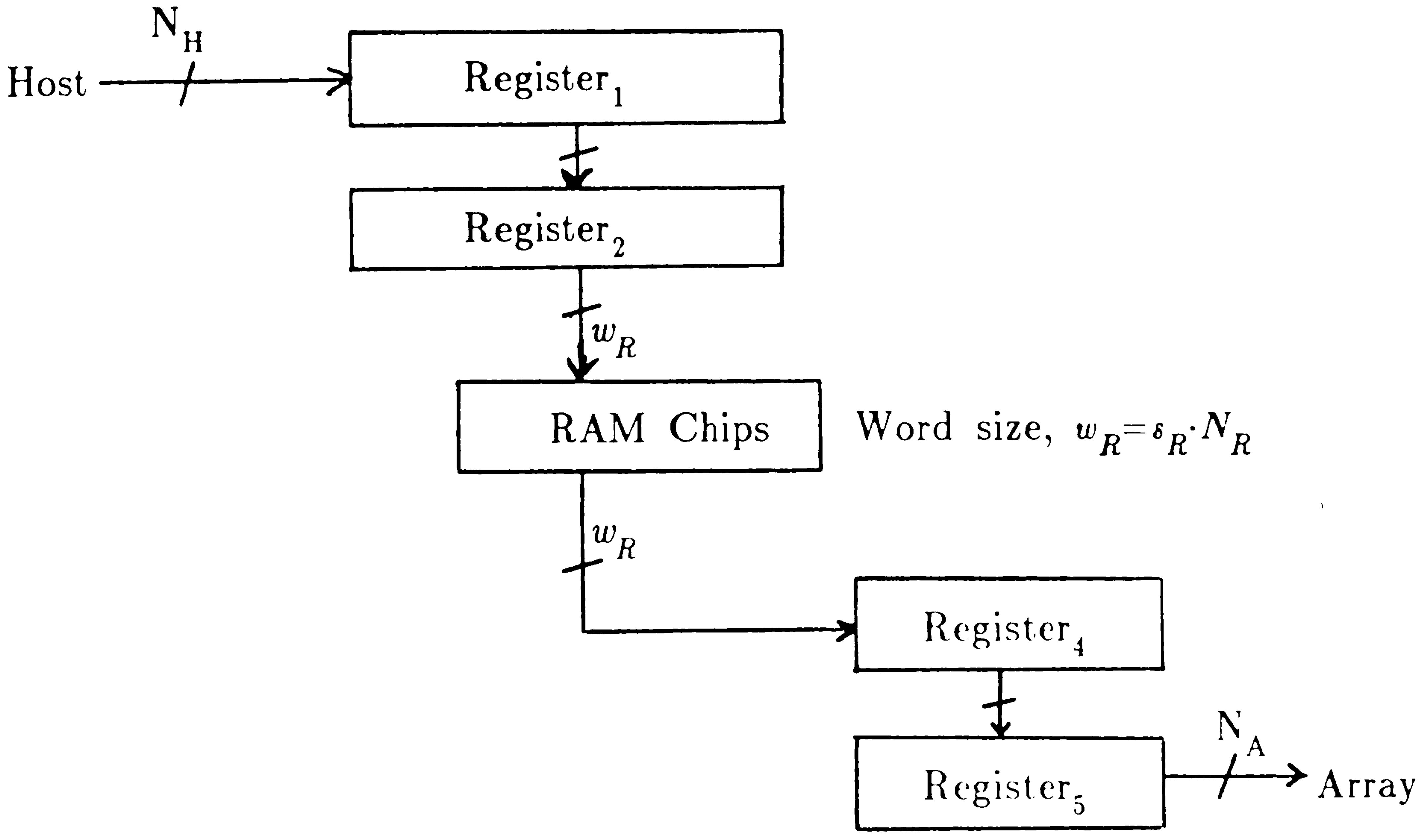


Figure 2-17: Memory Architecture

Notice in figure 2-17 that for both the read and write there is two-level buffering. This is done to match word sizes between the host and the RAMs and between the RAMs and the array. Sizes of the four registers are chosen in a specific manner. Since Register$_2$ and Register$_4$ communicate with the RAM, each is a multiple of the RAM word size w_R ($w_R=s_R \cdot N_R$) though not necessarily the same size. Define s_A and s_H as follows:

$$|\text{Register}_2| = s_H \cdot w_R$$

and

$$|\text{Register}_4| = s_A \cdot w_R$$

The question, then, is how s_A and s_H should be chosen. The answer comes from the fact that the registers are used to match word sizes. Consider the pair Register$_1$ and Register$_2$. We know that Register$_2$ is written to the RAM banks w_R bits at a time. Therefore, both Register$_1$ and Register$_2$ should be a length that is a multiple of w_R. We also know that Register$_1$ is filled N_H bits at a time (size of one host word). Therefore, the length of the registers should be a multiple of N_H too:

$$N_H | w_R \cdot s_H$$

or

$$s_H = N_H / \gcd(N_H, w_R). \tag{2.4}$$

Applying similar arguments for the array and its word size to Register$_4$ and Register$_5$ yields

$$s_A = N_A / \gcd(N_A, w_R). \tag{2.5}$$

The registers' functions are conceptually straightforward. Register$_1$ is filled at a rate of N_H bits (the host's word size) every T_H seconds until it is filled ($s_H \cdot w_R / N_H$ host words). All of Register$_1$ is then dumped to Register$_2$. Register$_2$ is then written to the RAM banks in s_H bytes of size w_R each. Analogously, RAM chips are read into Register$_4$ in w_R bytes until Register$_4$ is full (s_A bytes). Register$_4$ is then dumped into Register$_5$ which clocks $s_A \cdot w_R / N_A$ bytes of size N_A each to the array of PEs.

Because the number of RAM chips was selected in such a manner that the RAMs' data handling capability meets or exceeds the combined demands of the host and the array, we know that the data transfer scheme just described is capable of servicing the host and array "simultaneously". That is, whenever the host wishes to send a data byte, there must be room for it in Register_1 and, whenever the array requests a data byte, one must be available in Register_5. The next section describes a method that might be used by a memory controller to effect this read/write scheduling for the memory.

2.7.1 Memory Control

Certain aspects of register loading are known exactly. Register_2 must be loaded with the contents of Register_1 every $(s_H \cdot w_R / N_H) \cdot T_H$ seconds. This is how long it takes the host to fill Register_1. If Register_2 isn't loaded when Register_1 fills, there will be no room in Register_1 when the next data word arrives from the host, resulting in a loss of data. Similarly, Register_4 must be filled at least every $(s_A \cdot w_R / N_A) \cdot T_A$ seconds, since this is how frequently the array empties Register_5. Failure to meet this demand results in the arrray trying to read data that isn't present.

Read/write control operation can now be discussed. Arbitrarily give the array higher priority than the host as the default condition. That is, memory will normally be accessed by the array to write to Register_4 (until it is filled) unless the host *demands* access to the memory. This demand is the result of a situation presented earlier: Register_1 is filled (or will be filled before Register_2 is emptied into the memory) and needs to be dumped into Register_2 to make room for the next word of data from the host. Under these circumstances, the host will have access to write to the array (remember that all these arguments are presented for data input to the array but are analgous for the output as well). Because the memory was chosen sufficiently large to meet or exceed all data I/O demands, this method of read/write control will work with no access conflicts. Figure 2-18 shows an activity table employing the technique just discussed for the following parameters:

$$\begin{aligned} N_H &= 12 & T_H &= 5T \\ N_A &= 10 & T_A &= 2T \\ N_R &= 8 & T_R &= 3T \end{aligned}$$

which yields:

$s_R = 3$ from equation (2.3)
$s_H = 1$ from equation (2.4)
$s_A = 5$ from equation (2.5)

2.7.2 Optimization of Memory Size

We have seen in equation (2.3) an expression for the minimum number of RAMs necessary to service both the host and the array simultaneously. Upon investigation, it was discovered that this number is not necessarily optimal. Since our goal is an efficient interface, hardware reduction is a prime concern. Define complexity here as simply:

$$C=w_R\cdot(s_A+s_H). \tag{2.6}$$

Calculating complexity for different s_R's, while the other parameters (N_H, N_A, N_R, T_H, T_A, and T_R) are held constant, demonstrated that the optimal number of RAMs is often not the minimum number. Results for two such cases are listed in figure 2-19, where each table begins with the minimum s_R calculated from equation (2.3). Note that if w_R = LCM(N_H,N_A), then $s_A = s_H = 1$ and all registers have the same width as the memory. In most cases, this w_R also yields minimum complexity.

Time	0					5					10					15					20					25					30				
Register$_1$					●					●					●					●					●					●					●
Register$_2$	●										●										●										●				
RAM										●									●						●										
Register$_4$				●			●						●			●						●						●			●			●	
Register$_5$	●																								●										

Time	35					40					45					50					55					60					65				
Register$_1$					●					●					●					●					●					●					●
Register$_2$						●										●										●									
RAM					●						●												●												●
Register$_4$		●						●									●			●						●			●			●			
Register$_5$														●																					

Time	70					75					80					85					90					95					100				
Register$_1$					●					●					●					●					●					●					●
Register$_2$	●										●										●										●				
RAM									●												●									●					
Register$_4$						●						●			●			●						●									●		
Register$_5$			●																								●								

Time	105					110					115					120					125					130					135				
Register$_1$					●					●					●					●					●					●					●
Register$_2$						●										●										●									
RAM				●												●									●									●	
Register$_4$	●						●			●			●						●			●						●			●				
Register$_5$																●																			

Figure 2-18: Memory/Register Activity Table

$N_A = 10$ $N_H = 8$ $N_R = 1$
$T_A = 2$ $T_H = 8$ $T_R = 5$

s_R	w_R	s_A	s_H	C
31	31	10	8	558
32	32	5	1	192
33	33	10	8	594
34	34	5	4	306
35	35	2	8	348
36	36	5	2	252
37	37	10	8	666
38	38	5	4	342
39	39	10	8	702
40	40	1	1	80
41	41	10	8	738
42	42	5	4	378
43	43	10	8	774

$N_A = 10$ $N_H = 8$ $N_R = 8$
$T_A = 2$ $T_H = 7$ $T_R = 5$

s_R	w_R	s_A	s_H	C
4	32	5	1	192
5	40	1	1	80
6	48	5	1	288
7	56	5	1	336
8	64	5	1	384
9	72	5	1	432
10	80	1	1	160
11	88	5	1	528
12	96	5	1	576
13	104	5	1	624
14	112	5	1	672
15	120	1	1	740
16	128	5	1	768

Figure 2-19: Complexity vs. Memory Size

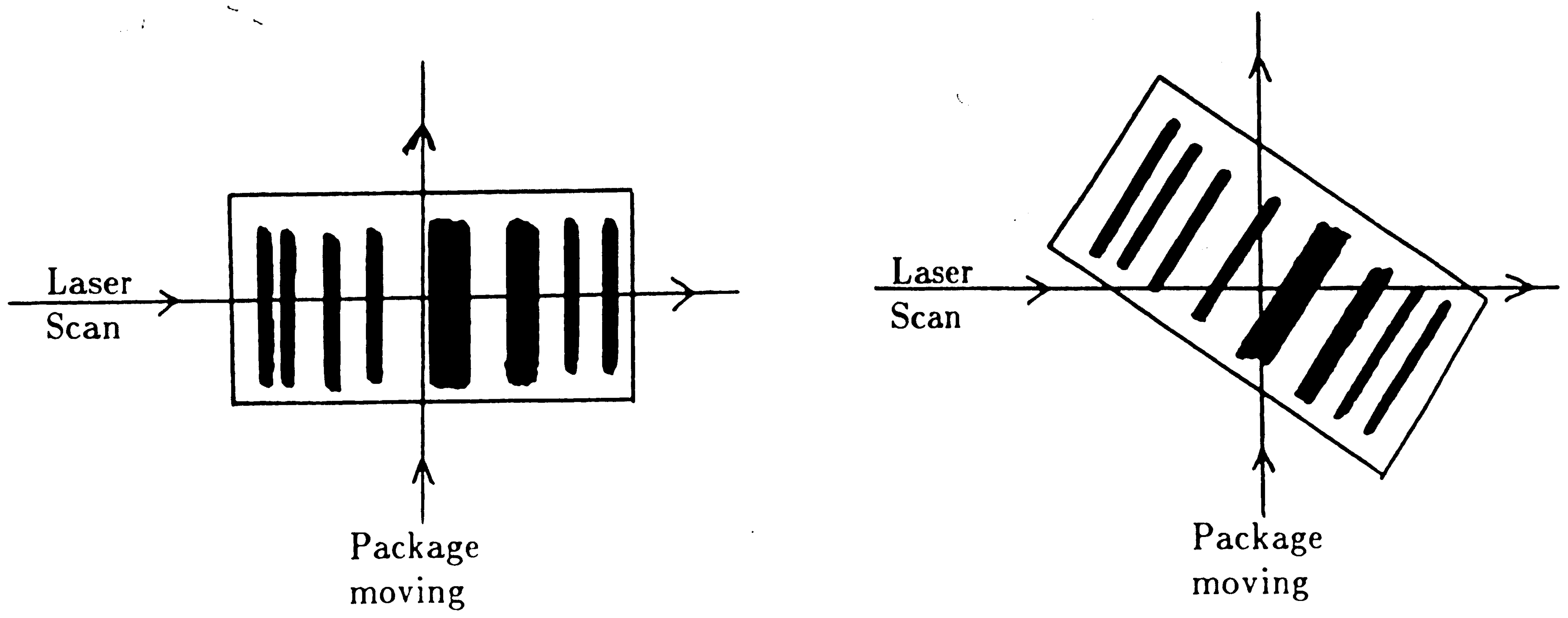
Chapter 3
A Specific Interface Design

3.1 Introduction

This chapter is devoted to the design of an interface circuit to be used in an ongoing research program sponsored by Accusort Inc. This design can be thought of as a specific example of the more general (and complex) interface described in the previous chapter. In this particular design, blocks of the general interface designated as the Input and Output Conditioner & Memory, the Input Stager, and the Permutation Network are not employed. The other blocks comprising the general interface have been incorporated in this design but are less complex than their counterparts of the general interface circuit previously described.

In the project under discussion, the task at hand is to provide an interface so that character strings resident in the host computer's memory can be transferred to a string matching module circuit. The host computer is based on an Intel 8085 microprocessor and the matching module is a bit-serial VLSI custom design [8].

The ultimate goal of the system is to be able to reconstruct bar code labels read by a linear laser scan from packages moving on a conveyor belt. In cases where the laser sweeps through the entire label in one scan, the reconstruction is trivial. However, when no single scan sweeps through the entire label, but only part of it, the label must be reconstructed from the fragments obtained from each scan. Figure 3-1a depicts a situation where a scan will sweep through an entire label. Figure 3-1b illustrates a situation



(a) Laser scans entire label.

(b) Laser scans only part of label.

Figure 3-1: Laser Scan

where no one scan will cut through the entire label. There will be several scans of the label as it moves through the sweep field of the laser providing label fragments with overlap. This overlap is caused by two consecutive scans cutting through some common area of the label.

The matching module is utilized in the reconstruction of the label by matching fragments' overlap. Conceptually, the module will compare a new string (called the A-string) to an existing string (called the X-string) and attempt to concatenate the A-string to the X-string. The strings are three-valued- 0, 1, W- where 0 and 1 are conventional Boolean logic values and W (wild card) can be thought of as a *don't care*. That is, when checking for a match with W the result is always true, whether comparing 0 and W, 1 and W, or W and W.

3.2 Description of Problem

The basic interfacing problem is twofold:

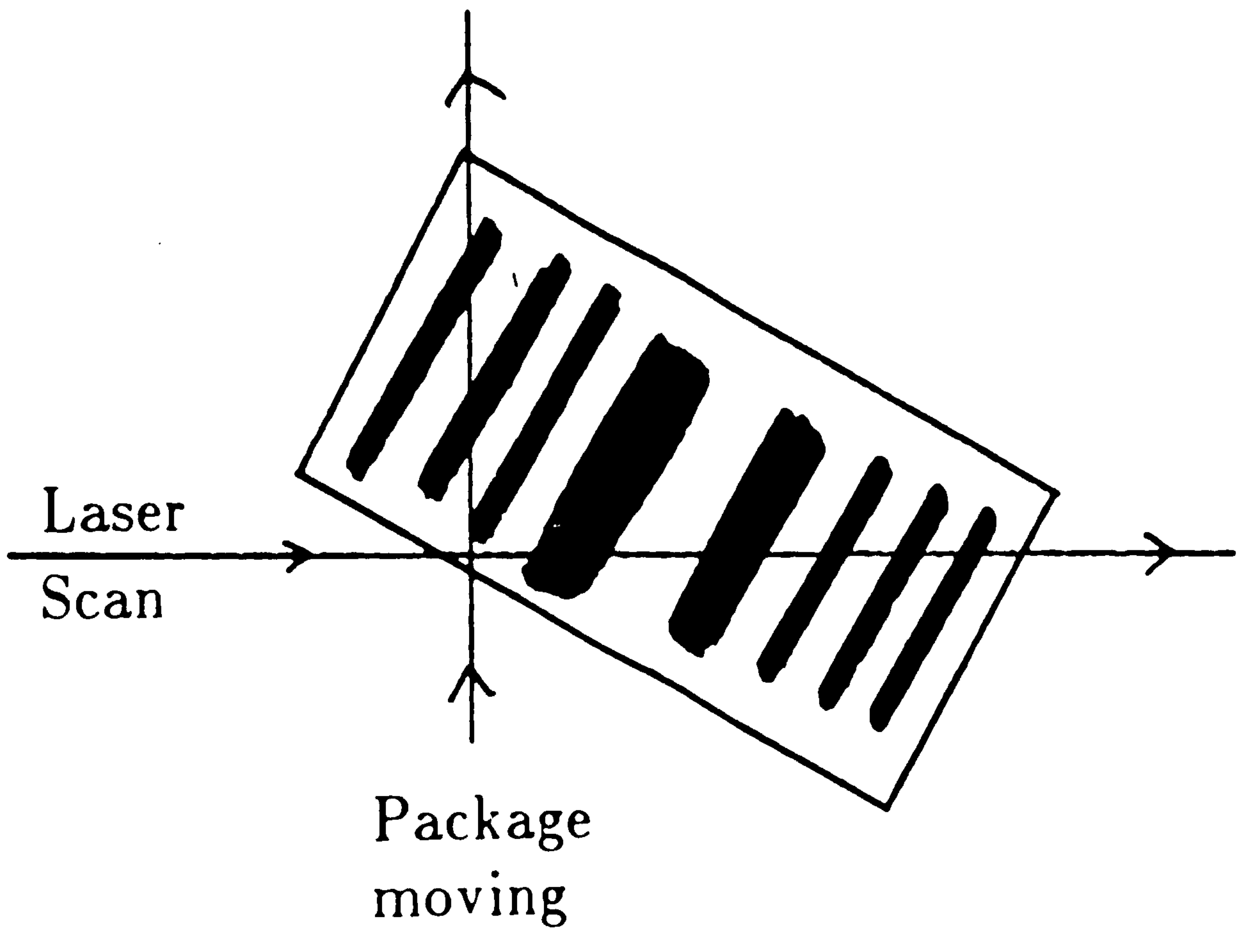
1. that data is available from the host computer in 8-bit words while the matching module needs to be loaded in serial form and

2. the matching module is capable of operating speeds far in excess of the host's ability to make data available.

For these reasons, a FIFO to hold an entire string of characters (maximum of 48 in the Accusort project) and parallel-to-serial shift registers are employed. The manner in which this alleviates the timing problem is simple: the host simply fills the FIFO with all the data as fast as it can and then ***instructs*** the module that the necessary data is now resident in the FIFO. The problem of parallel to serial conversion to load the strings into the matching module is more complicated.

There are two basic loading modes for the matching module: X-string load and A-string load. An X-string load is done when an entirely new label reconstruction process is to commence (there is no current string to be concatenated) and the initial string (X-string) is to be loaded into the matching module. In the case of an A-string load, a partially reconstructed X-string is already resident in the module and this new A-string is to be concatenated to it. Because of the design of the matching module, an X-string load is performed by serially loading one character per clock cycle while an A-string load requires that two characters be loaded per clock cycle. Obviously, implementing an X-string load is the easier of the two tasks.

Each of the strings consists of 1's and 0's representing wide and narrow bars and spaces encountered by the laser scans. Difficulty arises based on the orientation of the label relative to the laser scan. Ideally, one would like the scan to always begin at one end of the label and proceed toward the other end (see figure 3-2).



Scan	Bits read
1st	b_0 b_1 b_2 ...
2nd	b_4 b_5 b_6 ...
3rd	b_7 b_8 b_9 ...

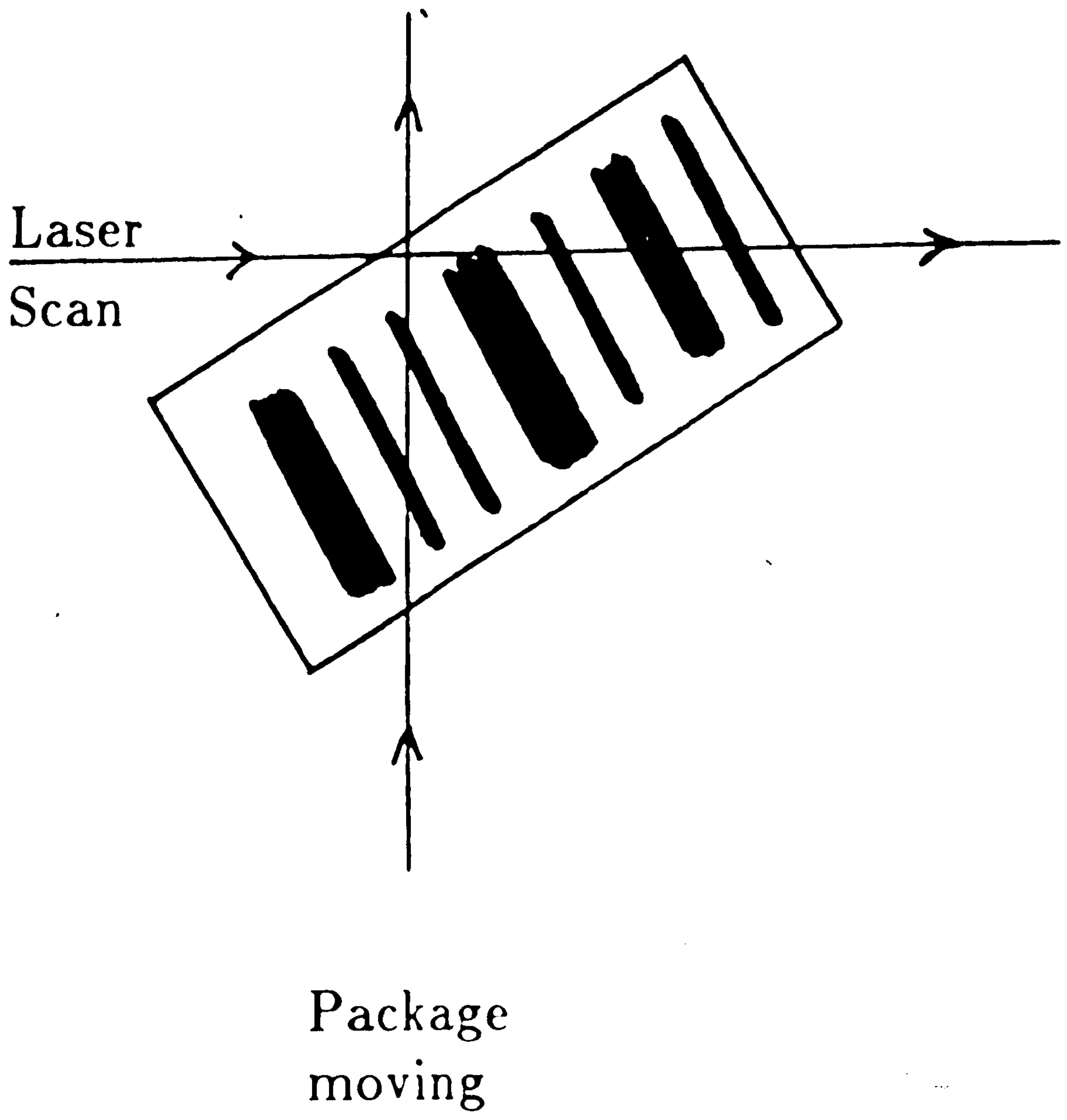
Figure 3-2: Good Scan

However, if the label is angled wrong, the bits will be read in reverse order as shown in figure 3-3. The host computer readily recognizes when a label has been read in the reverse order and will output the data words to the FIFO in proper order. However, it would take the host a prohibitively long time to properly reorder the bits within each byte. Instead, the host signals the interface hardware that this situation exists through a control signal "**Flip**" and the bit juggling is done by the interface.

3.3 Problem Solution

There are four cases of string loading which need to be addressed:

1. Normal X-string load;
2. Flip X-string load;
3. Normal A-string load;
4. Flip A-string load.

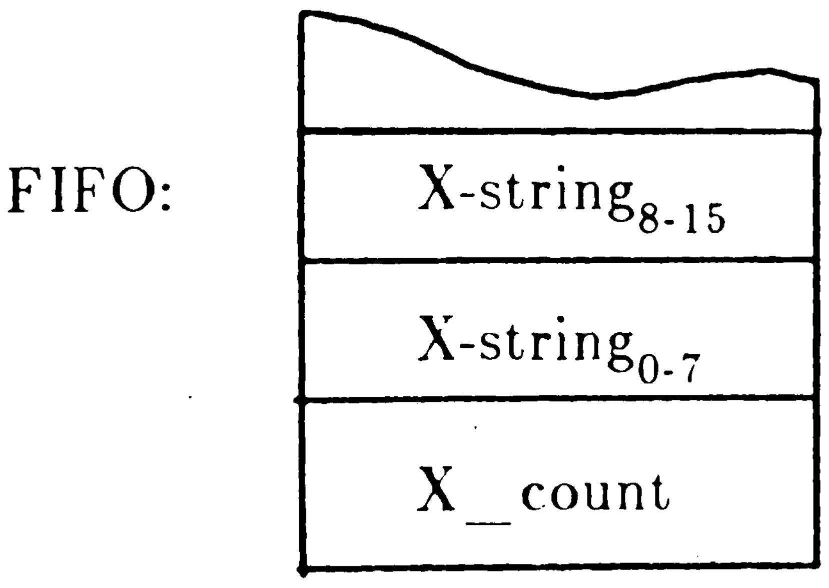


Scan	Bits read
1st	$b_2\ b_1\ b_0$ ...
2nd	$b_7\ b_6\ b_5$ ...
3rd	$b_{10}\ b_9\ b_8$ ...

Figure 3-3: Inside-out Scan

Case 1 is the most straightforward to implement and easiest to understand. It will therefore be discussed first.

For a normal X-string load, the data is resident in the FIFO as depicted in figure 3-4 when the loading is to begin.


FIFO:

X-string$_{8\text{-}15}$

X-string$_{0\text{-}7}$

X_count

Figure 3-4: Data Configuration

The first word in the FIFO is X_count, an 8-bit word whose magnitude is the number of non-wild-card characters in the X-string. The remaining words in the FIFO are the non-wild-card characters (0 or 1) themselves arranged as in figure 3-5.

X_{8n+7}	X_{8n+6}	X_{8n+5}	X_{8n+4}	X_{8n+3}	X_{8n+2}	X_{8n+1}	X_{8n}

$n = 0, 1, 2, 3, 4, 5$

Figure 3-5: Organization of a string in FIFO

Since the X-string is three-valued, there are two X-string input lines to the matching module: X and W_x. When $W_x=0$ the binary value on the X line is the character of the X-string at that time. When $W_x=1$ the character of the X-string is a wild card at that time, regardless of the value of X. The three-valued A-string is similarly implemented.

To load a normal X-string resident in the FIFO into the matching module, the X_count is first loaded into a count-down counter (X_counter) and then the first word of the X-string is loaded into a shift-right register. Also, a count-down counter is loaded with 48, the expected reconstructed string length in the Accusort project. This counter will be referred to as the 48_counter. The shift register, X_counter, and the 48_counter are now all enabled. Additionally, at every eighth clock pulse, the shift register is loaded with the next 8 string bits from the FIFO by a control signal from the microcontroller. W_x is 0 as long as the X_counter is non-zero (indicating there are still non-wild-card characters). When the X_counter reaches 0 it stops counting and W_x=1, indicating that the remaining string characters are wild-cards. When the 48_counter reaches 0 all 48 characters have been loaded into the matching module and the load instruction has been executed.

In the case of a **Flip** X-string load, complications are introduced: the characters in the X-string word are in reverse order and, if the number of non-wild-card characters isn't a multiple of 8, there is some initial set-up of the data to be performed. As an example, let the number of non-wild-card characters be 11 (X_count=11). The string words in the FIFO would be arranged as shown in figure 3-6.

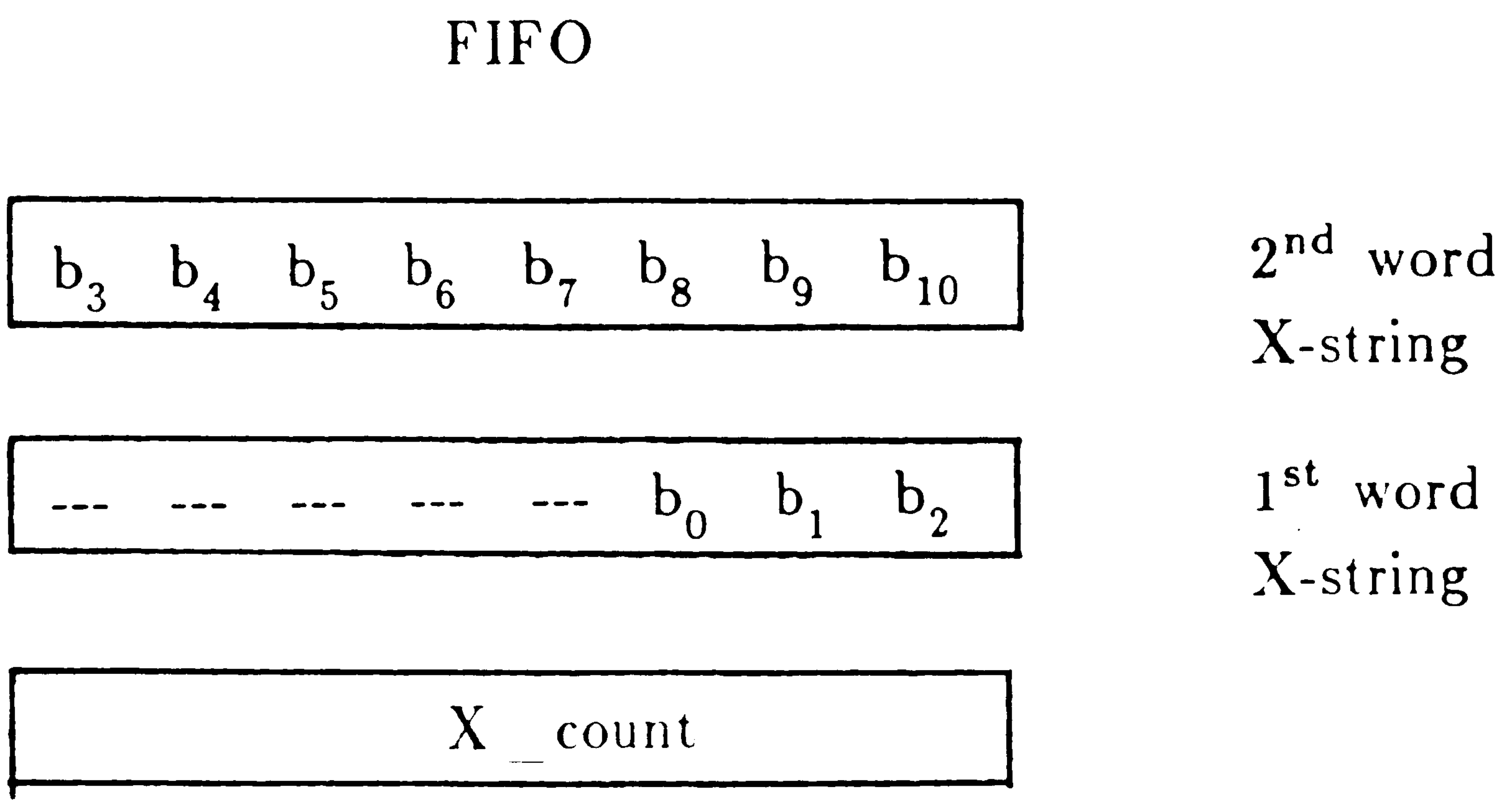

FIFO
b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10}
2nd word X-string
--- --- --- --- --- b_0 b_1 b_2
1st word X-string
X_count

Figure 3-6: FIFO word organization- **Flip**

Clearly, for the second and succeeding words of the X-string, all that needs to be done is to use a multiplexer to route the right-most bit of the FIFO into the left-most bit of the shift register, etc. (see figure 3-7). However, for the first word of the X-string, the shift register contents will have five (in this example) initial junk bits (see figure 3-8) after going through the multiplexer. Therefore, for the first word of the X-string in the case of a **Flip** X-string load, the shift register must be clocked N times (where N=[8-(X_count MOD 8)])

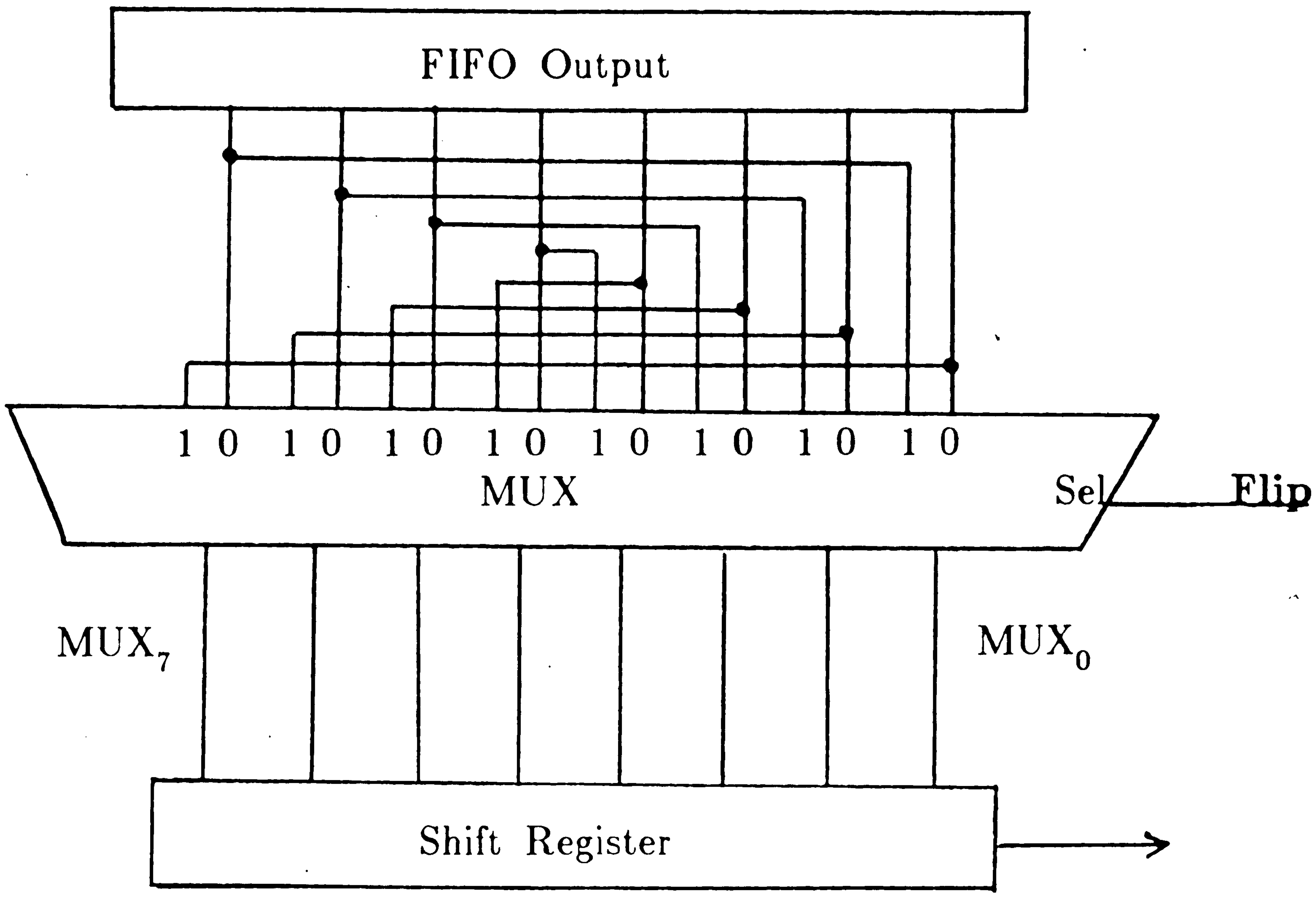


Figure 3-7: Flip MUX

b_2	b_1	b_0	---	---	---	---	---

Shift Register

Figure 3-8: Shift Register

unless X_count is a multiple of 8 in which case ***no*** initial shifts are done. This scheme of initial shifts is implemented through control signals from the microcontroller which looks at the X_count to determine how many initial shifts need to be performed by the shift register. During these initial shifts, the X_counter and 48_counter are disabled. After the initial shifting, the down-counters are enabled and loading proceeds as a normal X-string load with the microcontroller signalling when the shift register is to be loaded from the FIFO (still every 8 clocks, but with an offset due to the initial shifts).

When executing an A-string load, which must load two characters per clock cycle, the problem is further complicated. As before, the normal A-string load will be discussed first since it is more straightforward than the **Flip A-string** load. When performing an A-string load, the FIFO contains data in the same format as for an X-string load: the first word in the FIFO is A_count and the remaining words are the A-string characters. Since the A-string is loaded two characters per clock cycle and there are two lines per character there are four A-string inputs to the matching module: A and W_a each for *upper cells* and *lower cells.*

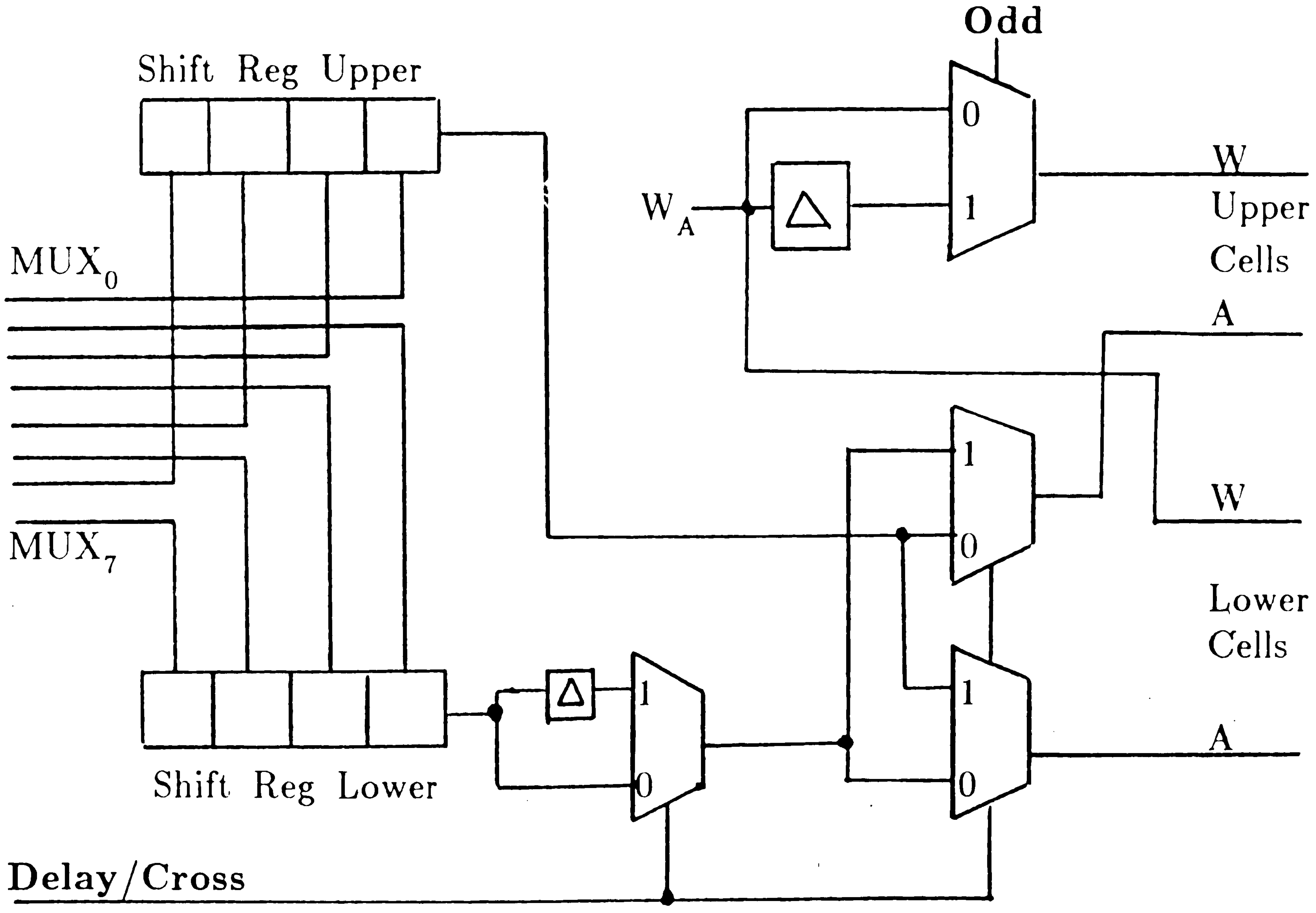
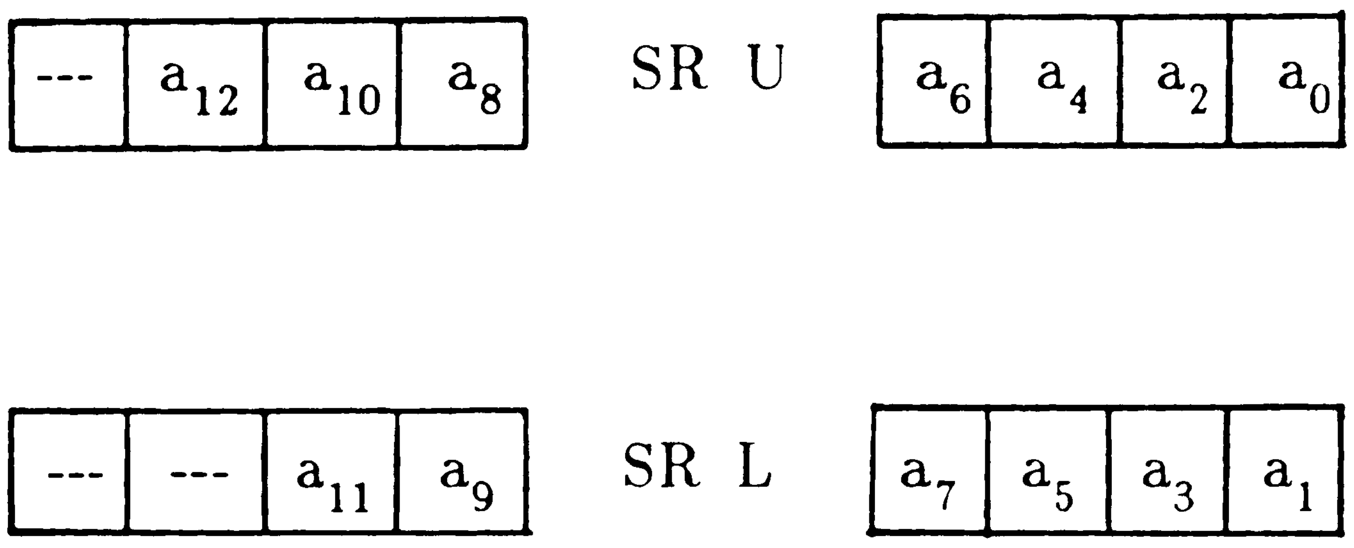


Figure 3-9: A-String Load

Figure 3-9 depicts in block diagram form how the A-string load is implemented. The upper cells of the matching module are to receive characters a_0, a_2, a_4... and the lower cells a_1, a_3, a_5... in that order. Again, as with an X-string load, the normal (not **Flip**) A-string load will be discussed first because it is conceptually clearer.

To simplify discussion, an example with A-string length of 13 will be used to explain how an A-string load is executed. For a normal A-string load the contents of the shift register will be as shown in figure 3-10.


--- | a_{12} | a_{10} | a_8
SR U
a_6 | a_4 | a_2 | a_0
--- | --- | a_{11} | a_9
SR L
a_7 | a_5 | a_3 | a_1

(b) 2nd word

(a) 1st word

Figure 3-10: SR Configuration

For an A-string load the total length counter is loaded with 24 (was 48 for an X-string load) and the A_counter with M (where M=[A_count DIV 2]). This is because two characters per clock cycle are loaded into the matching module. Since this is a normal (not **Flip**) A-string load, the microcontroller sets **Delay/Cross**- the select line of a multiplexer-, to 0 (this MUX will be further explained later). Figure 3-10a shows the first word of the A-string in the shift registers. Everything is now ready for execution and the shift registers, 24_counter, and A_counter are enabled. The microcontroller signals the FIFO to load the shift registers with a new word every *fourth* clock pulse, since two bits are shifted each clock cycle. In the example, after the sixth clock cycle, the shift registers look like figure 3-11 and the A_counter is 0 (13 DIV 2 = 6).

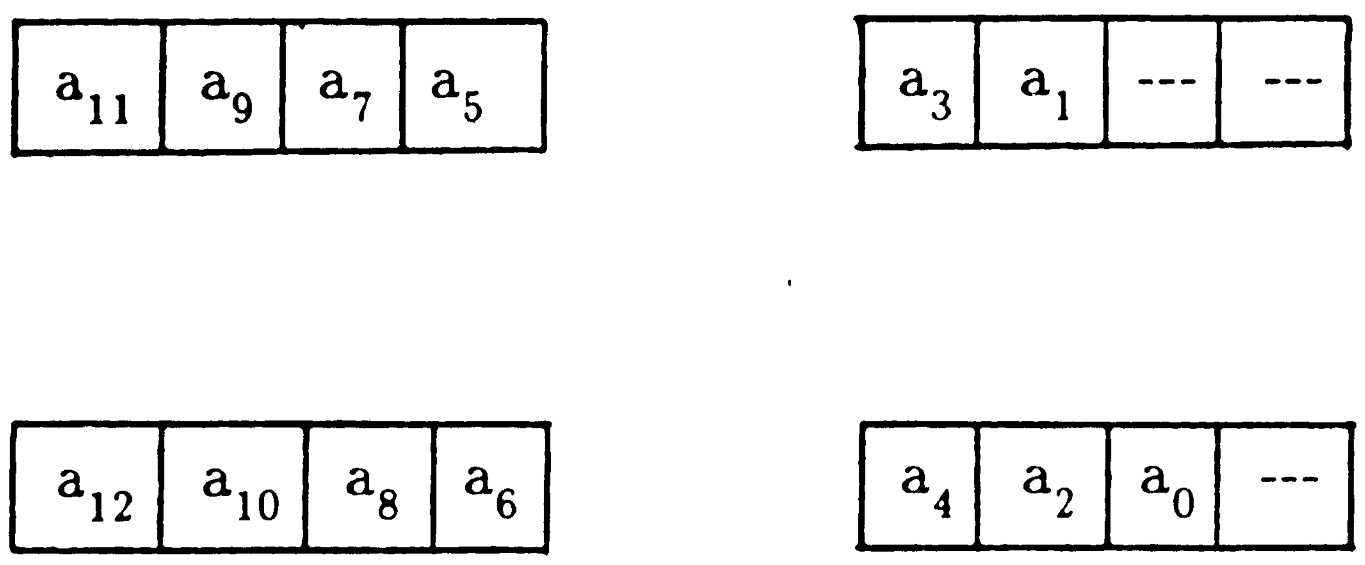
---	---	---	a_{12}

---	---	---	---

Figure 3-11: SR After 6 Clocks

Notice that on the next clock cycle, the lower cells should be loaded with a wild-card character while the upper cells should clock in a_{12}. The wild-card MUX select line should be high (**Odd**=1) to accomplish the necessary one clock delay for W_a to the upper cells (refer to figure 3-9). This delay in wild card generation is necessary any time the number of non-wild-card characters in the A-string is odd. The microcontroller tests the LSB of A_count (first word in the FIFO before load execution) and if it is 1, sets **Odd** to 1 and consequently the delayed W_a is selected for the upper cells by the MUX. Things proceed in much the same manner as for an X-string load: when the A_counter reaches 0 it stops counting and W_a=1. Execution is complete when the 24_counter reaches 0.

In the case where the label is read in reverse order (**Flip** is true) there is a further complexity similar to that found in an X-string load. For our example of 13 non-wild-card characters, figure 3-12 shows the state of the shift registers for the words.



(b) 2nd Word (a) 1st Word

Figure 3-12: SR Initial, **Flip** True

Notice that to prepare a_1 to be loaded into the matching module, the upper shift register must be initially shifted twice. Since the upper and lower shift registers are clocked simultaneously, the lower shift register will be shifted twice also and a_0 will then reside in the one-bit delay register (D in figure 3-9). Remember that when doing an A-string load, a_0 and a_1 are loaded simultaneously, as are a_2 and a_3, etc. and that even-numbered components are to be loaded into the upper cells and the odd-numbered components in the lower cells. For this reason, the **Delay/Cross** control line in figure 3-9 is driven high by the microcontroller when doing an A-string load if both **Flip** and **Odd** are true.

Figure 3-13 demonstrates the loading of our example string from clock pulse 2 through clock pulse 4. As before, $W_a=1$ when the A_counter reaches 0. Also as before, W_a to the upper cells is delayed one clock cycle since the

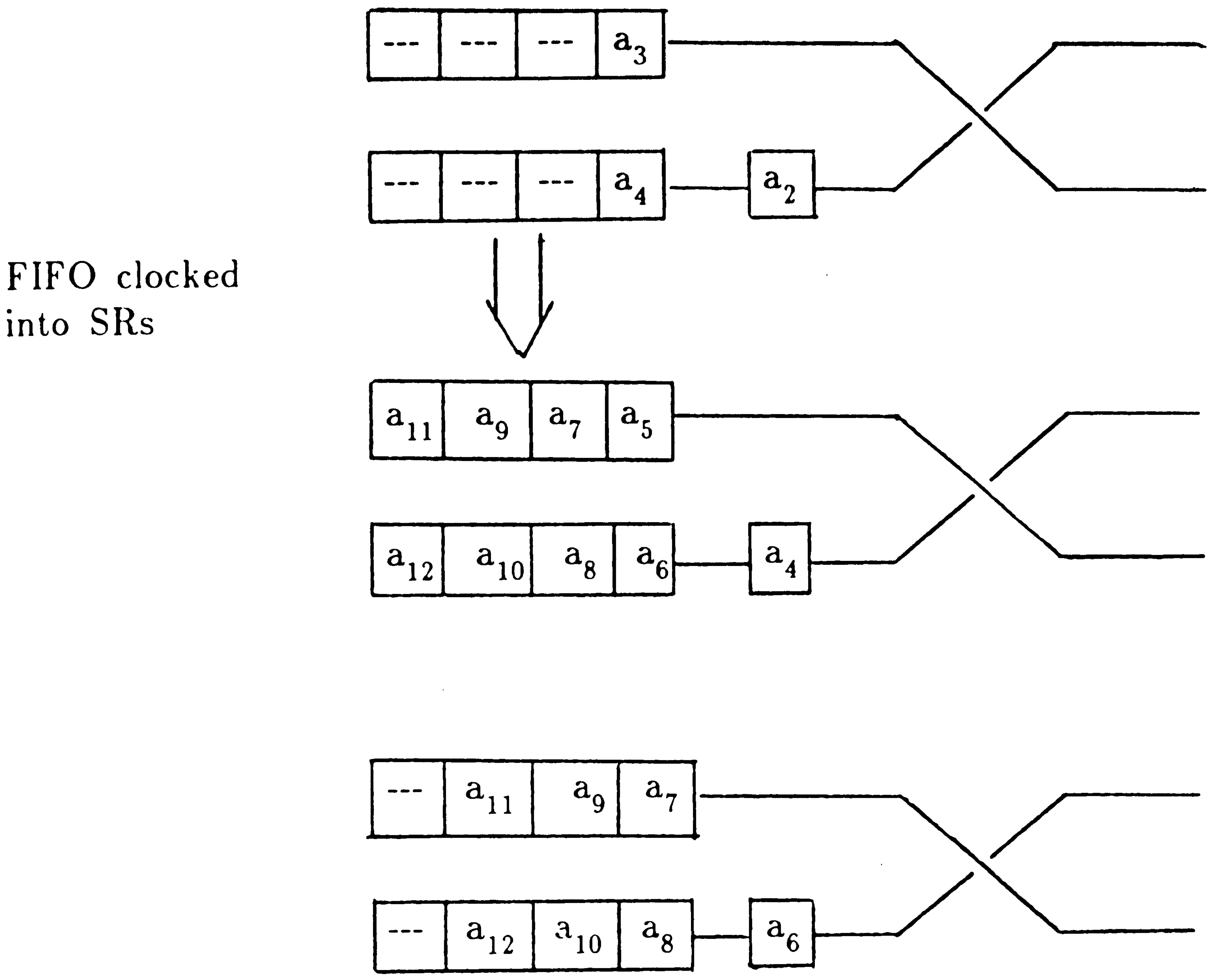


Figure 3-13: SR: Clocks 2 thru 4

string length is odd. Loading is complete when the 24_counter reaches 0.

3.4 Design Principles

There are three functions that the matching module and the interface hardware must perform:

1. begin a totally new string;
2. attempt to add to an existing, partially reconstructed string;
3. remove some characters from the end of an existing string.

The instructions for these three functions are, respectively:

1. Reset;

2. Add;

3. Undo.

The microcode to execute these instructions is resident in memory beginning at locations 04H(Reset), 74H(Add), and FCH(Undo). Examining the binary representation of each (below)

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
04H	0	0	0	0	0	1	0	0
74H	0	1	1	1	0	1	0	0
FCH	1	1	1	1	1	1	0	0

recognize that the last three bits (b_2, b_1, b_0) don't change and that $b_3=b_7$ and $b_4=b_5=b_6$. Therefore, only two bits are needed from the host computer to specify the instruction to be executed if the 8-bit microprogram address is generated as shown in figure 3-14.

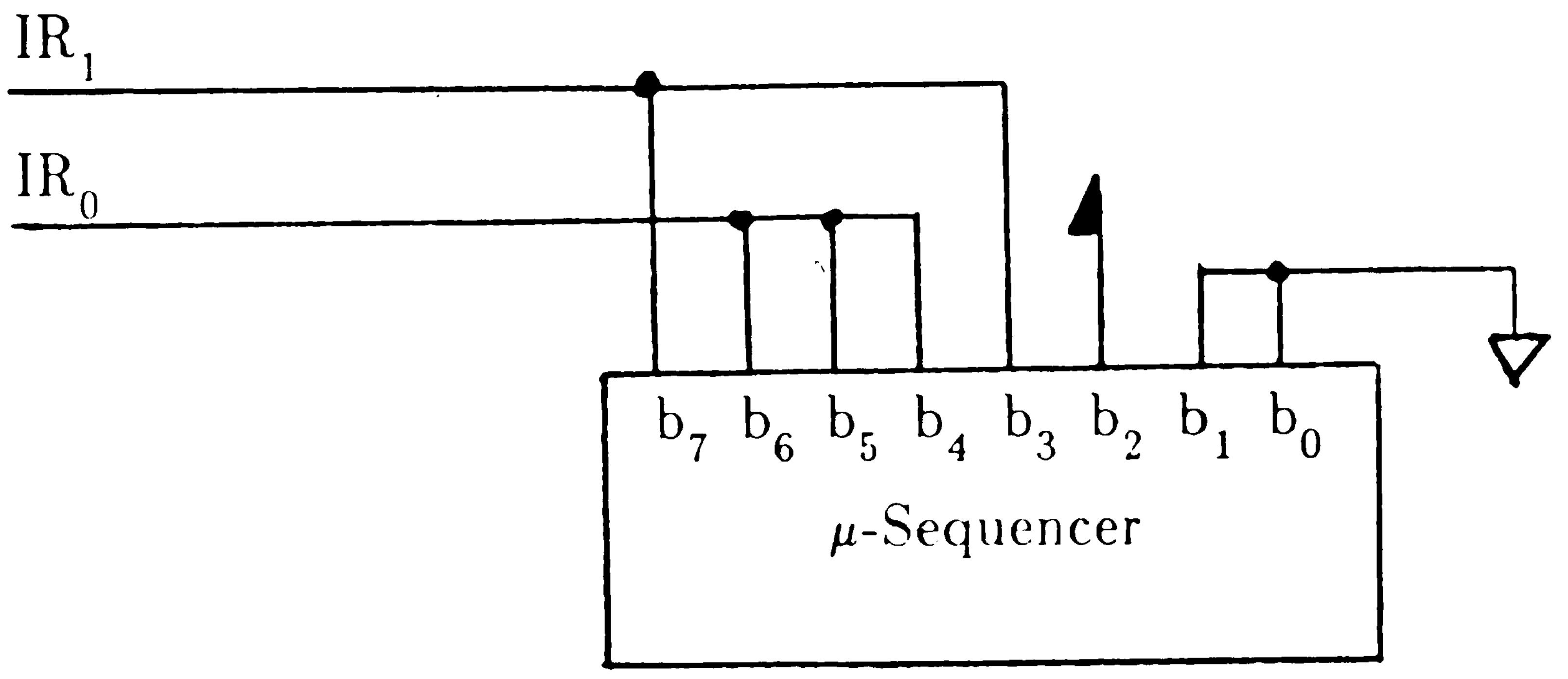


Figure 3-14: Microsequencer

In addition to the two bits of the instruction register specifying what the general instruction is, there are two other bits in the instruction word- **Flip** and **Bar/Space**. **Flip** and its function have been discussed previously- it is an indication of whether or not the laser scanned the label "inside out". **Bar/Space** indicates whether the first character of the A-string is the result of the laser having read a bar (black stripe on the label) or a space between two bars (both bars and spaces contain information about the label). The matching module uses the **Bar/Space** bit to ensure that a **1** (or **0**) of the A-string that corresponds to a bar doesn't get matched up with a **1** (or **0**) of the X-string corresponding to a space between bars and vice versa. A summary of the 4-bit instruction set as implemented is given in figure 3-15.

Four-bit Instruction Codes:

IR_3	IR_2	IR_1	IR_0	
x	0	0	0	Totally new string, characters inside out
x	1	0	0	Totally new string, characters in order
0	0	0	1	A-string to be added, inside out, **Space** first
1	0	0	1	A-string to be added, inside out, **Bar** first
1	1	0	1	A-string to be added, in order, **Bar** first
0	1	0	1	A-string to be added, in order, **Space** first
x	x	1	1	String characters to be removed

Interpretations of Specific Instruction Bits:

IR_3: **Bar/Space** = 0 → First character is a space

IR_2: **Flip** = 0 → Inside-out scan

$IR_{1,0}$: 00 Reset
01 Add
1x Undo

Figure 3-15: Instruction Set

The interface is address mapped by the host computer with very simple handshaking on the multi-bus of the 8085. The individual addresses on the interface are:

1. Input FIFO (data);
2. Instruction register (instruction word);
3. Instruction_in FF;
4. Input FIFO Clear;
5. Undo Counter (length data);
6. Output FIFO (data output to host);
7. Current count (length data output to host).

These addresses are decoded and the necessary enabling signals for the individual devices generated. For 1 through 5, the device is clocked when the handshake signal MWTC from the multi-bus goes low and the handshake signal XACK is then generated by the interface. For 6 and 7, the devices are read when MRTC is driven low by the host. This action does not affect any other function occurring at the time.

3.5 Collecting Results

Valid data begins to propagate from the matching module circuit 24 clock cycles after an A-string load has been completed. This data is the newly reconstructed X and W_x, on two separate lines. As this data is generated, a counter is enabled as long as W_x=0, indicating that the string character at that time is not a wild card. At the completion of 48 clock cycles, the counter value will be the number of non-wild-card characters in the newly reconstructed string.

The X line from the matching module circuit is input to an 8-bit serial-input parallel-output shift register. The contents of this shift register are clocked into an output FIFO every eight clock cycles on command from the interface controller. The FIFO is then read upon demand by the host computer.

3.6 Control

All control signals in the project under discussion (both for the interface and the matching module itself) are generated by a microcontroller. This unit consists of a microsequencer (AM2911), PROMs, and multiplexers. This approach to the design of the control unit was selected because it has the advantage of good speed (faster than a microprocessor), flexibility (easy to expand the number of control signals or change them by simply adding or re-burning EPROMS), and ease of design.

3.6.1 Microcontroller Hardware

The AM2911 microsequencer block diagram is shown in figure 3-16. The control logic in the Accusort project is rather simple, so neither the 2911's Stack nor its Register are used (shaded blocks in figure 3-16). The AM2911 is used simply to generate the next μPC address by incrementing the current address ($\mu PC \leftarrow \mu PC+1$) and selecting either this next instruction address or the address on the D/R lines (a JUMP) depending on whether S_1S_0 are 00 or 11. In this project, the JUMP address is either generated within the microprogram (address has been burned into the PROM) or is external (namely the starting address of our 3-instruction set). Additionally, conditions must be tested to determine whether or not a JUMP should be performed. Figure 3-17 illustrates the microsequencer with the additional support logic.

In figure 3-17, the three bits from the PROM labeled *Branch Select*

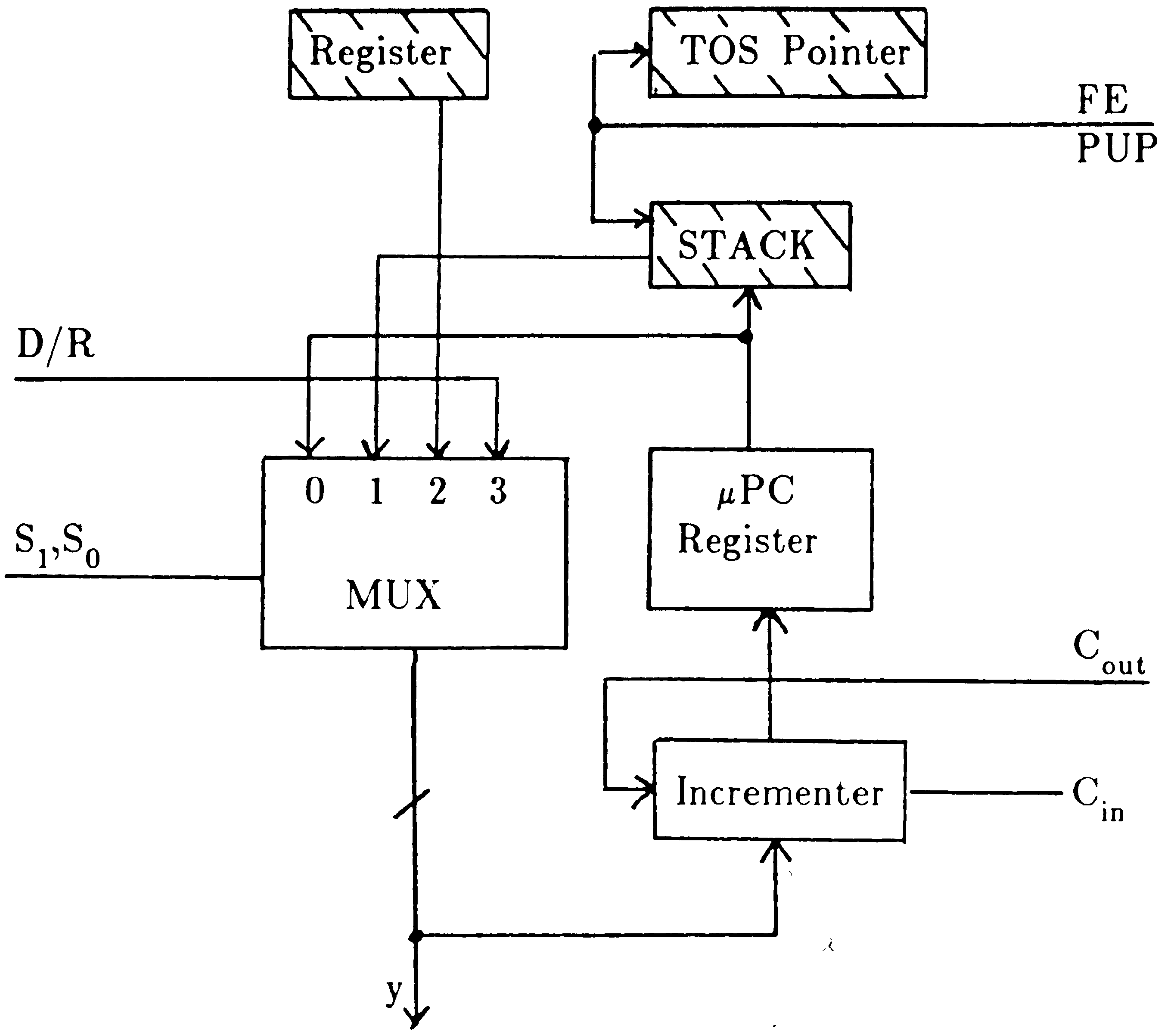


Figure 3-16: μ-Sequencer Block Diagram

determine which condition should be tested. For example, if *Branch Select* is 111 then logic 1 is selected from the test condition MUX and an unconditional JUMP will be executed. The destination address will either be an instruction if *Address Select* is 1 or will be the address in the jump-address-field of the PROM word if *Address Select* is 0. If *Branch Select* is 000 then logic 0 is selected by the test condition MUX and no jump will be executed- the next address will be the current address plus one. If *Branch Select* is 001 through 110 then a JUMP will occur if that condition selected for test is true (logic 1) at the time of the next clock.

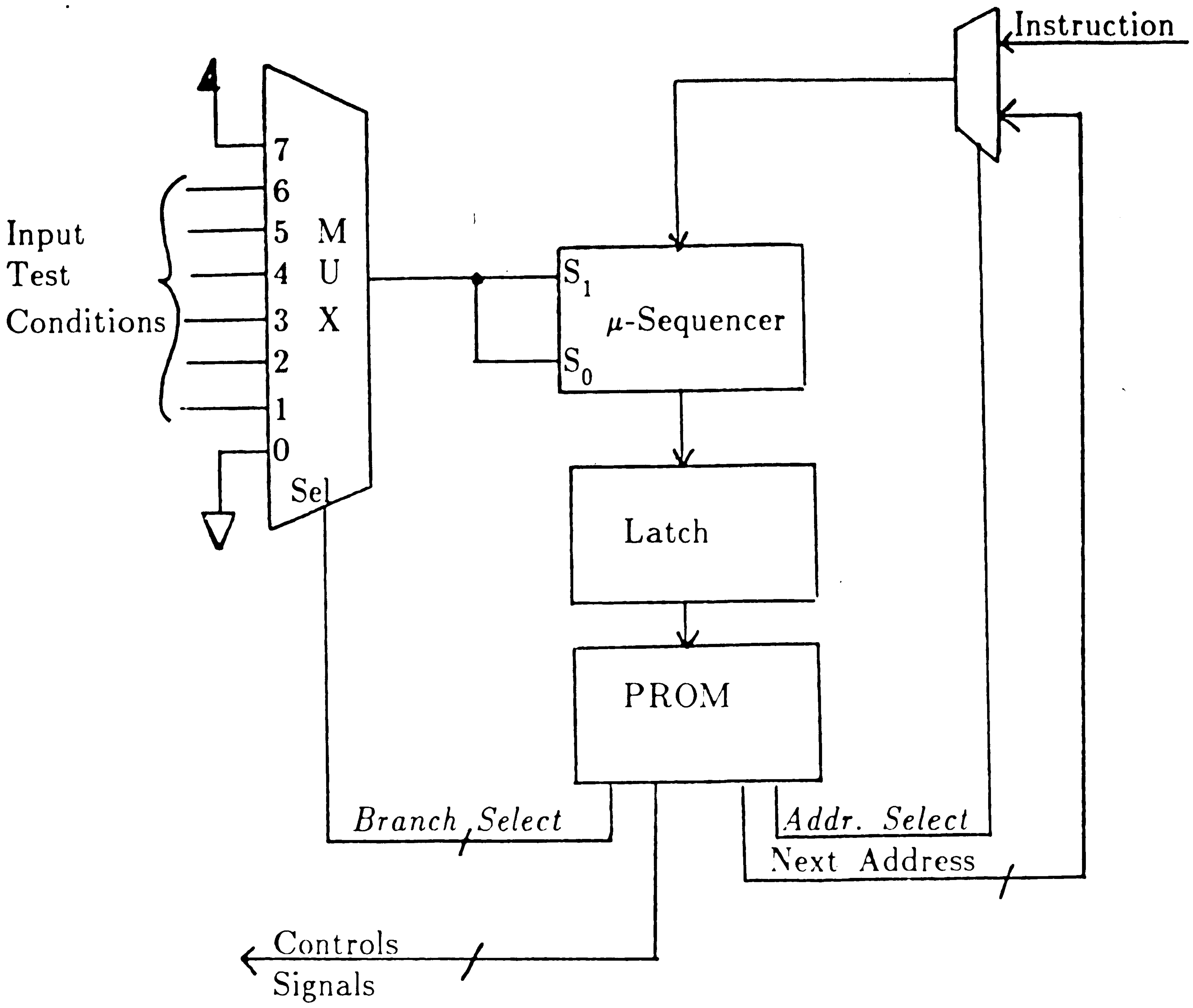


Figure 3-17: μController Block Diagram

3.8 Microsequencer Software

Development of the code for the microsequencer will be discussed in the following order: waiting for an instruction; X-string load; A-string load; Undo. Initial address at power-up and after completion of any instruction is 00H. At location 00H, the Instruction_in-FF bit (001 *Branch Select*) is tested and, if it has been set (indicating that there is an instruction to be executed), a JUMP to the instruction address (either 04H, 74H, or FCH) is taken. Otherwise, the next address is 01H which simply jumps unconditionally back to 00H. Until the microcontroller is informed that an instruction is to be executed, the μPC waits in the 00H to 01H loop.

The X-string load instruction routine begins at 04H. Here two counters are loaded: **Scratch** with the 3 LSBs of X_count (remember that X_count is the number of non-wild-card characters in the X-string) and **6_count** with the value six.

The **6_count** counter is responsible for counting the number of times that an X-string load software loop has been executed (each time through the loop loads one 8-character data word into a shift register and clocks the FIFO for the next word). When the **6_count** has reached 0, all 48 characters of the X-string have been loaded and the X-string load instruction is completed. The program will then jump back to 00H to await the next instruction.

The **Scratch** counter is used when a string has been read "inside-out" (described earlier in this chapter) and the number of non-wild-card characters is not a multiple of 8. As previously mentioned, the first word of characters must be initially shifted [8-(X_count MOD 8)] times for this case. Since **Scratch** is loaded with the 3 LSBs of X_count (this equals X_count MOD 8), the first data word is shifted and the **Scratch** counter incremented until **Scratch** becomes eight. The first word has then been prepared and X-string loading can commence.

The program will now jump to a load routine related to the three LSBs of X_count. Different routines are necessary because the second data word must be fetched from the FIFO at different times for different X-string lengths (all this applies *only* when **Flip** is true). For example, if X_count is 37, then there are five characters in the first data word (37 MOD 8 = 5) and, after X-string loading has commenced, the second data word must be fetched after five clock cycles. On the other hand, if X_count is 19 then the second data word must be fetched after three clock cycles (19 MOD 8 = 3). In all cases, after the first word, future data words must be fetched every eight clock cycles.

An example of the "code" for an X-string load (for X_count MOD 8 = 5) is shown in figure 3-18.

```
Begin:   Enable X_count, X-shift
         Enable X_count, X-shift
         Enable X_count, X-shift
         Enable X_count, X-shift
         Enable X_count, X-shift; fetch next word
         Enable X_count, X-shift
         Enable X_count, X-shift
         Enable X_count, X-shift, 6_count; jump to Begin if $TC_6 \neq 0$
         Routine done; jump to 00H for next instruction
```

Figure 3-18: X-string load routine

Notice that the loop that is repeated is eight lines long. This means that a new data word will be fetched on the 5th, 13th, 21st, 29th, etc. clock cycles. In other words, after the initial word is loaded, each succeeding word is fetched every eight clock cycles until all words have been fetched. Figure 3-19 shows the flow chart for an X-string load.

The software for an A-string load is structured in much the same manner as an X-string load. There are some differences, however. For instance, since the A-string is loaded two characters per clock cycle, a new data word must be fetched from the FIFO every fourth clock cycle rather than every eighth as with an X-string load. Just as with an X-string load, no initial shifting of the first word of the A-string is necessary if the label was not scanned inside out or if the length of the string is a multiple of eight.

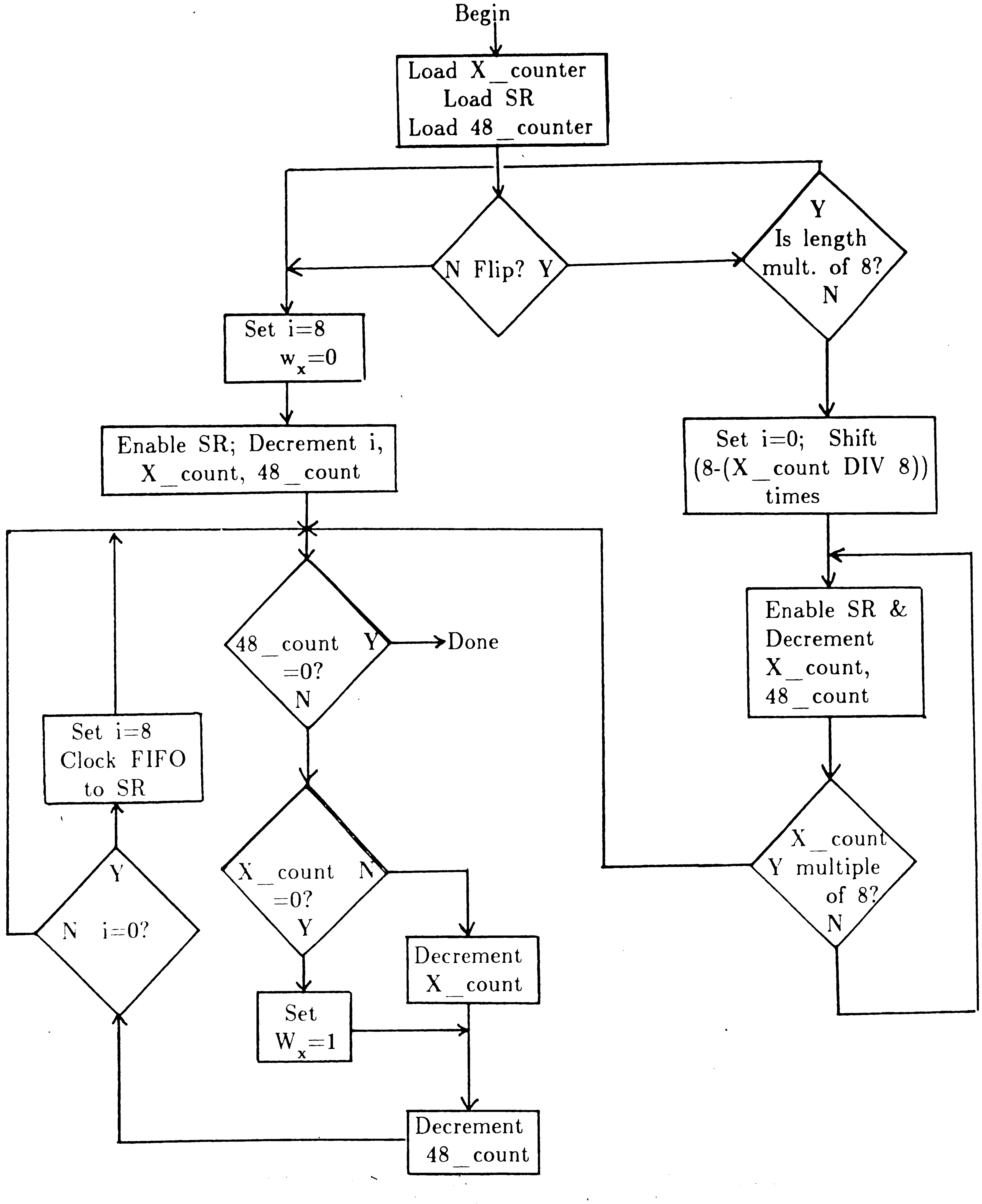


Figure 3-19: X-string load Flowchart

If **Flip** is true (label scanned inside out) and the string length (A_count) is not a multiple of eight then the first word of the A-string in the shift register must be shifted initially (refer to figure 3-12). The number of initial shifts to be performed is P, where P = [4 - (A_count MOD 8) DIV 2]. To perform this, the **Scratch** counter is loaded with A_count_2,A_count_1 (this is P). When the **Scratch** counter reaches four, the initial shifting is done and the A-string load can proceed.

For an A-string load, the **6_count** is again loaded with the value six and the software loops are each four lines long. Since two characters are loaded per line, six times through the loop will load all 48 characters. Figure 3-20 gives the code for loading an A-string where P is 2. The flowchart for an A-string load is shown in figure 3-21.

```
Begin:  Enable A_count, A-shift
        Enable A_count, A-shift; fetch next word
        Enable A_count, A-shift
        Enable A_count, A-shift, 6_count; jump to Begin if $TC_6 \neq 0$
        Jump to 00H for next instruction
```

Figure 3-20: A-string load routine

An Undo instruction is executed when the host computer recognizes that some part of the reconstructed string does not belong where placed. It then requests that all characters in the reconstructed string after a certain number (Undo_count) be converted to wild-cards, effectively erasing these characters. The implementation is rather simple- the host computer memory writes the Undo_count to a down counter and, when this counter reaches 0, the remaining (48 - Undo_count) characters are made wild-cards. When finished, the program again returns to 00H to wait for the next instruction. A flowchart for an Undo instruction is given in figure 3-22.

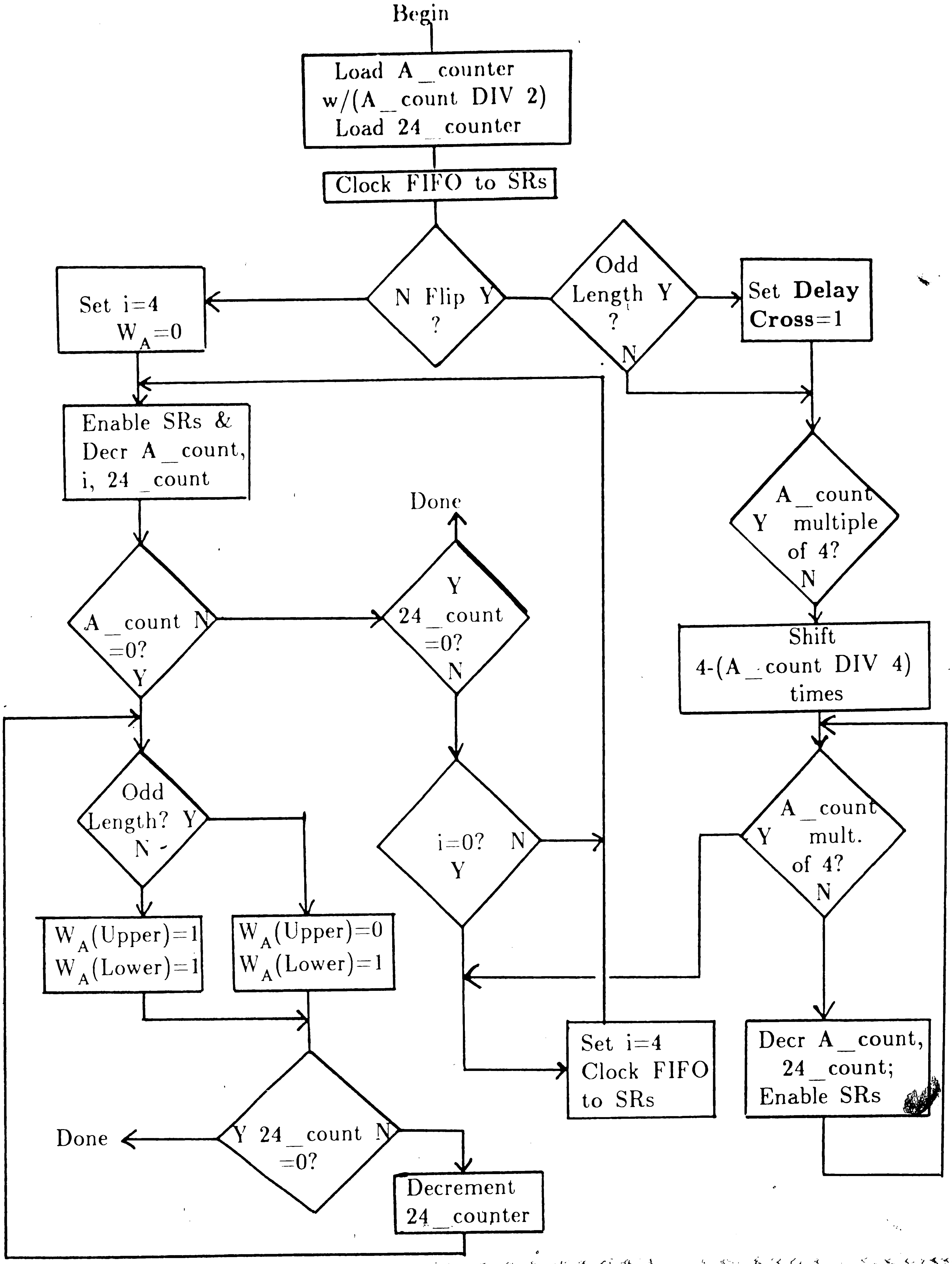

Begin
Load A_counter w/(A_count DIV 2) Load 24_counter
Clock FIFO to SRs
N Flip Y ?
Odd Length Y ? N
Set Delay Cross=1
Set i=4 W_A=0
Enable SRs & Decr A_count, i, 24_count
A_count Y multiple of 4? N
Shift 4-(A_count DIV 4) times
Done
A_count N =0? Y
Y 24_count =0? N
Odd Length? Y N
i=0? N Y
A_count Y mult. of 4? N
W_A(Upper)=1 W_A(Lower)=1
W_A(Upper)=0 W_A(Lower)=1
Set i=4 Clock FIFO to SRs
Decr A_count, 24_count; Enable SRs
Done
Y 24_count N =0?
Decrement 24_counter

Figure 3-21: A-string load Flowchart

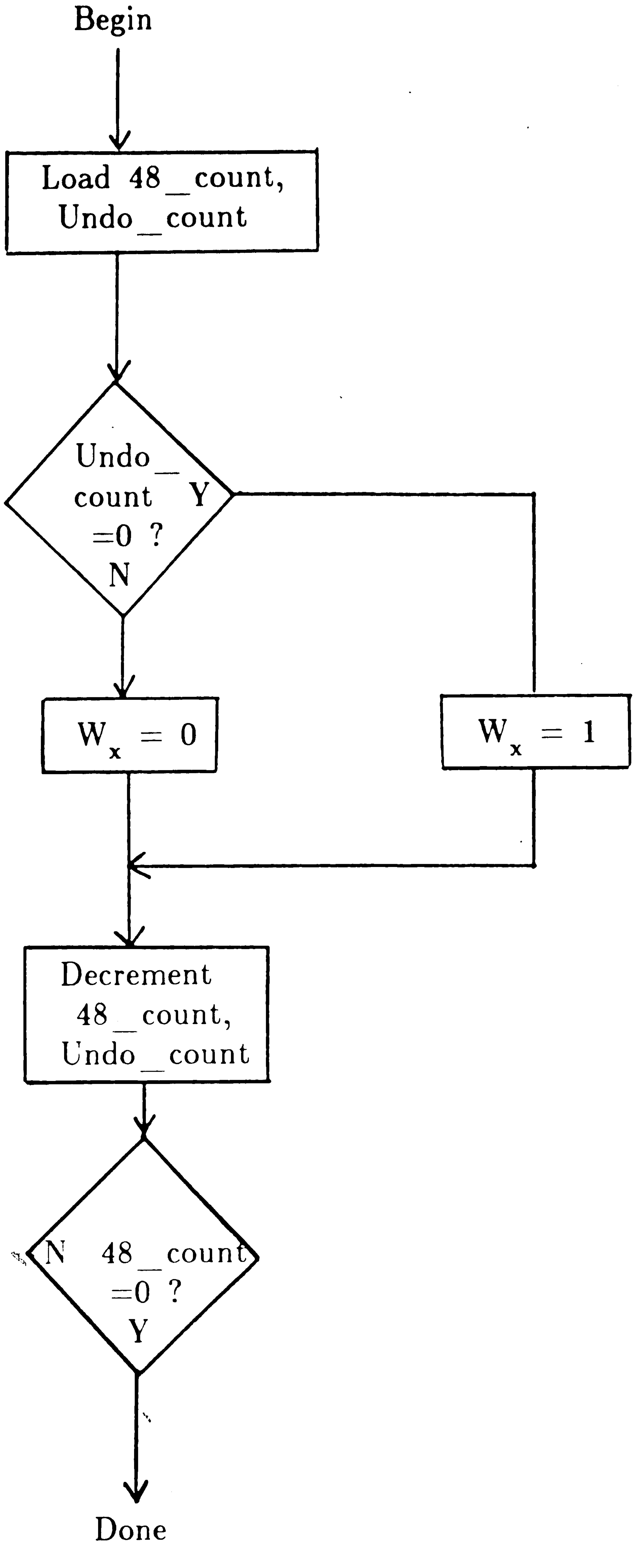


Figure 3-22: Undo Flowchart

3.7 Comparisons to General Interface

The Accusort project interface can be thought of as a specific example of the general interface design presented in the previous chapter. Because the range of functions to be performed by the Accusort interface is limited, much of the flexibility of the general interface (and the associated circuitry) is not implemented. However, there are definite parallels between the two designs.

Obviously, the use of FIFOs to queue data and to bridge the speed gap between the array and the host is the same in both interfaces. Also, both the X-string load and the A-string load utilize parallel-load serial-output shift registers. Because available off-the-shelf parts had to be used, conventional 8-bit shift registers were used in lieu of the Parallel-to-Serial Converter of the general interface design. Nonetheless, the function performed by these discrete components strongly matches that of the general interface's Parallel-to-Serial Converter. The Switch Network in the general design is directly analgous to the multiplexer of figure 3-9 that routes A to the upper and lower cells of the matching module circuit.

There is no Permutation Network in the Accusort interface since output generated by the matching module circuit is of one format only. Output generated bit-serially by the matching module circuit is routed to an 8-bit serial-input parallel-output shift register (commercially available). This is a very basic Output Stager circuit consisting of one Output Stager Module, since N=1 is the number of "rows" (as defined in the general interface) of the array simultaneously generating output. The output of this shift register is clocked directly into an output FIFO.

Chapter 4
Conclusion

4.1 Summary of Important Results

Having demonstrated the need for a high-speed general purpose interface to connect a host computer to a bit-sequential Systolic Array, this thesis has attempted to highlight important considerations in designing such a circuit. Many aspects of such a design pose significant hurdles to an efficient interface. These aspects, and solutions which would yield an efficient design, were examined in depth and are summarized in the paragraphs below.

The interface must accomodate differences in data formats and operating speeds between the host and the array. This implies that the interface have its own memory- to buffer bulk data transfers; its own data staging circuitry- to satisfy array I/O requirements; and its own controller- to intelligently interpret instructions from the host computer and configure itself accordingly.

Great effort was concentrated in the area of array I/O, especially in the efficient collection of array output. The data collection and routing strategy presented in section 2.6.4 is optimal and is applicable to any situation in which independent processes time-share a single resource. As such, its potential application reaches far beyond use solely in a host-to-array interface.

The mismatch in bandwidths of the host, the array, and the interface memory requires a solution that is multi-faceted. Because the host and the array do not operate synchronously, FIFOs are employed to ensure that there are no memory access conflicts. The proper selection of RAM memory word size (section 2.7) is fundamental to the most efficient data transfer between host and array.

Bit-slice microcontroller technology proved an effective solution to the problem of interface control logic. This approach provides the flexibility, speed, and intelligence to perform the necessary control functions, as demonstrated in section 3.6.

Although time constraints prohibit our investigating all possible aspects of the design, the major points of a general interface have been addressed in this work. Areas which merit further consideration are discussed in the next section.

4.2 Future Work

The effect of changing array word size (N_A) or host word size (N_H) on the Conditioner & Memory circuits deserves examination. Optimality for one set of N_A and N_H may be a miserable choice when one of the word sizes changes. A possible solution to this problem may be the use of programmable Conditioner & Memory subsystems that can be reconfigured for a variety of host and array word sizes.

Additional future work should be focused on design of the controller, as this unit is the nervous system of the interface. As mentioned earlier, bit-slice microcontrollers have desirable properties and, therefore, deserve further study. Effort should also be concentrated on the design of algorithms and instruction sets for the microsequencers.

In considering data formats for the array I/O, only word sizes that are a power of 2 were allowed. These word sizes permit a wide range of computations to be performed and, so, are not terribly restrictive. However, methods for generating and collecting array words of any size could be investigated.

Additional attention should be paid to universal "bit shuffling" within the array words. The Input Stager described in Chapter 2 is capable of performing row-to-column permutations. Others have studied methods of bit shuffling [9,10] and incorporating their techniques into the design of the Input Stager would make it- and, consequently, the general interface- more universal and powerful.

References

[1] Kung, H.T. and Leiserson, C.E. "Systolic arrays (for VLSI)", *Sparse Matrix Symposium, 1978*, Duff, I.S. and Stewart, G.W., ed., SIAM, pp. 256-282, 1978.

[2] Davis, R.H. and Thomas, D., "Systolic array chip matches the pace of high-speed processing", *Electronic Design*, pp. 207-218, Oct. 1984.

[3] Kung, S.Y., "On supercomputing with systolic/wavefront array processors", *Proc. IEEE*, vol. 72, pp. 867-884, July 1984.

[4] Kung, S.Y., "VLSI array processors", *IEEE ASSP Magazine*, vol. 2, pp. 4-22, July 1985.

[5] Batcher, K.E., "Design of a massively parallel processor", *IEEE Trans. Comput.*, vol. c-29, pp. 836-840, Sept. 1980.

[6] Batcher, K.E., "Staging memory", *The Massively Parallel Processor*, Potter, J.L., ed., MIT Press, pp. 191-204, 1985.

[7] Hannaway, W.H., Shea, G., and Bishop, W.R., "Handling real-time images comes naturally to systolic array chip", *Electronic Design*, pp. 289-300, Nov. 1984.

[8] Tewari, N. and Wagh, M., "Bit-sequential array for pattern matching", *Proc. IEEE*, vol. 74, No. 10, pp. 1465-1466, Oct. 1986.

[9] Bauer, L.H., "Implementation of data manipulating functions on the STARAN associative processor", *Parallel Processing. Proceedings of the Sagamore Computer Conference, August 20-23, 1974*, Feng, T., ed., Springer-Verlag Berlin - Heidelberg, pp. 209-227, 1975.

[10] Feng, T., "Data manipulating functions in parallel processors and their implementations", *IEEE Trans. Comput.*, vol. c-23, pp. 309-318, Mar. 1974.

Vita

The author was born on January 26, 1959 in Bethlehem, Pa. to Mr. and Mrs. William C. Seaman. After graduating from Bethlehem Catholic High School he attended Lehigh University and earned the Bachelor of Science degree in Electrical Engineering, graduating with honors in May, 1981. He then was employed by Westinghouse Electric Corp. for four years as a design engineer before returning to Lehigh to pursue a Master of Science degree in Electrical Engineering.