NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

A SEGMENTED MICROPROGRAMMED PROCESSOR

by

Steven Vaillancourt

A thesis submitted to the School of Graduate Studies and Research of the University of Ottawa in partial fulfillment of the requirements for the degree of Master of Applied Science in Electrical Engineering.

Ottawa, Canada, 1980.

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

I hereby declare that I am the sole author of this thesis.

I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Steven Vaillancourt

I further authorize the University of Ottawa to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Steven Vaillancourt

ABSTRACT

The use of Large Scale Integrated circuitry in processor design coupled with the demands for increased system reliability and testability have shown the need for new design methodologies in computer architecture. This thesis presents a modular design procedure based on the separability of the control actions during the different phases of instruction execution. The processor hardware is segmented into functional units based on an analysis of the instruction set, with a microprogrammed controller for each segment. The instruction microroutines correspond to low-level tasks that are executed on the functional units under the direction of a task sequencer communicating with the segment controllers by means of handshake signals. In order to prove its practicality the design technique was applied to a bit-slice based implementation of the PDP-11 computer. The major advantage of this approach is the simplicity of each module, which can be designed, tested and upgraded separately.

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to his supervisor, Professor M. Krieger, for his guidance, suggestions and patience, making this thesis possible. He would also like to thank Dr. A.M. Smith for the many helpful discussions while preparing the text. The financial support of the National Science and Engineering Research Council is gratefully acknowledged.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter I

INTRODUCTION

The advent of Large Scale Integration has transformed the design process of computers. The introduction of well defined modules of inexpensive, highly complex elements has led to a more systems oriented design approach. High systems software costs, coupled with the long design times and the availability of inexpensive hardware, have in turn led to more hardware intensive computer architectures.

However, the use of LSI and VLSI circuitry in processors increases the difficulty of completely testing the hardware because internal test points are often not available and the number of tests that must be performed is very large. This problem of testability has by itself stressed the need for new design philosophies. In addition, over the past few years increased emphasis has been placed on the availability aspect of computer systems because most of their cost is now tied up in the software that runs on the comparatively inexpensive Central Processing Units. Increasing the availability of the CPU involves first improving the reliability of the computer architecture through structured design techniques, which then simplify the testing and fault location phases.

The first structured design technique was introduced by Wilkes in 1951. The concept of Microprogramming provided designers with a systematic method to implement the control structure of computers. Microprogramming only became widely accepted in the last decade because of the availability of high speed, high density, low cost memory chips for the microprogram memories as well as complex support chips. In the last few years its use has become increasingly common because of the introduction of bit-slice bipolar logic processing elements and their support circuits. Their use has enabled designers to build custom-tailored, modular, high speed microprogrammed computers.

An alternate approach to the design of large LSI processors is the use of multi-processor systems. Microprocessors and single chip microcomputers are also often being used to implement intelligent computer subsystems.

The objective of modern design techniques is to develop flexible architectures that easily adapt to specification changes, simplify testing and fault location, improve the reliability and can be easily modified for future expansion or upgrading. One way to achieve this is by modularity of design.

This thesis presents the design of a modular processor with interacting segments using a distributed microprogrammed control structure. By analyzing the required information flow during the execution of the various instructions, it is shown that the processor hardware can be segmented into functional units. Each functional unit can be then designed with its own microprogrammed control section. The segments execute microroutines under the direction of a task sequencer, which coordinates the interaction of the segments through handshake signals. Although applied here to the design of a single processor, by using the proper segmentation the concept could be applied to a wide range of digital processors.

Chapter 2 provides the necessary background for the control structure developed in subsequent Chapters. The concept of microprogrammed CPU control is explained by relating it to conventional control unit designs. Then, the two features which characterize microprogrammed control units, the control store microinstruction formats and the facilities for sequencing the microinstructions, are explained in detail to show the strong and weak points of each method. The next step considers strategies to reduce the amount of control memory and thereby the cost of the processor. The final section contrasts microprogrammed control with the hardwired approach to show its advantages and disadvantages in terms

of cost of implementation, speed of operation and flexibility and maintainability of the design.

Chapter 3 presents the concept of segmented microprogramming by considering the processor architecture defined by the instruction set. It is then shown that through the analysis of the instruction set, the data path resources of the processor can be segmented into interconnected but functionally distinct units. Segmented microprogramming is developed as a suitable control structure for the processor with partitioned resources after considering the inadequacies and inefficiencies of adapting a conventional centralized control structure. This control technique is then examined in detail to show its positive and negative aspects with respect to conventional microprogramming schemes and to determine its applicability to different processor architectures.

Chapters 4 and 5 use the design of a segmented microprogrammed emulator for the PDP-11 minicomputer to show the practicality and feasability of the concept. A general description of the PDP-11's functional architecture and instruction set are presented, then the steps for the design procedure based on a segmented microprogrammed control structure are given and applied to the architecture described. The last section of Chapter 4 specifies the architecture of the emulator and provides an outline of the mi-

croprograms needed. Chapter 5 explains the implementation of the computational facilities and control units in sufficient detail to allow the units to be realized.

The final Chapter of this thesis gives a critical evaluation of the concept of segmented microprogrammed control by comparing the efficiency of the implementation, the design time, the performance and the cost estimates with those of other implementations of the PDP-11, since the processor was not actually built. Finally, considerations for future research into the overlapped operation of the segments, reliability enhancements and the application of the concept of task segmentation to other levels of computer architecture are proposed.

Chapter II

MICROPROGRAMMING AND MICROPROGRAMMED ARCHITECTURES

2.1 MICROPROGRAMMED CPU CONTROL

In a computer with a Von-Neumann type architecture, the central processing unit can be separated into a data-flow section concerned with the movement and transformation of information and a control section to direct the operation of the data-flow section and interface with memory and I/O devices.

The data flow section consists of data transformation circuitry capable of performing arithmetic and logical operations, some local storage facilities for operands, addresses and status information, registers to interface with memory and peripheral devices, and the interconnections to enable the transfer of data between storage and transformation units.

The control section interprets and executes instructions supplied incrementally from a program residing in memory. These instructions cause the control unit to issue sequences of control signals which select the operands and the transformation operations that are to be performed upon them,

move data between locations and select and fetch the next instruction to be executed.

There are two general approaches for implementing the control unit. One is to design the controller using combinational and sequential logic circuits that function as a finite state machine. Designed with the usual goals of minimizing the amount of hardware used and maximizing the speed of operation, once the controller is constructed changes can be made only by redesigning and phisically rewiring the unit, hence the name "hardwired control unit". The resulting pattern of interconnected gates and flip-flops is referred to as "random logic", since the control unit thus designed has little apparent structure.

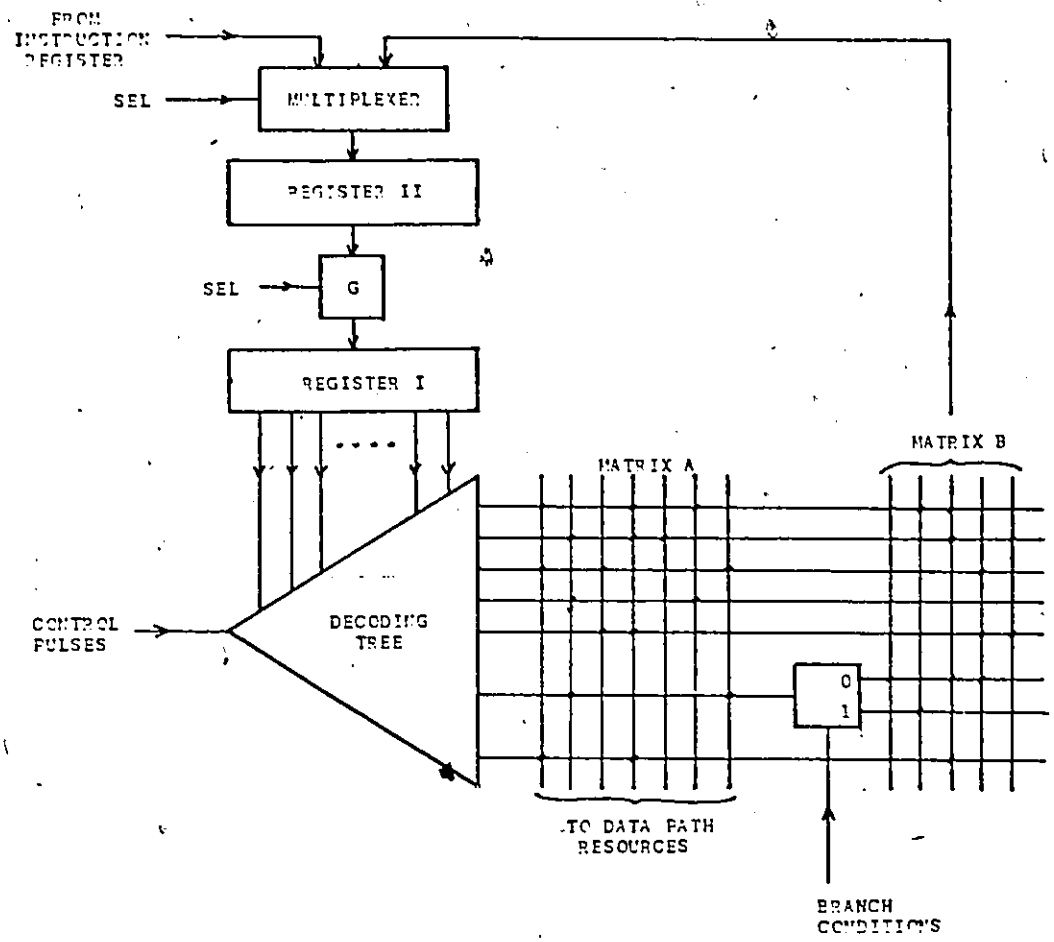
Several methods can be used to realize the hardwired controller. The most formal approach, using sequential circuit design theory, employs state diagrams and state tables to systematically minimize the number of gates and flip-flops. The number of states and input variable combinations for even mildly complex controllers render this technique ineffective because of the size of the state tables, their complexity and the amount of computations required. Other hardwired design techniques are less formal "ad hoc" derivations of the logic circuits from flowchart descriptions of the proposed behaviour of the control unit. They are usually

used in an iterative manner to reduce the number of components .

The other approach is to consider the actions of control signals as elementary low-level steps or "micro-operations". Each machine language instruction can be decomposed into sequences of microoperations called microroutines. The ensemble of these sequences form a microprogram, which is stored in some form of control matrix.

The concept of the microprogrammed control unit was originally introduced by M.V.Wilkes in 1951 as a systematic alternative to the usual ad-hoc design techniques. Microoperations corresponding to each machine-language instruction selected a particular row of the control matrix; the columns of the selected row were output as the control signals. A separate matrix contained the information required to select subsequent rows, based on the current row address and conditional status information. Each row of the combined matrix corresponds to a single microoperation with next address control and is called a microinstruction.

A microprogrammed architecture can be described from the perspective of different users by using the concept of programming levels. The microprograms of a digital computer are stored in a control matrix, usually implemented as a



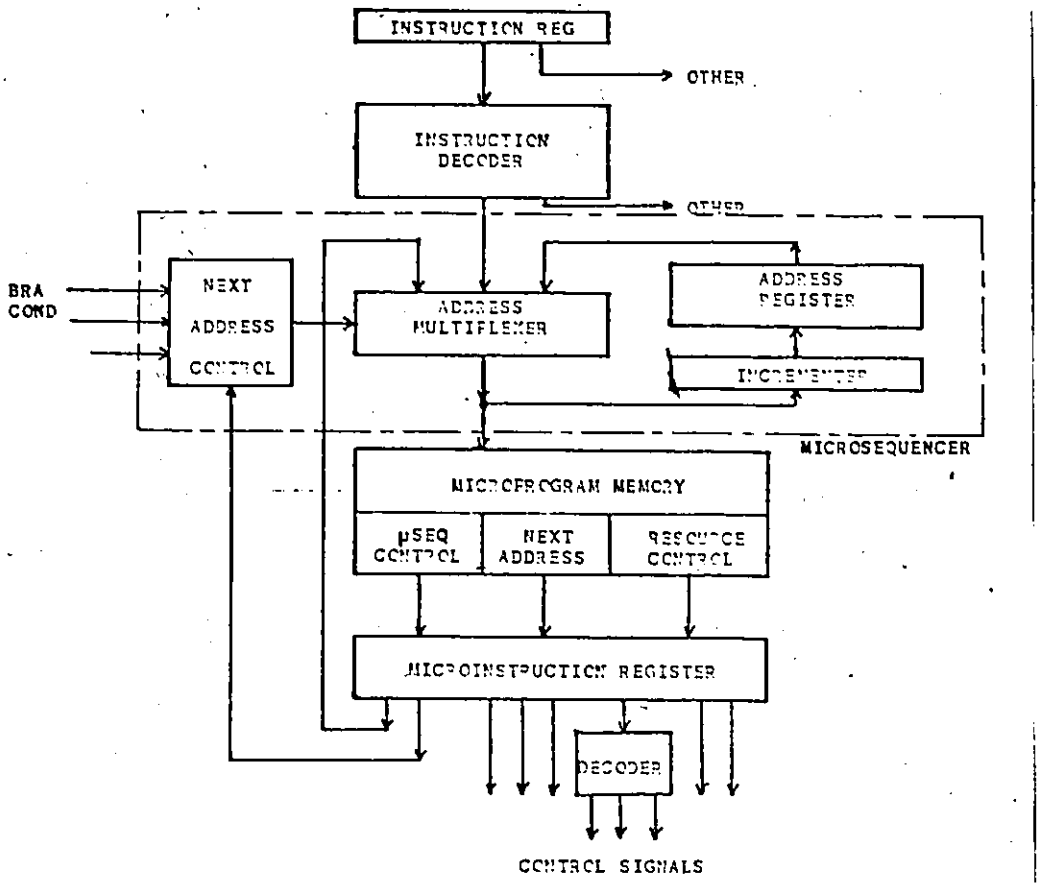
CONTROL SIGNALS for the data path resources are generated by routing control pulses to one of the output lines of a decoding tree as selected by the address contained in register I. The output lines all pass into a rectifier matrix A which encodes the pulses to operate the various gates associated with microoperations. The output lines also pass into a second matrix B, which has its outputs connected through a multiplexer to register II. This register contains the address of the next microinstruction to be performed and is loaded into register I just before the next control pulse is applied. By inserting a two-way switch controlled by external conditions, the microprogram can select either of two separate branch addresses in matrix B. Finally, the initial microprogram address corresponding to the machine language instruction can be input from the instruction register through the multiplexer feeding register II.

Figure 1: Wilkes Microprogrammed Control Unit

high speed memory or logic array internal to the CPU. The microprograms and the data section elements they control can be considered as a "host" computer, fetching from its control memory the proper sequences of microinstructions required for the execution of different machine language instructions. Consequently, the host computer and the externally supplied programs of machine language instructions define a virtual or target computer as seen by the microprogrammer.

Each word read from the control store is called a microword and may represent one or more microinstructions. As in Wilkes' microprogramming scheme, each microword must contain sufficient information to control the data path elements and select the microprogram address that is to succeed the current one. The next address is supplied by a microsequencer which contains logic circuitry that can increment the present control store address, replace it by the contents of a next address field of the microinstruction or generate a control store address based on information in the microinstruction, the present microprogram address and external conditions.

A block diagram of a microprogrammed control unit is shown in Figure 2. A machine language instruction is loaded into the instruction register (IR) and decoded as the start



MICROSEQUENCERS generate the address of the next microinstruction to be executed. This next address can be derived from the previous address by incrementing it, contained in a field in the current microinstruction or can be input as an initial address by the instruction decoder. The selection of the next address is controlled by the currently executing microinstruction and can depend on external branch conditions.

Figure 2: A Microprogrammed Control Unit

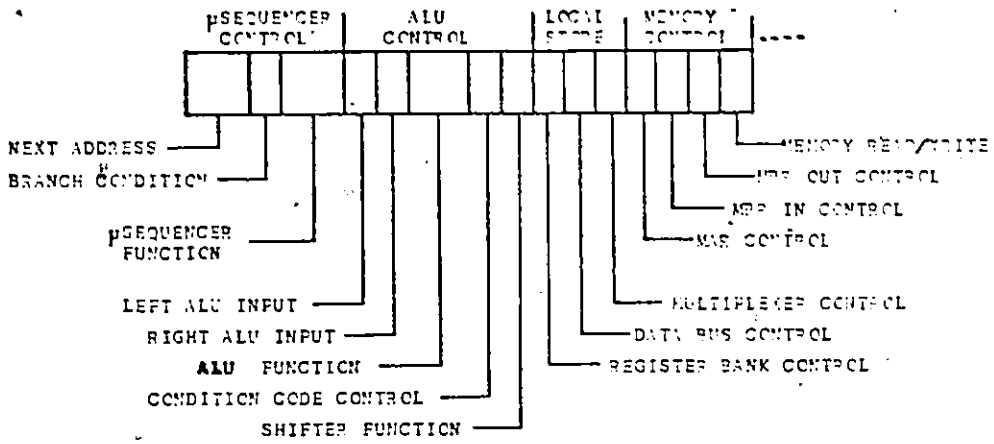
address of the microroutine in the control memory. The addressed microword is read out of the control store into a microinstruction register. The control signals are generated either directly from this register or through subsequent decoding and are applied to the elements of the data flow section. The next address field of the microinstruction is applied to the microsequencer, which selects the succeeding microprogram address as designated by its own control field.

The characteristics of different microprogrammed control units are defined by the microinstruction format used and the configuration of the sequencing capabilities. These two aspects determine the size requirements of the microprogram memory in terms of words and bits per word, the speed of execution of the microinstructions, and the complexity and readability of the resulting microcode.

2.2 MICROINSTRUCTION FORMATS

The format of a microinstruction relates the bits of the microword to the control signals required by the data flow and microsequencer sections of the processor. The simplest way of organizing the information contained in a microinstruction is to allocate one bit of the microword to each control line. Control signals going to the same functional unit or

elementary facility can be grouped into 'fields' so as to structure the microinstructions. This allows all of the microprogrammable facilities to be controlled concurrently, maximizing parallelism in the execution of microoperations. The microcode for these machines is usually quite difficult to generate because the microprogrammer must have detailed knowledge of the microlevel architecture comprising all the data paths, registers and processing elements, as well as a thorough understanding of the timing of the different steps.



CONCURRENCY of execution of microoperations is possible by controlling all the microprogrammable resources in parallel. This involves both the data path resources and the next address control. The example shows the control fields necessary for a simple CPU with a single ALU and a separate shifter, a general purpose register bank and a multiplexed data bus. The microsequencer control fields indicate that the next address generation is quite complex, in order to decrease the size of the microprogram memory.

Figure 3: Horizontal Microinstruction Format

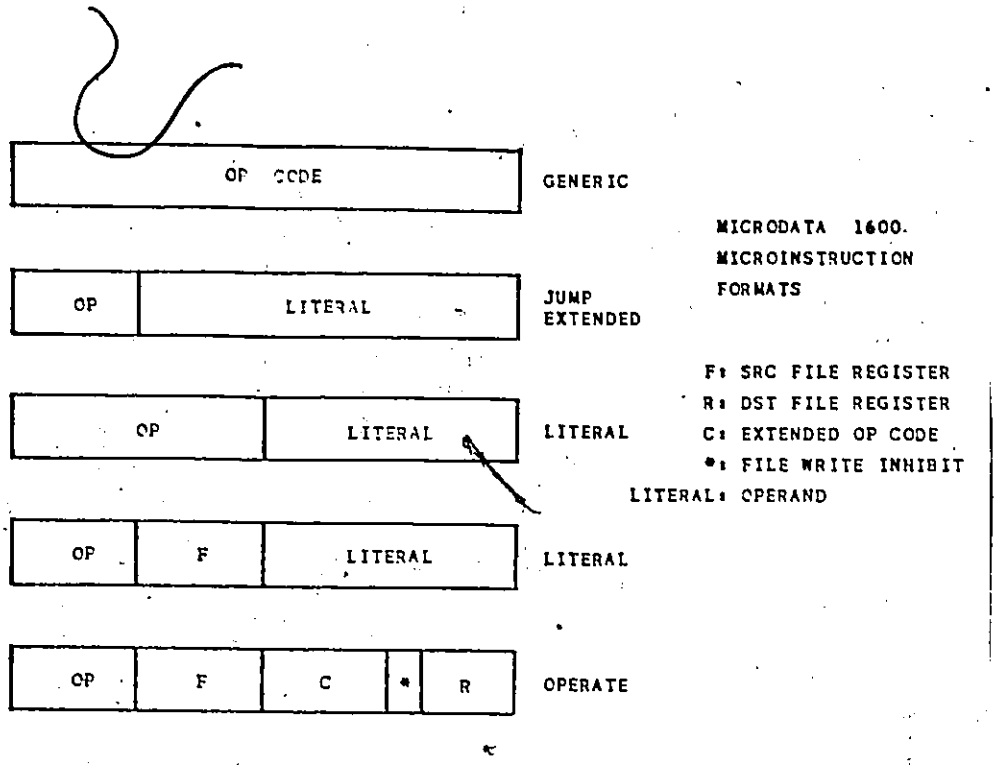
In order to decrease the cost of the control store, the number of bits in the microword can be reduced by encoding the bits in the fields or occasionally encoding groups of fields. The latter technique is used when several microprogrammable resources operate in a mutually exclusive fashion. When all the microprogrammable resources can be controlled in parallel, the microinstruction format is said to be horizontal. Designers often reduce the amount of parallelism possible when control store word size is at a premium, resulting in longer execution times for a few of the microinstructions. This encoding may increase the amount of external logic and overall cycle time because the information in the fields must be decoded before being sent to the unit they control. Recently the increasing use of LSI microprogrammable resources with internally decoded control lines has led to fewer bits in the microword to perform a comparable number of functions.

Further reductions in microword size are possible using microinstructions containing control information in a variable format. One or several fields can be used to determine the routing of other fields in the same microinstruction. The simplest form, called bit steering, is a single level encoding in which a control field routes the other microinstruction fields to the proper decoders or functional units. Multilevel encoding uses the bits of a format control field

to define the format of the rest of the microinstruction. The remaining bits may contain a second level control field that specifies the decoding and routing of some or all of the other fields of the microinstruction. The encoding generally reduces the number of microlevel facilities that can be controlled concurrently per microinstruction and necessitates more time for the more complex decoding circuits.

When the amount of parallelism is reduced to where only a single or very few microoperations can be executed per clock cycle, the format is of the vertical type. The control store width is usually small but there is a large number of microwords. The microinstructions of this class generally contain an operation code field and a control field possibly containing an address or an operand. These microinstructions perform single microoperations, thus bearing a resemblance to machine language instructions. Vertical microinstructions usually have a variable format but may or may not be encoded. Since all sequences of microoperations have to be executed serially in this type of machine, execution time is considerably slower than that of a machine designed with a highly parallel horizontal type microword.

Different types of microprogramming formats can be combined in the design of a machine. One example of this is a technique called Nanoprogramming or address driven micropro-



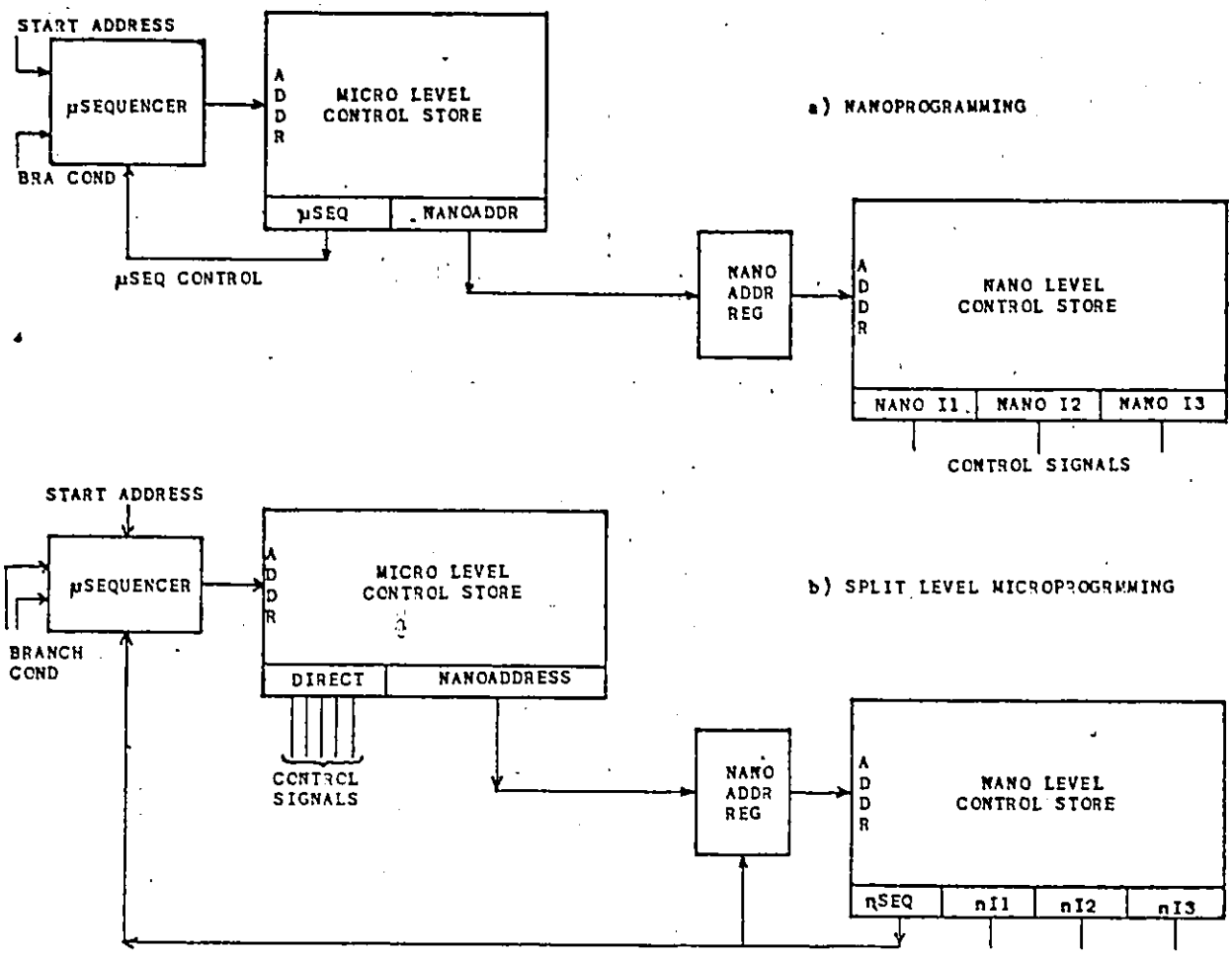
LIMITED parallelism is apparent in this example of vertical microinstruction formats (Microdata 1600). The microinstructions resemble machine language instructions in that only a single arithmetic or logic operation can be performed at a time; also, operands are often transmitted as arguments of several functions. Although each microinstruction can perform only a small number of operations as limited by the machine's architecture, many of these operations are quite complex.

Figure 4: Vertical Microinstruction Format

gramming, where a two-level control store is used. The lower or nanoprogramming level contains nanoinstructions of a horizontal type; the vertical upper or microprogramming level contains addreses or sequences of addresses of nanoinstructions to be executed. The nanoinstructions, and nano-level architecture forming the host computer can present a target

architecture at the microprogramming level so as to emulate other machines. Both the microprogram and nanoprogram control stores can be writeable and the nanostore can have its own nanosequencer. The nanocontrol store usually contains several nanoinstructions per nanoword in order to reduce the control store access overhead.

A variant of nanoprogramming uses split control stores, in which the microinstructions contain some direct control information in addition to the addresses for a second level control store. Either or both levels may be encoded and horizontally or vertically structured. A third technique called residual control is based on the idea that some "environmental" control information is static, as opposed to dynamic, with respect to the frequency with which it is altered. For a given emulation or perhaps only the execution of a single machine language instruction, some control information need only be set up at the start of a series of microinstructions. It can be stored in control registers commonly referred to as "stats" or set-up registers. This technique reduces the microprogramming to the more dynamic control information and decreases microword width by eliminating redundant information. A related technique allows control of multiple resources in parallel by using vertical instruction formats. This is effected by combining microinstruction fields with setup register information to alter the meaning of the field.



TWO LEVELS of control store are used in both nanoprogramming and split-level control stores, but the difference is that the split level scheme uses control signals generated from both control memories at the same time. Several variations exist in the implementation of the nanostore and the microsequencer control, of which two are depicted here. The nanoprogrammed structure of a) uses the microprogram to control the sequencing of microinstructions and uses several nanoinstructions in each nanoword addressed by the microstore in order to reduce the control store access overhead. An alternative control method is shown in b), where the microsequencer is controlled by a nanosequencer field, which also directs the order of execution of the nanoinstructions. Other possibilities exist, including the use of a nanosequencer instead of the nanoaddress register.

Figure 5: Nanoprogramming and Split-Level Microprogramming

2.3 SEQUENCING MICROINSTRUCTIONS

Once the microinstruction format has been defined, interpretation of machine language instructions can be accomplished by executing sequences of microinstructions from the control store. The microsequencer controls the timing of the fetch and execution of the microinstructions and generates the address of the next microinstruction. In the simplest case, the microroutines correspond to consecutive locations in control store, with few branches required. The next microprogram address is obtained by incrementing the control store address register or by loading it with another address. Microprogram addresses may originate from the instruction decoder, special logic, hardwired inputs or a field in the microinstruction. Each microinstruction must contain a microsequencer control field to specify whether to continue sequentially in the microroutine or to load a different address. Entering a new address performs a $N = 2^n$-way unconditional branch, n being the number of bits in the address field. This jump may simply overwrite the present microprogram address, be added to it as an offset or perform a restricted range jump by changing only some of the low order microaddress bits.

Conditions that may produce alternate execution paths require decision making capabilities in the microsequencer. A "no branch" test result causes the microinstruction at the

next sequential microaddress to be executed. If the branch condition is true, there may be one or more branch addresses. In the simplest case the microsequencer would use a branch condition test field to choose between the next sequential microinstruction and the microword at the address defined by a next address field in the current microinstruction. A 2^n-way branch can be implemented by storing two addresses in the next address field and selecting the outcome with a wider test selection field. Cascaded branches can be used where there are more than two branch paths. A more complex branch scheme assembles the next address by combining the contents of the next address field of the microinstruction with the condition flags. This is a powerful technique for multiway branching and leads to compact microcode because of the high utilization of the microinstructions by the different microroutines.

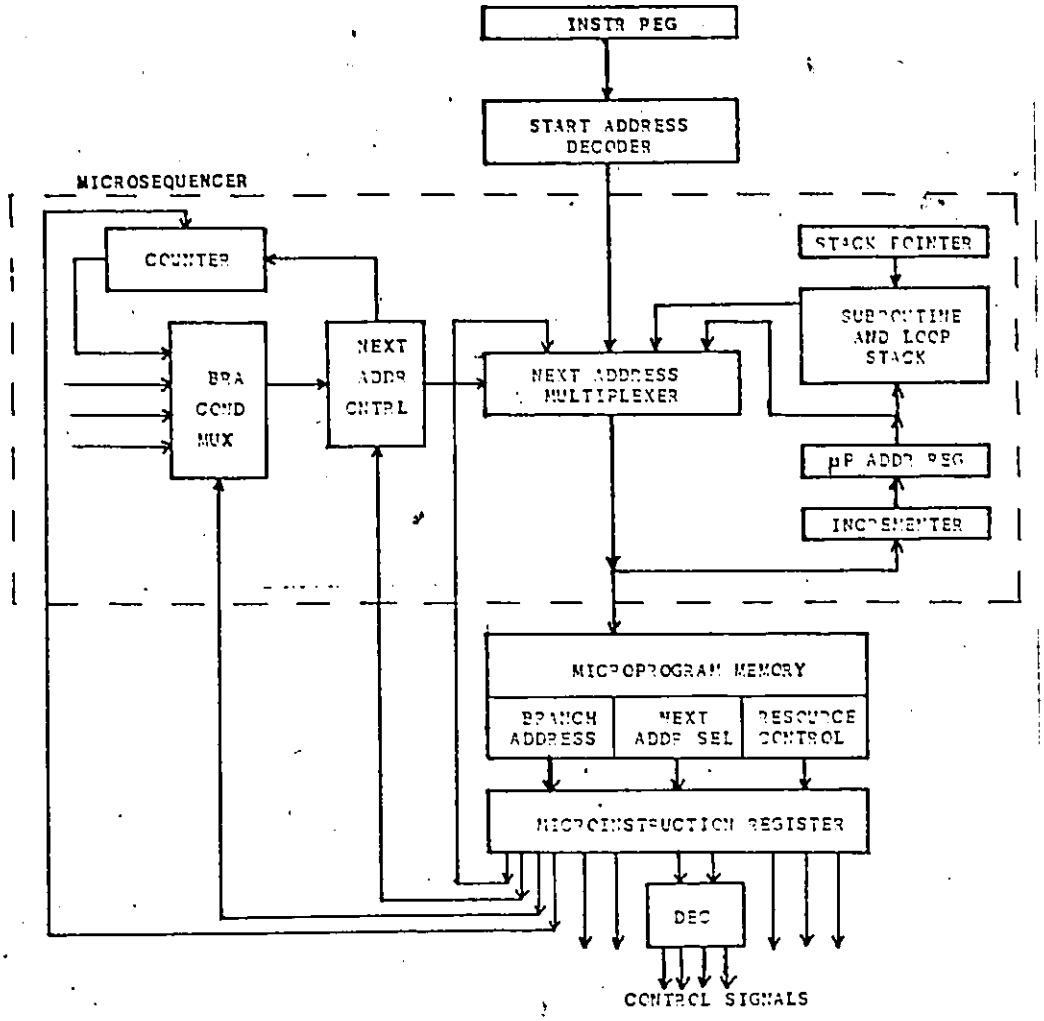
All of the above sequencing methods can be applied to both vertical and horizontal microinstructions. Vertical type microinstructions commonly have sequential microcode, N-way branch and branch-on-condition microinstructions strongly resembling machine language instructions. Because of the few microoperations performed by each microinstruction of this type, it becomes easier and more desirable to share microinstructions between microroutines. Horizontal microinstructions usually contain a next address field and

use highly complex branch logic in order to share as many of the microinstructions as possible between microroutines.

. Another way to reduce the amount of redundant microcode is to use subroutines. The sharing of microinstructions results in an overall reduction in the number of microwords, although it does require more bits in the next address control field. In order to jump to a subroutine, the microsequencer must store the address to which it will return on completion of execution of the subroutine. The return address can be stored by adding to the hardware a subroutine and loop return address stack with a stack pointer to indicate the top element of the stack. A stack permits more than one level of subroutine or loop to be used. However, the microsequencer field must contain bits to control the stack and stack pointer, as well as an expanded source address selection.

The second function performed by the microsequencer is the timing of the microoperations in a microinstruction, which can be either synchronous or asynchronous.

Synchronous clocking of all the microoperations fixes the execution times of all microinstructions, although they can execute in a monophase or polyphase manner. Monophase execution occurs when there is a single simultaneous issue of



COMPLEX microsequencers enhance the capabilities of the hardware by allowing greater variations in the order of execution of microoperations. In addition to the circuitry shown in figure 2, there is a counter for use with repetitive microinstructions and a subroutine and loop stack to enable the microprograms to use common sections of microcode. However, these additional functions require a larger and more complex microsequencer field in the microinstructions.

Figure 6: A Complex Microsequencer

control signals in a clock cycle. When the major clock cycle is subdivided into sub-cycles or phases the execution is termed polyphase. This technique allows control signals to be generated at each phase and interaction amongst the microprogrammable resources, albeit at greater expense and complexity of the microsequencer. A related scheme has several microinstructions contained in a single microword, reducing the control store accessing overhead at the expense of a wider microword and more complex timing circuitry.

The timing relationships between phases in polyphase execution are usually fixed but can be altered by including a clock control field in the microinstruction which acts with cycle stretching logic in the clock circuitry. Varying the cycle length can achieve a higher overall throughput by extending the cycle length for the slower microinstructions, thereby reducing the number of control store accesses for the operation. variable timing is often used with polyphase operation when the memory access time is a multiple or submultiple of the basic cycle length or when certain microoperations cannot be performed in a single cycle.

Asynchronous execution of microinstructions can be accomplished by including a "wait" or "fetch" field in the microinstruction that inhibits further execution of microoperations until the reception of completion signals from other

(specified) elements in the processor. This field can be used to decrease processor idle time on memory references where the access time is not an integral multiple of the cycle time (eg, only wait 2.5 cycles instead of 3 complete cycles). This scheme is well suited for a processor with distributed control or one in which the designer wants to add processing elements with different execution speeds, for instance, different family models or later upgrading the system. Few systems are completely asynchronous, most having a system clock and using the wait or fetch indication field only occasionally.

The microsequencer also controls the timing of the fetching of the next microinstruction. This fetch can occur serially upon completion of the current microinstruction or be overlapped with the execution of the currently addressed microinstruction. Serial fetching is simplest to implement and eliminates problems arising from conditional branches depending on information only available at the end of the microinstruction. The cycle time is considerably lengthened since the microinstruction fetch time is not negligeable in comparaison to the shortest microinstruction execution time.

Overlapped fetching can be made possible by storing the current microinstruction in a control store data register, thereby permitting the output of the control store to change

during its execution. Pipelining greatly increases the efficiency in the case of the shorter microinstructions, but requires more complex circuitry and might lead to an additional fetch in the case of conditional branches. The sequencer can be designed to overcome this drawback by making a "best guess" of the outcome of the branch condition. If the selected outcome occurs, no time is lost; if it doesn't, a null microinstruction is executed and a cycle is lost. Alternately, the cycle length can be extended for microinstructions with branch conditions, solving the problem but slowing execution somewhat.

2.4 MICROPROGRAM OPTIMIZATION

In most processors, there are benefits to be derived from "clever" microcode, which requires an intricate knowledge of microprogrammable architectural resources, machine timing and microinstruction format specifications. This is to be contrasted with conventional software which emphasizes clarity and maintainability. Hence there is a tradeoff between the speed of execution, the size of microprogram memory required and the testability and reliability of the microcode.

Whatever the choice of microinstruction format and sequencer it is usually desirable to reduce the size of the microprogram memory to a minimum. From an architectural point of view, large control stores need more addressing

bits and hence wider microinstructions. More microcode means more debugging time for the prototype. In terms of production costs, larger microprogram memories usually mean higher memory costs, more printed circuit board area, greater power supply demands and longer assembly times. The greater number of chips in the microprogram memory and the sequencer also reduces the testability and reliability of the control unit.

In some cases larger control stores are used to keep the decoding and microsequencer logic simple and the cycle time down. Microprogrammed VLSI chips introduce reasons to reduce the size of the control memory but not necessarily to minimize it. Although chip area is at a premium, memory arrays are the most compact form of integrated circuitry. Recent analyses of die area requirements of dynamic RAMs (16) show that memory arrays occupy only about 40% of the total area, the rest being taken up by address decoders, sense amplifiers, latching circuitry and bonding pads.

Microprogram optimization is one way to increase operational efficiency and can be crucial in some applications, resulting in a reduction or possibly minimization of the size and format of the microinstructions and/or execution time of the microprograms. The numerous strategies are classified under four broad categories : word dimension reduction, bit dimension reduction, state reduction and heuristic reduction (6).

Reducing the word dimension decreases the control memory size, the number of addressing bits required and the execution time. This type of optimization is usually performed by a microassembler in the microinstruction generation phase from code written in a higher level language in terms of single microoperations. These techniques are usually used with horizontal microprogramming formats to merge microoperations across several microinstructions. This is accomplished by combining a means of detecting microoperations that can be executed in parallel, a resource allocation scheme which can overcome conflicting demands from several sources and a scheduling method to take into account the data and operational dependencies between microoperations.

The reduction of the number of bits in the microword depends on the encoding used in the microinstruction format. In completely decoded microinstructions, mutually exclusive microoperations can be encoded to reduce microword length before being grouped into fields. The number of fields in the microinstructions can be reduced by using a variable format. Most techniques formulate the problem in the framework of switching theory in terms of set coverings of the prime-implicant type to strive for minimality of control store size. However, there is the tradeoff involved between control memory size and the flexibility and modifyability of the microprograms. If the microinstruction format is ini-

tially horizontal and completely decoded and is subsequently minimally encoded, most of the advantages of microprogramming are lost because of the extensive output decoder necessary to generate the control signals, and the microprogram is used only to sequence the words in the control store.

The state reduction technique considers the microprogrammed computer as two interacting finite-state machines : a control part and an operate part, described by state transition tables. The number of states of the control part can be reduced through state minimization techniques.

The three methods outlined require large memory and execution time overheads, even in the case of the algorithms not considering all possibilities. The state reduction method is virtually useless for digital computer design because of the very large number of states of the control part of the processor.

Ad-hoc heuristic reduction techniques can be easily applied with a small amount of effort but don't guarantee a minimal control store size. The application of these techniques depends upon the size of the planned control store and the type of encoding of the control fields required by the microprogrammable facilities. Some of the simplest methods involve reducing the size of the next address field, by

shuffling the microroutines to lessen the branch range or by using local branching with a base address register and an offset contained in the microinstruction. Other techniques use external conditions to activate output selectors or combine particular microinstruction fields with the contents of residual control registers to define the function of a resource. Many of the other reduction methods are of academic interest only at present, but are oriented toward microprogram generation from higher level languages which allow the user to write microprograms without being too familiar with the host machine architecture.

2.5 ADVANTAGES AND DISADVANTAGES OF MICROPROGRAMMED CONTROL

Wilkes' primary reason for introducing the concept of microprogramming was to provide a systematic and structured approach to the organization and design of digital computers. Instead of the hardwired interconnection of combinatorial and sequential logic circuits to generate the necessary control and timing siganls, the control information could be stored in a regular memory or logic array and accessed through machine language type programming procedures. This results in a more modular machine organizationn and in designs that are easier to produce and understand; they also require less design time and less use of low density logic circuits.

A microprogrammed control unit is more adaptable to design changes. Final definition of the instruction set can be postponed or modified late in the design cycle without substantially affecting the hardware design, by simply altering the content of the microprogram memory. Conversely, changes in data flow hardware while in the design stage are easily accomodated by redefining or adding bits in the microinstruction fields.

Upgrading an instruction set after completion of the design stage is also feasible, by using additional locations in control store to implement the new or redesigned machine language instructions in microcode. No additional hardware is required, although the macroinstructions will execute more slowly than they would had they been implemented with extra circuitry. In an architecture designed with possible upgrading in mind, extra hardware can be added on and its control assumed by new microinstructions. The microword may have to be widened in order to provide the additional control signals.

Microprogramming produces a virtual architecture as seen by the machine language level user, while implementing it on a host architecture which may be radically different. This feature is often used when an older machine is replaced by a new one. In order to retain compatibility with the user's

software investment, which is usually significantly higher than the cost of the hardware, the manufacturer can provide additional microcode to emulate the older machine while allowing the user to take full advantage of the new hardware facilities by using the 'native-mode' microprograms.

A more recent approach is to use a writable control store (WCS). This technique can solve the problem of providing two or more target machine environments with a single host architecture. It also simplifies later extensions to the instruction set, as well as allowing in some cases user microprogramming. This last concept involves changing the control store content under programmer control to permit the user to tailor the machine to the specific needs of the application, thus producing a target machine that is more efficient and conceptually simpler to program for the application. In some microprogrammable processors there is a basic set of microinstructions provided in firmware, with a separate area for user microprograms. In other machines, the entire control store can be changed and loaded from or stored in main memory.

A writeable control store (WCS) is also useful to implement often used sequences of code or subroutines. Microprogramming these simple routines saves the hardware that would be required in a hardwired implementation and eliminates the

memory space required by a purely software realization. In many applications microprogramming is more efficient than software and more flexible than special dedicated hardware.

Microprogramming can be used to interpret directly High Level Language (HLL) instructions. There is much ongoing research and controversy in this area. Performance and efficiency is ihcreased by microprogramming language translators and/or compilers instead of using conventional machine language software instructions. An extension of the WCS idea called dynamic microprogramming allows the microprograms to be changed by the operating system, so that one microprogram could be used to compile a HLL program while another microprogram could be used to execute the machine language instructions. However, this approach can lead to problems in multiprogramming environments where the computer is continually changing the user program it is executing according to an operating system scheduling program.

Finally there is the maintenance aspect to consider. The data flow section of the machine is often more uniform in design and conceptually cleaner in layout and structure when microprogrammed. The actual number of components may or may not be greater, but the number of different kinds of devices is often reduced due to the regularity of the structure. Similarly the interconnection patterns are simpler and easier

to trace. Documentaton of the processor for servicing purposes is also simplified.

Microprogramming permits simplification of the diagnostic programs, which are used for checking, locating and isolating defective hardware components. Most of the important paths in the processor must be intact in order to execute the diagnostics since the routines are loaded from main memory. Microprogrammed diagnostics or microdiagnostics can either reside permanently in the control store, be implemented as plug in compatible modules or stored in main memory and used through a WCS. Host-architecture level diagnostics permit selected bit patterns to test each individual control path, data path and data processing unit. Once a defective section is located, specific tests for that section can be initiated so as to pinpoint the malfunction more accurately. In the case of a defective processing component in the data flow section, a writeable control store could load a set of microroutines from main memory to execute the operations regularly performed by the malfunctioning unit on some other unit, resulting in a graceful degradation of system performance, as opposed to a complete system failure.

There are certain drawbacks to using microprogrammed control units. The design of a control unit with a control memory and a microsequencer may be too expensive as compared

to the cost of a hardwired implementation. This cost differential is usually offset by the enhanced testability and maintainability of the microcode, as well as the reduced cost of servicing over the lifetime of the machine. However, there are a few cases where this cost may not be warranted. If the digital system to be controlled does not require many microoperations or control decisions, the hardware approach is more economical. As system complexity increases the microprogramming costs rise much more slowly than those of the hardwired control unit. After the initially higher outlay for the microsequencer and control memory, the additional cost is mostly due to microcode that must be written and its support documentation. The complexity at which both implementations are equal in cost has been decreasing steadily over the past ten years, because of a lower initial cost for the microprogrammed control units, so that hardwired systems are becoming less cost-effective with time.

When the production runs are very small, hardwired systems are usually preferred. The cost of mask programming the ROM's, which is spread over all the devices, is still high with respect to the cost of the hardwired alternative. This problem has been alleviated by the introduction of user programmable ROM's or PROMs of the same size, organization and speed of the ROMs.

The major drawback of microprogramming is the reduction in execution speed of the processor. In high speed applications, the cycle time may be longer when microprogrammed because control store fetches occur sequentially, although several microoperations can be overlapped or executed in parallel. This aspect can be remedied somewhat by the use of a decentralized control structure which allows more parallelism, although the hardwired approach will always be faster by a slight margin because both logic and memory improve in speed and density with new technological developments.

A writeable control store can lead to operating system level problems because of the multiplicity of execution environments and libraries of microcode, causing an operational loss of efficiency. There are also some disadvantages of using microdiagnostics, such as not being able to catch intermittent failures or check to see if the machine meets the architectural specifications.

To conclude, there are several facts that can be gathered about microprogrammed Control Unit implementations. The cost of the implementation is determined by the size of the control store, the amount and complexity of the hardware needed external to the control memory to decode and sequence the microinstructions and the maintainability of the micro-

code. Its speed of operation is dependant upon the timing scheme and the levels of complexity of the microsequencer, the branch logic and microinstruction decoding circuitry. Finally, the flexibility of a control unit results directly from the simplicity of the encoding of the fields and the writing of the microprograms by utilizing the sequencing facilities provided by the hardware of the microsequencer.

Chapter III

FUNCTIONAL DIVISION OF THE CPU

The instruction set is the starting point of any design technique and defines the architecture of the computer. Classification of the instructions provides a structured way of examining the functions performed by the architectural resources of the CPU. By first breaking down the execution of an instruction into a series of linked low-level tasks and then considering the hardware resources needed by these tasks, it will be shown that the CPU can be segmented into functional units and that a decentralized microprogrammed control scheme is the simplest way of controlling these functional units.

3.1 INSTRUCTION BREAKDOWN

3.1.1 Instruction Classes

Instructions for general-register or accumulator type processors can be separated into five different classes based on the number of operands and the type of operation to be performed : double operand, single operand, move (load/store), program control and system control. Using Backus-Naur Form, these classes can be expressed as shown in Table 1.

TABLE 1

Instruction Classes

```
 double operand ::=<op code 1><operand 1><operand 2><result>
 single operand ::=<op code 2><operand><result>
          move ::=<op code 3><source><destination>
program control ::=<op code 4><address>
 system control ::=<op code 5><address>

where <op code i> ::=<operation j><modifier k>
                    |<operation j><condition k>
                    |<operand value>
                    |<addressing mode><address>
      <address> ::=<register address>|<memory address>
                    |<displacement>|<null field>
```

The main characteristics of these instruction classes are detailed below.

Double operand (Dop) instructions perform an arithmetic or logic operation between two operands and store the result at the destination address provided. Single operand (Sop) class instructions operate on a single operand from a source address and store the transformed result at a specified destination address. The last source address referenced before the op phase is usually also used as the result destination address, except for computers which have 3-address instruction formats. Also, the Sop class of instructions have a greater range of arithmetic and logic operations than those of the Dop class because more bits of the instruction are available to encode additional functions.

Move-type instructions transfer data or instructions from one register or memory location to another. There is no modification of the data during the operation. This class includes move, load, store and exchange operations, which may be performed between memory, CPU registers or input/output (I/O) registers. These instructions contain both the source and destination addresses either implicitly or explicitly, although one of the addresses may be implied by the opcode for special operations such as stack manipulations or the loading of special purpose registers.

The last two classes deal with the sequencing of instructions to the CPU and the interfacing with the console and peripheral devices. The program control class consists of the branch and short jump, subroutine call and return, extended address jump and condition code modification instructions. The system controls are the basic operator CPU commands such as halt, wait, reset, etc, as well as software interrupt and return instructions and input/output control instructions.

A general characteristic of the instruction classes is that each instruction can be broken down or pipelined into five distinct phases consisting of an Instruction Fetch phase, three execution phases to execute the decoded instructions and a final phase to service interrupts. The in-

struction fetch and decode phase is performed for all instructions.

The three execution phases deal with the different aspects of the execution of the instruction as it affects the physical resources of the processor. These execution phases are centered around the arithmetic or logic operation to be performed and are labelled the pre-operation (pre-op) phase, the operation (op) phase and the post-operation (post-op) phase. Each of these execution phases gives rise to a task or tasks out of a subset available only to that particular phase. These tasks perform the sub-operations necessary for the completion of each execution phase.

The Service phase is only executed when an interrupt occurs. The CPU then stores the current processor state in memory and changes the program address to the first instruction of the interrupting routine. This instruction is fetched when the processor executes the next Instruction Fetch phase.

The execution phases and tasks performed in a typical double operand instruction are as follows. First, the operands are fetched and sent to buffers in the arithmetic logic circuit during the pre-op phase. Next there is the arithmetic or logic operation phase, which transforms the

data, places it in a result buffer and updates the condition codes. Finally, during the post-op phase, the data from the result buffer is stored in general purpose registers or memory.

The Instruction phase breakdown described is a general model and thus not directly applicable to all instructions a processor may execute, since all three execution phases are not always performed. Double and single operand instructions execute all three phases except on "test" type instructions, where there is no post-op phase because the result is not stored. Move instructions theoretically should execute only the pre- and post-operation phases, but in some processors, such as the PDP-11, the condition codes are modified, so that these instructions must be considered as performing a null op phase operation. Program and system control instructions that affect the status flags must also perform an op phase; otherwise they may execute any combination of the phases. Finally, many of the complex system control instructions are actually "hardwired" macros of machine language instructions. They are included for completeness in the instruction set and execute as pre- and/or post-op sequences.

3.1.2 Execution Phases

The execution phases will be examined in detail as they would be performed on a conventional multiple register CPU in order to establish the hardware resource requirements of each phase. The pre-operation phase fetches one or two operands from the general purpose registers or from memory and stores them in the ALU operand buffers. In accessing the operands it may use the ALU to perform simple calculations with the addresses for indexing, incrementing and decrementing. This phase may also store values in the general registers or memory. These functions require access to the memory address register, the memory data buffer register, the general purpose registers, the ALU input and output buffers and the output of the processor status register.

The operation phase is concerned with the actual arithmetic or logical operation and is the only phase in which the contents of the Processor Status Word (PSW) can be changed. This phase needs an ALU circuit with a wide range of functions. It must also access the ALU input buffers, the result buffer, the condition codes in the PSW and a scratch pad memory for intermediate results.

The post-op phase stores the op-phase result into the general registers or memory for the Dop, Sop and Move classes of instructions. The program and system control

classes require more complex post-op tasks that may calculate addresses and interact with the console, processor status and I/O functions. This phase accesses the general registers, the memory address register, the memory data register, the ALU, the PSW output, the I/O hardware and the console. It also accesses the Instruction Register in order to extract constants and offsets for program control instructions and when fetching new instructions.

The above shows that the resources required for the pre- and post-op phases consist of memory and operand accessing hardware which is the same for both phases except for the Instruction Register and the I/O functions, although the latter can be functionally isolated. All three phases need an ALU, but the ALU of the op phase must perform more complex arithmetic operations as well as logical functions. Therefore it's circuitry would be more complex and several times slower than the ALU required by the other phases. In most cases it is more efficient in terms of execution time to provide a separate address modification unit (AMU) for the pre- and post-op phases. The addition of this AMU allows the op-phase ALU and PSW to be separated from the rest of the resources.

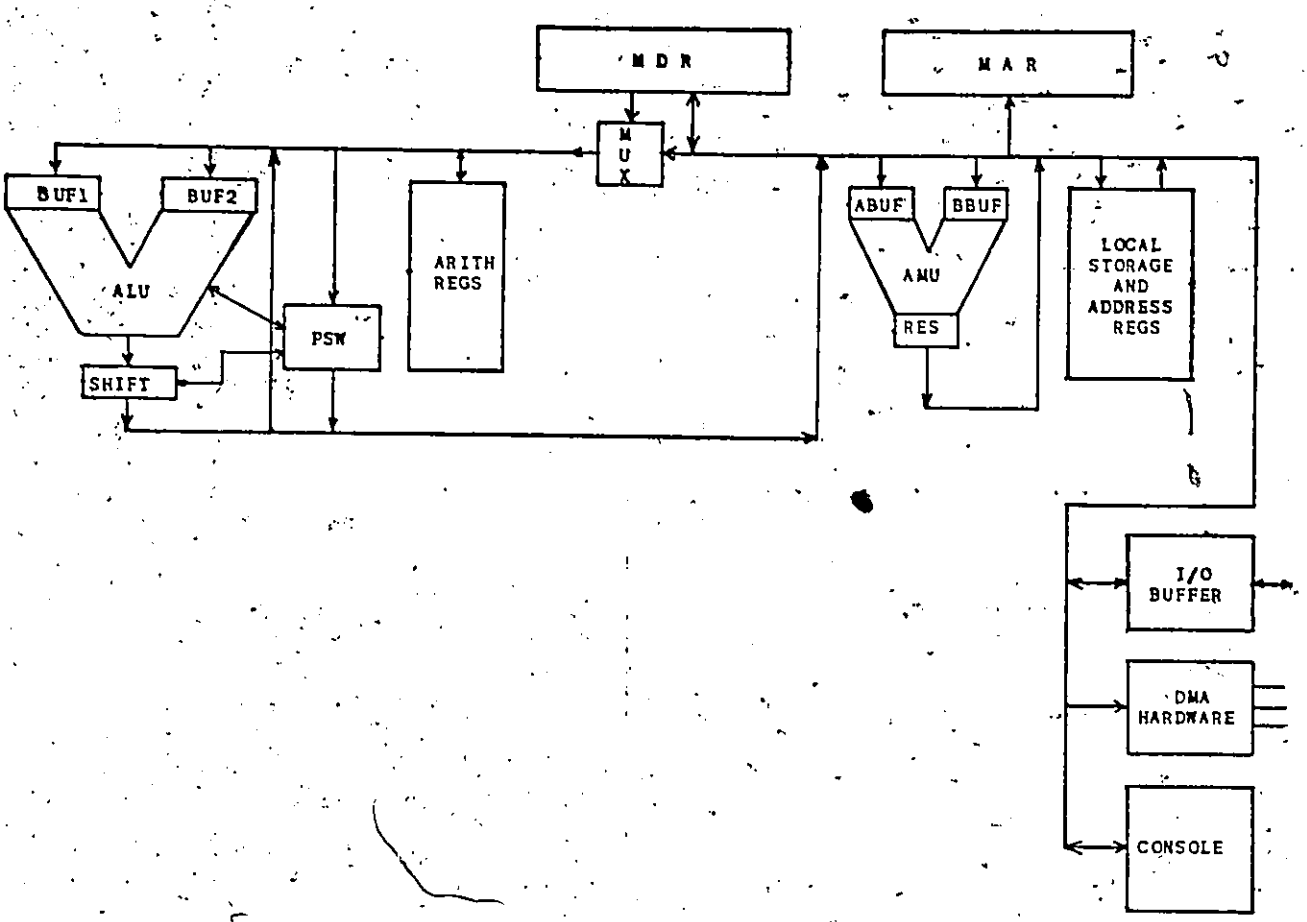
The hardware can now be partitioned into three functionally distinct segments : the op-phase ALU and PSW, the me-

mory and operand accessing resources and the I/O hardware (see Figure 7). Each of these segments can perform a set of tasks that is distinct from those executed by the other segments. The decoded instruction merely has to specify the tasks and the order in which they are to be executed.

3.1.3 Segmented Processor Architecture

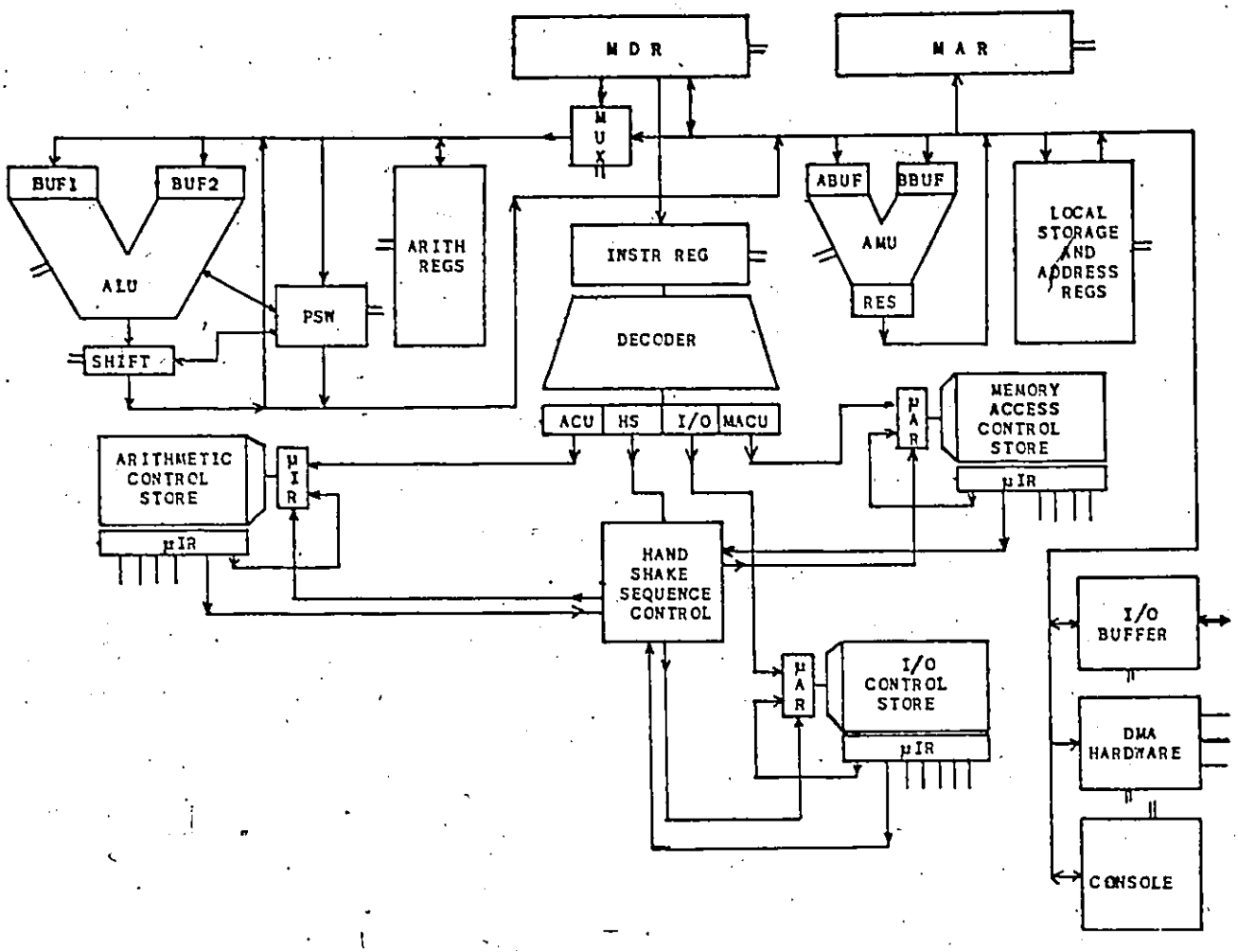
Originally a centralized control structure was considered. However, segmenting the hardware resources and breaking the instructions down into phases and tasks suggested restructuring the control mechanism so that each segment would become a separate functional unit with its own controller. Each segment could then function independently and interact with the other segments by transmitting or receiving data over interconnecting lines. The interaction between the segments would be controlled by means of handshake signals exchanged with a sequencing unit, which would initiate and terminate the different tasks according to the order specified by the instruction.

The architecture of the Segmented Processor can be refined with these developments so that each segment is tailored to its specific functions. A block diagram of the General Purpose Segmented Processor is shown in Figure 8.



RESTRUCTURING the processor's data flow section hardware into segments based on functionality permits the tasks for each segment to run on hardware locally optimized for these functions. The Arithmetic segment contains a complex Arithmetic Logic Unit, a shift network, the Processor Status Word containing the condition codes and some local storage for use by the arithmetic operations. The Memory Access segment interfaces with the MBR and MAR and has an Address Modification Unit to calculate offsets and indices with the data contained in the local storage and address registers. The I/O segment contains I/O buffers, Direct Memory Access hardware for high speed peripheral-memory transfers and also interacts with the console.

Figure 7: Segmentation of the Hardware



SEGMENTATION of the processor is extended to the control structure as shown. Central to the control of the CPU is the Handshake Sequence control unit, which orders the execution of the tasks specified by the instruction decoder for the ACU, MACU and I/CCU through handshake signalling lines to and from these units. This distributed control scheme allows the data flow and control hardware for each segment to be designed and modified separately.

Figure 8: General Purpose Segmented Processor

In the architecture of Figure 8, the Arithmetic segment performs all the arithmetic and logic operations specified by the op codes of the instructions. It contains all the data registers of the processor, which can be general purpose, scratch pad or dedicated registers for integer, floating point, extended length or other types of operands. There can be more than one general arithmetic/logic circuit, such as multi bit shifters, hardware multipliers and dividers as well as floating point units.

The Memory Access segment interacts with main memory and has access to the arithmetic segment's input and output buffers. It consists of most of the non-data registers, which include the program counter, stack pointer, index and other addressing registers that can be used for indirect addressing and table accessing. There is a simple arithmetic circuit to perform additions and subtractions for offsets and loops, as well as incrementing or decrementing.

The I/O segment communicates directly with all external devices except memory. It contains the input and output data buffers and the priority interrupt system. This segment may also have a direct memory access controller and event counters to facilitate interfacing with high speed or special purpose peripherals.

The rest of the processor circuitry decodes the instructions and controls the order of execution of the phases and tasks by sequencing handshake signals to and from the control units of each segment through a Handshake Controller. In order to show the validity of the proposed distributed control structure, it will now be developed from the concept of execution phases and tasks.

3.2 CONTROL STRUCTURE

The control structure of the processor with segmented resources has to decode the instructions into execution phases, each phase specifying a task or set of tasks to be executed on the associated segment. The Instruction Decoder also must specify the order of execution of the tasks in the phases.

With this information the control structure must be able to initiate and terminate the phases and tasks in the right sequence, allow consecutive or concurrent operaton of the phases and tasks as determined by their resource requirements and supervise their interaction without incurring significant logic or delay overhead. It must also be a flexible design so as to allow changes to be made in the tasks of each segment. Finally, the control structure is to be microprogrammed in order to provide a more structured and systematic design approach. Each execution phase task can

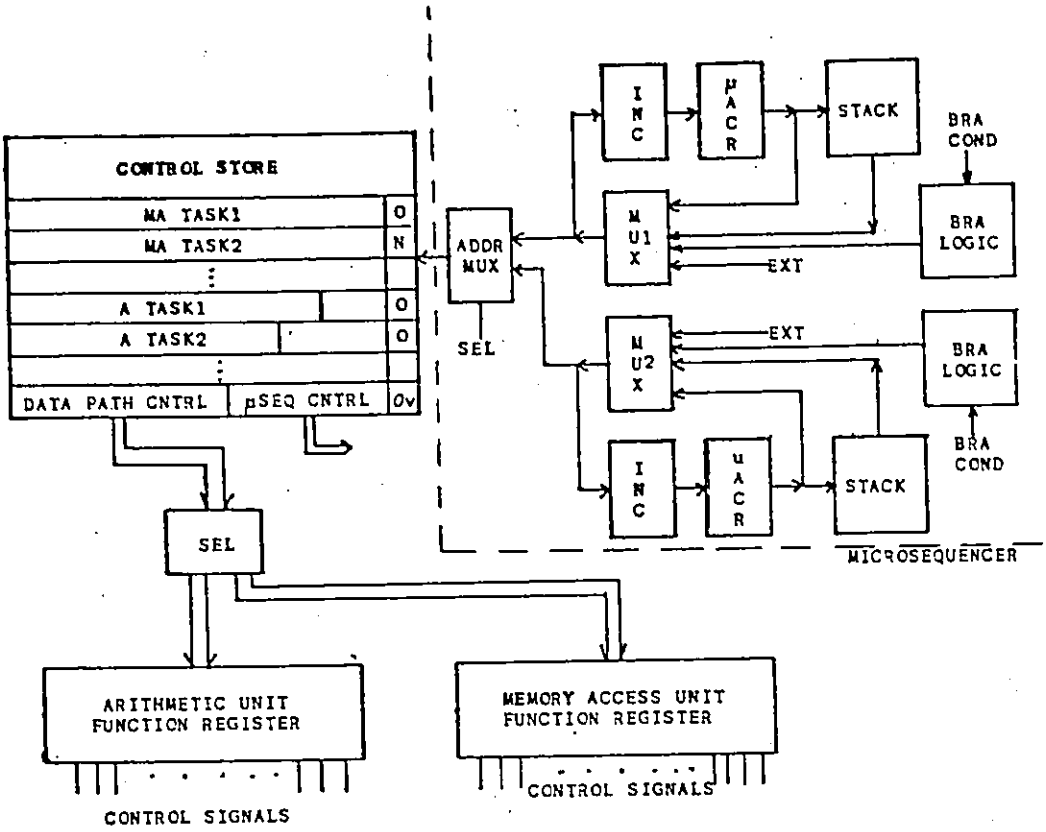
then be made to correspond with a microroutine in the control store.

Several conventional centralized control techniques described in Chapter 2 could be used to implement the control structure of the processor with segmented resources. However, by examining each possibility it will be shown that the microsequencer facilities are very complex, the microcode contains many redundant or unused bits, overlapped operation of the functional units is very involved and that modification of the microcode for one or more functional units is very difficult once the microprograms are written.

The concept of a task of sequential elementary operations suggests the use of vertical microprogramming. The different functional units could be controlled by using a variable format encoding scheme, although the encoding would be necessarily quite complex because a number of different tasks can be executed by each functional unit and these tasks may use different hardware resources. Also, a highly complex microinstruction format makes it difficult to change the microcode when modifying the hardware it controls. The major drawback of this scheme is that there is no possible overlapping of the operation of the functional units.

This situation can be remedied by using a form of residual control in order to allow several tasks to execute concurrently by switching control from one task to another. This scheme would require very complex microsequencer facilities in order to be able to alternate between separate task sequences executing in different areas of control store. The control sequences would be interleaved or time shared because of the need to continually change the contents of the residual control registers as the task sequences progress. Therefore this type of control unit would overlap the execution of the functional units only if the microoperations executed by the different functional units take longer than the time to set up a residual control register and if the microoperations performed on one of the functional units take at least twice as long as those executed on the other functional unit. Subsequently, the throughput would be dependant upon the particular tasks to be overlapped and may not result in any improvement over sequential execution of the tasks. A system of this type is shown in Figure 9.

Horizontal microprogramming allows maximum concurrency since it can be designed to control all functional units in parallel. To control the segmented resources, a completely decoded horizontal format would require a microword of considerable length because of the multiplicity of the functional units, whereas an encoded format would reduce the ex-



OVERLAPPED operation of a segmented resource processor with memory mapped I/O and a vertically microprogrammed control store requires the use of a double microsequencer and a form of residual control which alternates between the functional units. Since the tasks for both units reside in the same microprogram memory and control substantially different hardware, each microinstruction must provide overlap information for the microsequencers while containing the control signals in a highly encoded format.

Figure 9: Vertical Microprogramming with Residual Control

ecution speed because of the complexity of the microinstruction decoder. Although the horizontal format has the capability to execute many microoperations concur-

rently, the tasks that are to be performed in parallel have to be determined and microprogrammed at the design stage. For example, in order to overlap a floating point arithmetic operation with a memory access, the microcode for all addressing modes that can be executed in parallel with the floating point operation have to be determined in advance and stored in the microprogram memory. A horizontal format would result in a greatly enlarged control store listing all possible combinations of overlapping microoperations. Much of this microcode would be redundant, since microoperations would be repeated in many microwords. Also, since not all tasks could execute concurrently, there would be a large number of null (NOP) fields in the microprogram memory (See Figure 10).

MICROPROGRAM MEMORY			
TASK1	TASK2	TASK3	TASK4
TASK1	NOP	TASK5	TASK4
TASK1	TASK2	NOP	TASK6
TASK1	NOP	TASK3	TASK6
TASK7	TASK8	TASK3	NOP
NOP	TASK8	TASK5	TASK4
OTHER MICROINSTRUCTIONS			

PARALLEL execution of tasks must be determined in advance by the microprogrammer. The tasks shown are divided according to the microoperations they perform on specific hardware instead of being grouped according to which segment they operate on, since the determination of the microoperations that can be overlapped depends on conflicts with the resources of other segments as well as contention within the same segment. As shown, a single task can often execute concurrently with many other combinations of tasks, leading to needless repetition of the microcode and in many cases the insertion of no-operation fields.

Figure 10: Horizontal Format Overlapped Microprograms

The use of nanoprogramming or split level control stores doesn't alleviate these problems because the lower level nanoinstructions would still have to store all possible overlapping combinations. Applying residual control techniques would allow interleaving of the control functions or overlapping of some slower microinstructions at the expense of complicated sequencers and decoders.

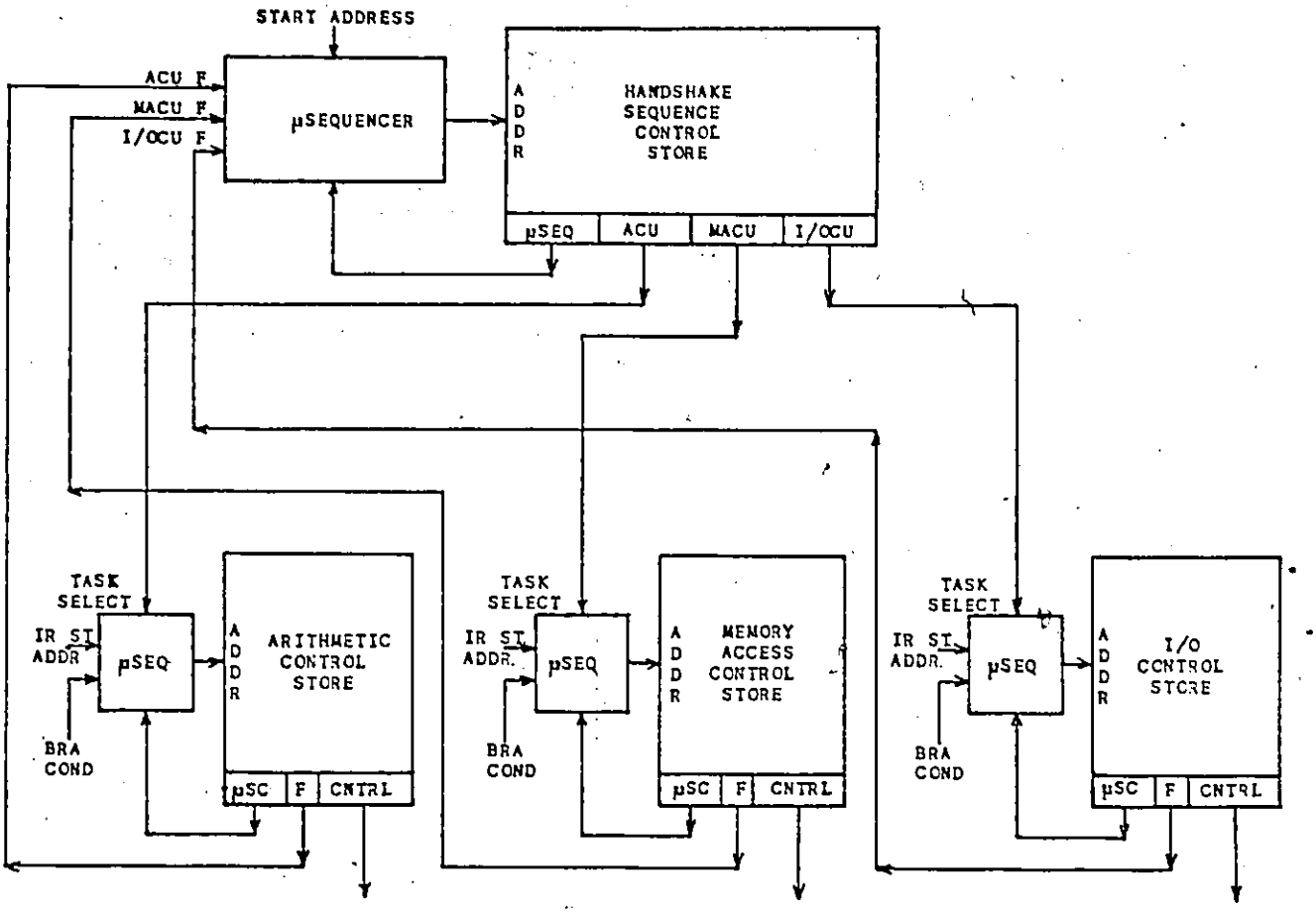
A distributed processor control scheme can overcome many of the difficulties encountered by the conventional methods previously outlined. The tasks and functional units, being distinct, can be logically segmented so that a control store can be used for each separate functional unit. Instead of a very complex central control store and microsequencer, each of the segmented control units can be designed with its own microprogram memory and sequencer, which can be made very simple.

The proposed method of control resembles nanoprogramming but has instead a nano-type control store dedicated to each segmented functional unit. The micro- and nano-controllers function in a hierarchical fashion with all communication between nano level controllers being routed through and interacting with the single micro level controller. This mi-

cro-controller acts as a handshake signal controller and task sequencer, receiving the task nanostore start addresses and the order in which they are to be performed from the instruction decoder. The tasks to be performed by the machine language instruction are sequenced by sending "enable" signals to and by receiving "finished" or "busy" signals from the segment controllers.
A system block diagram is shown in Figure 11.

The concept of separating a processor into different intelligent units is not new. Distributed processing appears in large computers in the form of Input/Output controllers, front-end processors and multiple arithmetic units. Bidirectional signalling between controllers was implemented on the Solomon computer and the IBM 360/91 floating point arithmetic unit. It was used most extensively in the Illiac 2 where it was found to require considerable overhead.

There has been a previous suggestion to use separate control memories for each functional unit (10). The application dealt with the overlapping of instructions on a stack type computer by using an interlock technique with inhibit bits generated by pending instructions. The object was not the definition of a design procedure based on the concept of separability of the control sequences, nor was there any indication of overlap within the instructions themselves. Also,



SEGMENTED microprogramming uses a two level control store. The upper level is termed the Handshake Sequence Control Store, the output of which selects the tasks to be performed by the second level Arithmetic, Memory Access and I/O Control Units. These controllers generate the signals for their segment's microprogrammable resources, control the sequencing of their microinstructions within a selected task and indicate the completion of a task to the microsequencer of the upper level control store. The HSCS microsequencer in turn waits for the proper completion signal to trigger it and then steps to the next task in the sequence specified initially by the instruction decoder.

Figure 11: Segmented Microprogrammed Processor Control Unit

the seqmentation used was more of a brute force method instead of a loqical seqmentation of the resources, as it may lead to excessive fraqmentation of the microcode and thus an inefficient control structure.

3.3 ADVANTAGES AND DISADVANTAGES

Seqmented microproqramminq has the same advantages and disadvantaqes that conventional microprogramminq techniques have with respect to hardwired control. However, it is different in that a control store is dedicated to each seqmented functional unit while the seqments are sequenced by a handshake siqnal controller. The advantages and disadvantaqes of seqmented microproqramminq as compared to conventional microproqramminq methods will be considered by coverinq the architecture of the processor, the reliability of the design, its expansion capabilities, its adaptability and the hardware required for implementation.

Seqmented microprogramming has several advantages over conventional techniques. Its main feature is the simplification of the design process by reducing it to the design of separable modules. Each functional segment is simpler, easier to design and is controlled separately. The interface between seqments consists of data buses that transmit the information to be operated upon by the tasks.

Several of the hardware components of the control section are simplified. The microsequencers for the control memories need only have increment and load capabilities, as the microprograms are sequential most of the time. The instruction decoder is used mainly to differentiate between the different classes of instructions and generate the handshaking bits required, instead of decoding the instructions into an address for a central control store.

The microcode is much simplified and smaller in size since each functional unit only performs simple tasks. The compaction of the microcode is achieved by the elimination of most of the redundant code used in conventional designs to control the inactive functional units. The dedicated control aspect of each microprogram memory allows inherent overlapping of the tasks of the different units. In terms of software support, the microprograms for each module would be developed and simulated separately, since they only interact through the handshaking bits, which could also be easily simulated. Finally, since the amount of code necessary for each unit is fairly small and mostly sequential, it is faster to design and easier to debug.

This control scheme can enhance the reliability of the CPU because the segments can be singled out and tested separately, greatly simplifying the debugging procedures. The

simplest type of test is the catastrophic failure type, where if a segment fails to respond after a known delay, a flag is set and a trap occurs. This could be implemented easily by using time-out circuits on the "finished" and "enable" signals interconnecting the Arithmetic controller, Handshake controller and Memory Access controller. This procedure could also be applied within each segment to detect total failure of the resources under microprogram control.

Segmented microprogramming makes it easier to expand present systems and/or selectively upgrade them by introducing new circuitry. This may be done by modifying the instruction decoder and replacing the control store of a segment with an upgraded version, by adding new microinstruction sequences to the segment's present control memory or by expanding it in length and/or width. For example, if a hardware multiplier is added to the present circuitry of the Arithmetic segment, either additional microinstructions or additional bits (width of microinstruction) can be used, depending on the complexity of the required control signals. Finally, if a floating point section were added on, due to its complexity a separate control store could be used to allow more overlapping or the existing microprogram memory could still be used with path selectors to route the control signals to the proper subunits. Thus, by segmenting the mi-

croprogramming, the processor organization is adaptable to design changes and segments can be modified separately without affecting the performance or design of the other units.

There are two major areas where the use of segmented microprogramming could be disadvantageous. A segmented microprogrammed processor needs more control hardware and possibly multiple ALUs and register banks. There is, however, a certain amount of tradeoff between the cost of the extra circuitry in terms of ICs, wiring, printed circuit board area, power supply requirements and cooling facilities, and the reduction in complexity of the chips used to implement the microsequencers and control stores. In the segmented microprogrammed version of a processor, the circuit complexity of the three interacting control units is about equivalent to the amount of circuitry required for a single, more complex conventionally microprogrammed control unit. (see Ch. 6 sect. 1) If the processor were implemented in LSI it could be implemented as 3 or 4 chips or in multiples of these numbers by using bit slice techniques. Each functional unit or group of integrated circuits could be upgraded individually.

Also, the general organization uses separate ALU's and register banks to allow overlap and a higher throughput, along with greater microprogramming flexibility. Multiplexors can be used to run a single ALU and register bank from

both the Arithmetic Segment and the Memory Access segment control stores if no overlap is required. These controllers would be of the same complexity as those used for the implementation with multiple ALUs, the control bits of the unused resources being generated by hardwired lines into the microinstruction buffer registers. Expansion for further arithmetic hardware devices would still be possible but would require a quasi-residual control technique to handle the hardware that would be inactive during the operation of the alternate controller.

The second major area concerns the execution speed. A segmented microprogrammed processor would be about as fast as a conventionally microprogrammed computer if both had a single ALU and scratch pad register bank. The handshake sequencer can cause delays if the cycle time is governed by the next microinstruction fetch timing. Otherwise its delay is hidden because its operation is overlapped with the execution of the current microinstruction and it acts in a lookahead fashion with the control unit "enable" and "finished" bits. Another source of delay could result from the use of a complex instruction decoder that the conventional implementation might not have. By making minor modifications to the handshaking controller, the instruction fetches could be overlapped with the execution of the current instruction, increasing throughput substantially.

Segmented microprogramming is not applicable to all processor architectures, as it was originally intended for use with multi-register and accumulator machines. The segmentation reduces instructions to tasks that are executed in a pipelined fashion. Therefore it is doubtful that segmented microprogramming would offer any advantages in completely pipelined machines, where the microprogram for each unit in the pipeline is either already separated or implemented as a separate field in a large microinstruction. In the latter case the timing of the system would be optimized for this type of microword and no further concurrency or improvements could be made without major redesign, due to the data and control dependencies in a pipeline.

Stack architecture processors execute each instruction as a single execution phase. For arithmetic and logic operations the processor fetches operands from a stack, performs the operation and stores the result back on the stack. There are separate instructions to push data onto the stack and pop data off the stack and store it in memory. Since the functions are already segmented, little reduction in the control store size would occur by using segmented microprogramming.

In summary, this Chapter presented a basis for the segmentation of a processor into intelligent functional units

by first examining the execution of the machine language instructions, as classified into five types depending on the number of operands and the operation to be performed. The instructions were then described in terms of three phases of execution, each phase consisting of a set of tasks of elementary operations. Determination of the hardware resource requirements for the tasks of the pre-op, op and post-op phases led to a partitioning of the processor's resources into two groups, those required by the pre- and post-op phases and the resources required by the op phase.

After examining different conventional control structures, a distributed scheme involving two levels of microprogramming communicating through handshaking lines was presented. Finally, the architectural features and drawbacks of segmented microprogramming were analysed by considering the simplicity of the microprograms and microsequencer, the testability of the design, its expansion capabilities, the cost of implementation, the execution speed and its applicability to various computer architectures.

Chapter IV

A CASE STUDY : THE PDP-11 BY SEGMENTED MICROPROGRAMMING

The practicality of the concept of Segmented Microprogramming as explained in Chapter 3 will be demonstrated by applying it to the design of a hardware emulator of a popular minicomputer. The first part of this Chapter provides an architectural overview of the chosen processor by describing its functional characteristics. The second part initially details the steps necessary to produce the organization of the data paths and control units and an outline of the microcode, then applies this design procedure to the processor whose architecture is described in the first part of the Chapter.

4.1 WHY THE PDP-11

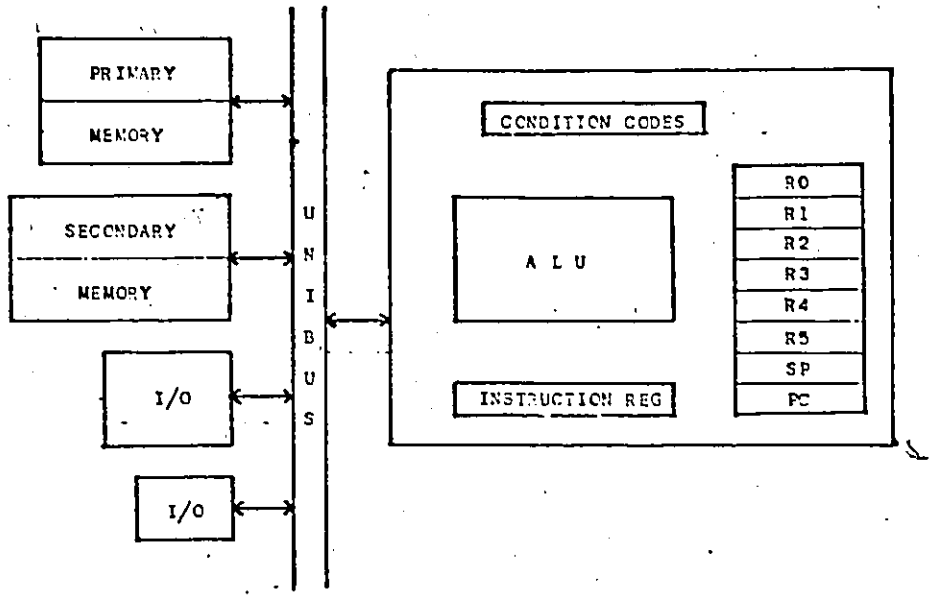
The emulation of a current minicomputer was proposed as a test of the practicality of the segmented microprogramming technique. This would avoid tailoring the architecture to the specific IC chips used and provide a reference for comparaison. The restrictions of the design technique required that the computer selected have a general register or accumulator organization.

The computers readily available at the university were an IBM 360/65, a few PDP-8s and a PDP-11/34. The IBM computer was considered to be too complicated for an initial design methodology test and the PDP-8 was judged obsolete and not worthwhile. The PDP-11 was chosen over other comparable minicomputers because it is reputed to have a relatively complete architecture with numerous addressing modes and instruction formats. Being one of the most popular minicomputers built, there is an abundance of papers published about its design and performance. Finally, the Department of Electrical Engineering's PDP-11/34 could provide detailed hardware documentation for comparaison and a prototype could be tested and its performance evaluated.

4.2 ARCHITECTURAL FEATURES OF THE PDP-11

The PDP-11 family consists of high speed, general purpose digital computers that operate primarily on either 8- or 16-bit binary numbers. The instruction set includes zero-, one- and two-address instructions which utilize a set of eight general purpose registers to specify operands in a variety of addressing modes. The PDP-11 is organized around a single general purpose bus called the UNIBUS, which interconnects the processor, memory and peripherals.

The UNIBUS carries bidirectional signals consisting of data, addresses and control information pertaining to the



UNIBUS concept is central to the architecture of the PDP-11 processor, which is connected to the bus as another peripheral device. The main resources of the CPU are the multifunction ALU, the processor status word containing the condition codes, the instruction register and decoder and the register file containing the general purpose registers, the Stack Pointer and the Program Counter. Although not explicitly shown, the memory is byte structured but can be accessed 16 bits at a time for UNIBUS transfers. I/O devices are assigned fixed addresses in memory space according to their specific function.

Figure 12: PDP-11 Architecture

use of the bus. Devices connected to the bus are assigned a bus address, with peripherals having control, status and data register addresses. The control of the UNIBUS can be dynamically obtained by the CPU or I/O devices through a separate, asynchronous bus request and grant protocol. Contention for the UNIBUS is resolved by an arbitrator using a basically centralized control scheme with all devices except the processor having a fixed priority request level. All bus

data transfers are asynchronous and interlocked between the bus master and its slave. This permits devices with a wide range of operating speeds to be connected to the bus and operate at their maximum rate. Devices can also exchange data, either 16-bit words or 8-bit bytes, without processor intervention.

The central processor, connected to the UNIBUS as a subsystem, controls the time allocation of the UNIBUS for peripherals and performs the arithmetic and logic operations. It contains eight general purpose registers which can be used as accumulators, index registers, autoincrement registers, autodecrement registers or simply as temporary storage for data. Arithmetic operations can be from one general register to another, from one memory location or device register to another, or between a memory location and a general register. Two of the eight general registers are used for special purposes : register R6 is used as a hardware stack pointer to indicate the last entry into the system stack. Register R7 is the machine's program counter and contains the address of the next instruction to be executed. This last register is used only for addressing purposes and not as an accumulator for arithmetic operations.

The other registers in the CPU accessible by the programmer are the Instruction Register and the Processor Status

Register. The Instruction Register is 16 bits wide and contains the instruction currently being executed. The Processor Status Word is also 16 bits wide and holds the current processor UNIBUS priority level, the condition codes describing the result of the last instruction and an indicator for detecting the execution of an instruction to be trapped during program debugging. The condition codes consist of the carry (C), overflow (V), negative (N) and zero (Z) flags associated with the arithmetic unit.

A feature not shown by the diagram of Figure 12 is the system stack, a Last-In First-Out storage area occupying fixed bounds in primary memory. This stack accomodates subroutine and interrupt responses by storing respectively the current program counter value or both the PC and the processor status values. The return instructions pop these values off the stack and restore them to the proper registers. It also provides for argument or argument address transmission through the linkage register associated with subroutine call and return instructions. Separate user stacks can be created by designating any one of the other general purpose registers (except R7, the program counter) as a stack pointer. In this way temporary storage areas can be allocated and accessed with a single address register.

The memory is organized in terms of bytes, although either bytes or 16-bit words consisting of two contiguous bytes can be addressed. The low byte of a 16-bit word, bits 0-7, is assigned an even numbered address while the high byte, bits 8-15, is assigned an odd numbered address. The memory is accessed by the CPU through a Bus Data Register (BDR), a Bus Address Register (BAR) and two memory operation control lines on the UNIBUS. The BDR sends data to and receives data from the UNIBUS. The BAR transmits the memory location to the bus and the control lines determine whether the memory is to read a word, undergo a read - modify - write cycle, store a word or store a byte at the indicated address.

The instruction set is implemented in 16 bit words, with instructions containing immediate data or offsets occupying several words. Most of the double operand and single operand instructions are available for both byte and word operations, with the exception of the Add, Subtract and Exclusive-Or operations, which can only be used for full words. Table 2 lists the instruction set as classified by the scheme explained in Chapter 3 and shows the different instructions formats for each class.

The eight general registers may be used with an instruction in any of the following ways :

TABLE 2

PDP-11/34 Instruction Set

```
Symbols
SRC : Source address : address of first operand
DST : Destination address : address of second or only operand and is
                            also used as the result address when stored
( ) : Contents of
 ←  : becomes
Reg or R : Register

Double operand      <op code><source address><destination address>

CMP,CMPB     (SRC) - (DST)                  compare
ADD          (DST) = (SRC) + (DST)          add
SUB          (DST) = (DST) - (SRC)          subtract
BIT,BITB     (SRC) and (DST)                bit test
BIC,BICB     (DST) = not(SRC) and (DST)     bit clear
BIS,BISB     (DST) = (SRC) or (DST)         bit set
XOR          (DST) = R exor (DST)           exclusive-or

SINGLE OPERAND     <op code><destination address>

CLR,CLRB     (DST) = 0                      clear
COM,COMB     (DST) = not (DST)              1's complement
INC,INCB     (DST) = (DST) + 1              increment
DEC,DECB     (DST) = (DST) + 1              decrement
NEG,NEGB     (DST) = -(DST)                 negate; 2's complement
TST,TSTB     (DST) -0                       test
ROR,RORB     rotate (DST) right one place with carry
ROL,ROLB     rotate (DST) left one place with carry
ASR,ASRB     arithmetic shift right one place into carry
ASL,ASLB     arithmetic shift left one place from carry
ADC,ADCB     (DST) = (DST) + (carry)
SBC,SBCB     (DST) = (DST) - (carry)
SXT          (DST) = (-1) and (N bit)       sign extend N bit into (DST)

MOVE       <op code><source address><destination address>

MOV,MOVB     (DST) = (SRC)                  move source to dest
SWAB         exchange upper and lower bytes of (DST)
i.sp 1
PROGRAM CONTROL     <op code><offset>

BR     PC = PC + 2x(offset)     branch unconditionally
BNE    if Z = 0                 branch if not equal to zero
BEQ    if Z = 1                 branch if equal to zero
BPL    if N = 0                 branch if positive
BMI    if N = 1                 branch if negative
BVC    if V = 0                 branch if overflow clear
BVS    if V = 1                 branch if overflow set
BCC    if C = 0                 branch if carry clear
BCS    if C = 1                 branch if carry set
BGE    if N exor V = 0          branch if greater or equal to
BLT    if N exor V = 1          branch if less than
BGT    if Z or(N exor V) = 0    branch if greater than
BLE    if Z or(N exor V) = 1    branch if less than or equal to
BHI    if C = 0 and Z = 0       branch if higher
BHIS   if C = 0                 branch if higher or the same
BLO    if C = 1                 branch if lower or the same
BLOS   if C or Z = 1            branch if lower or the same

<op code><destination address>

JMP  PC = (DST)    jump to dest address

<op code><source register><destination address>

JSR     jump to subroutine
        push (Reg) onto processor stack
        transfer (PC) into Reg
        transfer (DST) into PC

<op code><source register>

RTS  return from subroutine
        transfer (Reg) to PC
        pop (top of stack) into Reg

<op code><number of parameters>

MARK          stack cleanup operations
        (SP) = (PC) + 2x(# of parameters)
        (PC) = (R5)
        pop (top of stack) into R5

<op code><source register><offset>

SOB      subtract one and branch if not zero
        (Reg) = (Reg) - 1
        if the result is not 0 then
        (PC) = (PC) - 2x(offset)

SYSTEM CONTROL     <op code><data>

EMT     emulate; software interrupt
        push PSW onto stack
        push PC onto stack
        load PC from Trap Vector
        load PSW from Trap Vector
TRAP    trap; software interrupt

<op code>

BPT     breakpoint trap; software interrupt
         push PSW onto stack
         push PC onto stack
         load PC from Trap Vector
         load PSW from Trap Vector
IOT     I/O trap; software interrupt
RTI     return from interrupt
         pop stack into PC
         pop stack into PSW
RTT     return from interrupt; disable trace trap
HALT    stop processor
WAIT    wait for interrupt
RESET   initialize system
CONDITION CODE OPERATIONS

CL(N,Z,V,C)   clear condition codes selectively
SE(N,Z,V,C)   set condition codes selectively

<op code><destination address>

MFPS     move lower byte from PSW to DST

<op code><source address>

MTPS     move byte from SRC to PSW
```

-as accumulators, where the data resides within the register

-as pointers, where the contents of the register are the address of the operand

-as pointers which automatically step through memory locations

-as index registers, in which case the contents of the register and the word following the instruction are summed to produce the address of the operand. There is no direct addressing, where the operand is contained in the instruction itself, because of the size of the instructions.

There are four basic addressing modes : register, autoincrement, autodecrement and indexed. In the register mode, any of the registers may be used as simple accumulators and the operand is contained in the selected register. The autoincrement and autodecrement modes provide for the automatic stepping of a pointer through sequential memory locations and can be used to access tables or implement user stacks in main memory. In the autoincrement mode, the register contains the address of the operand and is incremented (by one for byte addresses, two for word addresses) after the operand access to point to the next sequential location. The autodecrement mode subtracts one or two, depending on the address, from the register contents then uses this decremented value as the operand address. Indexing uses the

contents of a register as a base for the calculation of several addresses. In this mode the contents of the selected general register and an index word following the instruction word are added to form the address of the operand.

The four basic modes may also be used with deferred (indirect) addressing, in which case the register contents select the address of the operand rather than the operand itself. The register deferred mode uses the contents of the selected register as the address of the operand. The autoincrement deferred mode uses the register as a pointer to the word containing the address of the operand, then invcrements the register by two, since it always points to a 16-bit address. The autodecrement mode first decrements the register by two then uses it as a pointer to the word containing the address of the operand. The index deferred mode uses the register contents as the base address of a pointer to the address of the operand. First the selected register is added to the index value in the word following the instruction and this sum becomes the pointer to the word containing the address of the operand.

The use of the program counter or stack pointer as a general register involves special features of the addressing modes. When the processor uses the PC to access memory, the PC is automatically incremented by two to point to the next

word of the instruction being executed or to the next instruction to be executed. Using the autoincrement mode with the PC provides an immediate addressing mode by accessing the word at the location following the instruction. Absolute addressing is achieved by using the autoincrement deferred mode, which considers the word following the instruction to be the address of the operand.

Relocatable programs can be written by using the indexed mode with the PC. This program relative mode uses the PC as a base register and adds it to the offset value following the instruction to provide the address of the operand. Indirect relative addressing by the program counter permits table accessing when the program is loaded into different areas of memory. Here the index deferred mode is used so that the index value following the instruction is summed with the PC to form a pointer to the address of the operand.

The processor stack pointer is used by the hardware for program and interrupt nesting and, like the PC, is always incremented by two. The use of the autodecrement and autoincrement addressing modes push and pop data off the system stack respectively. Also, indexed addressing with the SP provides random access to the items on the stack.

4.3 DESIGN TECHNIQUE

The following figure shows the steps involved in the design of a processor using segmented microprogramming.

1. DEFINE THE INSTRUCTION SET TO DETERMINE THE NUMBER AND TYPES OF THE PROCESSOR SEGMENTS.
2. BREAK DOWN THE INSTRUCTION SET INTO EXECUTION PHASES TO DETERMINE THE PHASE SEQUENCING.
3. DEFINE THE SET OF TASKS EXECUTABLE BY EACH PHASE AND THEIR SEQUENCING RELATIONSHIPS.
4. DESCRIBE THE TASKS OF ALL PHASES IN DETAIL SO AS TO DETERMINE THE COMPUTATIONAL RESOURCES REQUIRED FOR EACH SEGMENT.
5. DEFINE THE MICROSEQUENCER CAPABILITIES NEEDED FOR EACH SEGMENT BASED ON THE HARDWARE ARCHITECTURE OF STEP 4 AND THE TASK SEQUENCES OF STEP 3.
6. DEFINE THE MICROINSTRUCTION FORMATS AS TO THE FUNCTION OF EACH FIELD.
7. GENERATE THE DETAILED HARDWARE SPECIFICATIONS.
8. DETAIL THE MICROINSTRUCTION FORMATS AND TRANSLATE THE TASKS INTO MICROROUTINES.
9. PERFORM COST REDUCTIONS ON THE ARCHITECTURE AND MICROCODE BY ITERATING THROUGH STEPS 4, 5, 6, 7 AND 8.

Figure 13: Design Procedure

4.3.1 Execution Phase Sequences

The instruction set has been defined in the first part of this chapter. The second step of the procedure is to break down each instruction into execution phases. The instruction

formats defined for each class do not specify the number of execution phases performed, so that there may be several different types of execution phase sequences in the same instruction class. The instructions listed in Table 3 represent all execution phase sequence types for each instruction class.

TABLE 3

Execution Phase Sequences of the Instructions

Class	mnemonic	execution phases	comments
Double operand	CMP	pre-op, op	test type, no post-op
	ADD	pre-op, op, post-op	
Single operand	CLR	op, post-op	no operand fetch
	INCB	pre-op, op, post-op	
	TST	pre-op, op	
Move	MOVB	pre-op, op, post-op	no 2nd operand fetch
Program Control	BNE	post-op	
	JMP	pre-op, post-op	
	JSP	pre-op, post-op	pre-op first sets up data
	EAT	pre-op, op	op to load PSW
System Control	HALT	post-op	
	SEC	op	affects CCs only
	RFPS	pre-op, op, post-op	
	RTPS	pre-op, op	

The six execution phase sequences used, out of a possible seven, are :

pre-op, op

pre-op, op, post-op

op

op, post-op

post-op

pre-op, post-op

4.3.2 Execution Phase Tasks

The third step is to define the sub-operation tasks of each execution phase and determine their sequencing orders. By comparing the execution phase breakdown with the instruction set, it can be seen that the pre-op phase corresponds to the operand fetches and a few special purpose register operations. The op phase performs all arithmetic and processor status modifications. Finally, the post-op phase takes care of the storage of the op phase result and also does most of the tasks required by the program and system control instructions.

The set of pre-op tasks can be divided into three subsets : the regular eight addressing modes, the destination addressing modes as modified by the move instructions and the stack and register manipulations required by some program and system control instructions. This last sub-set of pre-op tasks is either used in order to group common operations in the same phase or may be necessary because of the restrictions on access to the PSW.

The first sub-set of tasks, the eight addressing modes described in section 3 of this Chapter, are expressed in terms of elementary register operations in order to show the micro-operations needed. Each row of Table 4 generally corresponds to a complete major clock cycle, with the semi co-

lon (;) separating operations which are either performed or initiated during the same clock cycle.

TABLE 4

Register Operations of Regular Addressing Modes

Addressing Mode	Description
Register	Reg to ALU buffer
Autoincrement	Reg to BAR; Reg + inc to Reg; memory read
	BDR to ALU buffer
Autodecrement	Reg - dec to Reg and BAR; memory read
	BDR to ALU buffer
Index	PC to BAR; PC + 2 to PC; memory read
	BDR + index Reg to BAR; memory read
	BDR to ALU buffer
Register deferred	Reg to BAR; memory read
	BDR to ALU buffer
Autoincrement deferred	Reg to BAR; Reg + 2 to Reg; memory read
	BDR to BAR; memory read
	BDR to ALU buffer
Autodecrement deferred	Reg - 2 to Reg and BAR; memory read
	BDR to BAR; memory read
	BDR to ALU buffer
Index deferred	PC to BAR; PC + 2 to PC; memory read
	BDR + index Reg to BAR; memory read
	BDR to BAR; memory read
	BDR to ALU buffer

The second sub-set of tasks, the MOVE-type destination addressing modes, omit the actual fetch of the operand as its current register or memory location will be overwritten during the post-op phase. The corresponding register operations are shown in Table 5

TABLE 5

Register Operations of Move-type Addressing Modes

Addressing Mode	Description
Register	no operation
Autoincrement	Reg to BAR; Reg + inc to Reg
Autodecrement	Reg - dec to Reg and BAR
Index	PC to BAR; PC + 2 to PC; memory read BDR to BAR
Register deferred	Reg to BAR
Autoincrement deferred	Reg to BAR; Reg + 2 to Reg; memory read BDR to BAR
Autodecrement deferred	Reg - 2 to Reg and BAR; memory read BDR to BAR
Index deferred	PC to BAR; PC + 2 to PC; memory read BDR + index Reg to BAR; memory read BDR to BAR

The remaining sub-set of pre-operation tasks are unique to specific instructions. These are shown in Table 6 and consist of sequences of register operations followed either by pre-op operand fetches and tasks of other phases, or they may be followed directly by op and post-op phase tasks.

TABLE 6

Register Operations of Special Pre-Op Tasks

Instruction	Description
JSR	SP - 2 to SP and BAR Reg to BDR; memory write PC to Reg
+ regular addressing	modes for destination fetch
EMT,TRAP,BPT,IOT	SP - 2 to SP and BAR PSW to BDR; memory write SP - 2 to SP and BAR PC to BDR; memory write Trap vector address to BAR and buffer; memory read BDR to PC buffer + 2 to BAR; memory read BDR to ALU buffer
RTI,RTT	SP + 2 to SP and BAR; memory read BDR to PC SP + 2 to SP and BAR; memory read BDR to ALU buffer
MFPS	PSW to ALU result buffer
+ regular addressing	modes for destination fetch

The op phase tasks consist of all the arithmetic and logic operations specified by the instruction op-codes, the selective loading of the PSW bits from the ALU input buffer and setting or clearing the condition code bits according to the instruction op-codes.

The post-op tasks listed in Table 7 consist of two types, one storing the ALU result buffer into a register or at the address defined by the BAR set up in the pre-op phase. The other type performs special operations with the registers and memory corresponding to branches, stack manipulations and interacting with the console.

TABLE 7

Register Operations of Post-Op Tasks

Instruction	Description
Branch	PC + 2x(signed offset) to PC; offset set to 0 if branch condition false
RTS	Reg to PC SP to BAR; SP + 2 to SP; memory read SP + 2 to SP BDR to Reg
MARK	PC + 2x(number of arguments) to SP Reg5 to PC SP to BAR; SP + 2 to SP; memory read SP + 2 to SP BDR to Reg5
SOB	Reg - 1 to Reg PC - 2x(offset) to PC; offset set to 0 if Reg = 0
HALT	Reg0 to console data display register PC to console address display register
WAIT, Reset	hardware functions

The task sequences used by the different phases are listed below.

Pre-op :

Dst fetch

Src fetch, Dst fetch

Src fetch, Dst fetch (Move type)

Special pre-fetch, Dst fetch

Special pre-fetch, Dst fetch (MOVE type)

Special operation

Op :

single cycle Arithmetic operations or PSW modifications.

Post-op :

single tasks only, consisting of :

operand storage

hardwired macroinstructions

hardware interaction routines (HALT, WAIT, RESET)

This information is combined with the execution phase sequences previously extracted and the instruction set to yield the instruction task sequences to be output by the handshake controller, as shown in Table 8 The important fact to recognize is that not all sequences of tasks and phases are actually used by the instruction set. This permits a sub-set of all possible task sequences to determine the operation of the machine.

TABLE 8

Task Sequences for Handshake Controller

PRE-OP phase			OP phase	POST-OP phase
SPE	SRC Fetch	DST Fetch		
	Src	Dst	op	post-op
	Src	Dst (move)	op	post-op
		Dst	op	post-op
			op	
				post-op
Spe		Dst (move)	op	post-op
Spe		Dst		post-op
Spe			op	

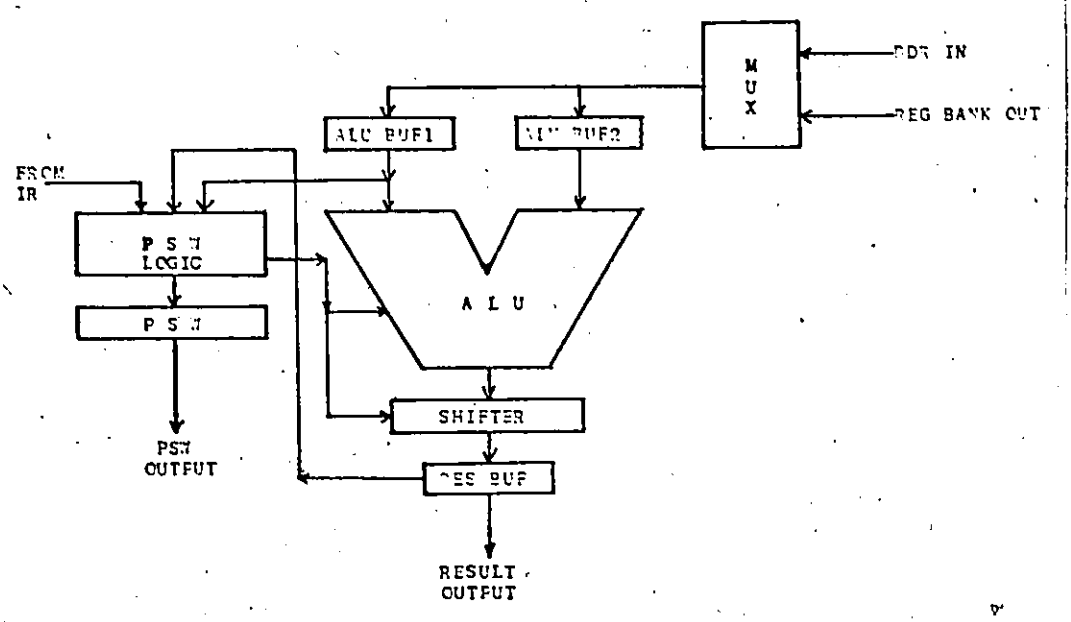
4.3.3 Definition of the Organization

The final step considered in this chapter determines the nature of the microprogrammable resources needed by each segment and the organization of the control structure.

The Arithmetic Segment executes OP phase tasks exclusively. By using the description of the tasks to be accomplished by this phase we can determine the structure of the arithmetic and logic unit and the interconnections between itself, the Processor Status Word and the Memory Access Segment.

The arithmetic and logic circuit requires two input buffers, whose loading is under the control of the Memory Access Segment. The output of the ALU passes through a shift multiplexer before going to the result buffer, where it is

temporarily stored. Both the ALU and the shifter receive inputs from the condition codes and output new CC values.



ARITHMETIC resources used by the Segmented Microprogrammed PDP-11 emulator (SMP-11) consist of a multifunction ALU and a shift network. The condition codes of the Processor Status Word can interact with both these devices and can also be modified directly by the instruction register. The two ALU input buffers and the ALU source multiplexer are controlled by the Memory Access segment in order to facilitate the loading of data into the buffers.

Figure 14: Arithmetic Segment Resources

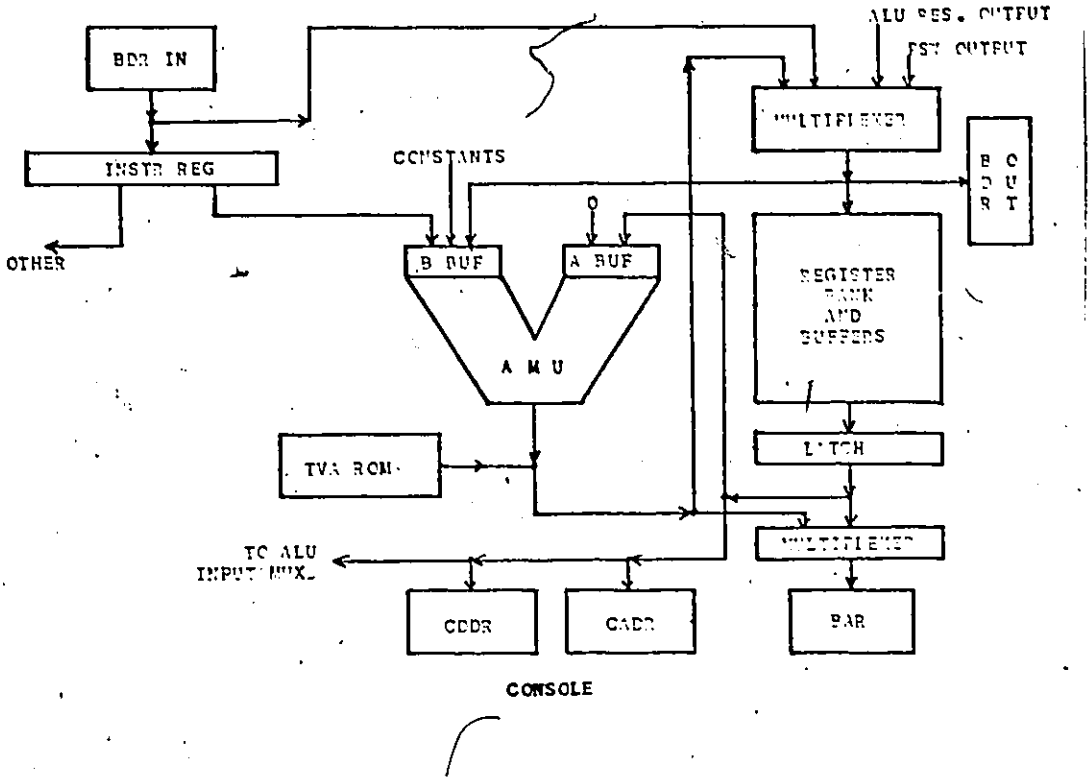
The Processor Status Word has inputs from an ALU buffer, the ALU and shifter circuitry as well as from the Instruction Register decoder. These inputs are processed by some select logic to determine which bits are to be modified and what their new value is to be.

The output of the result buffer and the PSW are routed to the Memory Access segment and are under its control.

The Memory Access Segment executes both pre- and post-op phase tasks. Its resources include the Bus Data Register, the Bus Address Register, a Register Bank and an Address Modification Unit.

The central resource is the Register Bank, which contains the eight general purpose registers defined by the instruction set as well as buffers for temporary storage. The inputs to the RB come from the BDR, the AMU result bus, and the Arithmetic Segment result buffer and PSW output. These inputs are routed through a multiplexer before being applied to the RB input bus. This multiplexer swaps the upper and lower bytes of a word either when specifically designated by the SWAB instruction or upon input from or output to the BDR with an odd byte address.

The Address Modification Unit is a simple arithmetic circuit which can perform the addition, subtraction and pass operations called for by the addressing modes. The AMU has inputs from the Register Bank, the BDR and hardwired constants, as well as offsets and input bytes from the instruction register. The RB is given access to the primary side of the AMU because it is used most often in computations in-



MEMORY Access segment resources interface with the UNIBUS through the Bus Data registers and the Bus Address register. Other resources required are an Address Modification unit to compute indexed and branch addresses; a register bank to hold the general purpose registers and buffers needed by special operations; a multiplexer to select between data inputs, computed addresses, ALU results and the processor status word output; a trap vector address memory for hardware interrupts and trap instructions; a Console Data Display Register and a Console Address Display Register.

Figure 15: Memory Access Segment Resources

volving offsets and constants. The Bus Data Register is connected to the secondary side of the AMU through the register bank input multiplexer so that it can be combined with a va-

lue from the Register Bank or pass through the AMU without modification. The AMU result bus is connected to the BAR and fed back to the input multiplexer of the Register Bank.

The Bus Data Register transmits information to the Instruction Register, Register Bank, Address Modification Unit and the Arithmetic Segment ALU buffers. It can also input data to the BAR indirectly by going through the AMU.

For the interrupt and trap instructions a Trap Vector Address Memory is connected to the AMU result bus so that vector addresses can be input to the BAR and stored in a buffer in the RB at the same time.

The control circuitry of the processor has four major elements : the Instruction Register and decoder, the Handshake Sequence Controller, the Memory Access Segment Controller and the Arithmetic Segment Controller. These last three units are microprogrammed and each is made up of a control store and a microsequencer with next address control.

The instruction decoder provides all the start addresses for the Handshake Sequencer, Memory Access Segment and Arithmetic Segment tasks. The Handshake Sequence Controller initiates tasks in the Arithmetic and Memory Access segments by activating 'enable' lines to these units. When a segment

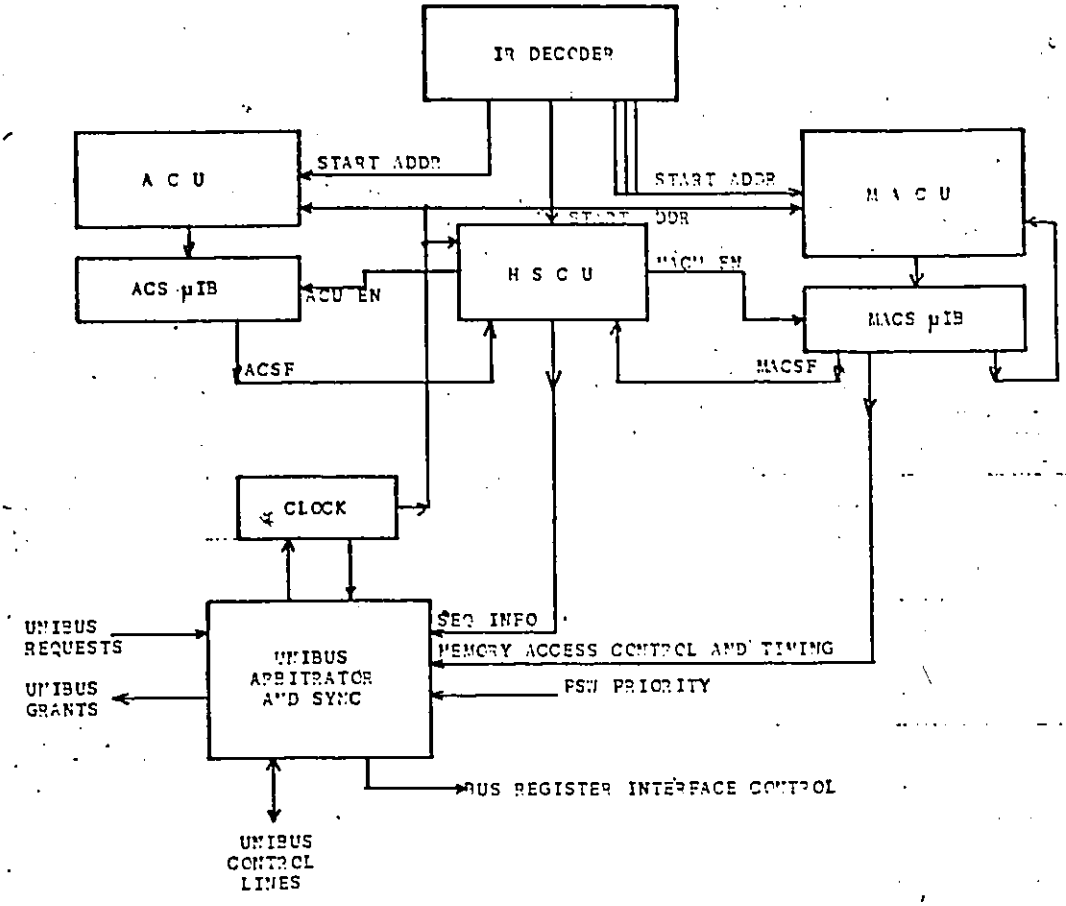
has finished its task, it signals the Handshake Controller by enabling its 'finished' line. The Handshake Sequencer waits for the proper input 'finished' line to be activated then proceeds to the next word of its control store, whose output enables the next task.

The complex arbitration and synchronization logic required for UNIBUS compatibility can be provided by Bus Interface Logic circuits designed outside of the processor. This logic handles interrupts generated by devices on the UNIBUS and stops processor operation on memory wait cycles. The BIL interacts with the processor through the handshake sequencing unit.

4.3.4 Realization Cost Reductions

The following data path changes were used to reduce the cost of realizing the circuits. It will be shown that there is very little effect on the structure of the control units.

A detailed study of the microcode required by a PDP-11 emulator with the architecture previously defined in this section showed that there is little or no overlap possible between the operation of the memory access segment and the arithmetic segment. This is because the basic PDP-11/34 instruction set can be implemented in a straightforward manner with single cycle ALU operations. Although two separate



HANDSHAKE lines carrying enable signals to the microsequencers and finished signals from the microinstructions of the segment control units permit the Handshake Sequencer to order task sequences defined by the instruction decoder. UNIBUS compatibility is achieved by using external bus interface, arbitration and synchronization logic. Based on the processor's priority and sequencing information from the HSCU, bus requests submitted by the Memory Access segment can be acted upon by controlling the clock circuitry and the bus input and output registers.

Figure 16: Control and Bus Interface

ALUs make a simple design and have many positive features with respect to testability and adaptability, the cost can be reduced substantially by incorporating the Arithmetic Segment functions into the Address Calculation Unit of the Memory Access Segment and reorganizing the buses accordingly. Furthermore, this structure can be directly realized through LSI bit slices containing a register bank, ALU and shift network, thus significatly reducing parts count while slowing down operation only marginally because of the lack of overlap.

Overlaying the Arithmetic and Memory Access controllers to result in a single control store would not reduce the size of this control memory and would require subroutine capabilities of the microsequencer in order to be able to switch between microcode segments. The merged control memory would be larger because the op phase tasks need to control all the functions of the bit-slices, and the PSW selection logic, while the pre- and post-op phase tasks don't use the logic operations and shift capabilities of the slices or the PSW input circuitry. Also, the register bank addresses are set up in advance by the Memory Access control store, as well as the bits controlling all the other registers and data path elements of the processor and the interface logic, except for the shifter and CC inputs.

The control structure therefore stays essentially the same but the control lines that must be shared are multiplexed into common control store microinstruction registers. The remaining control lines are held constant by the inactive control unit through 'residual control' microinstruction registers.

In summary, the design steps outlined in the second part of this Chapter have two salient features, namely the concepts of execution phases and execution phase tasks. The subdivision of instructions into execution phases provides a logical basis for the segmentation of the computational resources into fuctional units. An instruction can then be described as performing a series of linked tasks in each execution phase. The detailed description of the tasks serves to organize the hardware of the segments to show the exact nature of the computational facilities required, the interconnections between them and provides an outline of the microprograms for the Memory Access and Arithmetic control stores. The linking of the tasks defines the contents of the Handshake Sequencer Control Store and establishes the requirements for the microsequencer capabilities of the Memory Access and Arithmetic segments.

Chapter V

REALIZATION : PDP-11 BY SEGMENTED MICROPROGRAMMING

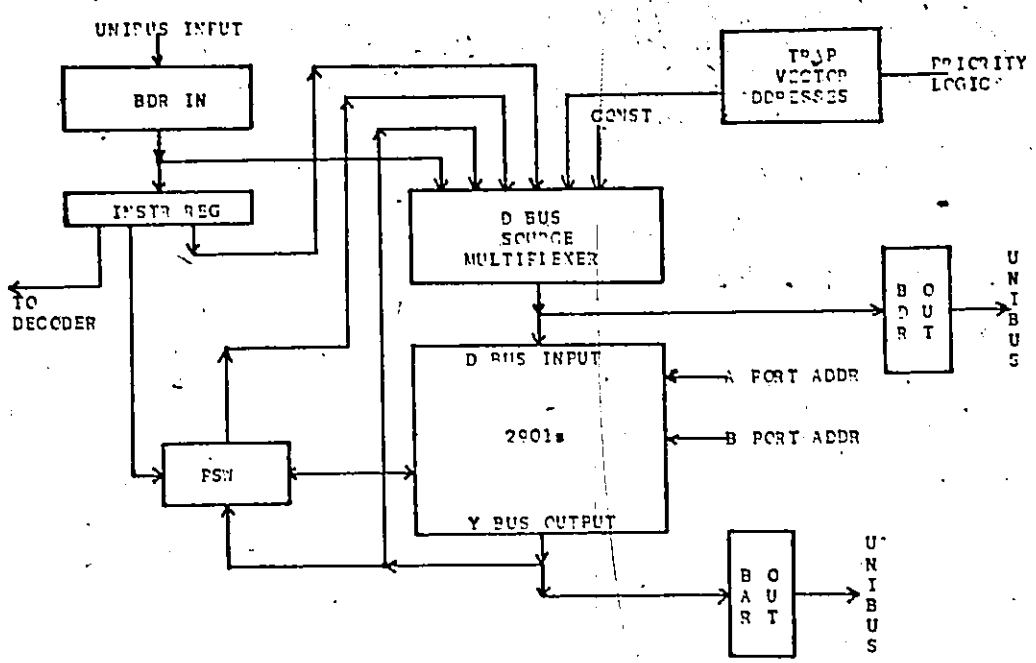
The contents of this Chapter correspond to the last step in the design procedure laid out in Chapter 4. The data flow section describes the microprogrammable computational facilities, which include the processor registers, the data bus multiplexer and the internal architecture of the bit-slice processor elements. The logic necessary for the Processor Status Word is especially detailed and the control fields of the bit slices are examined.

The control section examines the function, operation and structure of the instruction decoder, the microprogrammed controllers and the bus interface and clock circuitry. The development of the three control units shows the simplicity of the microsequencers and the correspondence between the actual microprograms and their outline based on the task description of the previous design step. Finally, there is a section on console requirements that relates to the testability and maintainability of the design.

5.3 COMPUTATIONAL FACILITIES

There were several bit-slice processor chips available to implement the architecture, consisting mainly of 2 bit-wide slices, 4 bit-slices and 4 bit-slice Emitter Coupled Logic high speed chips. The ECL units (MC10800) suited the general architecture requirements well but would have needed external register files and either ECL-TTL interfacing or extensive ECL design experience, as well as being expensive and not readily available at the time the design was being planned. The 2-bit slice Intel 3000 series chips were not used because of their limited internal architecture, wide control field and higher cost, since twice as many are required as compared to the 4-bit slices. The unit chosen to perform the ALU/RB functions was the AMD 2901, a 4-bit slice TTL IC with internally available ALU, shift circuitry, extension register and 16 word 2 port register bank.

The computational facilities of the SMP-11 can be grouped into three types : the bit-slice processing elements, the 2901 input bus source multiplexers and the Bus Interface and Processor Status registers. The general purpose registers of the instruction set architecture are realized internally in the 2901s in the low order 8 registers, while the higher address 8 registers are used for ALU buffers and temporary storage. The bus data input, bus data output and bus address registers are connected to the UNIBUS through bus drivers or receivers controlled by the Bus Interface Logic.



DATA PATH resources of the SMP-11 consist of the 2901 bit-slice processors, an externally implemented Processor Status Word and its logic, a Trap Vector Address memory, bus interface registers and an extensive input bus multiplexer for the single input bus, single output bus 2901s. This bus multiplexer selects between bus input data, instruction register provided offsets, the PSW output, the output of the 2901s, trap vector addresses and hardwired constants under microprogrammed control and also permits byte swapping to be performed on incoming data.

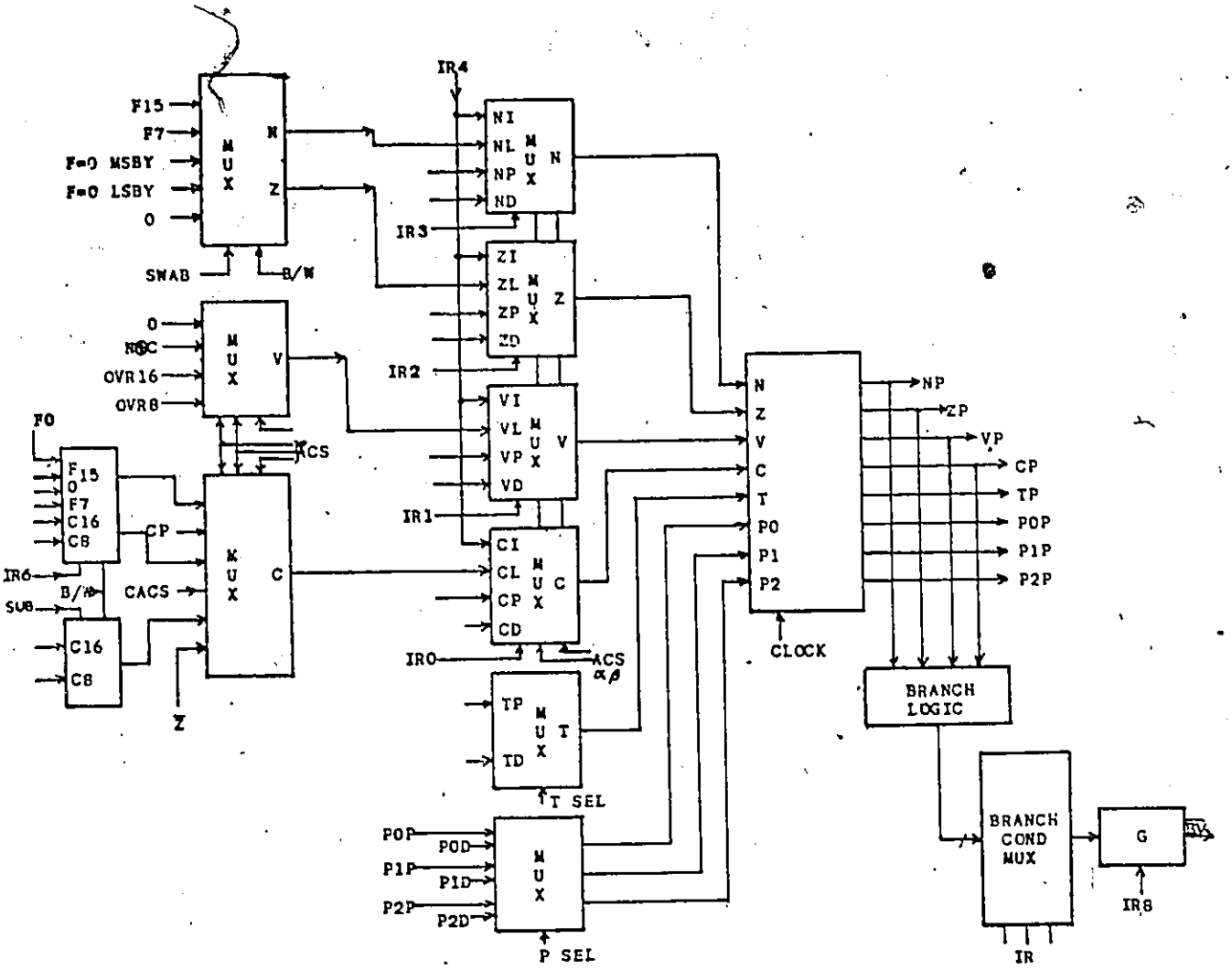
Figure 17: Computational Facilities of SMP-11

The processor status register contains the four condition codes (N,Z,V,C), the trace bit and the three encoded priority bits. Only the low order byte of the PSW is implemented, the upper byte being used only in processor models with memory management schemes. The data in the condition code register is updated at the end of the current microin-

struction cycle, although it is only changed during the "OP" phase. The condition codes of the PSW are used to generate the different logic conditions for the branch instructions. The branch condition to be tested is selected by the instruction decoder and the response (true or false) is routed to the Handshake controller in order to conditionally alter the instruction flow.

The inputs to the PSW depend on the bit or bits to be affected. The T bit can only be loaded from the D bus in an interrupt response. The priority bits can be changed by the MTPS instruction or by an interrupt. A multiplexer tree is used to select the flags NZVC as generated by the logical, arithmetic or shift instructions, the instruction register value as determined by the instructions that set or reset the condition codes, such as SCC, SCZ, etc., the data bus value or the previous flag value.

The 2901 source input multiplexer is implemented as several multiplexers in cascade so as to allow data to be input from the bus data input register, the output of the 2901s, the PSW output, hardwired trap vectors, and arithmetic constants. It also allows byte swapping to take place on data being input from the UNIBUS or from the output of the 2901s. Two separate multiplexer trees are used, one for the lower byte and one for the upper byte, in order to take care of



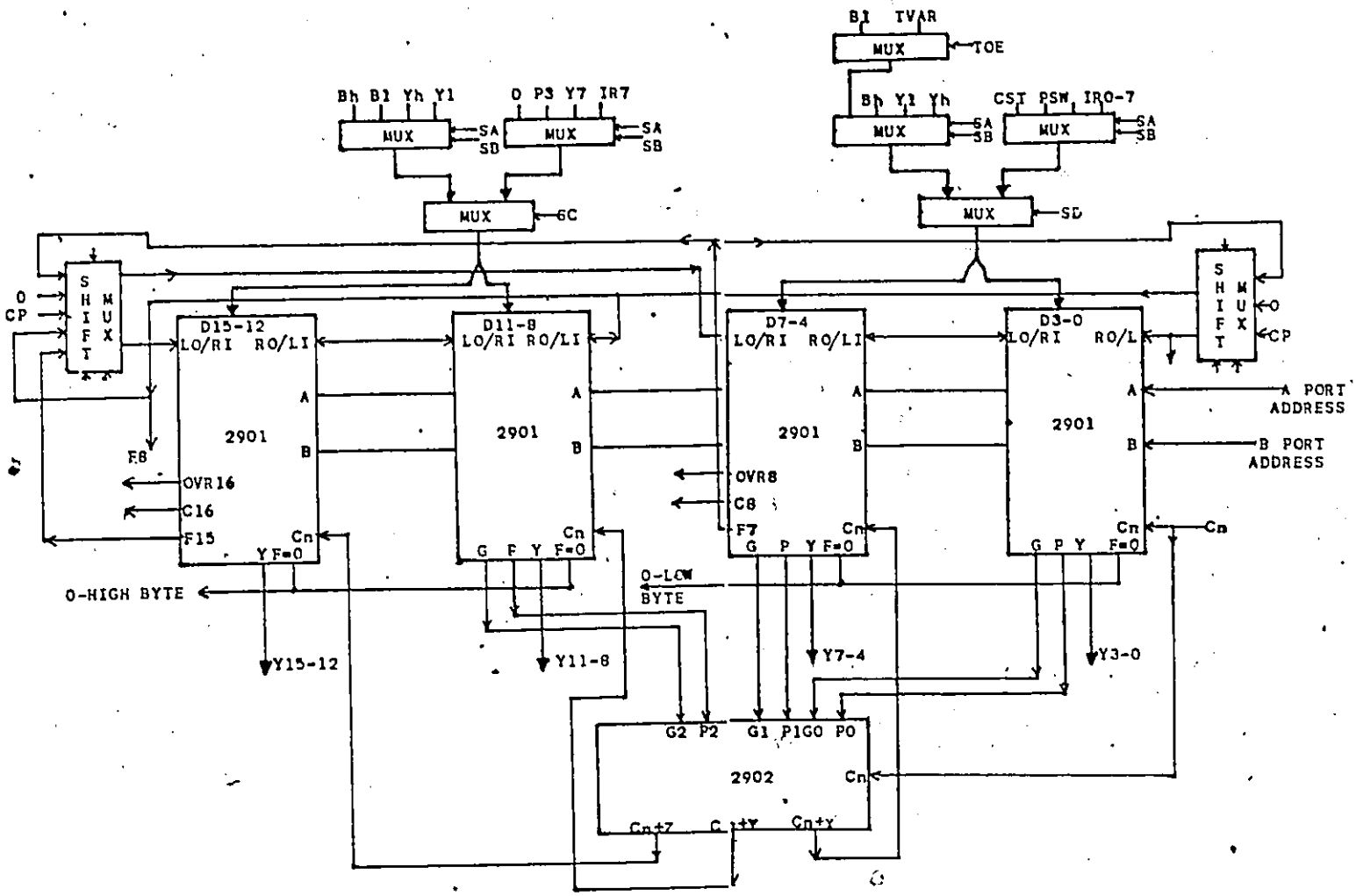
MULTIPLEXER trees controlled by signals from the instruction decoder and microinstruction registers of the MACS and ACS provide the selection logic for the Processor Status Word inputs. The condition codes have to select between inputs from the instruction register, the generated logic value, the previous value and the data bus value depending on the particular instruction being executed. Feeding the previous outputs back as inputs allow the condition codes to remain unchanged. The condition code outputs are also used to determine branch conditions selected by the instruction decoder.

Figure 18: Processor Status Word Implementation

anomalous conditions raised by some of the machine language instructions, such as sign extension of the lower byte. The output of the multiplexers feeds the data (D) bus input of the 2901s and the Bus Data Output register.

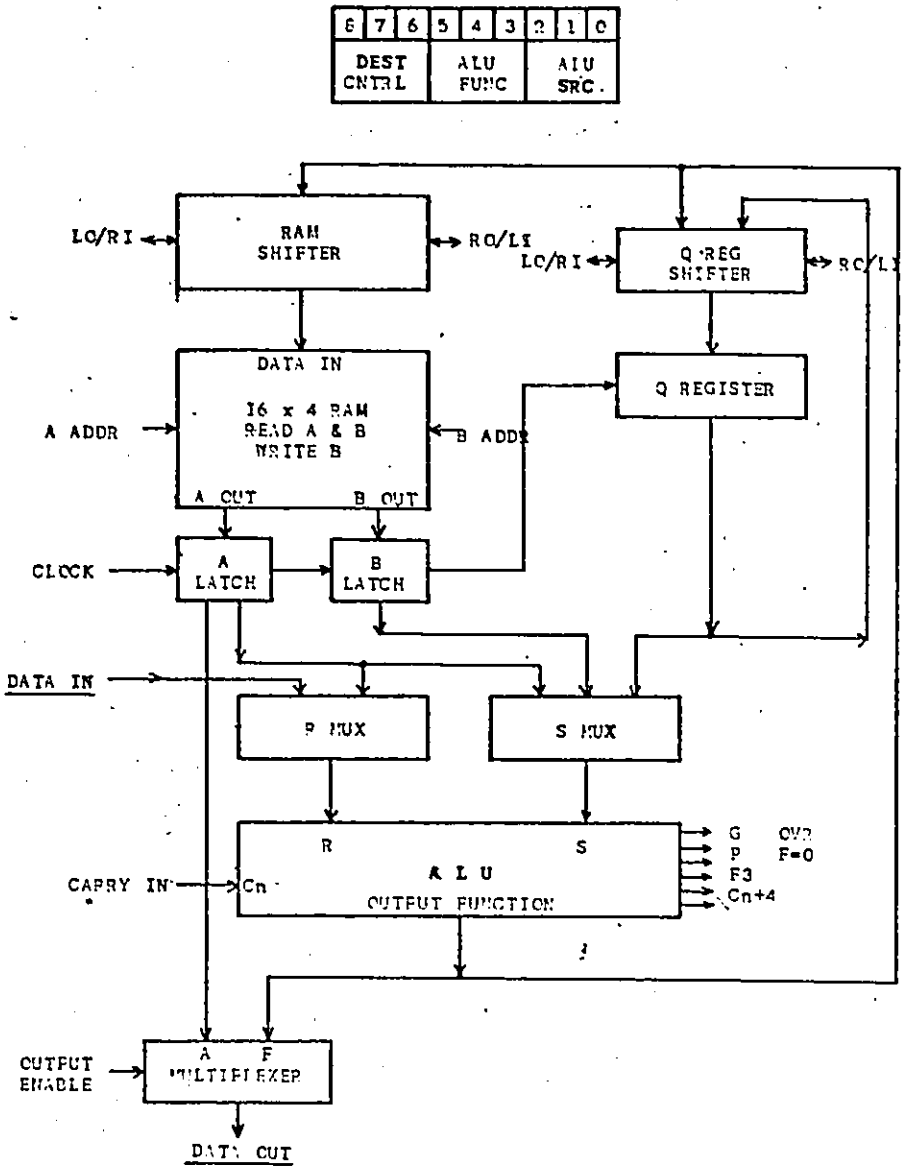
The heart of the system is made up of the four 4-bit slice 2901s and the 2902 look-ahead carry generator. The 2902 accepts carry generate and propagate signals from each slice and provides the carry-in signal to the next stage and the anticipatory signals to successive stages. To allow for various types of shift operations an external set of multiplexers provides a way to link the carry flag and a logical 0 or 1 into the shift path.

A detailed block diagram of the microprogrammable facilities of the 2901 is shown in Figure 20. Data in any of the 16 words of the scratch pad RAM can be read from the A port of the RAM as controlled by the 4-bit A address field input. Likewise, data in any of the 16 words of the RAM as selected by the B address fiels can be simultaneously and independently read from the B port of the RAM. The RAM data input is driven by a three input multiplexer, which permits 1 bit left or right shifts to be made on the incoming data to be stored at the word accessed by the B address field. Data read by the A and B addresses is latched at the beginning of the clock cycle so that data can be written into the RAM during the same clock cycle.



CASCADING the 2901 bit-slices with carry lookahead logic (2902) allows easy and inexpensive implementation of the majority of the data transformation functions and provides all the condition code inputs needed. The internal shifter of the 2901s is used with external shift multiplexers in order to be able to perform arithmetic shifts and rotates on words and bytes. The tree-structured D bus input multiplexer is controlled by 4 microprogrammed signals and can swap bytes on its UNIBUS and y-bus inputs.

Figure 19: Bit-slice Processor Organization



BIT-SLICE 2901 provides a 4-bit wide cascadable computing element with a 16 word two-port RAM, eight function ALU, separate shift unit and various status flag outputs. The ALU output passes through the shift unit before entering the RAM so as to allow single cycle add-and-shift operations. The output of the RAM is latched for read-modify-write operations. The multiplexer network selects the source operands of the ALU and determines whether the ALU output or RAM A port data is to be placed on the output bus. There is also an extension register (Q) with its own shifter to simplify microcoded multiply and divide operations. The nine bit microinstruction word is organized into three groups of three bits each to select the ALU source operands, the ALU function and the ALU destination and output bus source.

Figure 20: 2901 RALU

The ALU is capable of performing three arithmetic and five logical operations on its two input words, designated R and S. The R input is driven by a two input multiplexer, while the S input is driven by a three input multiplexer. Both multiplexers also have the capability to inhibit inputs, resulting in a zero (0) operand. The R multiplexer's sources are the RAM's A port latch and the external direct data input (D bus). The S multiplexer can select between the RAM A port latch, B port latch and the internal extension (Q) register. This Q register is a separate 4-bit cascadable file that can store the output of the ALU or its own contents shifted right or left. The ALU's output (F) is fed back to the inputs of the scratch pad RAM and Q register as well as being applied to a 2 input Y bus output multiplexer. The other input of the Y multiplexer is the RAM's A port latch, permitting data to be read from RAM while a read-modify-write operation is performed via the ALU and F bus.

The ALU provides the carry generate (G) and carry propagate (P) outputs necessary for cascading several slices in a lookahead carry mode. There is also an external carry-in input Cin and carry-out output Cn+4, the latter being used on the most significant slice to generate the carry flag input to the PSW. The ALU has three other status oriented outputs, F3, F=0, and overflow (OVR). The F3 output provides the most significant bit of the ALU result separately from the Y-bus

outputs and on the 2nd and 4th most significant devices is used to generate the sign bit for byte and word operations. The F=0 output is used to detect a zero arithmetic result and is used as the Zero flag input. The overflow (OVR) output is used to flag arithmetic operations that exceed the availble 2's complement range.

The 2901 is controlled by 19 lines. The clock line controls the reading of data into the A and B port latches, the writing of data into the RAM and enables inputs to the Q register. The output enable signal controls the condition of the three-state Y-bus output lines. The 4-bit A and B addresses of the 2 port RAM make up another 8 control lines. The remaining 9 lines are designated 0-8 and are separated into three 3-bit fields called Source-operand control, ALU function control and Destination control. The source operand control field selects the ALU source operands R and S through their multiplexers. The ALU function control field specifies the arithmetic or logic operation to be performed upon the R and S operands. The destination control field is used to select the output of the Y-bus multiplexer and the RAM and Q register input shift multiplexer functions.

5.2 CONTROL UNIT DETAILS

5.2.1 Overview

The control unit has five parts consisting of the instruction register and instruction decoder, the handshake sequencing controller, the Arithmetic control unit, the Memory Access control unit and finally the Interrupt control, Clock (timing) and Bus Interface logic.

Control of the computational facilities proceeds as follows : in the instruction fetch microroutine the instruction held in the Bus Data Input register is loaded into the instruction register and is decoded. This decoding separates the arithmetic operation to be performed from the operand fetch and/or store operations and determines the sequencing of these two classes of operation. The instruction decoder sends information to the handshake controller which specifies whether the instruction is a single operand, double operand or special type. It also generates the start address of the task sequence for the handshake sequence control store.

The ACU is sent the start address of the microroutine corresponding to the arithmetic or logical operation to be performed. The MACU may receive one or more start addresses from the instruction decoder, depending on the type of macro-operation considered and its breakdown in terms of the preop and postop components. At the end of each instruction,

the processor may execute a Service routine if an interrupt or a trap of sufficiently high priority is detected by the interrupt control and bus interface logic units.

As stated in the literature (13), the instruction op-codes of the PDP-11 were not laid out for microprogramming. Generalities exist in the format for the addressing modes and for the arithmetic or logic operations which can be translated in a fairly straightforward manner into microcode start addresses. The branch conditions, as will be seen, can also be used with practically no decoding. The diagram in Figure 21 is a generalized configuration based on that used to realize the SMP-11. The operation of the ACU and MACU as well as the initialization of the Bus interface logic is controled by the handshaking sequences stored in the HS controller through enable bits. The ACU, MACU and BIL can indicate to the HS controller the end of a microroutine by a "finished" or "data ready" line.

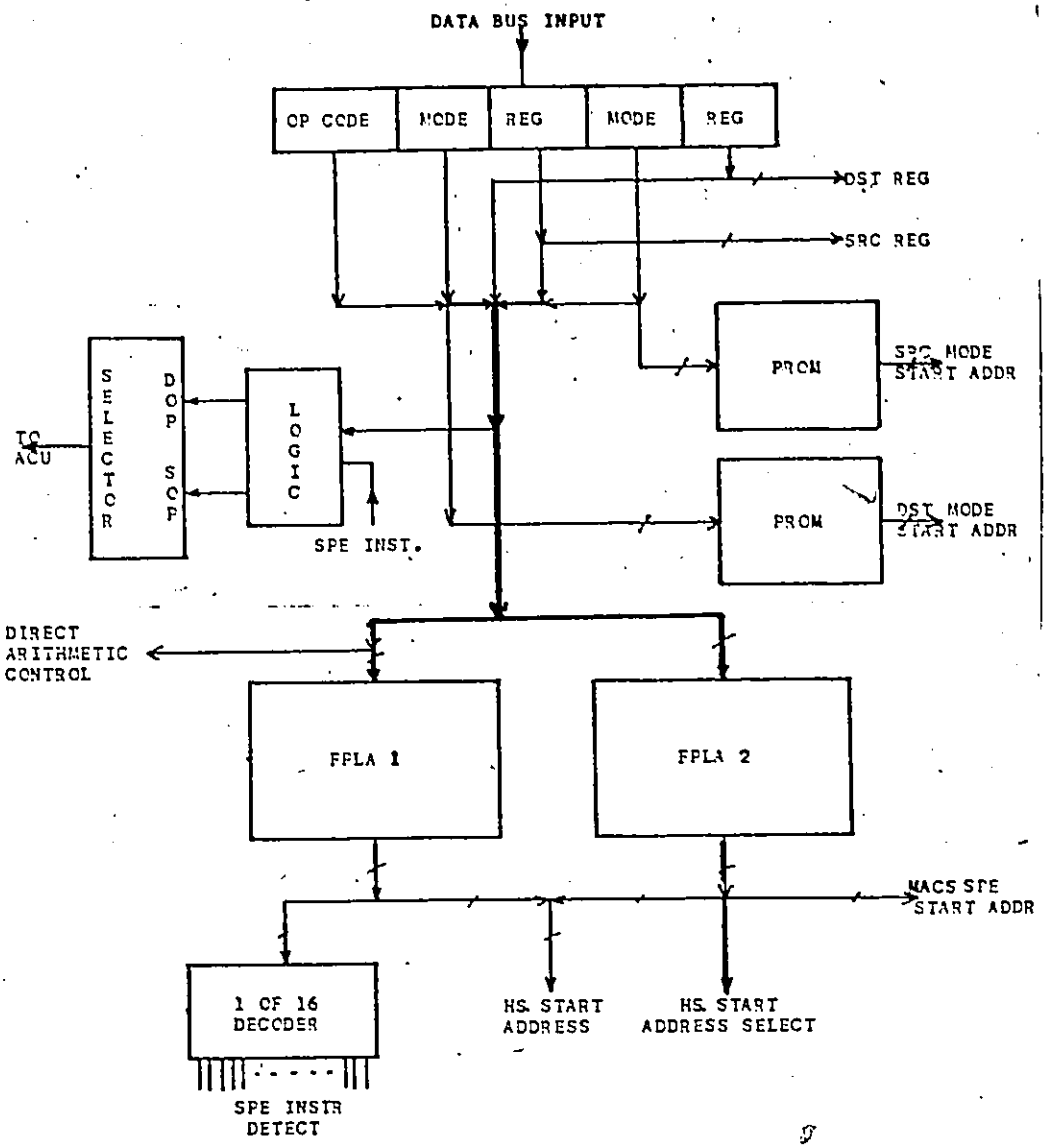
It is to be noted that the information from the instruction decoder is available for the entire duration of the macro-instruction, ie. until the next IR input enable from the MACU's instruction fetch cycle. Also, although the system is designed to work asynchronously, all elements in the data path (MSI and LSI circuitry) are clocked, so that "asynchronous" in this context means that a controller may wait a

variable but predetermined number of clock cycles for the completion of an operation by another controller.

5.2.2 Instruction Decoder

The instruction register is loaded at the beginning of the clock cycle following an acknowledgement of the "data ready" signal from the UNIBUS interface. The instruction decoder is structurally inhomogeneous because some control information can be derived directly from the operation code, while other information must be derived by increasing degrees of decoding.

The main elements of the decoder are two 16-input 8-output Field Programmable Logic Arrays (FPLAs). The sixteen outputs of these FPLAs can be subdivided into four fields of unequal size. A 5-bit field from FPLA2 generates the start address corresponding to one of the sixteen special instructions (SPE) that don't execute the standard task sequences defined by the DOP and SOP instructions. A 4-bit field from FPLA1 is routed to a one-of-sixteen decoder. Its output identifies the SPE instruction (if any), for use elsewhere in the decoder and in the Memory Access control unit. The separate decoding of these SPEcial instructions is necessary because of the haphazard instruction encoding and inconsistencies in the operation of conceptually identical or similar instructions. The remaining two fields of the



DECODING of machine language instructions in the SMP-11 can be separated into three parts, each one corresponding to a microprogrammed controller. Field Programmable Logic Arrays are used as lookup tables to generate the Handshake Sequence start address and selection code, the MACS special mode start addresses and to detect SPEcial type instructions. The MACU derives its register addresses directly from the instruction register and transforms the instruction addressing modes into microroutine start addresses by using PROMs. The ACU uses some control signals directly from the instruction register and generates the microroutine start addresses by gating several instruction register bits with the SPEcial mode instruction detect lines.

Figure 21: Instruction Register Decoder

FPLAs hold the 5-bit SPE handshake sequence start addresses and a 2-bit HS sequence start address select field.

The 6 bit Memory Access seqment microprogram start addresses for the Destination and Source operand addressing modes are stored in two identical PROMs. Their output is selected according to the 3 bit DST and SRC mode fields of the instruction op-code. The general register addresses in the Scratch Pad RAM are provided directly by the 3 bit DST and SRC register fields in the instruction's op-code.

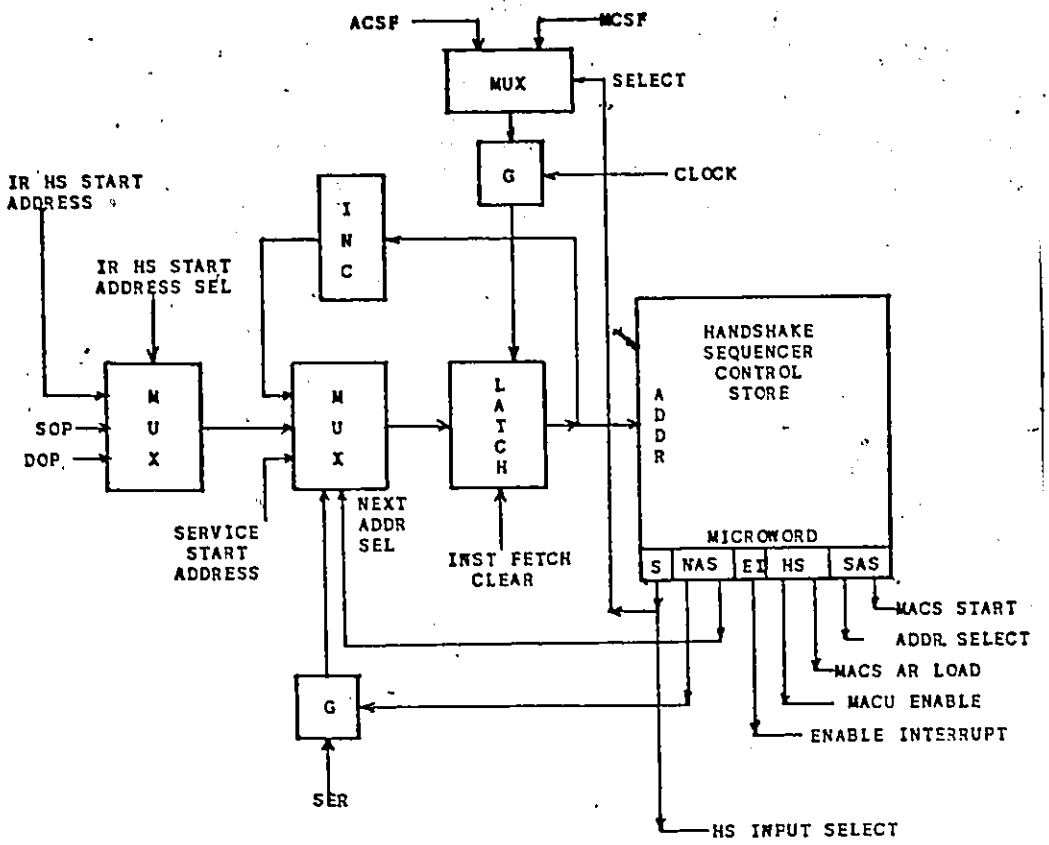
There are two classes of instruction which can generate arithmetic or logical operations. These are the double operand and single operand instructions, although the SPE type of instructions that affect the PSW must be accomodated by the ACU, since only the ACU has been permitted to affect the PSW. The DOP class must also include the MOV instructions, which modify the carry bit. The DOP class address is generated by IR bits 12-14 with some gating of bits 13 and 14 because of the SUBtract operation. The SOP class addresses are generated by IR bits 6-10, bits 9 and 10 used directly and with the three other bits being modified by the presence of the MFPS, EMT and TRAP instructions. The gated signals are applied to a multiplexer whose output is the ACS microroutine start address. The selection of the multiplexer is controlled by the DOP/SOP bit produced by the HS controller output.

5.2.3 Handshake Sequencer

The Handshake Sequencer is a microprogrammed controller which interfaces with the MACU and the ACU through enable and finished lines to determine which unit is to be enabled at any time. Upon receiving the "finished" signal from the unit executing a pending operation, the HS microcode sequencer steps to the next microword in its control store. The bits of the microinstruction determine the unit(s) to be or to remain activated and select the source of the next control signal that will cause the sequencer to step. The microsequencer only has to load a start address during an instruction fetch cycle and then sequence until the service mode. Then it may wait while the rest of the controllers execute an interrupt or proceed directly to the next instruction fetch if there is no interrupt.

Although several units could be activated at the same time, the implementation described herein is a non-pipelined version designed to demonstrate the usefulness of the design technique.

All possible sequences of machine states (SPE, SRC, DST, OP, POSTOP, SEB) are stored in the HS control store in an overlapped fashion in order to reduce the overall number of microwords needed. It is not an optimal encoding as simplicity of the Handshake sequencer was wanted and so there is no



SEQUENCING of the tasks executed by the ACU and MACU is accomplished by using an enable bit to activate either of the control units and a select bit to determine which "finished" signal will cause the next microaddress to access the HS control store. The next address can either be the start address of a sequence of tasks, provided by the instruction decoder or by hardwired inputs, or it can be the previous address incremented by one in order to step through the task sequence. Other outputs of the HSCS are signals to load the MACS Address Register and control selection of the MACS microroutine start address, and an interrupt enable line for use with the Service microroutine.

Figure 22: Handshake Sequencing Controller

"next microaddress" field. The 7 bits output from the control store PROM can be separated into a 2-bit microsequencer control field, a 2-bit processor state field and a 3-bit field used by the MACU.

TABLE 9

Handshake Sequencer Microcode

Address		Type	Sequence
0	100---01	IF 1	
1	101---00	IF 2	
2	10010000	SRC	1,2
3	10010100	DST	1,2,3
4	0000--00	OP	1,2,3
5	11101110	POST-OP	1,2,3,5
6	011-1110	OP	4
7	10011000	SPE	6
8	10010100	DST	6
9	0000--00	OP	6
10	11101110	POST-OP	6
11	10011000	SPE	7
12	10010100	DST	7
13	11101110	POST-OP	7
14	10011000	SPE SER	8
15	011-1100	OP SER	8

Note : " - " represents a don't care condition

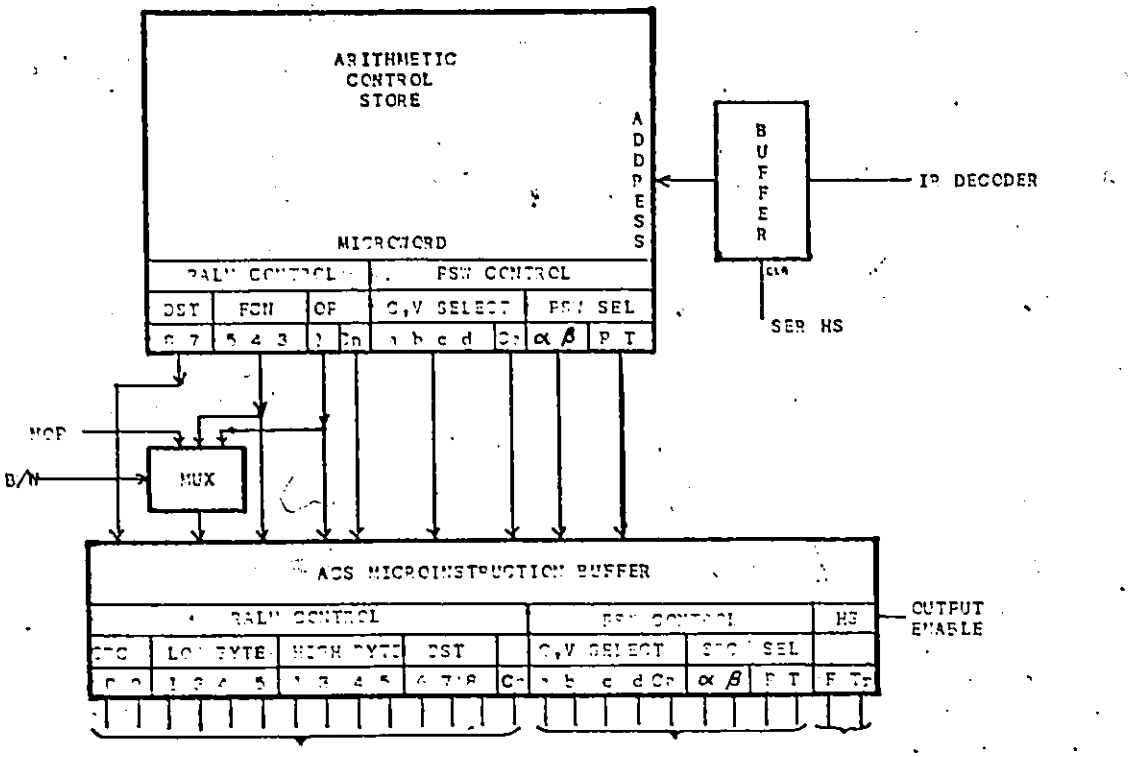
The HSCS microsequencer consists of a microaddress register loaded through a multiplexer and clocked by the output of a Select multiplexer "AND"ed with the first clock phase (CLK2). The sources of the address multiplexer are the incremented value of the present HSCS address, an external address and the address of the microroutine for the Service phase. This multiplexer is controlled by the HS address source select bits of the HSCS Next Address Select field. The other microsequencer control bit selects the "finished" input which will clock the new address value into the address register. The external HS address is supplied through a multiplexer controlled by the 2-bit HS sequence source select field from the instruction decoder's FPLAs. This multi-

plexer enables either a hardwired SOP or DOP start address or the SPE start address supplied by the instruction decoder to be selected as input.

5.2.4 Arithmetic Control Unit

The Arithmetic Control Unit takes control of the RALU (2901s) when it is enabled. It performs all DOP and SOP operations, as well as modifying the condition codes and the other bits of the PSW during special instructions. The hardware required to realize this unit consists of the control store PROMs, the microinstruction buffer and some microinstruction multiplexers. No microinstruction sequencer is needed to implement the basic instruction set because each arithmetic or logical operation and PSW operation can be executed in a single cycle.

The ACS microword can be split into two control fields. There is a 5-bit field used to control the 2901s and a 7-bit field to determine the source of the inputs to the PSW. The ACU "finished" bit is always set and is stored in the ACS microinstruction buffer. This bit is activated when the ACU is enabled, triggering the sequencing function of the HS controller immediately after the ACU is enabled. There is an additional ACS trigger bit which is used in conjunction with the BIL during the Service microroutine.



SIMPLICITY of the Arithmetic Control Unit is evident from the lack of microsequencer, made possible by the single cycle OP phase arithmetic, logic and processor status modification operations. The Arithmetic Control Store is 16 bits wide and contains 7 Register-ALU control bits routed to the 2901s and 9 PSW control signals selecting the inputs to the condition code, trap and priority bits. The outputs of the 3-State ACS microinstruction buffer alternate control of the PSW and 2901 functions with the lines from the MACS microinstruction buffer and also generate the handshaking bits for the HS controller. The RALU control section of the ACS microinstruction buffer has a separate high and low byte function code for the 2901s because byte operations in the PDP-11 modify only the lower byte of a 16 bit word.

Figure 23: Arithmetic Control Unit

Because byte operations execute only on the lower byte of a word and leave the upper byte unaffected, there are two ALU function control fields required, one for each byte. A

multiplexer controlled by the byte/word detect line from the instruction decoder selects which field to load into the microinstruction buffer for the control of the upper byte. This multiplexer chooses between the microoperation of the lower byte (for word operations) and a hardwired no-operation code for the byte operations.

TABLE 10

Arithmetic Control Store Microcode

ACS Microcode

Address	8754321cabcdCnABPT	operation
0	010110-----01011	PC to PSW
1	----------	
2	010110------1100	CC
3	010110-010100100	SWAB
4	010110-----01011	TRAP, EMT
5	----------	
6	----------	
7	----------	
8	011001-01-100100	CLR
9	011111-01-110100	COM
10	010001110000-0100	INC
11	010011010010-0100	DEC
12	010101111100-0100	NEG
13	000001011111-0100	ADC
14	000011010011-0100	SBC
15	010111-01-100100	TST
16	100111-0000-0100	ROR
17	110111-0000-0100	ROL
18	100111-0000-0100	ASR
19	110111-0000-0100	ASL
20	110100------1010	MTPS
21	----------	
22	010110-10-0-0100	MFPS
23	010011-10-0-0100	SXT
24	010010101110-0100	SUB
25	011010-10-0-0100	BIC
26	010100101100-0100	CMP
27	010000000011-0100	ADD
28	010110-10-0-0100	MOVE
29	010110-10-0-0100	BIS
30	011000-10-0-0100	BIT
31	011100-10-0-0100	XOR

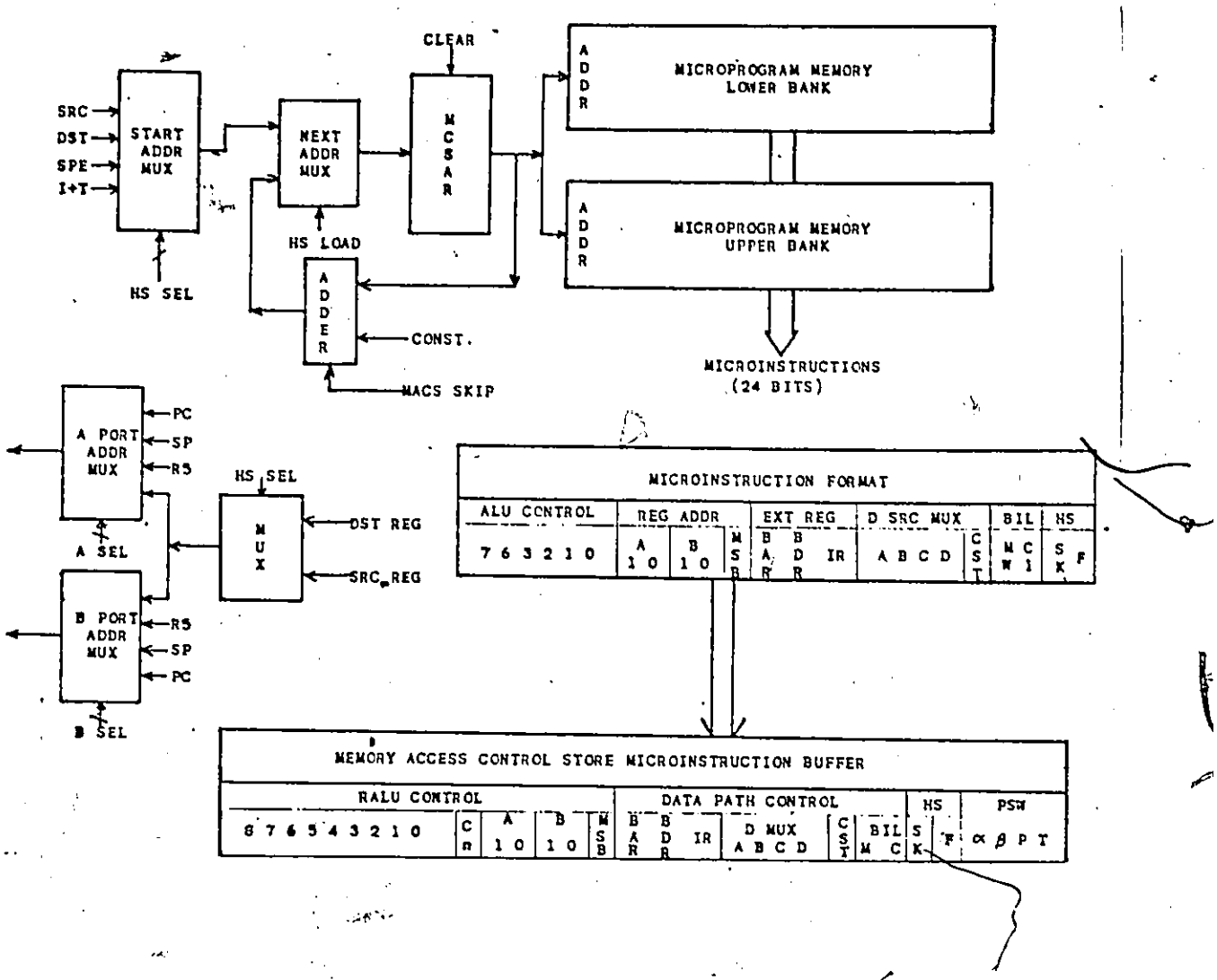
5.2.5 Memory Access Control Unit

The Memory Access Control Unit circuitry is considerably more complicated than that of the ACU because it controls all the data paths in the processor and the microroutines

last several clock cycles. This controller is made up of four main units : the microprogram memory, the microinstruction decoders and the microinstruction register, the microsequencer and the start address selection circuitry.

The Memory Access Control Store is organized as two banks, the upper holding the microcode used by the SPE instructions and the lower bank containing the microroutines used by the machine language memory addressing modes. This functional subdivision allows easier debugging of the microroutines, as well as more flexibility in writing the software since each bank can be upgraded or even redefined separately while in the design stage. However, the fields of the microinstructions are the same for both banks and can be grouped under the headings of ALU and data path control, microsequencing control and Bus Interface Logic control. The ALU and data path field is itself split into 3 parts, with 7 bits controlling the ALU and data routing functions of the 2901s, 5 bits addressing the registers inside the 2901s and 4 bits selecting the D bus multiplexer source and gating the instruction register and the Bus Data and Address registers.

The microsequencer control field consists of the MACS "finished" bit and the increment to be added to the present control store address. The Bus Interface control lines de-



MICROPROGRAM addresses can be directed to either of two banks of control memory, the lower bank containing all the memory and register accessing modes and the upper bank holding the microroutines for the SPEcial instructions and Service calls. The handshake sequencer controls the start address selection for Source mode, Destination mode, SPEcial instructions and interrupt and trap service routines. The microinstructions are sequenced by incrementing by one in an adder, or by incrementing by two when the skip enable is true. The MACU also selects the 2901 A and B port addresses, which can be hardwired and correspond to the Program Counter, Stack Pointer and Register 5 in the lower 8 registers, buffers at the same addresses in the upper 8 registers or register addresses input directly from the instruction register. The microinstructions contain 6 fields that define the operation of the 2901s, specify its internal register addresses, control the bus and instruction registers, route inputs through the D bus multiplexer, interface with the Bus Interface Logic through bus control and memory wait bits, determine the next microprogram address and indicate the end of a microroutine to the HSCU. The MACS microinstruction buffer contains the entire 9 control lines for the 2901 and hardwired select lines for the PSW input multiplexer.

Figure 24: Memory Access Control Unit

signate the type of memory operation to be performed and stop the clock at the end of the cycle until a "data ready" is received from the BIL.

Some of the data path control lines are mutually exclusive in activation, so that the width of the microinstructions can be reduced by encoding these bits in the microstore and using an external decoder before they are clocked into the microinstruction register at the beginning of the next major clock cycle, when enabled by HS controller.

The MACS microsequencer contains a 6-bit MACS address register which points to the next location to be accessed in the control store. This register can be loaded from the Start Address multiplexer or from the output of an adder which increments the present value of the register and can perform short forward branches in the microprograms. No subroutining capability is required as there are no loops or backwards branches to lower microaddresses since the microcode is implemented in a simple sequential manner.

The microsequencer is operated in a pipelined mode starting its next microinstruction fetch by clocking the next microaddress into the MACS microaddress register halfway through the major clock cycle, after the HS unit has determined which control unit is to be enabled during the succeeding cycle.

TABLE 11

Memory Access Control Store Microcode

MACS Microcode 01234567890123456789012 3	operation
100101111101000----01001	IF 1
01010011--10110000100001	IF 2
11010000111000----00001	REG 1
10010011001000010000000	REG 2
10010100000100030011111010	A INC 1
110111--11100000100030001	A INC 2
10010011--10100110001101	A INC 3
11110100000100300111110	A DEC 1
110111--11100000100300001	A DEC 2
01010011--10100110001101	A DEC 3
10010111110100000111000	IX 1
01010100--01000000001010	IX 2
110111--11100000100000001	IX 3
01010011--10100110001101	IX 4
01010000--01000----01010	REG DEF 1
110111--11100000100000001	REG DEF 2
01010011--10100110001101	REG DEF 3
10010100000100000111000	A INC DEF 1
010111-----10000000001010	A INC DEF 2
110111--11100000100000001	A INC DEF 3
11010011--10100110001101	A INC DEF 4
11110100000100000111000	A DEC DEF 1
010111-----10000000001010	A DEC DEF 2
110111--11100000100000001	A DEC DEF 3
11010011--10100110001101	A DEC DEF 4
10010111110100000111000	IX DEF 1
01010100--01000000001000	IX DEF 2
010111-----10000000001010	IX DEF 3
110111--11100000100000001	IX DEF 4
11010011--10100110001101	IX DEF 5
11010111110000300111100001	BR
11010111100003003111100000	MARK 1
11010001100000----00000	MARK 2
10110110100100000111000	MARK 3
010111--01000030000000001	MARK 4
11010000110000----00000	RTS 1
10010110100100003011111000	RTS 2
010111--00900030000000001	RTS 3
10010110100100000111000	RTI/T 1
110111--11000000000000000	RTI/T 2
10010110100100000111000	RTI/T 3
11011101--00000000000001	RTI/T 4
11110100000000000110000	SOB 1
11110110101000000100010	SOB 2
11110111110000011110001	SOB 3
11110110100100000110000	JSR 1
01001100000-01000000001100	JSR 2
11010011000000000100001	JSR 3
11110110100100000110000	TRAP 1
01011101011010000111001100	TRAP 2
11110110100100000110000	TRAP 3
01001111010010000111001100	TRAP 4
11011101111100010010001000	TRAP 5
11011101110000000000000000	TRAP 6
11010111111100000111000	TRAP 7
110111--11100000000000001	TRAP 8
110111--11100000011100001	RFFS

A multiplexer selects between the pre-op, post-op and service microroutine start addresses. Pre-operation start addresses correspond to the SRC or DST addressing modes and to the SPE instruction using pre-op phases. Pre operation addresses can be SRC or DST mode starting locations, as well as those of some SPEcial microroutines. The post operation start addresses are the remaining SPE instructions that execute as hardwired macro instructions and generate longer sequences of microcode. The other input to the multiplexer is the start address of the interrupt and trap routine in the Memory Access segment Control Store. This microroutine is identical for all traps since it is only the trap vector address that changes. As can be seen in the microcode listings of table there are many redundant microwords in the microprograms, leading to seemingly inefficient packing of the microinstructions in the control store. There are 28 fundamental microinstructions combined in the different microroutines. The number of redundant microinstructions could be reduced by using a more complex microsequencer accessing only the 28 basic microinstructions, although a few repetitions would still be necessary for those microroutines which start with the same microinstruction.

This microsequencer would need 32-way (5-bit) branching capability as well as the ability to jump to subroutines of common code and then return to the microinstruction at the

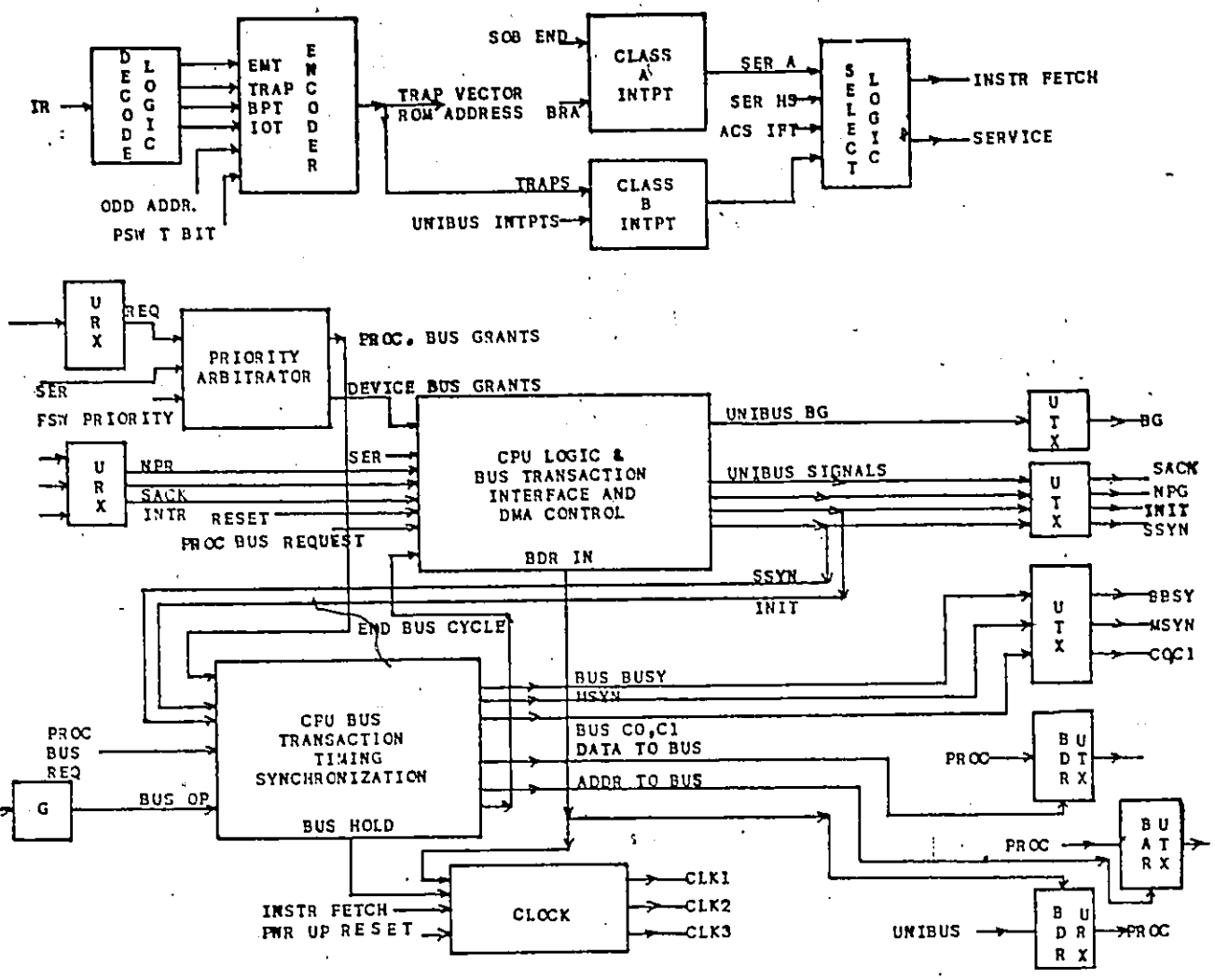
address following the subroutine call. These facilities are available in several LSI bit-slice microsequencers, although extra microword bits would be needed to provide a next address field, a microsequencer function select code and a microbranch condition select field. This would typically add 12 bits to the current 24-bit wide microword, to save 1 memory chip and 384 of the original 1536 bits. These gains for the reduced system are offset by the complexity and cost of the microsequencer circuitry, the slower operating speed and much more complicated microsoftware.

For a prototype design the positive aspects of simpler microcode and microsequencer circuitry as well as lower cost were considered to outweigh the advantages of the reduced microcode system.

5.2.6 Bus Interface, Clock and CPU Interrupt Logic

The last unit in the control structure contains three modules, the Bus Interface, the Clock circuitry and the CPU Interrupt logic. Although not essential to the understanding of Segmented Microprogramming, this section is used to provide the necessary link between the implementation and the realization of the architecture.

The CPU interrupt module accepts interrupt requests from the Bus Interface module, CPU hardware traps and special in-



INTERFACE logic between the processor, interrupt circuitry and the UNIBUS controller can be divided into three interacting sections. The first part involves the processing of CPU traps and UNIBUS interrupts to determine whether the service microroutine can be executed before the next instruction fetch cycle. The second section arbitrates UNIBUS requests from peripheral devices and from the processor and provides the bus grant selection and acknowledge signals. The third section controls the timing of the processor operations, provides all the synchronization signals for processor UNIBUS transactions under the direction of control signals from the MACS microinstruction buffer and gates the data and addresses between the processor and the UNIBUS.

Figure 25: Bus Interface, Clock and Interrupt logic

struction traps. It arbitrates the requests according to a fixed priority scheme and permits the highest priority request to interrupt the CPU at the start of the POSTOP cycle. This interrupt controller also selects between an external Trap Vector address input through the Bus Data input register or a trap address read from a PROM containing the addresses corresponding to the hardwired interrupt and trap routines.

The Bus Interface logic creates the link between the processor and the UNIBUS. It consists of all the UNIBUS drivers and receivers and the UNIBUS control and interrupt logic. The UNIBUS interrupt logic arbitrates all interrupts external to the processor and transmits the highest priority signal to the CPU interrupt module. The UNIBUS control logic processes all bus requests, sequences all the required bus transactions and grants the bus to devices having a higher priority than that of the CPU. The control logic also interacts with the Handshake Sequencer, the Memory Access Control Store BIL control lines and the clock circuitry to generate the CPU bus request signal for the UNIBUS interrupt logic, the strobe signals for the UNIBUS latches and the clock start and stop signals. At the end of the service routine it generates the Instruction Fetch signal for the Handshake Sequencer and Memory Access Control unit.

The clock circuit generates a major clock cycle length of 200 nano seconds, as well as two one quarter cycle (50 ns) non-overlapping phases, CLK2 and CLK3. The main cycle (CLK1) clocks the IR and microinstruction buffers, the 2901s and the PSW; phase CLK2 clocks the HS controller and phase CLK3 clocks the MACS address register. Signals from the BIL can affect the cycle length by stopping the master clock at the end of a major cycle until a Data Ready is received by the BIL. Timing diagrams for all the task sequences are listed in Appendix A.

5.2.7 Console Requirements

The console for the SMP-11 has not been designed, although its requirements will be outlined here. The machine's console must operate at 2 levels : the instruction set processor level, as in conventional designs, and the microprogramming level, in order to facilitate hardware debugging. The instruction level commands that must be available are :load memory address, data register or PSW; examine contents of a location (CPU or memory); halt; reset; and allow input of data and addresses to the UNIBUS. These functions can be implemented either by microcode in the CPU or by an external microprocessor interfaced to a keyboard, as in the PDP-11/34.

The microprogramming console must interface directly with the hardware in the CPU. It must be able to monitor, not necessarily concurrently, the data on the internal D and Y buses, the main registers such as the PSW, Bus Data In, Bus Data Out, Bus Address and Instruction Register; the output of the HSCS, ACS and the MACS and their address registers; the addresses of the operands being accessed inside the RALU and the status/state of the BIL.

The complete testing of the units or subunits requires the ability to enter data into the Instruction Register, microinstruction data registers, the PSW, the input and output buses of the 2901 RALUs, the addresses of the registers inside the RALUs and access to the shift inputs and outputs of the RALUs. Most of these functions can be performed by inserting multiplexers into the microinstruction and data bus paths with the input selected by the console panel. Finally, the clock circuitry also should be modified to allow single step operation.

Chapter VI

EVALUATION, CONCLUSIONS AND FUTURE RESEACH

The purpose of this Chapter is to evaluate the design technique as it was applied to the SMP-11. First the design is compared to other implementations of the PDP-11 architecture in order to give a measure of the efficiency of the implementation in terms of control store size and design time. The comparaisons are also used to obtain an estimate of its performance in terms of operating speed and cost. An investigation of the expandability and testability of the SMP-11 shows that the architecture after the cost reduction overlay of the ALUs is more difficult to test and modify than a separated implementation but that these features are still available. The final section of the Chapter indicates areas for future research in the overlapped operation of segmented microprogrammed processors, a more detailed look at the reliability of the processor and the application of the segmentation ideas to other areas of computer architecture.

6.1 CONTROL STORE SIZE

The size of the microprogram used to implement a target architecture on a host machine gives a measure of the efficiency of the emulation. Table 12 shows the control store

sizes of different implementations of the microprogrammed PDP-11 computers (8). The commercially available models have a highly PDP-11 oriented microprogramming level architecture, resulting in compact microprograms ("only" 256 words of micromemory in the 11/40 (13)).

TABLE 12

Control Store size of PDP-11 implementations

	PDP-11/10	CMU-11	SMP-11	PDP-11/40	PDP-11/45
	40b x 239w	32b x 287w	24b x 57w 16b x 27w 8b x 16w	56b x 251w	64b x 256w
TOTALS :	9560	9184	1928	14056	16384

b : number of bits per word
w : number of words in the micromemory

At the microlevel organization there is a tradeoff between generality of design and efficiency of execution of the microinstructions. Although the SMP-11 hardware organization was tailored to emulate the PDP-11 basic instruction set, it retains several generalities of the design technique relating to the modification of the PSW and the memory and register addressing modes. Despite its more general organization, the SMP-11 has the smallest control store size of all the implementations of Table 12.

The SMP-11 takes less than one-quarter of the microcode required for the most "efficient" regular microprogrammed version, the CMU-11, designed by the Carnegie Mellon University using bit-slice technology and Computer Aided Design techniques (15). These last two factors make the CMU-11 a good reference for comparaison. The SMP-11 and the CMU-11 were both implemented with approximately the same number of integrated circuit packages, the CMU-11 using 144 chips and the SMP-11 using 155 chips. The distribution of the ICs amongst the processor sections is shown in table 13

TABLE 13

Integrated Circuit Distribution in the Processors

PROCESSOR COMPONENT	NO. OF IC PACKAGES SMP-11	CMU-11
DATA PART		
PE Array	4	8
PSW and Instruction register	14	6
Miscellaneous	35	4
subtotal	54	18
CONTROL PART		
Control Store ROMs	9	8
Microinstruction Buffer	7	10
Sequencer	17	1
MicroBranch Logic		26
PSW control	5	16
Misc	15	18
subtotal	53	79
UNIBUS INTERFACE		
Bus Receivers and Transceivers	16	19
UNIBUS control	32	28
subtotal	48	47
TOTAL	155	144

and is also similar. The differences are due to the architecture and bus structure of the bit-slice processor chips chosen. The 3002 element of the CMU-11 has three input buses and two separate output buses, while the 2901 has only a single input bus and a single output bus, making it neces-

sary to use an extensive input multiplexer on the 2901's input bus. However, the 2901 can perform single microcycle register read-modify-write and ADD-and-SHIFT operations internally. It also readily provides the flag signals for the condition codes.

It is claimed (15) that the main reason for the smaller control store size of the CMU-11 as compared to the other PDP-11 implementations lies in the fact that commercial PDP-11s (except the LSI-11 and PDP-11/20) use 4-bit slice MSI ALUs (74181) that do not have as useful a set of primitive operations as the 3002 RALUs of the CMU-11. Since there are certain tradeoffs in using either the 3002s or the 2901s, it does not seem likely that the set of primitive operations would cause a factor of four reduction in the microcode from one bit-slice implementation to another.

The 2901s used by the SMP-11 can perform fewer operations than the 74181s but have the advantage of an architecture based on a two-port register bank. This feature was used to locally optimize the memory access microroutines but was not used to simplify register to register operations. If the microprograms of the MACS are rewritten for the 2901 using only one port of the register bank an additional 9 microinstructions (216 bits) are added to the control store. Since the microprograms would still be one quarter the size of

those of the CMU-11, it is proposed that the major reason behind the reduction in microcode size is the segmentation of the control structure into dedicated microcontrollers based on the instruction set breakdown introduced in Chapter 3. As previously explained, this segmentation eliminates the need for similar microroutines for different instructions and reduces the length of the microword because of the lesser number of control lines for each segment.

6.2 DESIGN TIME

The CMU-11 also provides a reference for comparing the complexity of the design procedures. The CMU-11 was designed in 39 (40 hour) man-weeks using Computer Aided Design techniques which automatically maintain correct and consistent documentation (schematic diagrams and wirelists), reduce the turnaround time for design alterations and free the designer from the bookeeping and clerical tasks. The CMU-11 was also simulated by software to test the layout of the data paths and control structure. Although timing, loading and signal propagation problems have to be debugged on the actual hardware, simulation simplifies debugging and can show the validity of a "fix" before trying it out on the actual hardware.

Starting with the instruction set breakdown and classification scheme of Chapter 3, the design technique outlined in

Chapter 4 was applied to design the SMP-11 in about 4 man-weeks, exclusive of the actual wiring diagrams, which take another few weeks to prepare without the use of CAD techniques, and the debugging time, which would also take a few weeks.

After the initial design was laid out, some time was spent on obtaining a reasonably-efficient hardware structure for the IR decoder and the control of the PSW. The generation of the condition codes is complicated by the fact that logically equivalent instructions don't always have consistent flag settings. This problem occurs accross the entire PDP-11 instruction set and there are many unexpected special cases. This "overspecification" of the condition codes requires extensive hardware to implement while, in many cases, not producing correct or consistent results (14).

The structured design technique of the SMP-11 permits modular design of each segment and produces very compact microroutines for each control store as well as very simple sequential microroutines. Based on the User's Manual description of the operations performed by each instruction, the complete set of microprograms for each of the control stores were written in less than one man-day apiece. The small size and simplicity of the microroutines also gave extremely fast turnaround time from design decision changes to the actual hardware and/or microcode changes.

6.3 PERFORMANCE ESTIMATES

Table 14 lists the projected Instruction execution times, number of microcycles required per instruction and the number of bus (memory) cycles for the SMP-11, as well as the actual values for the PDP-11/34 and /40 models as reported in (8). There are 7 regions examined in the table consisting of the DOP, SOP, MOVE, program control and system control instruction types plus the instruction fetch and addressing mode times. All execution times for the SMP-11 use a projected 200 nanosecond cycle time.

Several conclusions can be drawn from Table 14.

1. The faster execution of the program control and system control instructions in the SMP-11 is due to a decreased number of microcycles needed to perform the same operations. This is also apparent in the operand addressing mode times for modes 6 and 7. The reduction in the number of microcycles used to execute instructions is a property of the microlevel architecture of the SMP-11 and does not directly result from the design technique.

2. The reduction of the number of Instruction Fetch cycles is due to a modification of the clock circuitry and is not a function of the microcode.

TABLE 14

Processor Instruction Execution Times

	SMP-11	PDP-11/34		PDP-11/40	
I fetch	1/2 .50	1/3 .72	1.66	1/4	1.12
Addressing Modes					
0 R	0/1 .20	0/1 .18		0/0	0
1 @R	1/2 .40	1/1 .18	1.12	1/3	.78
2 (R)+	1/2 .40	1/2 .36	1.30	1/3	.84
3 @(R)+	2/3 .60	2/3 .54	2.42	2/5	1.72
4 -(R)	1/2 .40	1/2 .36	1.30	1/3	.84
5 @-(R)	2/3 .60	2/3 .54	2.42	2/5	1.72
6 X(R)	2/3 .60	2/4 .72	2.60	2/5	1.34
7 @X(R)	3/4 .80	3/5 .90	3.72	3/7	2.12
Instructions					
MOV/B	1/1 .40	1/1 .24	.78	1/3	.64
ADD	1/2 .40	1/1 .24	.78	1/3	.54
SUB	1/2 .40	1/1 .24	.78	1/4	.68
BIC/B	1/2 .40	1/1 .24	.78		.54
BIS/B	1/2 .40	1/1 .24	.78		.54
BIT/B	0/1 .20	0/1 .18		0/3 .48	
CMP/B	0/1 .20	0/1 .18		0/3 .48	
XOR	1/2 .40	1/1 .24	.78	----------	
CLR/B	1/2 .40	1/1 .24	.78	1/4	.62
COM/B	1/2 .40	1/1 .24	.78	1/4	.62
INC/B	1/2 .40	1/1 .24	.78	1/4	.62
DEC/B	1/2 .40	1/1 .24	.78	1/4	.62
ADC/B	1/2 .40	1/1 .24	.78	1/4	.62
SBC/B	1/2 .40	1/1 .24	.78	1/4	.62
ROL/B	1/2 .40	1/2 .42	.96	1/4	.62
ASL/B	1/2 .40	1/2 .42	.96	1/4	.62
ROR/B	1/2 .40	1/2 .42	.96	1/4	.84
ASR/B	1/2 .40	1/2 .42	.96	1/4	.84
TST/B	0/1 .20	0/1 .18		0/4	.62
NEG/B	1/2 .40	1/2 .42	.96	1/3	.54
SWAB	1/2 .40	1/1 .24	.78	1/3	.54
SXT	1/2 .40	1/1 .24	.78	1/4	.62
BRA(true)	0/1 .20	0/3 .54		0/3 .64	
BRA(false)	0/0 .00	0/0 .00		0/2 .28	
SOB(true)	0/2 .40	0/4 .78		0/5 1.24	
SOB(false)	0/1 .20	0/2 .42		0/5 .92	
JMP	0/0 .00	0/1 .18		0/2 .34	
JSR	1/3 .60	1/5 .96	1.50	1/6	1.48
CC ops	0/1 .20	0/2 .36		0/2 .60	
MARK	1/4 .80	1/8 1.44	2.38	1/6	1.54
RTS	1/3 .60	1/4 .72	1.66	1/4	1.28
RTI/T	2/4 .80	2/6 1.08	2.96	2/6	2.32
IOT,EMT	2/9 1.80	2/13 2.46	5.42	2/14	4.18
TRAP,BPT					

11/34 cycle times : 180 and 240 ns.
11/40 cycle times : 140, 200 and 300 ns.
Format : r/m t1 t2
where r : # memory reads/writes
m : # microcycles
t1: time in microseconds without UNIBUS access
t2: time in microseconds including UNIBUS access

3.

The SMP-11 is slower for the SOP, DOP and MOVE op-

erations because it uses a pipelined mode of execution requiring 2 microcycles, the first cycle performing the OP phase operation and the second cycle storing the result in the POSTOP phase.

4. It should be easier to test the execution time of the instructions on the SMP-11 because the only special case occuring in Table 14 is when the destination data fetch for MOVE instructions is skipped.

In (15) the CMU-11 is evaluated to run about 1.6 times slower than the 11/40, as measured on an operational system by benchmarking and testing a few particular tasks. Taking into account its reduced instruction fetch time, the SMP-11 should execute at approximately the same speed as the PDP-11/34, which is 1.45 times slower than the 11/40. Proper evaluation of the SMP-11 should be made by examining the actual execution rate of all instructions on a working model.

A package count and cost listing for each section of the processor is given in Appendix B. This data is used in Table 15 to compare the cost of the CMU-11 (1976 prices) with that of the SMP-11 (1980 prices). Overall, the SMP-11 is less expensive because of the smaller number of high density chips used. The cost differences for the LSI and SSI/MSI

sections reflect this fact but also show that circuit costs have decreased with time. The difference in cost of the PROMs reflects the large reductions in memory cost since 1976 and the significant savings obtained by using smaller capacity chips in the SMP-11.

TABLE 15

Processor Cost Breakdown

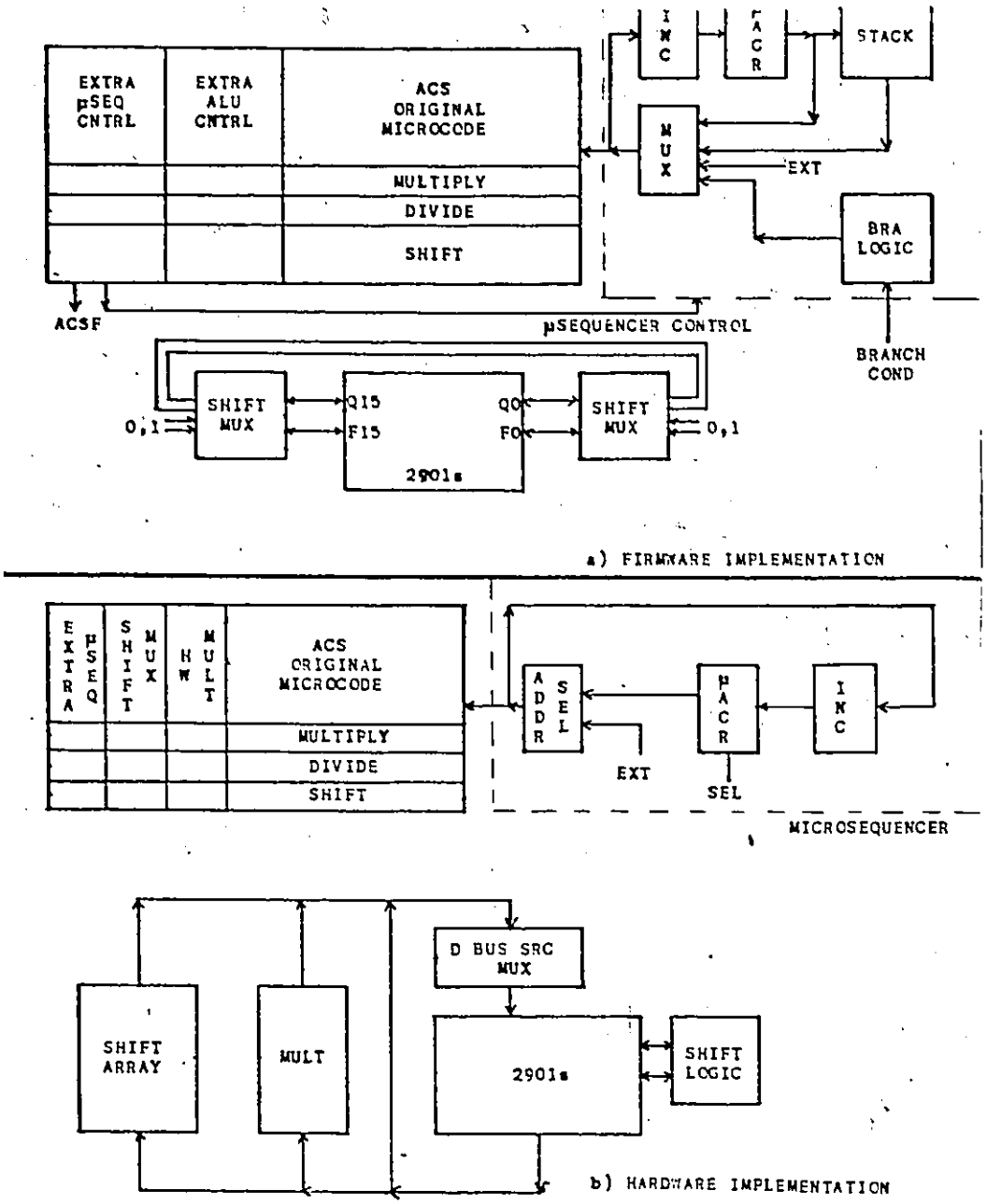
	COMPONENTS		PRICES	
	SMP-11	CMU-11	SMP-11	CMU-11
LSI	2901,2902	3001,3002,3003	$74.77	$188.00
PROMS	PROM,FPLA	PROM	$72.70	$204.00
SSI/MSI	misc	misc	$93.07	$179.00

6.4 EXPANSION OF SMP-11

Although the SMP-11 was designed to emulate the PDP-11, optimization of register-to-register operations and overlap of bus cycles and arithmetic operations were not implemented in order to keep the design close to the general architecture. The SMP-11 is not specifically laid out for easy expandability, as would be the case of the general purpose segmented microprogrammed processor of figure .

In order to show that the structured design technique allows modifications to be easily made, the case of adding the Extended Instruction Set (EIS) as defined by the Digital Equipment Corporation (MUL,DIV,ASH,ASHC) will be examined. MUL and DIV are 16 bit integer multiplication and division routines which form a 32 bit result. ASH and ASHC are the arithmetic shift and arithmetic shift combined instructions respectively. They perform left or right shifts of up to 32 places, as specified by a signed 6 bit binary number. ASH operates on 16 bit words and ASHC on 32 bit doublewords or can perform rotates on 16 bit words.

These instructions can be added by providing extra microroutines and hardware. The major modificatins would be applied to the Instruction Register decoder and the Arithmetic Control Store. The ASH and ASHC instructions could be implemented by executing microroutines of variable length determined by the number of places to shift as specified in the instruction op-code. This means that the ACS microsequencer would need increment and loop or branch capabilities. The microinstruction format would have to be extended to allow the controller to utilize the full capabilities of the 2901s, such as accessing other register bank addresses and using the Q register. The MUL and DIV microroutines could be written easily using the branching capabilities of the microsequencer.



EXPANSION of the arithmetic functions of the SMP-11 can be achieved by making either firmware or hardware based additions to the ACU. Part a) shows the ACU modifications necessary to implement the Extended Instruction Set of the PDP-11/34. Hardware modifications to take full advantage of the 2901 FALU's capabilities consist of restructuring the shift logic to include the Q register and the addition of microsequencer increment, subroutine and branch capabilities. These changes require more control bits in the ACS, in addition to the microroutines for the multiply, divide and shift operations. Part b) of the figure shows a hardware-oriented solution, where a 16 bit multiplier and a shift array are added. Some extra shift logic is needed to implement the divide operation, which would still be microcode based. The microsequencer would also have to be extended to include increment capabilities for the multiply and divide operations, which involve 32 bit results. This hardware intensive implementation would require extra microcode in the ACS to provide the additional hardware control lines as well as the few microinstructions for the operations.

Figure 26: Microcode and Hardware Expansion Alternatives

The segmentation of the design allows another approach to be considered. Specialized hardware could be added to perform the ASH, ASHC and MUL functions. The DIVide operation could be implented in sequential microcode using an ACS microsequencer which could increment and some microinstruction modification hardware (a few gates) for the 2901s. The MUL operation could be accomplished by adding a 16 bit multiplier such as the TRW MPY-16HJ, which can form a 32-bit product in 100 nanoseconds. By connecting the multiplier input bus to the Y bus and its output bus to the D bus input multiplexer, the Arithmetic controller could set up, multiply and store the result in 4 microcycles (800 nanoseconds). The ASH and ASHC functions could be performed by a shift tree controlled directly by the instruction op-code and inserted in parallel with the hardware multiplier, allowing complex shifts to be performed in a single microcycle. The two specialized units could be controlled by an additional arithmetic/shift unit selection field and a control field in the microinstruction.

The point to note is that if additional arithmetic or logical operations are to be added, only the instruction decoder and the ACU need be changed, since there would be no changes in the addressing modes or in the sequencing of the microroutines as controlled by the HandShake unit. If new addressing modes or special instructions not involving the

PSW are to be added, only the instruction decoder, HS controller and memory access control unit need be changed. If the PSW is to be modified also, then the ACS might have to be modified to accomodate it. If the operation of existing instructions is to be modified, then only the ACU, MACU or the HSU or some combination of the three would have to be changed.

6.5 TESTABILITY OF THE DESIGN

Segmented microprogramming has built-in module sensorialization, in that testing of large scale integrated circuits requires fault detection at the functional unit level instead of the internal gate level. The test patterns to be applied can be supplied by a process known as algorithmic pattern generation, which creates test data through an initial pattern and some digital feedback to change the pattern after each cycle.

The different segments can be tested by diagnostics microprograms separately while not in use, resulting in an inherently distributed hardcore. This overcomes the limitation of processors using a single hardcore, where a small portion of the CPU hardware is first tested and the area under test is gradually expanded, using the error-free portions, to diagnose untested portions of the hardware. If the hardcore of such a bootstrapped diagnostic testing system is faulty,

the rest of the processor cannot be tested. In segmented microprogramming, by having separate segments with their own control, each segment constitutes a hardcore subsystem. Alternately, each segment could be connected to an external and possibly detachable test system so that the the hardcore need be examined fully only during testing.

The multiplicity of the circuitry for the control stores such as sequencers, output buffers and ROMs can decrease the reliability of the control logic, although it is more testable. In the system designed, using the single RALU has reduced or eliminated a lot of the separate testing concepts and the design reverts to a single hardcore system consisting of the RALU. This RALU can be tested by the ACS, which would also have to test the PSW. The rest of the data paths could then be bootstrapped by the MACS using the RALUs, so that testing would be the same as if there were a single control store except that the RALU can be tested separately.

6.6 FUTURE RESEARCH

The SMP-11 was designed to operate in a non-overlapped manner because of the nature of the arithmetic and logic operations considered. The addition of the microprogrammed EIS would make a good case for the parallel operation with the MACU, which could pre-fetch instructions and operands.

The parallel operation of units inside a processor, such as that described by Tomasulo (19) for the floating point unit of the IBM 360/91 and the method proposed by Higbie (10) use an interlock mechanism which inserts inhibit bits in the instructions. This interlock control could be accomplished by the handshake sequencing unit of a segmented microprogrammed processor since the function of this device is to determine which segments are active as well as the tasks to be executed by each segment. It is also felt that the breakdown of the instructions into execution phases could be used to segment the hardware differently, so that the tasks of each phase could be performed by pipelined segments to achieve higher throughput. Overlapped execution of several instructions could then be accomplished by operating the handshake sequencer in an interrupt mode from a stack of task sequences generated by the instruction decoder.

Testability, failure rates and failure modes are becoming increasingly important in processor design. New processor designs use extensive parity checking in the microprogram memory, redundant arithmetic and local storage units and special fault detection registers to aid in diagnosing the source of failure. Fail-soft designs, where failed dedicated functional units can be replaced by microprograms operating on the remaining 'good' hardware, are also becoming practical because of the increasing costs of the software being

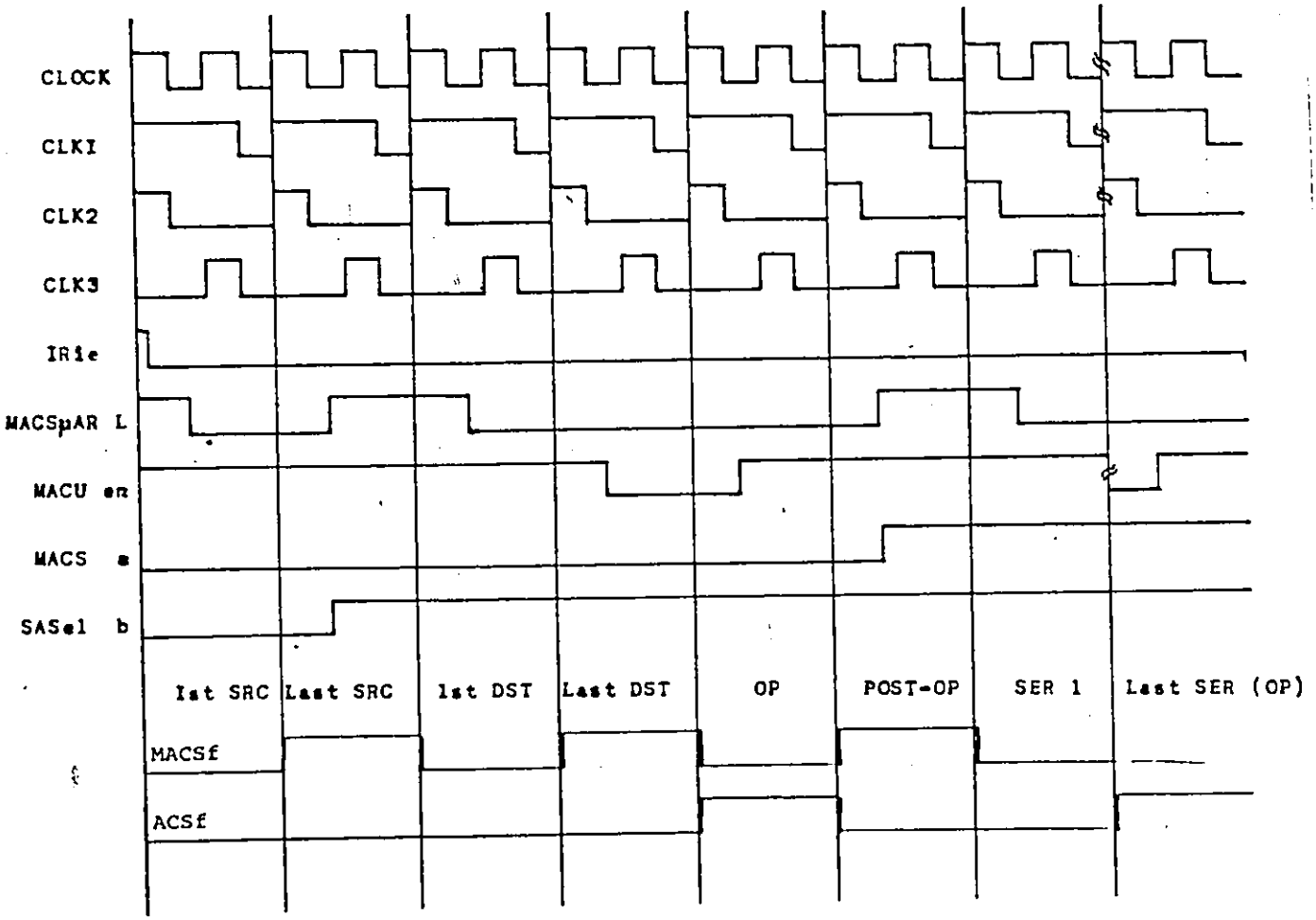
run. The present configuration of the SMP-11 would allow testing of the arithmetic only during UNIBUS access delays, but a fully segmented design could use microdiagnostics to test functional units not executing HS driven tasks. Also, by building a fully segmented processor, its performance could be determined and hardware modifications for fail-soft capability could be investigated.

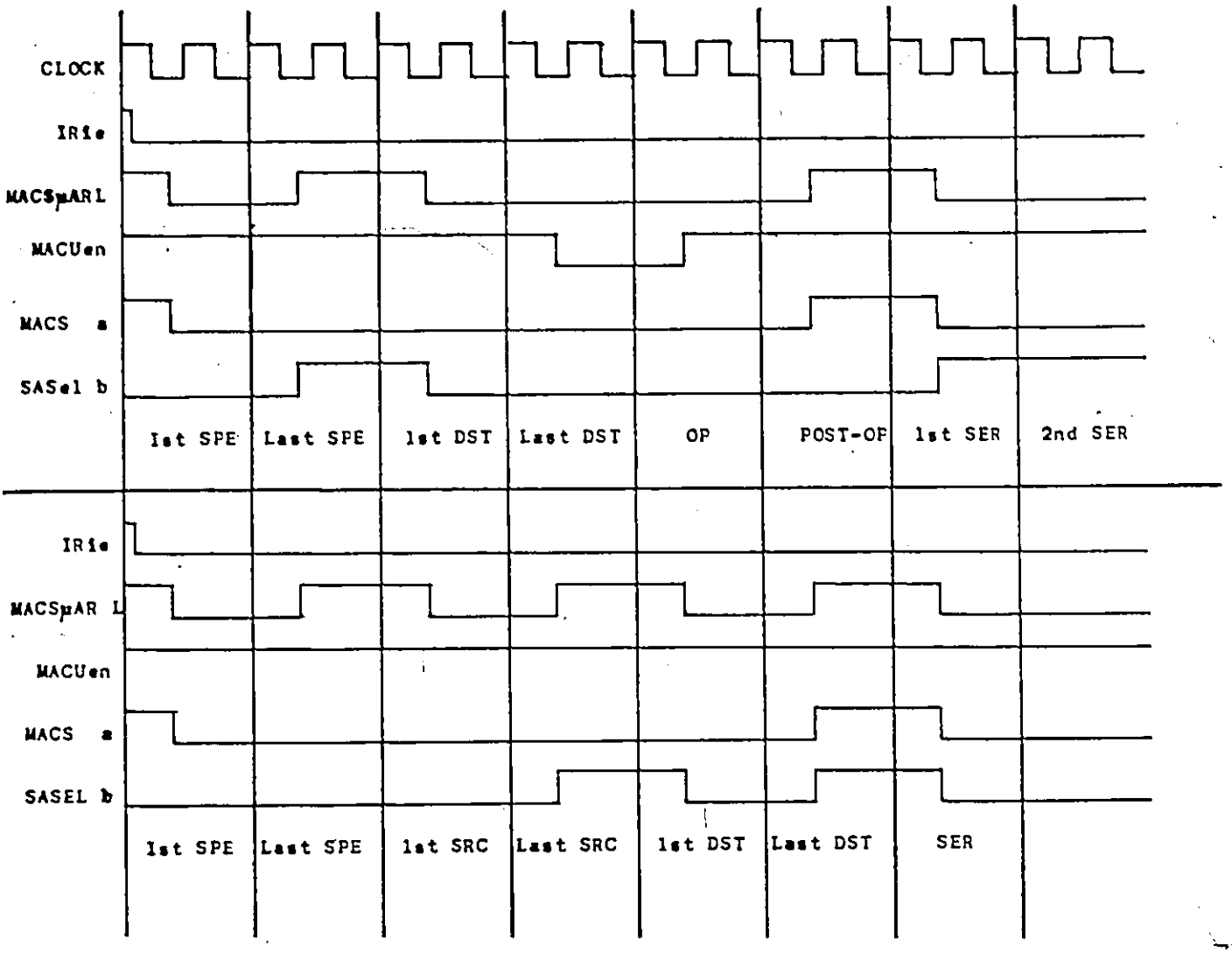
The concept of segmenting the processing resources of a computer to execute independant tasks can also be applied to other areas of computer architecture. In (17) a multi-microprocessor system was described which used a number of dedicated microprocessors, each able to execute a restricted set of tasks or subtasks. These microprocessors were controlled by a task driver which activated the appropriate microprocessor or microprocessors to execute the requested task. As in segmented microprogramming, this organization allowed separate implementation of each task executor, which could then be subdivided or upgraded independently. It is felt that the modularity of the design technique and its inherent testability would make segmented microprogramming a good design tool for new computer families, multiprocessor systems and highly reliable and testable special purpose computers.

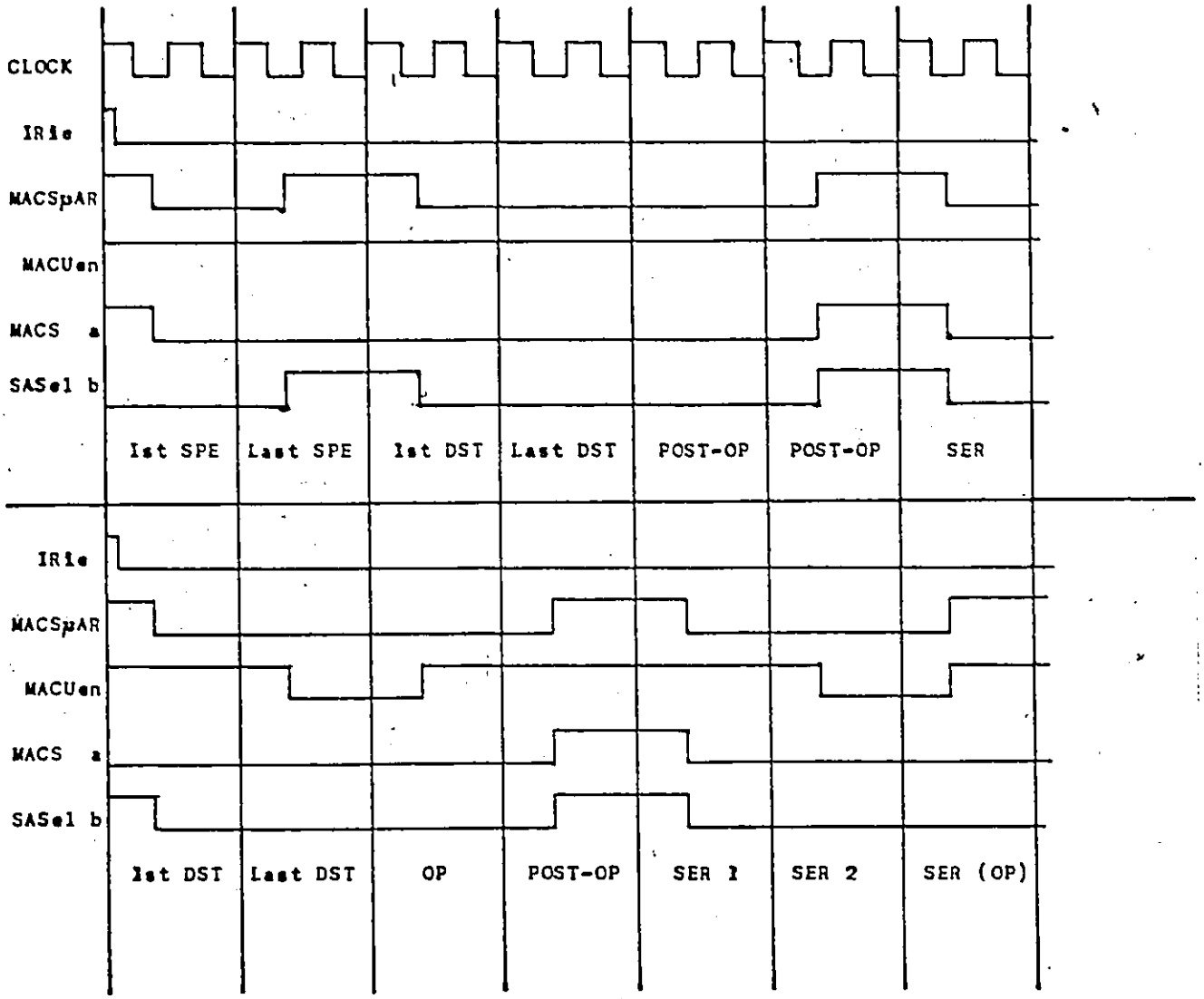
Appendix A

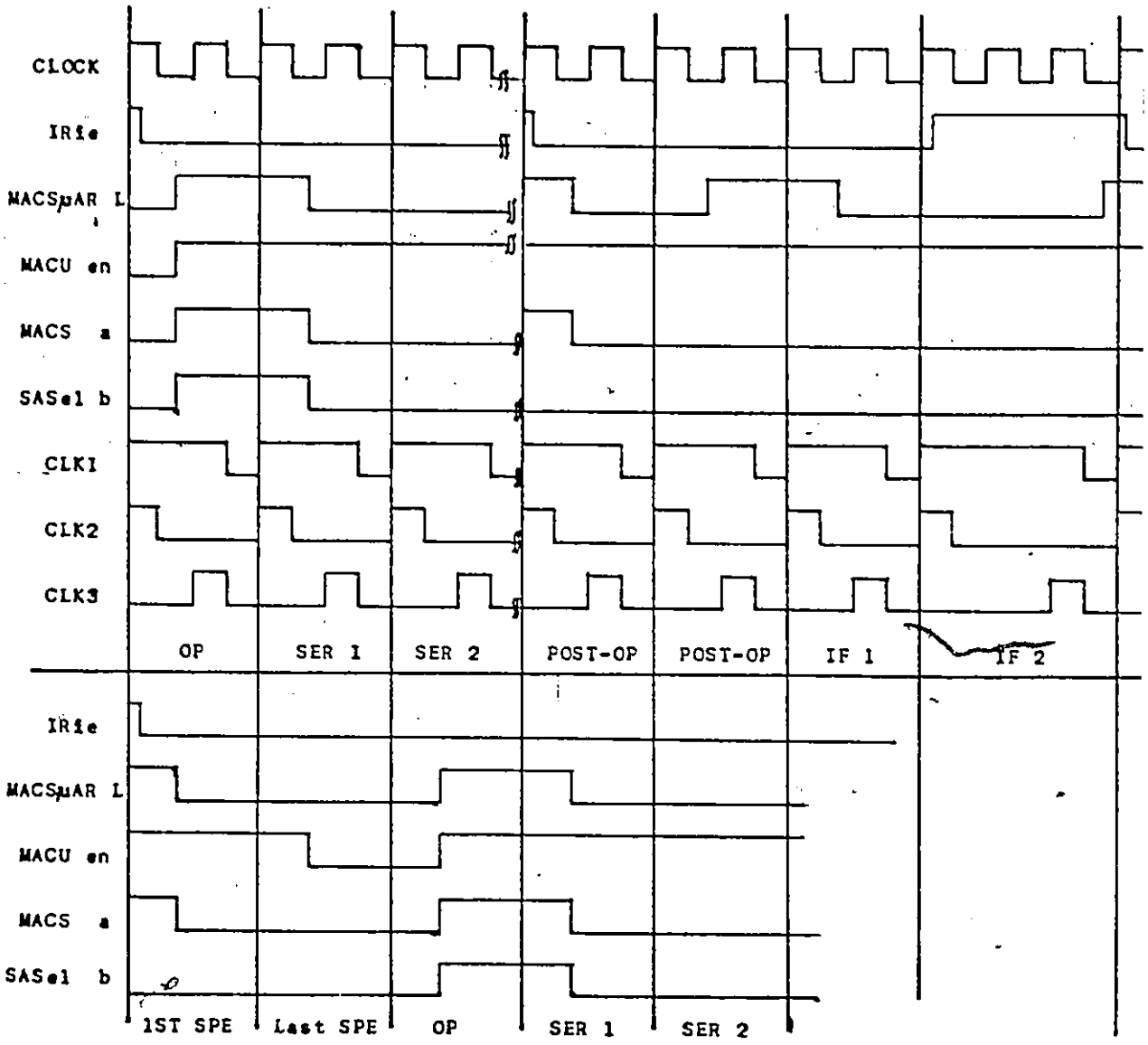
HANDSHAKE SEQUENCE TIMING DIAGRAMS

These timing diagrams show the relationship between the clock pulses and the control signals of the Handshake Sequencer. The 2901s, the microinstruction buffers and the PSW are clocked by CLK1. CLK2 triggers the Handshake Sequencer and CLK3 is used for the MACS microaddress register.




CLOCK
IRie
MACSμARL
MACUen
MACS a
SASel b
1st SPE
Last SPE
1st DST
Last DST
OP
POST-OP
1st SER
2nd SER
IRie
MACSμAR L
MACUen
MACS a
SASEL b
1st SPE
Last SPE
1st SRC
Last SRC
1st DST
Last DST
SER


CLOCK
IRie
MACSpAR
MACUen
MACS a
SASel b
1st SPE
Last SPE
1st DST
Last DST
POST-OP
POST-OP
SER
IRie
MACSpAR
MACUen
MACS a
SASel b
1st DST
Last DST
OP
POST-OP
SER 1
SER 2
SER (OP)


CLOCK
IRie
MACSμAR L
MACU en
MACS a
SASel b
CLK1
CLK2
CLK3
OP
SER 1
SER 2
POST-OP
POST-OP
IF 1
IF 2
IRie
MACSμAR L
MACU en
MACS a
SASel b
1ST SPE
Last SPE
OP
SER 1
SER 2

Appendix B

SMP-11 COST BREAKDOWN

#	Fcn of chip	Unit	Cost	
7	8:1 mux	PSW	5.04	
3	4:1 mux	PSW	2.55	
2	2:1 mux	PSW	1.70	
2	Q2I XOR	PSW	.58	
1	Hex inv	PSW	.17	
1	T3I Nand	PSW	.14	
1	8-bit clocked D-FF	PSW	1.15	
				subtotal: $11.33
2	FPLA 16 input 8 output	IR Decoder	54.70	
2	32x8 Prom	IR Decoder	3.00	
1	1 of 16 decoder	IR Decoder	.65	
1	T3I Nor	IR decoder	.24	
1	Q2I And	IR Decoder	.17	
1	Q2I Nand	IR decoder	.14	
1	Q 2:1 mux	IR Decoder	.85	
				subtotal: $59.75
3	4:1 mux	HSCU	2.55	
2	2:1 mux	HSCU	1.70	
2	8-bit clocked D-FF	HSCU	7.00	
1	4-bit adder	HSCU	.64	
1	32x8 Prom	HSCU	1.50	
1	Q2I XOR	HSCU	.29	
				subtotal: $13.68
19	4:1 mux	Data Path	16.15	
8	8-bit clocked D-FF	Data Path	9.20	
7	2:1 mux	Data Path	5.95	
4	2901 bit slice CPU	Data Path	71.40	
1	2902 CLA	Data Path	3.37	
1	32x8 Prom	Data Path	1.50	
				subtotal:$107.57
8	32x8 Prom	Control Units	12.00	
7	8-bit clocked D-FF	Control Units	8.05	
3	2:1 mux	Control Units	2.55	
3	4:1 mux	Control Units	2.55	
2	4-bit adder	Control Units	1.28	
				subtotal: $26.43
6	D-FF	Bus Interface Logic	.45	
11	UNIBUS trans.	Bus Interface Logic	25.00	
4	Monostable	Bus Interface logic	1.40	
5	UNIBUS receiver	Bus Interface Logic	10.00	
2	Q2I Nand	Bus Interface Logic	.28	
2	Q2I And	Bus Interface Logic	.34	
2	Q2I Nor	Bus Interface Logic	.28	
2	Hex Inverter	Bus Interface Logic	.34	
1	2:4 decoder	Bus Interface Logic	.45	
1	8:3 encoder	Bus Interface Logic	1.15	
1	4 bit comparator	Bus Interface Logic	.89	
1	4 bit clocked FF	Bus Interface Logic	1.09	
1	Hex Schmitt trigger	Bus Interface Logic	.40	
1	4 input Nand	Bus Interface Logic	.14	
				subtotal: $43.20
1	555 timer	Clock	.27	
1	Schottky JK FF	Clock	.59	
1	Q2I Nand	Clock	.14	
1	Hex Inv	Clock	.17	
1	Q2I And	Clock	.17	
1	T3I And	Clock	.24	
				subtotal: $ 1.58
			Total cost :	$267.98 (US)
				approx $325.00 Can

REFERENCES

1. PDP-11 04/34/45/55 Processor Handbook Digital Equipment Corporation, 1976

2. PDP-11 Peripherals Handbook Digital Equipment Corporation, 1976

3. Logic Handbook Digital Equipment Corporation, 1976

4. Advanced Micro Devices Microprogramming Handbook, 1977

5. Signetics Data Book Signetics Inc., 1976

6. Agerwala, Tilak Microprogram Optimization : A Survey IEEE Transactions on Computers, Vol C-25, No. 10, Oct. 1976

7. Armstrong, C.V.W. MultiMicroprocessor Architectures and the use of Multi-Level Encoding in Microinstruction Formats Proc 15th IEEE Computer Society International Conference, COMPCON Fall '77, Sept. 6-9 1977, pp. 144-147

8. Bell, C.G., Mudge, J.C., McNamara, J.E. Computer Engineering A DEC View of Hardware Systems Design Digital Equipment Corporation, 1978

9. Bell, C.G. and Newell, A. Computer Structures and Readings (Ch. 28) Microprogramming and the design of the control circuits in an electronic digital computer M.V. Wilkes, J.B. Stringer

10. Higbie, L.C. Overlapped Operation with Microprogramming IEEE Transactions on Computers, No. 3, Mar. 1978, vol C-27

11. Husson, S.S. Microprogramming Principles and Practice Prentice Hall, 1970

12. Krieger, M. Microprocessors and Microprocessor Based Systems Unpublished Manuscript

13. O'Loughlin, J.F. Microprogramming a Fixed Architecture Machine Microprogramming and Systems Architecture, Infotech State of the Art Report 23, Maidenhead, 1975

14. Russell, R.D. The PDP-11 : A Case Study of how NOT to design Condition Codes ACM-SIGARCH Newsletter, Vol 6 No. 7 Apr. 1978

15. McWilliams, T.M., Fuller, S.H. and Sherwood, W.H. Using LSI processor bit slices to build a PDP-11 -A case study in microcomputer design AFIPS Conference Proceedings, Vol 46, 1977 National Computer Conference

16. Rideout, V.L. One-Device Cells for Dynamic Random-Access Memories : A Tutorial IEEE Transactions on Electrom Devices Vol ED-26 No.6 June 1979

17. Krieger, M. Task driven multi-microprocessor system Proceedings of the First Canadian Workshop on the Design and Development of Computer Systems DDCSI May 23-24 1979

18. Wolfe, C.F. Bit-Slice processors come to mainframe design Electronics, Vol 53, No.5 Feb 28 1980

19. Tomasulo, R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units IBM Journal of Research and Development Vol 11, No.1 Jan 1967

Microprogramming Handbook, 2nd edition Microdata Corporation April 1972

A Microprogrammed 16-bit Computer Advanced Micro Devices

M2900 Bipolar (TTL) Processor Family Advanced Micro Devices

N. A. Alexandriis Bit-Sliced Microprocessor Architecture IEEE Computer June 1978

Ramamoorthy, C.V., Chang, L.C. System Modeling and Testing Procedures for Microdiagnostics IEEE Transactions on Computers Nov 1972 Vol C-21 No. 11

Rauscher, T.G., Adams, P.N. Microprogramming : A Tutorial and Survey of Recent Developments IEEE Transactions on Computers Jan 1980 Vol C-29 No. 1

Chiang, A.C.L., McCaskill, R. Two New Approaches Simplify Testing of Microprocessors Electronics Jan 22, 1976 Vol 49 No. 2

Davies, P.M. Readings in Microprogramming IBM System Journal Vol 11 No. 1 1972

Gordon, P. Stallard, S. Microprogrammed CPU Architecture Offers User-Alterable Minicomputer Performance Computer Design June 1978

Salisbury, A.B. Microprogrammable Computer Architectures Elsevier 1978

Podraza, G.V., Gregg, R.S.Jr, Slager, J.R. Efficient MSI Partitioning for a Digital Computer IEEE Transactions on Computers Vol C-19 No. 11 Nov 1970

Rosenfeld, J.L., and Villani, R.D. Micromultiprocessing : An Approach to Multiprocessing at the Level of Very Small Tasks IEEE Transactions on Computers Vol C-22 No. 2 Feb 1973

Chang, H.Y-P., Heimbinger, G.W., Senese, D.J., Smith, T.L. Maintenance Techniques of a Microprogrammed Self-Checking Control Complex of an Electronic Switching System

McCaskill, R. Wring out 4-bit microprocessor slices Electronic Design 10 May 10, 1977