NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

STRUCTURE AND FUNCTION OF A GENERAL PURPOSE INPUT/OUTPUT PROCESSOR

by

A.C. Parker*, A. Nagle*, and James Gault*

DRC-18-13-79

May 1979

* Department of Electrical Engineering
Carnegie-Mellon University
Pittsburgh, PA 15213

** Department of Electrical Engineering
North Carolina State University

ABSTRACT

This paper describes a processor architecture designed specifically to perform input/output and interfacing functions for any central-processor-peripheral configuration. This architecture is justified on the basis of functional I/O requirements which are discussed in detail. This processor is microprogrammable with a writeable control store, allowing dynamic configuration of the processor for different input/output and interfacing applications. Underlying the microcontrol is a ROM-resident nanoprogram which performs the complex timing, handshaking, and bookkeeping control tasks. The processor architecture is modular and bus oriented.

Keywords,. Input/Output, Interfacing, Microprogramming, Processor Architecture

CARNEGIE-MELLON UNIVERSITY

pITTSBURGH-FENNSYLVANiA 15213

STRUCTURE AND FUNCTION OF A GENERAL PURPOSE INPUT/OUTPUT PROCESSOR

I. INTRODUCTION

The intent of this paper is to present an architecture of a general purpose input/output processor, Pio, which is based on design goals and constraints specific to the input/output environment. Present input/output processors - including channels, communication processors, data link controllers and device controllers - do not exhibit an architectural style which is optimal for input/output. In fact, I/O processors have architectures ranging from those which can be characterized as Von Neuman to ad hoc or even hardwired systems which cannot even be partitioned into data-memory and control parts. Examples of these systems will be discussed in Section II.

While a strong argument could be made for an end to the ad hoc, problem-specific design of I/O processors, the motivation for abandoning or severely modifying the Von Neuman style architecture must be presented. A comparison of the goals and design constraints of CPU design and I/O processor design, partitioned into the four categories of control, data manipulation, data input/output, and data storage, will illustrate the desirability and the need for a differe .t architectural style for I/O processors.

[1]The PMS notation of Bell & Newell [Bel. 7.1] is used in this paper to abbreviate structural entities in the processor. A single capital letter symbolizes a genre of components: P for processor, M for memory, S for switch, K for controller, T for transducer. Small letters characterize the particular instance of the component under discussion. Thus P_{io} is an input/output processor.

The research described in this paper was partially supported by the U.S. Army Research Office under grant # DAAG29-76-G-0224.

The following generalizations about I/O processors can be drawn from the information in Table 1:

- o Simple bit manipulations and control over the states and transitions of individual I/O lines are important
- o Precise timing and synchronization of register transfers and I/O operations are important
- o Data storage can be restricted to FIFO queues and registers
- o The overall system functions must be controlled at a lower level than in a CPU
- o I/O processors contain multiple independent, asynchronous processes.

In addition, there is one other constraint on digital systems design - available technology. A major factor in central processor performance is main memory cycle time, or cache cycle time if that scheme is used. Since data and sometimes program storage requirements for I/O processing could be met with registers and fast memories, speed of processing could be optimized by altering the architecture in ways which would not have been effective for CPU optimization.

Underlying the goals and constraints discussed above is an overall conceptual difference between central processors and input/output processors. CPUs might be said to be "introverted" and Pios "extroverted". Central processors interpret an instruction set for manipulating arithmetic, logical and symbolic data - types while input/output processors manage peripherals and transmit information without change except for error checking/detecting, encoding, formatting, and searching. For this reason, the performance requirements applied to CPUs (such as number of bits processed per second) do not apply to Pios; data through-put is a more valid measure. These differences in performance criteria, along with inherent functional differences, imply structural differences also.

DESIGN GOALS AMD CONSTRAINTS

FUNCTION	CENTRAL PROCESSORS	INPUT/OUTPUT PROCESSORS
DATA MANIPULATION	COMPLEX DATA OPERATIONS DESIRED; SPEED OF OPERATIONS IMPORTANT; ARITHMETIC OPERATIONS (FLOATING POINT FOR EXAMPLE) DESIRABLE: . (WORD PROCESSING DESIRABLE)	SIMPLE OPERATIONS REPEATED ON LARGE AMOUNTS OF DATA (FORMATTING, ENCODING, SERIAL/PARALLEL CONVERSIONS. PACKING, ERROR CHECKING)
CONTROL	SPEED OF INSTRUCTION FETCH, DECODE AND EXECUTE IMPORTANT;FLEXIBLE SEQUENCING OF INSTRUCTIONS AND DATA DEPENDENT SEQUENCING IMPORTANT; POWERFUL, HIGH-LEVEL INSTRUCTION SETS DESIRABLE; TIMING AND SYNCHRONIZATION OPERATIONS TRANSPARENT TO PROGRAMMER; BIT MANIPULATIONS LESS IMPORTANT; RARELY ASYNCHRONOUS OPERATIONS CONCURRENT IN A SINGLE PROCESSOR	LOW LEVEL INSTRUCTIONS (BIT MANIPULATIONS) IMPORTANT; SEQUENCING OF INSTRUCTIONS MUST BE TIMED AND SYNCHRON- IZED; CONTROL OF REGISTER TRANSFERS MUST BE CAREFULLY TIMED - RAW SPEED LESS IMPORTANT THAN CORRECT TIMING; CONTROL OF PROCESSOR MUST BE PARTIALLY BASED ON THE STATES AND TRANSITIONS OF EXTERNAL LINES; MAY HAVE MULTI- PLE PROCESSES EXECUTING ASYNCHRONOUSLY IN A SINGLE PROCESSOR
DATA STORAGE	RANDOM ACCESSING OF DATA AND INSTRUCTIONS NECESSARY; EASY/FAST ACCESS TO A SMALL NUMBER OF OPERANDS IMPORTANT	VERY LITTLE OR NO RANDOM ACCESSING OF DATA AND INSTRUC- TIONS NEEDED; FIFO ACCESSING OF DATA DESIRABLE #

TABLE 1.1 A Comparison of Design Goals and Constraints for CPU architectures and I/O architectures

DESIGN GOALS AND CONSTRAINTS (CON'T)

DATA INPUT/OUTPUT	SYNCHRONOUS OPERATIONS AND MOST CONTROL TRANSPARENT. TO THE USER; I/O ASSUMES A SECONDARY ROLE TO DATA MANIPULATIONS; SPEED OF I/O OPTIMAL ONLY WHEN SPECIAL PROCESSORS EMPLOYED (DMA FOR EXAMPLE)	MAXIMIZE SPEED OF DATA THROUGHOUT; ALLOW FLEXIBLE HANDSHAKING OPERATIONS; CONTROL DATA I/O PRECISELY

TABLET,1 A comparison of Design Goals and Constraints for CPU architectures and PIO architectures

II. A HEIRARCHY OF INTERFACING PRIMITIVES

A Survey of the I/O, interfacing, and communication environments reveals a set of common functions which collectively form primitive operations. Their implementations vary widely; some salient examples of this are given in Table 2.1. The functions of the primitives are clarified by matching them with implementation "levels", shown in figure 2.1. This heirarchy of levels is evident in the IMP hardware and software, as shown in figure 2.2. The hardware displays the signal, and gate and flip flop levels, while the software has a modular structure which allows routines to exist on the system level (link routines'), register transfer level (MODEM-TO-IMP), and gate and flip flop level (TIMEOUT).[HEA70] It can be seen that the highest level, the "system[11] level, has long been the only level made available for software modification in Pios. At all other levels, the primitives have been bound by hardware, tailored to meet the needs of a single CPU and peripheral device. The I/O processor presented later in this paper is programmable across all levels of the implementation hierarchy, and therefore can emulate a variety of interfaces, matching any CPU/peripheral environment. What microprogramming has done for general purpose central processors is applied here to a general purpose I/O processor. Vital to this application are the interfacing primitives, which are detailed below in groups: CONTROL, DATA I/O, DATA MANIPULATION, and DATA STORAGE.

. In the following paragraphs, we give examples of current implementations of these primitives and point out the flexibility inherent in these implementations, in contrast to the flexibility of a general-purpose Pio.

FUNCTIONAL CLASS	PRIMITIVE OPERATION	EXAMPLES
CONTROL	Protocol	TTY™ STOP and START bits, UNIBUS™ bus request and grant lines;RFD,DAV and DAC tines on the IEEE488 bus
	Sequencing	Microprogram, flipflops or CPU instructions which cause the PIO to change internal state
	Timing	Timeout while waiting for a response; timing of pulse trains
	Synchronization	Simultaneous input and output of data through a PIO, timing of latches to input synchronous data
	Priority Allocation	Interrupt request and grant circuitry; allocation of multiplexer
DATA INPUT/OUTPUT	Latching and I/O	Involves the I/O of information on data lines ,the associated data paths and hardware
	Electrical Compatibility	Involves level shifting, line drivers and receivers, impedance matching and other interface circuitry
DATA MANIPULATION	Error Checking	Parity; redundancy checks; message counts for data link transmission
	Formatting	Flags for data link transmission; packing/unpacking bits/bytes and words
DATA STORAGE	Buffering	Queues, shift registers, registers, latches, memories for temporary storage of data

TABLE 2.1 INPUT/OUTPUT AND INTERFACING FUNCTIONAL PRIMITIVES, EXAMPLES AND/OR DESCRIPTIONS

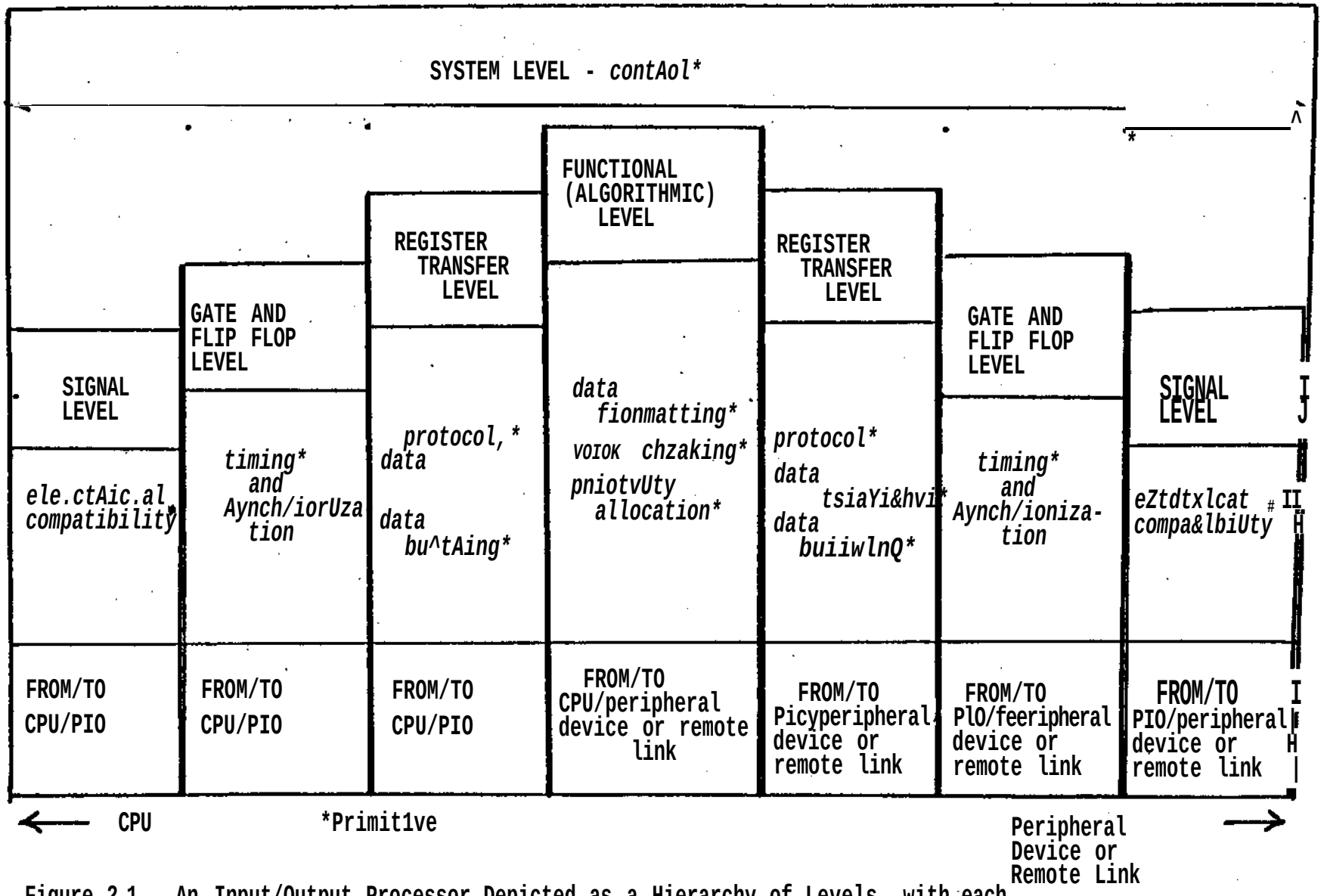
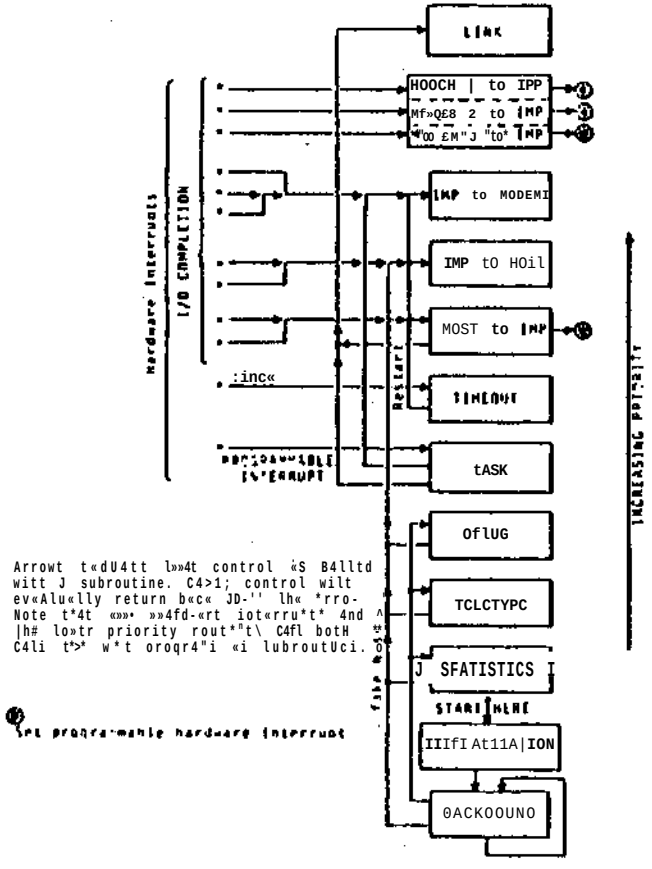
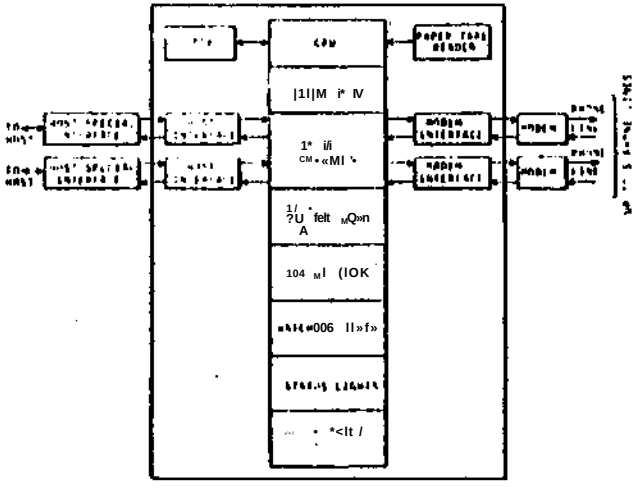


Figure 2.1 An Input/Output Processor Depicted as a Hierarchy of Levels, with each I/O Primitive shown at the appropriate level.



-Program control structure



" —I MP c»ni(iK«nalion

Figure 2.2 .The Hardware and Software Structure of the IMP (From [HEA 70J). Note the Heirarchy of levels similar to figure 2.1.

Control

In the CONTROL group are PROTOCOL,SEQUENCING, TIMING, SYNCHRONIZATION, and PRIORITY ALLOCATION,

The PROTOCOL primitive controls the handshaking operations which accompany the flow of information from or to a processor. This ranges from the insertion, detection, and deletion of start and stop bits of Teletype™ I/O to the manipulation of bus lines such as "data available/ "data received", and "ready for data". The implementation of this primitive is transparent to the user of IBM channels, and on the CD6600 PPUs (peripheral processor units), for example, and is performed entirely by the hardware, allowing no user flexibility in interfacing these processors to nonstandard modules in a system. The DEC POM/70 (programmable data mover), designed to control data aquisition in a laboratory or industrial environment, is similarly hardwired; it uses a single strobe pulse for parallel data transfers, waiting until an external device signals that data has been received or is ready. For serial data transfers the standard start and stop bits are used. An experimental disk controller, [TNT 74] built from INTEL 3000 series microprocessor modules, contains special hardwired logic for the bus protocol on the CPU side of the controller, and for the pulse capturing on the disk side. The Motorola Peripheral Interface Adaptor (PIA) chip has programmable protocols as one of its most flexible features. Hardwired implementations of the protocol primitive abound; few have the flexibility required to emulate different protocols.

The second CONTROL primitive, SEQUENCING, moves the processor or controller through states, whether by instruction execution or hardwired state transitions. Implementations of this primitive display more flexibility. For example, IBM channel controllers execute "programs[11] stored

in primary memory. Channel command words (CCW) are fetched and executed sequentially until an interrupt condition arises (e.g. end of data transfer); a limited branching facility also exists to permit storing CCW's in random locations. Looping, conditional branching, test and skip, and other control features are not provided. An early computer, the PILOT at the NBS, had wired plugboards for I/O system flexibility. The INTEL 3000 disk controller is microprogrammed, and currently implemented with a ROM; flexibility could be enhanced by substituting a writeable control store. The IMPs (Interface Message Processors) on the ARPA network are minicomputers and can be reprogrammed. The PDM70 is programmable from a keyboard and the Motorola 6820 PIA device is programmed by conroands from the processor. SEL (Systems Electronic Laboratory) minicomputers use microprogrammed I/O processors, but the microprograms are stored in ROMs. Note that the flexibility allowed in the above examples is at high implementation levels.

Control over TIMING can occur at different levels in a digital system and so comparisons across levels are somewhat inaccurate. For the present discussion, TIMING is intended to include the pulse timing of bits over a serial data link, the timeout in micro or milliseconds while waiting for a handshaking or error signal, the time between data transmissions in terms of seconds, and the counting of clock pulses. The implementation of TIMING structures in I/O processors and controllers is varied. The PDM/70 has program control over data I/O in terms of seconds, and the INTEL 3000 controller can measure time delays in microseconds under microprogram control. UART (Universal Asynchronous Receiver/transmitter) chips contain precise timing control for transmitting/detecting single character bit strings with start and stop bits.

Also many serial asynchronous data link controllers have selectable Baud rates-I/O processors and controllers for channels and peripheral devices in general do not have any flexible control over timing.

There are essentially three levels of SYNCHRONIZATION which occur in I/O, communications, and interfacing. The lowest level of SYNCHRONIZATION involves the transmission/detection of data bits synchronously over a data link or to/from a disk or magnetic tape. The data bits arrive at a fixed ratfe, sometimes with the clock alternating with the data bits, sometimes with encoding which allows the receiving device to synchronize on the transmitted data, sometimes with a combination of both. Because of the speeds involved, any attempt to allow flexibility*in this type of SYNCHRONIZATION is limited to changing the data rates of the transmission/detection.

The second level of SYNCHRONIZATION which occurs in I/O processors and controllers is the SYNCHRONIZATION between different hardware processes in a single processor. In order to discuss this problem, the notion of a hardware process must be explored. A hardware process is a sequence of actions which is controlled independently.of other sequences of actions. In a disk controller, for example, the process of forming words from single bits runs in parallel both with the process of testing the cyclic redundancy check (CRC) bits for errors, and with the process of sending the assembled and tested word to the central processor. Due to the synchronous nature of the word assembly, and the time constraints on the input process, the only conmunication with the other processes may be through a signal that-a word has been assembled, and the return signals that indicate enough words have been assembled or an error has occured.

A second process, the CRC, checks the assembled word, independent of the assembly process, as long as it knows the location of the word and its readiness for checking. A third process is activated when it is signalled by the second process that a word is ready to be transmitted to the central processor: Ignoring memory contention problems and variations in communications between processes, there is still the basic synchronization problem to resolve. Even hardware language descriptions of concurrent independent, asynchronous processes are difficult to construct and do not really represent the operations of the hardware. Addition, in controlling separate hardware processes with a central processor executing a single program is virtually impossible with current notions of instruction execution. As a result, the processes are implemented in hardware, obviating the flexibility desired in a general purpose I/O processor. For example, the Intel 3000 disk controller has implemented each of the processes described above in separate hardware subprocessors. The highest level of synchronization is the control over devices transmitting/receiving data at different rates, synchronously or asynchronously, or in different quanta of information. Flexibility on this level would require variable buffer memories, programmable hardware for data rate variation, and the ability to adapt to synchronous or asynchronous transmission.

PRIORITY ALLOCATION, the last CONTROL primitive to be discussed, is one of the primitives often implemented within the peripheral device hardware, with the device interconnection scheme or as a central processor function. Devices are often "daisy-chained[11] together so that interrupt priorities are wired into the system. When central processors issue commands to I/O processors and controllers, priority allocation is often

done hy the central processor prior to the command issuance- I/O processors and controllers linked to more than one device often service in a "round-robin[11] fashion. On a higher level, the IMP is the best example of PRIORITY ALLOCATION in a communications processor. It responds to a message with a Ready For Next Message (RFNM) acknowledgement and does not allow reception of a second message over the same logical link until the first has been acknowledged. On a lower level when transmitting messages it uses a head-of-the-line (HOL) scheme « . It allocates priority to incoming packets which form one message depending on the order in which they were sent. The only exception is that acknowledgement messages have priority over data traffic.

The next division of primitives, the DATA I/O section, contains the primitives DATA TRANSFER and ELECTRICAL COMPATIBILITY. DATA TRANSFER refers to the movement of data into/out of the I/O processor, interface, or controller. This primitive differs in implementation depending not so much on the specific system or processor but on the type of data to be input/output. If the data is static (is.valid on the I/O lines until the receiver signals data received) then simple gating into registers solves the I/O problem. If, however, the data input is signalled by strobe pulses, start bits, special flags preceeding the data or other means, and the data changes dynamically without intervention from the receiver, then special consideration must be given to the capture of each word or bit as it is available. A second complication can occur when the data bits are represented not by voltage levels which signify Is and 0's but by transitions in voltages. Decoding must occur at the time the data is input, and encoding at the time data is output.

Since data transfer as a primitive refers to levels at and below the register transfer level, flexibility of this primitive function most often occurs as the I/O design is underway, and not under program control, or even console switch control.[2]

ELECTRICAL COMPATIBILITY is the lowest level of I/O and interfacing functions, and maybe should not be considered with the others, were it not for the fact that integrated circuits exist which perform most of the electrical interfacing tasks required, and these ICs occur in specific places in an I/O processor, communications controller, peripheral controller, or interface architecture. Hence they can be used as modules in a modular architecture, and various implementations can contain different circuits, as needed, in the electrical compatibility module locations. Further discussion of the nature of ELECTRICAL COMPATIBILITY is beyond the scope of this paper.

Data Manipulation

DATA MANIPULATION as a functional division is present in all dit-ital systems. The principal types of data manipulation found in I/O and interfacing are ERROR CHECKING and FORMATTING. The two basic methods of error checking to be discussed here are parity checking and parity bit generation, and redundancy checks. Cyclic redundancy checks (CRCs) are often used with disks, and latitudinal and longitudinal redundancy checks are used with magnetic tapes. Parity bits are used most often for data I/O that involves single word transfers and in particular for binary, BCD and ASCII I/O. Another type of error checking that occurs is the counting of

^However, flexibility can be made available in a general purpose P_{io} by providing several different I/O modules which can be addressed under program control.

messages sent, received, and acknowledged over synchronous data* links. Although all of these checks could be done by software, the message counting is the method most often implemented in that manner. The introduction of integrated circuits for parity andredundancy checks has further reduced the likelihood of realizing these checks in software, and in most cases software is relatively slow. However, the flexibility needed for general-purpose I/O is lost when wired checks are implemented.

FORMATTING of data is a primitive operation which covers any bit manipulations which do not change the information content of the data. Examples of this include packing and unpacking of bytes into words, the insertion and deletion of flags on messages, the insertion and'deletion of stop and start buts on ASCII characters, BCD to binary conversion, and other low level procedures which rearrange data. In addition, FORMATTING includes the data dependent rearrangement of data. This encompasses sorting procedures most often carried out by the central processor but in some cases (the CDC6600 PPUs for example) by the I/O processors. Bit insertion and flag insertion, along with data packing/unpacking are often done by the hardware, while code conversion, sorting and searching are done by firmware of software.

Data Storage

DATA STORAGE in central processors usually refers to register storage (direct access) and random access storage. Any other data structures (linked lists, stacks, queues, for example) are implemented with software. In I/O processors, data is either buffered in a save register as it is transferred through the system, or in a FIFO queue which contains a string of data words, bits or bytes. These queues are- implemented with software in the majority of cases, although IC queues are available for limited applications. Software queues are used in the ARPA Network IMP, for example. In

the INTEL 3000 disk controller, data is moved to the processor memory as fast as it is accessed, and so the use of a queue is not necessary. It is necessary, however, to maintain a memory for block transfers in general purpose I/O processors, but there is rarely a demand for random access capa^ bilities in these memories. An exception to this occurs if certain queue items have a higher priority and are to be renioved before other items.

Summary

The I/O primitive functions discussed above are quite different from the functions one might describe as primitive for central processors. In addition, the range of levels covered by these primitives is broader than CPU primitives, and each primitive itself covers a broader functional concept. The architecture designed to implement this set of primitives is therefore somewhat different from the architecture of a central processor.

III A GENERAL-PURPOSE, MODULAR INPUT/OUTPUT ARCHITECTURE

A review of the constraints and design goals of I/O processors indicates three fundamental principles of I/O processing:

- The data-memory portion of the hardware should be designed to optimize data through-put,
- The processor should be able to support multiple, asynchronous operations or sequences of operations (processes)
- The user should have control over timing, synchronization, and bit manipulations

The above goals and constraints, in addition to the general-purpose nature of the processor, force the following design decisions:

- The data-memory architecture should be modular, each module containing programmable hardware in order to maintain multiple processes without complex central control
- The flexible nature of the data paths indicates a bus for data transfers, but the asynchronous, concurrent operations and the need for optimized data flow through the processor indicate a multiple data path architecture. A dual bus structure is intended to solve these problems,
- The data-memory structure should support first-in first-out store and access
- The architecture should accomodate variable data widths
- The architecture should support a pipelined sequence of data operations to optimize the speed of data flow

- The processor must address its own program memory
- The program memory should be supported by an underlying control structure which can manage internal handshaking operations, bookkeeping tasks, and other processor operations which should be transparent to the user
- The processor should be programmable at the register-transfer level, and in some cases at the gate/flop-flop level

These architectural features, in combination with a design which can be implemented with high-speed circuitry to meet the speed requirements of I/O controllers, produce a processor which is generalized to the extent that it can perform under the following circumstances:

variable data widths

variable flag formats on synchronous data

variable formats of data (packing densities, for example)

variable types of error checking

variable handshaking requirements

variable priority allocation schemes for multiple servicing

variable encoding and decoding operations

variable buffer lengths and word widths

variable timing of synchronous and asynchronous data I/O

The generalized Pio emulates a variety of processors, interfaces, and controllers with the same hardware. Thus, the generalized Pio assumes the role of host processor to a set of target processors spanning a range of possible Pios. This type emulation is more difficult than central processor emulation because the I/O processor must emulate, for the central processor, the interface the central processor expects to see, and must also emulate, for the device, data link, device controller, or other processor, the interface it expects to see, all with the correct timing. In addition, these two

emulations must be synchronized within the generalized host Pio. An alternative view is to consider the generalized I/O processor to be the base machine and the processors implemented to be virtual machines. It should be noted that the Pio is not designed to support multiple emulations on a dynamic basis. Hence, if the CPU linked to the Pio should force the Pio to handle more than one configuration at a time, the control microprogram will have to deal with the ensuing data changes in buffers and registers. This mechanism is presented later in this section in more detail.

This need for control on a lower level than with conventional processors implies the requirement of a writeable control store. However, at the same time the complexity of microcoded interfacing and I/O operations precludes user programming. Thus a two-level microprogram/nanoprogram combination is used to allow the user the freedom to program sequences of operations and some timing parameters without microcoding each individual control signal.

The nanoprogram control performs the ultimate control and reconfiguring functions of the processor, keeping track of addresses, buffers, and hardware programming, while being transparent to the user. It also controls instruction fetch and execution for the microstore.

Some of the microword fields in each microinstruction cause initiation of nanoinstruction sequences while others control the processor directly. Thus, the control signals in the processor originate in both the microinstruction register and the nanoinstruction register. This configuration, along with the level of operations evoked by the micro and nanoinstructions, illustrates a level of control lower than the two level combination of assembly language/microprogramming commonly implemented in CPU's.

In general, it can be said that the microprogram describes what the processor is to do, while the nanoprogram controls the timing of each task, the synchronization between multiple tasks executed simultaneously, and the handshaking and internal control signals needed to perform the operations. Thus, the microprogram describes a target processor and the nanoprogram performs the actual mapping of the target I/O processor onto the host processor. This feature has been described and used by others in the past: by Lesser ' to define two levels of control, the conventional level and a global level of control [LES 73] and by Nanodata Corporation in the QM-1 [NAN 74].

In order to program this processor, the user writes a program which consists of a main body and one or more processes. The main body merely defines the hardware configuration to be maintained by each module inside the Pio, and describes the conditions for initiation of each process. Each process can be given a priority by the user, if needed, and can be initiated individually by data and control conditions specified by the user. Each process consists of a set of statements which perform a particular I/O function in a logical time dependent order. For example, if the processor is to emulate a disk controller, the read operation from the disk would be a separate process from the write operation.

IV. THE OVERALL PIO DATA-MEMORY ARCHITECTURE

In order to discuss the processor performance and function an idea of the structure has to be developed.

The generalized processor data-memory structure consists of modules interconnected asynchronously by a dual data bus and a control bus. This interconnection is shown as a PMS[3] diagram in figure 4.1. This structure is similar to the Honeywell emulation machine described by Jensen [JEN 77]. The data-memory modules can be grouped into functional classes, as shown below, corresponding tp the primitives discussed

Data Manipulation Modules:

ALU Module
Code Converter
Parity Check Module
Redundancy Check Module
Unpacking Module
Packing Module
Decoding Module
Encoding Module
Format Module

Data Storage Modules:

Buffer
Register Module

Data Input/Output Modifies:

Input Shift Register Module
Output Shift Register Module

Control Modules:

Initiation Module
Interrupt/Protocol Modules
Nanostore
Microstore
Timer
Synchronization Modules
Arithmetic and Logic Unit
Registers

Some of the data storaae. data I/O and data maniDulation modules are similar in architecture to the QED modules specified by Dejka [DEJ 73]

[3] The Processor-Memory-Switch notation [BEL 71]

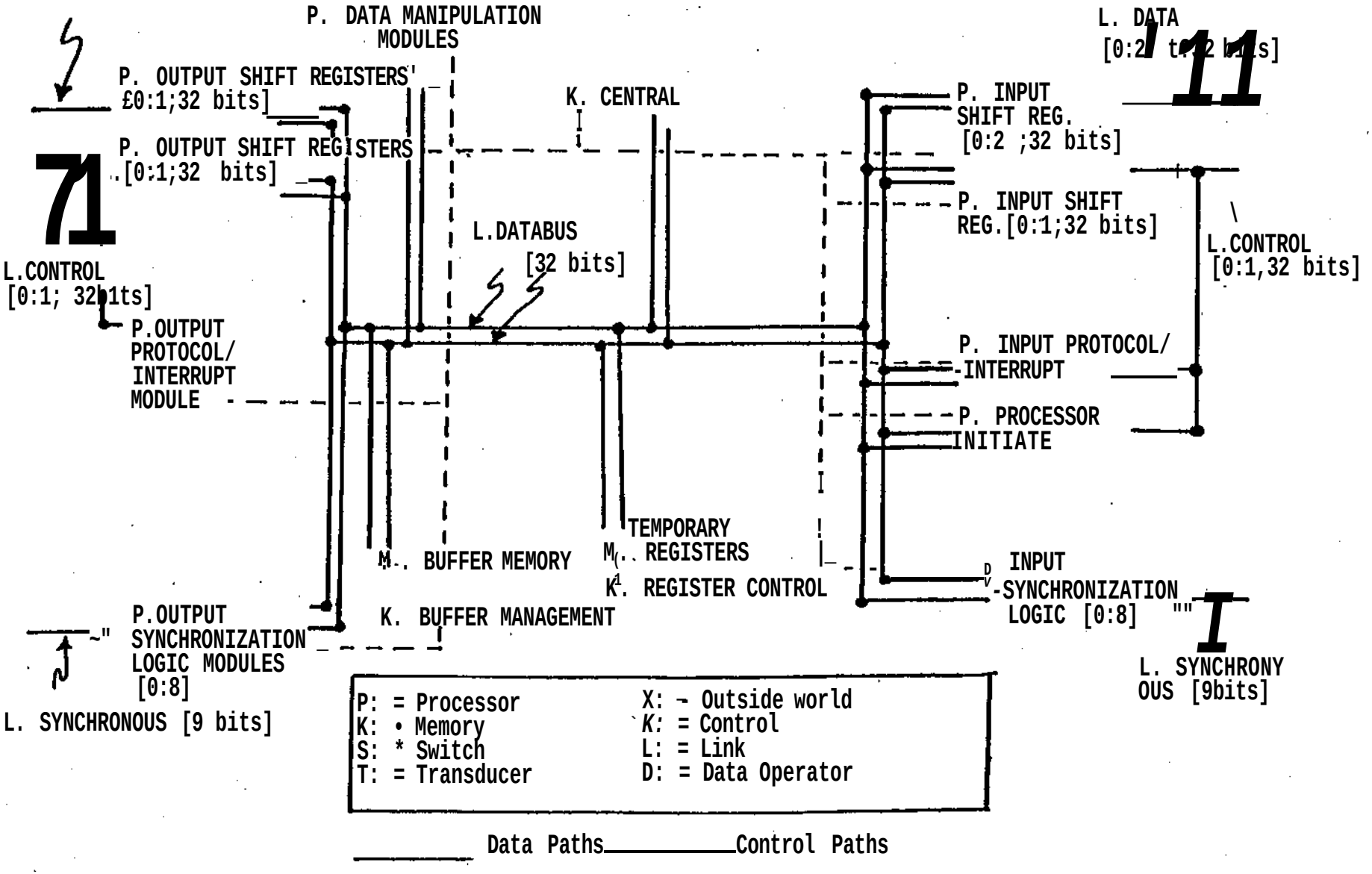


Figure 4.1 PMS Diagram of the P.I/O Data- Memory Structure

Most of the data-memory modules have the following capabilities and features:

- o The capability to be addressed by the control
- o The capability to perform handshaking with the control in order to be programmed or to transfer data
- o The capability of being programmed over the control lines to perform an operation or sequence of operations
- o Residual control: the capability to be preprogrammed for an entire process execution or indefinitely
- o The capability to transfer data on/off the internal data buses under microprogram control
- o The capability of accepting variable data widths as programmed by the control
- o The capability of raising an error line for data errors or hardware malfunction
- o The capability to output status information to the control
- o High output impedances (tri-state logic) and TTL Compatible I/O lines
- o A 10ns clock rate, the minor cycle time of the control

The modules must possess programmable sequential logic in order to realize these capabilities, requiring the Pio to possess distributed control. The modules represent special purpose processors activated by signals from the control to perform functions determined by module type. For example, the buffer module [PAR 77] actually-can contain up to four queues, and the width of the words stored in each queue can vary from 4 to 32 bits. Once queue lengths and word widths are preset by commands from the central control, the buffer module itself updates queue pointers, checks for full and empty conditions and maintains the present bit widths and queue lengths, all o* which is transparent to the central control. The PMS structure of a typical data-manipulation module is shown in figure 4.2.

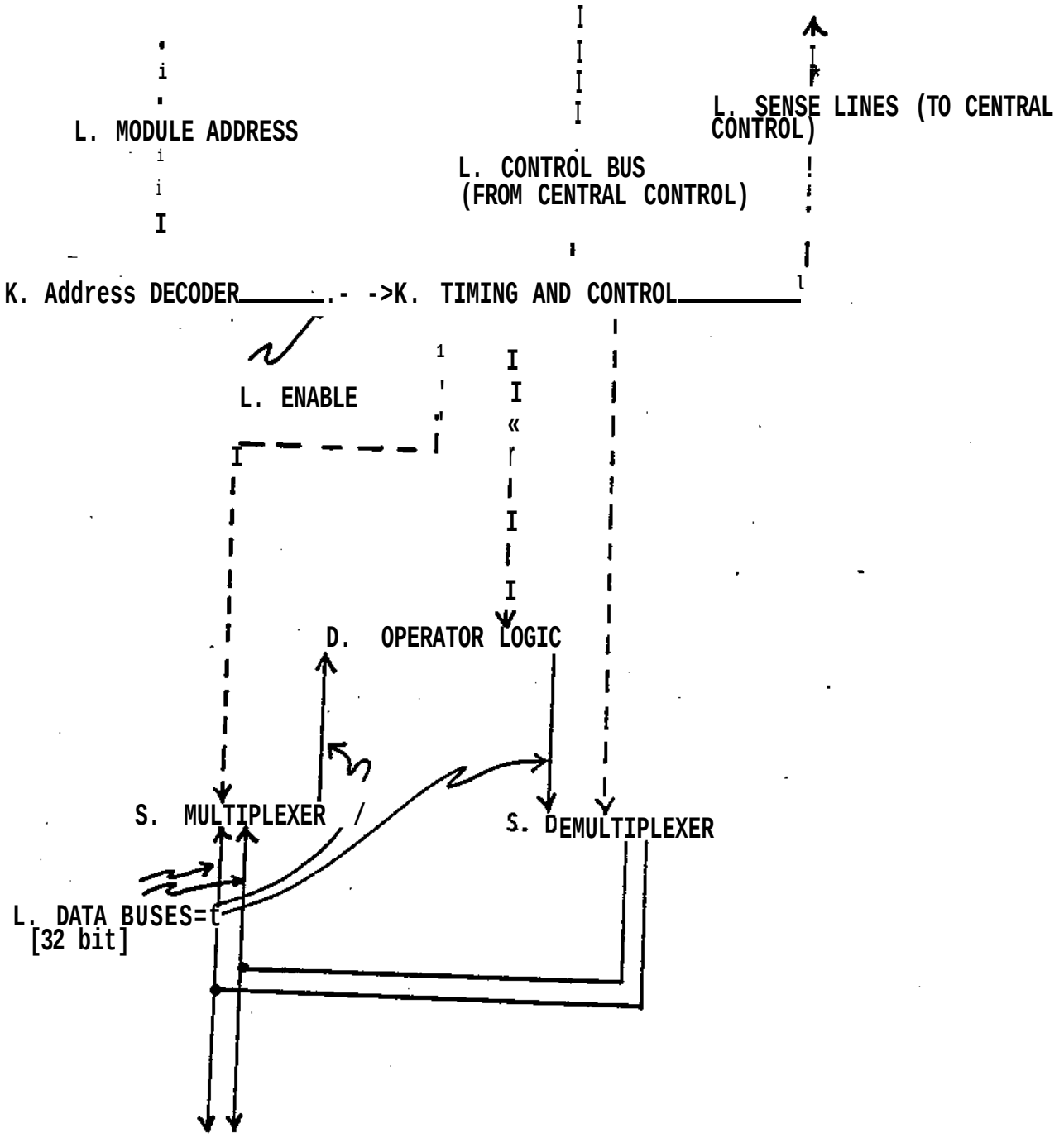


Figure 4.2 PMS Diagram of A Typical Oata Manipulation Module

Each module is addressed by the control module via the control bus, as shown in figure 4,1, The use of dedicated control and enable lines is used in modular designs such as Torode's logic machine, [TOR 74} where the number of modules is small. However, for variable-function ., reconfigurable systems with many modules, the wiring rapidly becomes complicated and the control store word width unwieldy when multiple enable lines rather than addressing is used. Using addressing, the functional module set is easily expanded, and the configuration has failsoft capability.

When a module is addressed, that module latches the control bus and performs the specified functions• In order to activate two or more modules to input data, to output data, to manipulate data or for concurrent operations to occur, each module must be addressed.

Concurrent operation of several modules is accomplished by addressing them sequentially, the deactivating them later by separate commands. The modules are activated in the order in which the functions naturally occur. For example, the signal to output data on the bus comes first, then the signal to another module to input data and operate on it and then the signal to output the data operated on. There is a signal from the control to each module when deactivation is to occur. The timing sequence lengths are variable in multiples of 10 nanoseconds and the nanoprogram word contains the timing information.

The timing and control signals required to store a word in the buffer module are drawn in figure 4.3. During the execution of a time STORE, data bus A is only used for a short period, data bus B is unused, and commands are issued from the central control only 25 % of the time, underutilizing the Pio resources. This can be critical in high data-throughput situations, and for this reason the control has the ability to pipeline data through the Pio. For examplp

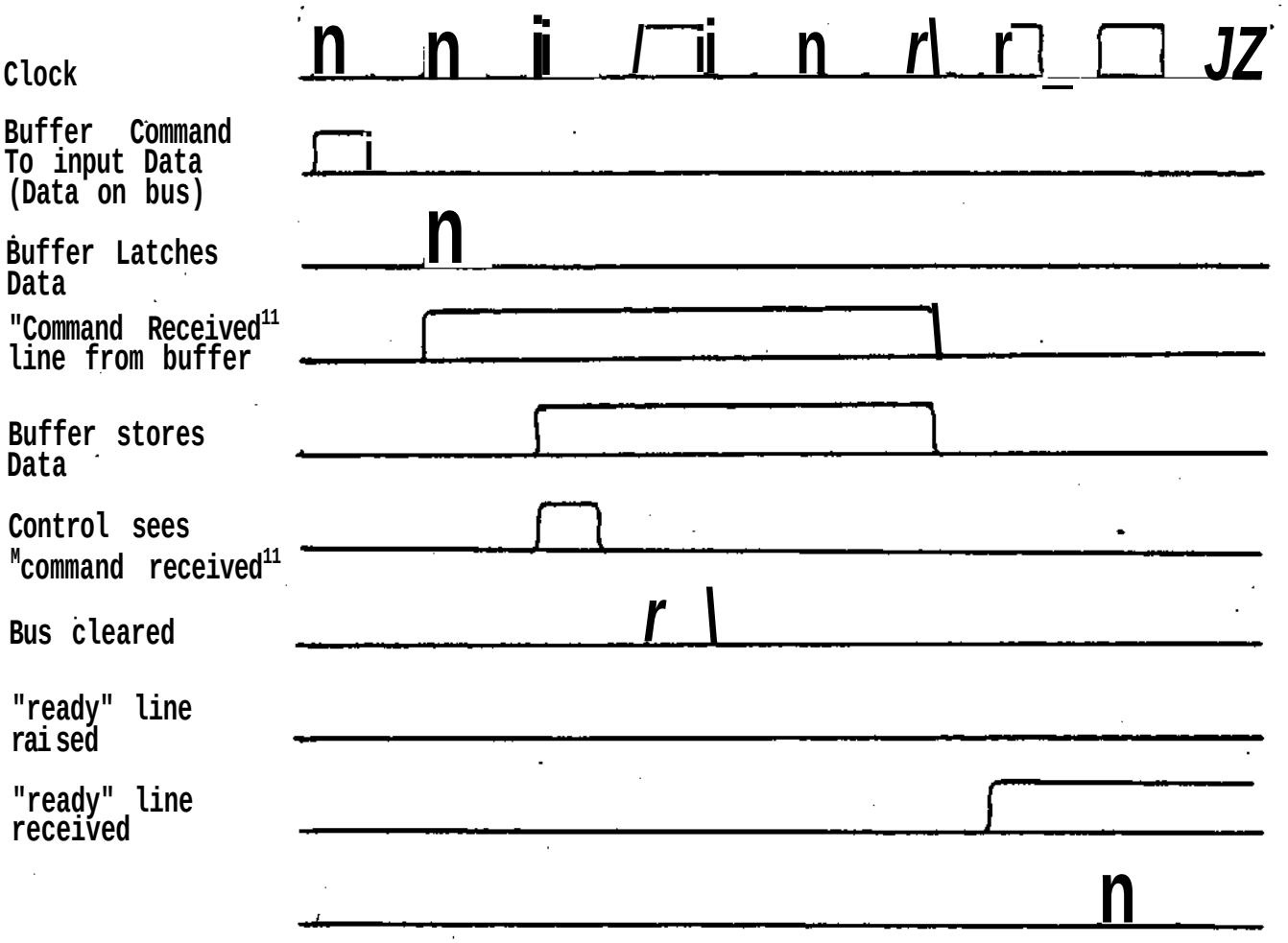


Figure 4.3 Timing and Control Signals Required to Store as Word in the Buffer Module

while data.$_i$ is being stored in the queue, data$_{i+1}$ can be input to the Pio, and data.j_{-n} accessed from the queue earlier, can have a parity bit generated. Meanwhile data $_{i-n-1}$ can be output. However, during this particular phase, the queue cannot be accessed to retrieve data. In fact, the data buses are only used to transfer data to the parity check and buffer modules - otherwise bus usage conflicts might occur.

In general, the contention problems which would arise if the data flow is pipelined include:

- o Module addressing - only one module can be addressed at a time
- o Use of the data buses
- o Use of a single module for two functions simultaneously

The control required to support this complex type of data flow is discussed in the next section.

Specific Module Functions

In addition to the buffer module, the other data/memory modules deserve some explanation. The other data storage module, the register module, is used for temporary storage of constants and contains accessible registers.

The data input/output modules include the input and output shift register modules. In addition, the synchronization module performs I/O of synchronous data, but is classified as a control module since the synchronization hardware performs mainly a control task.

The input shift register modules act as latches for input data of $\leq$ 32 bits in width and also align the data so that it is right justified when transferred onto the internal A and B buses. This way 8 bit wide data can be input on lines 1-8 and 9-16 of the shift register module at different times, and transfered through the interface properly aligned on lines 1-8

of the internal buses. The output shift register modules perform the opposite function, transferring justified data from the A and B buses to the output lines. All of the data manipulation modules are capable of interacting with either bus as programmed by the control. The ALU module performs standard arithmetic and logic functions. The code converter contains a memory for code conversion from any code to any other, using table lookup. The decoding module is prewired to perform ASCII, BCD and EBCDIC to binary conversions, and the encoding module performs the reverse operations. The parity check module generates an even/odd parity bit and in the generate mode appends it to the left or right of the data. In the check mode, it checks the generated bit and compares to the existing parity bit alerting the control if a parity error has occured. The redundancy check module performs a CRC (cyclic redundancy check) on an arbitrary string of data, and returns a check byte.

In the generate mode this byte is output when commanded by the central control. In the check mode, the module then accepts a check byte, compares it to the generated byte, and reports errors. The packing module can be programmed to accept a variable width word and pack n words into a single larger word, outputting the packed word when commanded. The unpacking module performs the reverse, accepting a word of variable width and unpacking it to output smaller words, one at a time, when commanded. The format module can insert bits at specified locations in a word by being presented with the bit pattern to be inserted, along with the word. Alternatively, it can delete bits at specified locations in a word (stripping flags, tag bits, etc). Implementation of all the data manipulation modules including control commands, timing and block diagrams is described in [PAR 75].

The data memory modules perform - for the most part - functions which could be performed by a single cleverly programmed microprocessor. However, the performance - as measured in terms of data throughput - and flexibility necessary for generalized I/O processing cannot be attained with a single-processor architecture

V THE OVERALL PIO CONTROL ARCHITECTURE

INTRODUCTION

The Pio control will be discussed in terms of function and structure. Although the overall system functioning implies a straightforward control structure, the constraints of user access to a control store and high-speed data throughput requirements create a complex control environment.

Pio Control Environment

The Pio Control can be subdivided into three basic functional categories:

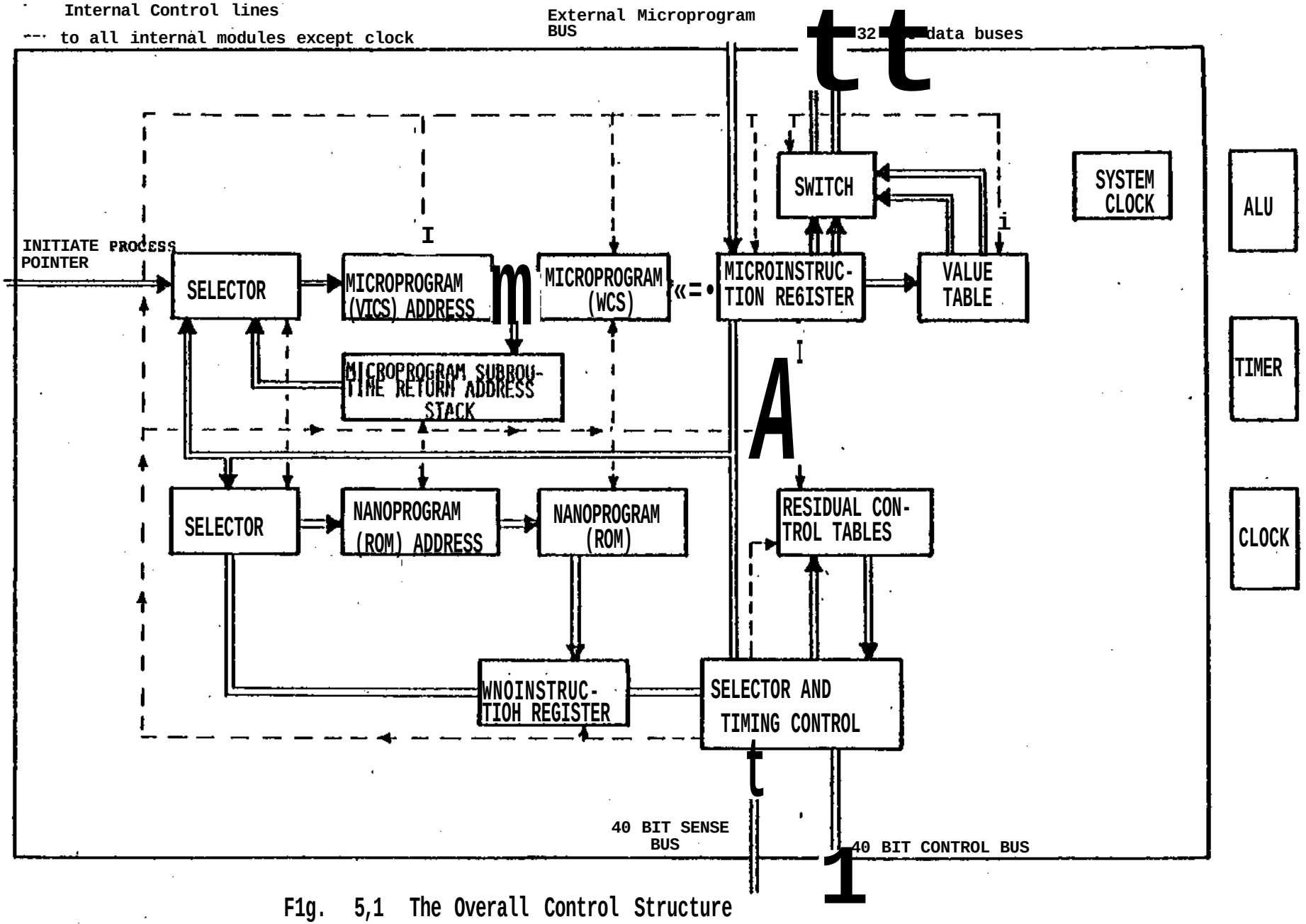
- o Functions external to Pio - Control over protocol and interrupt signals, initiation of Pio processes, and synchronous/asynchronous data I/O.
- o Functions internal to Pio - control over programming of data-memory modules, Pio configuration and internal data flow.
- o Functions internal to the control itself - control instruction fetch, decoding, execution and branching.

Since the second and third categories deal with internal Pio control functions, they will be covered after a discussion of the control structure.

In specific, the Pio control must be responsible for and responsive to external events in the following way. The control must first program the initiation module to respond to certain configurations of the I/O control and interrupt lines by issuing to the central control the address of a process to be initiated. In essence, the central control has passed control over process initiation(to the initiation module)This control is not passed back to the central control until conditions for process initiation have been met. The central control then determines whether the process priority is higher than the currently executing process. If so, the central control interrupts the current process at a suitable break point and begins the new process. If

the process desiring initiation has a priority equal to or lower than the current process, the process address and priority are saved until the current process finishes execution. It should be noted that lower priority, current; processes are-, interrupted at a marked break point in the program executing the process. The placement of break points and the resultant effects of process interruption have not been investigated in detail at this point.

The central control has direct control over .transitions on output protocol and interrupt lines by issuing commands to the interrupt/protocol modules. In addition, the central control detects incoming interrupts either via the initiation modules (if they are to cause a process initiation) or by programming the input.protocol/interrupt module to signal the central control when an interrupt has occurred. This second option,for other than real-time applications, is also used by the control to inspect protocol lines. The only other central control functions external to the Pio are the data transfer commands, which, in general, are straight forward. The exceptions are the synchronous I/O commands, which tell the input synchronization module to begin detecting synchronous data (in nbit words), and to the output module to begin outputting synchronous data. The synchronous modules themselves contain hardwired control over the synchronization process including the detection/generation of sync bits, flags, imbedded clock pulses and other synchronization information.



F1g. 5,1 The Overall Control Structure

The Control Structure

To an educated observer, the control structure of a generalized Pio seems overly complex. We make no effort to justify the complexity at this point; rather we point out particularly complex structures and reserve comments on control complexity until the discussion on control function-

The overall central control structure is shown in Figure 5.1. The major points of this structure are the nanostore and microstore modules which contain a nanoprogram (ultimate control over the Pio) and a user-writeable microprogram. It should become evident as the discussion continues why this is different from the conventional machine-level/microstore combination.

The microcontrol and nanocontrol modules are the sources of the control inputs to the other modules. Control also includes clock and timing modules and synchronization circuits for the input and output of synchronous data. An arithmetic and logic unit is used for instruction sequencing, internal variable incrementing and other operations of the microprogram and nanoprogram. There are also registers to hold variables and constants used by the control. Due to the overlap of functional classes, the synchronization modules can be considered to be control modules as well as input/output modules. A system clock completes the control modules.

The timing module has a clock, a counter, and an output which is set after a time delay specified by a command from the control. The synchronization modules are activated by synchronization microinstructions or I/O microinstructions involving synchronized input or output lines. These modules can be addressed individually or in groups of nine for parallel synchronous data transfer. They include a programmable clock which can be phase locked with an external clock and a shift register for either inputting or outputting data. In addition, the modules contain logic for gating of the data on and off

external lines and for shifting of the registers. A save register is used to transfer data on or off an internal Pio data bus while synchronized input or output of data continues.

The ALU module performs addition, subtraction, multiplication and comparison for use by the control, but is not used to implement the data manipulation primitives,which have their own modules attached to the data bus.

The microcontrol module contains a random access memory, an address register/counter (WCSAR), a data register (WCSDR) and a selector to control the loading of the address register. In addition, there is a hardware stack used to contain return addresses from subroutines and a value table which contains constants used by the microprogram.

The microstore is 80 bits wide. Two 40 bit word instructions or one 80 bit full word instruction can be stored in each memory location. This allows two shorter instructions to be accessed simultaneously saving about one access time every four instruction executions if instructions are executed sequentially, and allowing pipelined execution of two shorter instructions. The half word-full word type of architecture is used to allow for variable instruction lengths and to increase execution speed of the interface.

The fixed fields of the microinstruction have specific meanings which depend on the opcode of the instruction and the location of the fields in the instruction word, using two level or indirect encoding.

As the microprogram is assembled constants specified in the program are loaded into a value table, and variables are assigned locations. The value table and variable pointers are placed in the microinstructions. This allows variable field length data to be replaced with addresses of constant length in the microstore. TIME, FREQUENCY and SCALE (multiplication) values specified in

microinstructions are represented in floating point format.

Since the instructions generally involve only one machine operation the microprogramming can be referred to as vertical. Generally, vertical microprogram words would be less than 80 bits wide but the reconfiguration capability of the hardware adds bits to the microinstruction.

The address space of the microstore and its proposed implementation in hardware are discussed in detail in (PAR 751. In addition, the bit widths of the microinstruction fields and an example microprogram for disk controller emulation may be found there.

Inputs to the microcontrol include an external microstore data bus for loading the WCS, a set of lines containing a pointer to a process desiring initiation, and control lines from the nanocontrol. Outputs from the microcontrol include the contents of the microdata register, and the two 32 bit internal data buses.

The nanocontrol module contains three main submodules; they are the read only memory (ROM) which contains the nanoprogram, the residual control table and the selector and timing control block. In addition there is a data register (ROMDR), an address register (ROMAR), an address selector for the ROM and a selector for the control bus.

The residual control table consists of random access memory submodules. These are pre-loaded with module addresses and can contain flags denoting process interruption and state information necessary for process restoration.

The selector and timing control block contains control logic, a microinstruction stack, a reservation table for modules and internal buses, a register for the sense bus input and a comparator.

It has a clock input, the 40 bit sense inputs and the nano and microinstructions as inputs. The outputs are the 40 control lines to the modules, 13 lines to control the microstore module and 16 lines to control the nanostore modules.

The Control Functions - Overall View

At this point it is desirable to consider the overall control flow. A WCS word is accessed if both halves of the previously accessed word have been executed. The word accessed is partially decoded to determine the opcode and instruction length. Appropriate control signals are sent to the modules needed to perform the specified function. SenseJines from the modules signal the completion of the task, error messages and branch conditions the control is to receive. If the information is completed without errors or branches, the remaining 40 bit instruction is executed or the next 80 bit physical word is fetched. Of course, if there are branching or error conditions, the address register is forced to the appropriate branch or error addresses. The microinstruction accessing, decoding and execution are all under control of the nanoprogram.

The microprogram is written in the form of processes. The main process contains conditions to be met for execution of underlying processes, imbedded in INITIATION statements. Usually the state of Pio input lines from the computer or peripheral determines a condition for-process initia-

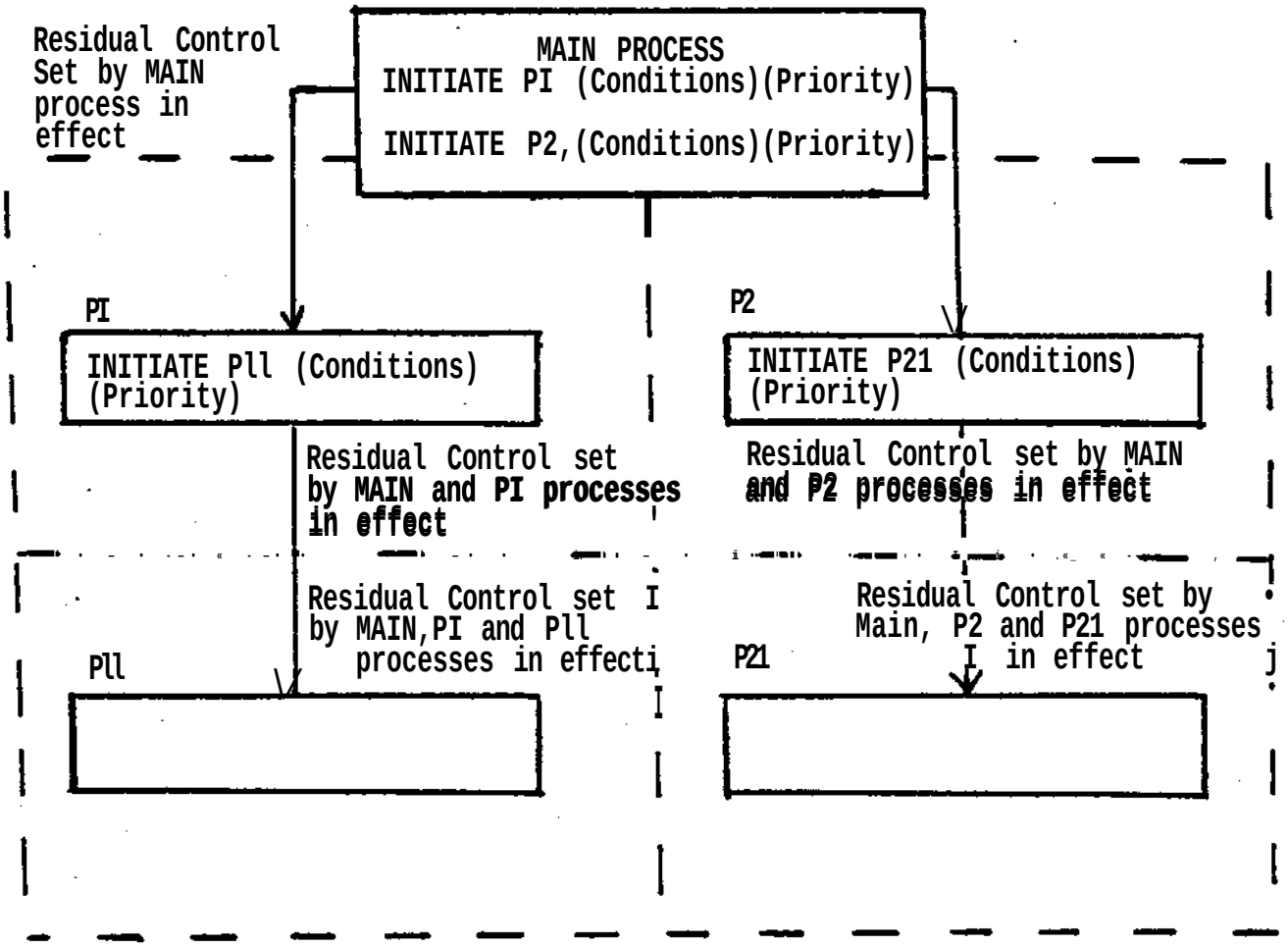
tion. An initiation module is programmed to check for this condition by the INITIATION microinstructions, and given a microstore address corresponding to the process to be initiated. When the condition occurs, the pointer is returned to the central control by the initiation module.

Control of the modules, the data bus widths and the buffer configurations can be either of single instruction duration or residual. In general, RESIDUAL CONTROL microinstructions are used to set module functions to be maintained for more than one microinstruction, and a CLEAR microinstruction allows reprogramming of module functions. For example, the process initiation conditions are stored in residual control registers inside the initiation modules. Also, residual control is used to control the data word width for the data storage modules. Data manipulation modules may or may not be programmed with residual control over data word widths, depending upon whether the data word width processed by a given module remains constant over more than a few microinstructions.

When a process has completed execution, all residual control functions set during the process are cleared. When a lower level process is initiated by conditions set within a process, all residual control functions are maintained. When a higher level process is initiated, the process residual control state is saved . The overall process hierarchy is illustrated in figure 5.2.

The Control Functions - Microprogram Execution

Execution of the microprogram begins with a fetch and execute of the first instruction in the microprogram and proceeds top down until the main process has been executed. Execution is implemented by fetching the microinstruction, partially decoding the opcode and then fetching a set of nanoinstructions in parallel based on the opcode of the microinstruction.



Note: If PI has higher priority than P2 and its conditions for initiation are met while P2 is executing, all of the P2 residual control information is saved, the modules are cleared, and PI begins execution.

Figure 5-2 Overall Process Hiearchy in the Pio

The microinstruction is then stored in an instruction stack in the nano-control module to be used by the nanocontrol. The nanoinstructions cause addresses, conmands and values to be output onto the control lines by selecting fields from the microinstruction, the value table and the nano-instruction itself. If the nanocontrol determines the microinstruction is. not a branch instruction it causes the next microinstruction to be fetched but not executed until the present instruction execution is com-pleted, unless special conditions exist.

If the microinstruction is a branch instruction the branch address is pretransferred from the branch field to the inputs to the WCSAR. If the branch conditions are met the address is then loaded into the WCSAR. The prefetching and preaddressing of microinstructions is done to achieve the high data-throughput rates demanded of a Pio. The microprogram execution is illustrated in figure 5.3, along with the nanoprogram execution.

The Control Functions - Nanoprogram Execution and Selector and Timing Control Operation

Ultimate system control resides in the nanoprogram and selector and timing logic. They control micro- and nanoinstruction fetching and execution, address incrementing and branching, the internal interface bus structure, the module functions including residual control and the data flow through the interface. Their operation depends upon the five major system states:

- o Wait for any process initiation conditions to be met
- 0 Initiate a process
- o Execute a process
- o Interrupt a Process, and save the state
- o End a process, clear residual control

These states have been described previously in functional terms. The state of interest in this section is the "execute a process[11] state, and it

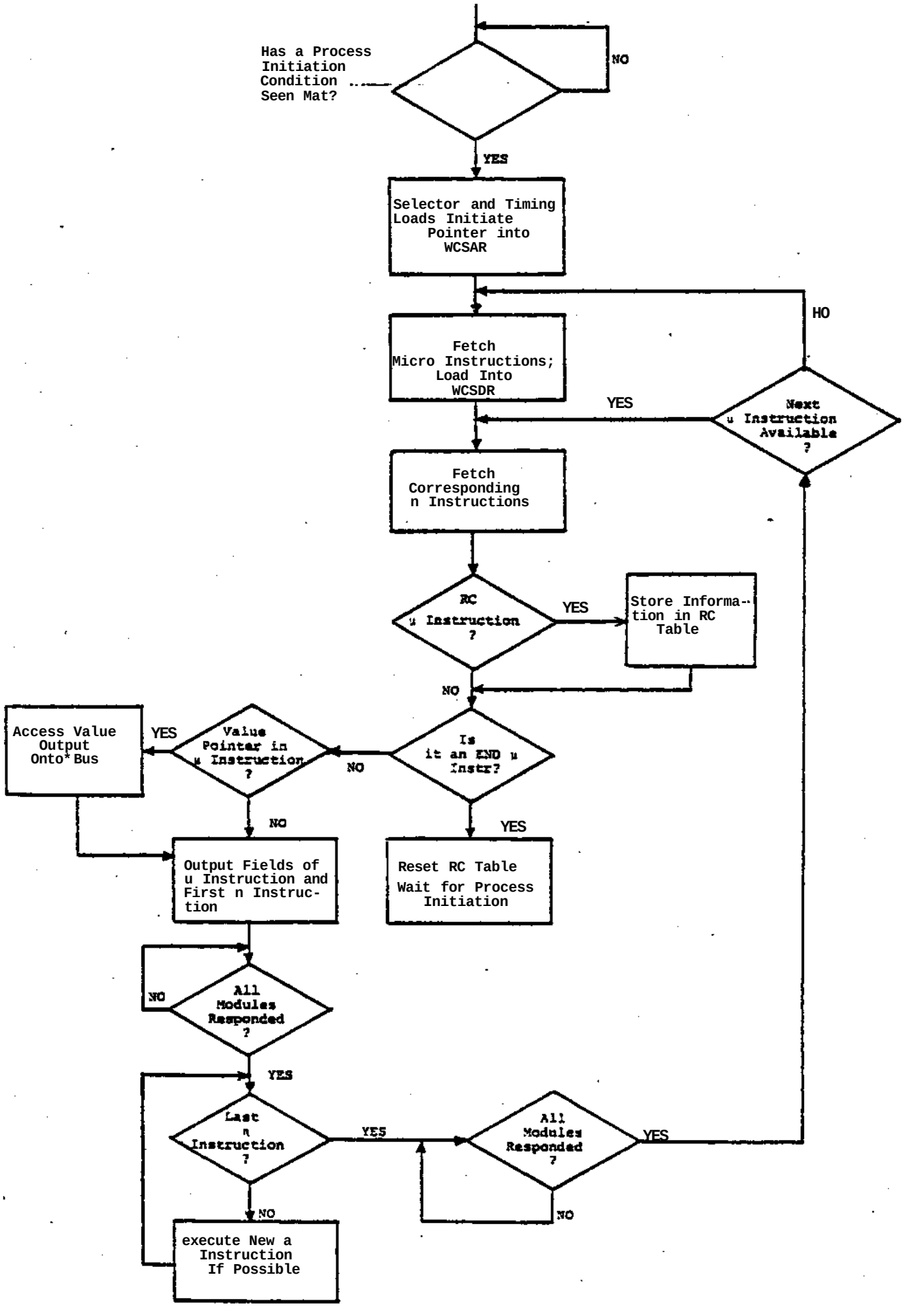


Figure 5.3 Microprogram Execution Control Flow

is discussed here in terms of the overall control flow.

When a process is being executed the nanocontrol accesses the microstore. It loads the microinstruction into the selector and timing control microinstruction register. The opcode of the microinstruction determines the address of the next nanoinstruction word fetched. The physical nanoword, containing seven nanoinstructions, is loaded into the selector and timing block. Each nanoinstruction contains three parts. The first 40 bit part contains fields either to be output onto the 40 bit control bus or to be compared to the 40 sense inputs. The second part and third part (40 bits total) contain bits which: *

- o control the output of the other nanoinstructinn and selected microinstruction fields onto the control bus-
- o are directly output onto the internal control lines tn the nanostore and microstore modules,
- o determine whether the nanoinstruction fields are to be output or be compared to sense lines,
- o determine branching and wait conditions and control fetching and execution of the next microinstruction.

The seven instructions are executed sequentially at minor clock cycle intervals. As soon as a nanoword has-been accessed another is fetched.

The first nanoinstruction fetched is usually combined with selected microinstruction fields and output onto the 40 bit control bus. The selector and timing control essentially outputs control information by multiplexing the nanoprogram and microprogram fields without editing.

If a response is anticipated on the sense lines the pattern expected

is contained in the next nanoinstruction. The nanocontrol either waits until the pattern expected is received or branches depending on the fields of the nanoinstruction which control internal functioning. Each nanoword fetch takes 3 minor clock cycles plus the nanostore access time, and the average microinstruction is executed in five or less nanoword access times. The functions of the selector and timing control are:

- o Output control words to the modules
- o Input sense words from the modules
- o Branch the nanoprogram on sense input conditions
- o Cycle the nanoprogram
- o Cycle the microprogram

On the whole, it is sensitive to input information by comparing the actual sense information with the nanoinstruction. This reduces the functioning of this block primarily to multiplexing,comparing, timing and synchronization. Table 5.1 gives a compilation of conmands which are output from the selector and timing control and which are directed toward the central control functioning. The selector and timing control essentially provides an interface between the control and data-memory part of the Pio.

VII Discussion

The Pio design presented here is in the early stages of development; the next step is simulation of the entire system at a high level. This design is the result of studying the requirements of the computer I/O environment, from high speed disk I/O to lower speed asynchronous I/O, and attempting to merge the necessities in a general purpose processor.

At this point, several conclusions can be drawn:

- o The feasibility of a general purpose I/O processor has been demonstrated.
- o The speeds required of the I/O processor combined with flexibility

I Destination	Command
Nanocontxol Module	Output from ROMDR Input to ROMDR Input to ROMAR Output from ROMAR Access ROM Increment ROM Address
Selector in Nano-control Module	Select ROMAR+1 Select ROMDR Select WCSDR
Output Selector in Nanocontrol Module	Select ROMDR Select WCSDR Select RC Table Output from Selector Access RC Table Load RC Table
Microstore Module	Select WCSAR+1 Select WCSDR Select Initiate Pointer Access WCS Input to WCSAR Output from WCSAR Output from WCSDR Select Bus #1 Select Bus #2 Access Value Table Input WCSDR to Address Value Table Output Value Decode Microinstruction

Table 5.1 Selector and Timing Control Commands

desired in a general purpose processor forces a high degree of complexity into the P_{io} controller,

- o It is difficult and in some cases impossible to achieve electrical compatibility with programmable hardware/ However, gate level compatibility is possible to sane extent and is being implemented in many microprocessors I/O chips,
- o Conventional digital system simulation techniques prove undesirable for I/O simulation since timing, synchronization and multiple asynchronous 'processes are difficult to handle, also and mostly, because conventional digital simulators require a gate level description as input,
- o I/O could be described with a formal high-level language (GLIDE- a Generalized Language for Interface Description and Evaluation) [PAR 75-11] which translated into the Pio microcode,
- o The GLIDE language and I/O primitives provided a basis for ongoing research into the automation pf digital interface and Pio design.

Future investigations of this general-purpose Pio include the possibility of redesigning the architecture into bit-slice data paths so that specific Pio's can be configured and hardwired, as desired. This would greatly simplify control complexity and system cost, with the loss of generality of course.

VIII REFERENCES

[BEL 71] Bell, C.G., and Newell, A., Computer Structures: Readings and Examples, McGraw Hill, 1971.

[DEJ Dejka, William, "Implementation of Mathematical Functions," 1973 Simulation and Modeling Conference, Pittsburgh, Pa. April, 1973.

[HEA 70] Heart, F.E., et.al., "The Interface Message Processor for the ARPA Computer Network", Spring Joint Computer Conference, 1970, pp. 551 - 567.

[INT 74] INTEL Corporation Application Note, 1974.

[JEN 77] Jensen, E. Douglas, "The Honeywell Modular Microprogram Machine: M^3," Proceedings of the 4th Annual Symposium on Computer Architecture, March, 1977, pp. 17-28.

[LES 73] Lesser, Victor R., "Dynamic Control Structures and their use in Emulation", Ph.D. Thesis, Stanford University, 1974.

[NAN 74] QM-1 Manual, Nanodata Corporation, 1974.

[PAR 75] Parker, Alice, "A Generalized Approach to Digital Interfacing," Ph.D. Thesis, North Carolina State University Raleigh.

[PAR 75II] Parker, Alice and Gault, J.W.,"A language for the Specification of Digital Interfacing Problems"? Proceedings of the 1975 International Symposium on Computer Hardware Description Languages and their Applications, pp, 85-90.

[PAR 77] Parker, Alice and Nagle, Andrew " Hardware/Software Tradeoffs in a Variable Word Width, Variable Queue Length Buffer Memory," Proceedings of the 4th Annual Symposium on Computer Architecture, March 1977 pp. 159-164.

[TOR 14] Torode, J.Q. and Kehl, T.H., "The Logic Machine:A Modular Computer System, "IEEE Transactions on Computers, Vol. C-23 No. 11, Nov. 1974, pp. 1164-1168.