LILITH COMPUTER

Hardware Manual

Modula Computer Systems
950N University Ave
Provo, Utah
84604

Foreword to Lilith Owner Whose Machine Has Just Died

This will happen from time to time. The important thing is not to overreact to the situation and possibly create greater damage in a misguided effort to repair the problem.

Most of the time, the cause of a non-functioning Lilith can be attributed to a transient inocuous problem, i.e. a disconnected cable or an improper switch setting. Such problems are easily corrected if recognized. It is therefore worthwhile to proceed cautiously to determine just how severe is the cause of the malfunction. Exploratory surgery always has the potential of creating more problems than it cures.

So, before tearing a non-functioning Lilith apart, it would be well to first check the following six items:

1. Is the boot file "clobbered"? Verify this by attempting to boot with another disk or by using the alternate boot file. If your boot file has neen clobbered by an errant program, restore it by the following commands:

   ```
   *copy
   from> PC.BootFile.Back
   to> PC.Bootfile
   ```

   The program "memfilexfer" could also be used to restore the file.

2. Is the keyboard cable disconnected or damaged?

3. Are the display connectors, power and logic, either disconnected or damaged?

4. Is the power supply fuse blown? Is the display fuse blown?

5. Has a surge of the line voltagea triggered the power supply overvoltage circuitry? To find out, turn power off for five minutes and try again.

6. Have the circuit cards been vibrated out of their connectors, possibly by moving the equipment or from vibration? Reseat cards to find out.

If none of the above investigations rectify the problem of the malfunctioning Lilith, then the maintenance engineer should begin diagnostic procedures according to Section 7 of this manual.

Foreword to Manual

This manual will attempt to cover the Lilith hardware at three levels:

1. At the highest level, it will cover the theoretical considerations involved in design of the Lilith for the benefit of those wishing to incorporate ideas from the Lilith project in other computers. This information will be contained for the most part in Section 5.

2. At the second level, the functioning of the circuits in each subsystem will be detailed in order to aid engineers and technicians in the debug of the logic of the computer. The circuit descriptions of Section 6, together with the schematics and wirelists in the appendices, will be most useful for such problems.

3. At the lowest level, a set of diagnostic procedures will be enumerated to enable an apprentice technician or even a *software type* to determine whether or not the machine is functioning, and possibly even which, if any, of the circuit boards are defective. The diagnostic procedures given in Section 7 should be understandable even to the uninitiated.

Those wishing to connect other input/output devices to the system will find Appendix A of interest.

If the reader encounters any errors within this manual, he is invited to notify R. Ohran at Modula Computer Systems, 950 N. University Ave, Provo, Utah 84604.

Table of Contents

Kit Documentation appendices

1 Introduction

One of the tenor themes of the Lilith project was the idea that conventional computer systems have been designed to be inefficient and inadequate because of the limited perspective used in their design phase, especially because of the limited consideration given the importance of high level programming languages in the final determination of the architecture of the machine. In the Lilith project, it was intended that the requirements of the Modula-2 programming language would determine the structure of the machine to be constructed. In fact, as it developed, the stack architecture dictated by the needs of the language proved inappropriate for direct implementation with the available components. Because of this, it was necessary to construct a computing engine of a more conventional nature and to microprogram it to emulate a more desirable stack architecture. This stack machine will be referred to as the Lilith "virtual machine." The conventional register based computing engine which supports the virtual machine will be referred to as the "real machine."

The main intent of this manual is to describe the Lilith hardware. Part of this description will review some of the basic aspects of the virtual machine. This will help the reader to have the proper perspective from which to contemplate the design decisions which were made. We will leave an in depth discussion of the relationship between Modula-2 and the virtual machine to the manual on software. In this manual we will mainly present the characteristics of the virtual machine without reference to the software and attempt to explain them as they relate to the actual hardware.

As the reader investigates this machine, he will discover that the Lilith computer system yields a remarkable degree of performance and efficiency for the investment of resources. As the hardware is described, an attempt will be made to explain the factors which caused this. Particular emphasis will be given to the unusual features of the hardware that were designed into the machine to accommodate the special nature of the virtual machine. However, a good deal of the success of the Lilith is based upon the features normally included in other computers which were left out in the Lilith. The reader should prepare himself to understand the success of this approach not by looking just for the features that were added to increase the power of the machine, but rather to consider as well the features which were left out. As an example, it is a worthwhile exercise to consider the many features of a machine such as the PDP-11 which are not found in the Lilith. Then, one should evaluate whether or not the Lilith has suffered from their absence.

2 Overview of the Hardware

In this section, an overview of the major Lilith components will be given with the intention of providing a perspective view of the entire system and the various interactions between its subsystems. Each subsystem will be investigated in greater depth in subsequent sections.

The basic Lilith system consists of a processor, memory, high resolution graphics display, 15-Megabyte Winchester disk with single-sided double-density floppy disk back-up, and miscellaneous I/O devices including: mouse pointing device, keyboard, real time clock, and serial line receiver/transmitter interface (UART). A block diagram of the system is given in Figure 1.1.

2.1 Processor

The processor is a microprogrammed implementation of a stack machine architecture. The stack architecture includes two stack memories, registers for global and local variable base addressing, and an arithmetic logic unit featuring a full barrel shifter. One of the stack memories is only sixteen levels deep but constructed from high speed bipolar memory circuitry. The other stack memory uses main storage with an addressing scheme based on registers of the processor. The smaller stack is called the "evaluation stack" to distinguish it from the typically larger stack in main memory which is simply called "the stack."

The heart of the processor is a 16-bit arithmetic unit based on the 2901 bit slice integrated circuits. The 2901 based arithmetic unit has also been augmented with a barrel shifter and an external 16 level stack. Information flows into and out of the arithmetic unit through a main processor bus (called simply BUS) which is 16 bits wide and connected to most subsystems of the Lilith.

There are two registers, the code frame register and the program counter register, which provide the mechanism for fetching instructions from memory. The program counter functions as an offset from the code frame register value. The stack machine instructions are called M-codes and may be 8, 16 or 24 bits long. M-code instructions are stored in memory in segments referred to as modules. The starting address for each module is kept in memory in a table of pointers known as the module frame table. The module frame table holds as many as 256 entries. Any reference to a procedure or a global data variable of a module implies the use of its entry in the module frame table; however, references to the module frame table are minimized by saving the entry points for a module in processor registers.

Execution of M-codes is handled interpretively by a microcontroller executing microinstructions from a control store. Typically, three or four microinstructions are required to interpret each M-code. The basic cycle time of the microinstructions is 150 nanoseconds. The microinstructions are 40 bits wide and a total of 4096 microinstructions are addressable. Currently, only 2048 instructions are used and they are stored in bipolar programmable ROMs.

The M-code instructions are fetched from main memory by a special instruction fetch unit. This unit fetches the M-codes and updates the program counter values. The IFU has a cache memory capable of holding 8 bytes of M-code instructions, allowing it to fetch M-code instructions in units of eight bytes at a time.

2.2 Display

The display controller refreshes a high resolution 15 inch cathode ray tube from the main memory of the Lilith. The display image is a so-called "bitmap display." The refresh rate is 25 frames per second, and the structure of the image is 768 bits across and 594 vertical lines high. The total memory used to refresh the image is 28,816 sixteen-bit words. The size of the image on the display can be reduced both in height and width. Horizontally, it can be reduced in increments of 64 bits from 768 to zero. Vertically, it can be reduced to any number of lines. All images displayed on the screen are constructed from "bitmap" images written into the display area of memory. The processor has several microprogrammed instructions which perform image-creation operations with greater efficiency that normal programming.

2.3 Memory

The display, the processor, and the instruction fetch unit all compete for cycles from the multiport main memory. The allocation of memory cycles is handled by an arbitration mechanism using a priority arbitration circuit. The display has the highest priority. For reading, the memory is organized basically as 32768 addresses of 64-bit memory words. For writing it is organized into 131,072 sixteen-bit words. A special multiplexing circuit allows the processor and I/O devices to read memory in 16-bit words, but the instruction fetch unit and the display exploit the full 64-bit word length for their memory cycles. There is an additional memory port allocated to a section of circuitry which periodically requests "dummy" memory read cycles. The circuitry requests these cycles to satisfy the requirement of the dynamic memory to refresh each of the 128 row addresses once each two milliseconds.

2.4 Secondary Storage

The secondary storage device of the Lilith is a 15-Megabyte Winchester disk with single-sided double-density floppy disk back-up. The disk controller transfers a single sector of data to an on-board buffer of 128 sixteen-bit words. Data transfers from secondary storage to main memory are not handled by direct memory access but are handled by the processor, using special microcoded routines which operate at full memory speed. Data transfers between the Winchester and FDU are handled serially via a UART.

2.5 Mouse Pointing Device and Keyboard

One of the significiant features of the Lilith is its well designed human interface facilitated by its high resolution display (already described), its "mouse" pointing device, and its keyboard. The mouse rests on the table and transmits relative surface movements to the processor. The processor uses this information to position a pointer on the screen. Through pointing and depressing the three buttons on the mouse, the user can give commands to the processor which are typically program control operations. The effectiveness of this interface can hardly be described; it must be experienced.

2.6 RS-232c Interface

For communication with other standard peripheral devices and computers, a RS-232-compatible serial interface is provided. The transmission rate is selectable from 75 to 9600 baud. The voltage levels for mark and space are standard.

2.7 Extra I/O Slots

For expansion of the machine's memory and connection of peripheral devices, the machine has additional uncommitted circuit board slots. The devices which are connected may be designed for programmed I/O or for direct memory access techniques using one of the four uncommitted memory ports.

3 The LILITH Virtual Machine

In the search for a suitable architecture to support programming in the high level Modula-2 programming language, some basic ideas were formulated as characteristics of the ideal computing engine. Because limited possibilities existed for the purpose of creating such hardware, it was necessary to define the nature of this ideal machine and then to emulate it as well as possible with the available components. This ideal machine is therefore referred to as a "virtual machine." The word "virtual" in Webster is given the following definition:

virtual--in essence or in effect, but not in fact

This is exactly what we have: a machine whose essence and effect is real, but which in fact does not exist, being an illusion created by the real hardware.

Some would argue that the use of the word "virtual" in this sense is unwarranted because of the real behaviour exhibited by the machine. One could simply say that the microprogram and the underlying hardware are only a different form of construction. Perhaps this is so. We are not particularly fond of the term "virtual," but it does serve a function. It is a valid designation which allows us to distinguish between two computing engines which are present in the Lilith: the logically described stack architecture machine, and the actual register machine which emulates the stack architecture.

3.1 Salient Characteristics of the Lilith Virtual Machine

The Lilith virtual machine exhibits characteristics reflecting the dominating influence of the high-level language software considerations. These characteristics cause the Lilith to have an appearance significantly different from the conventional "von Neuman" computers. Some of these characteristics are presented here for the purpose of giving the reader a perspective comparison between the Lilith and more conventional computers.

3.1.1 Use of Stacks

The Lilith virtual machine is what one commonly refers to as a "stack machine." It is so named because a stack, otherwise known as last-in-first-out memory, is the the prominent structure used in the processing of information. There are other elements of the machine equally important in the processing of information such as registers, memories, and arithmetic units, but the machine derives its name from *stack*, because this is the element which distinguishes it most from other computers.

The Use of a Stack for Data Operations

Conventional machines transfer data from memory locations to registers in the processor where they may be operated upon by the arithmetic/logic unit of the machine. The Lilith has no registers for such a purpose. Instead, a stack, 16 levels deep, is used to receive the contents of memory locations. Arithmetic instructions in the Lilith have no address fields associated with them. The top two levels of the stack are always implicitly referenced as operands in binary operations, and the top level alone is used for unary operations. When a result is generated by the operation, it is returned to the top level of the stack. There is no other mechanism in the machine to perform arithmetic and logic operations on variables stored in memory. The stack is also used to pass parameters in procedure calls. And it may be used in address calculations as may be necessary in operations such as array indexing.

Use of Main Memory as a Stack for Procedure Activations and Local Variables

Main memory is usable in the conventional manner of address register and data cycle. In the Lilith, it is also possible to use main memory as a stack. There are registers in the processor that hold addresses which are incremented and decremented as necessary to give the appearance of a stack. This feature of "stack machines" has been incorporated into most of the new microprocessor designs. However, in the Lilith, the usage of main memory as a stack is much more sophisticated. With each procedure call, an elaborate activation record is created on the stack which does more than simply save the return address onto the stack. It also saves the state of several registers used as base addresses for the variables of the previous procedure, and builds a chain of pointers which allows the new procedure to access all variables at higher levels according to the scope rules of Modula-2. Storage for the local variables and parameters of the procedure are also allocated from the stack just above the procedure activation record.

3.1.2 Dynamically Allocated Storage in the Heap

The higher extremity of the unused memory space allocated to the stack in memory is available for allocation as the *heap.* When the program wishes to allocate storage dynamically during execution and assign this storage to a pointer variable, it obtains this storage from the top end of the stack. Naturally, the stack has fewer memory locations available for procedure activation records when a portion of its space is taken away and assigned to the heap. Management of the heap is a program function in the current Lilith software. There is no system "garbage collector" to retrieve used and no longer needed areas of the heap.

3.1.3 Variable Addressing via Base Registers

The Lilith architecture enables the compiler to produce smaller compiled programs by providing a facility for addressing data variables using base address registers. Instructions capitalize on this feature by using offset values requiring only four or eight bits which, together with a base register, give the complete address of a variable. Since the offsets require fewer bits for their representation in the instruction, this results in significant savings in memory space used for program storage. This feature is enhanced by the fact that the base register contents are automatically updated when the processor moves into another procedure, or another module, or another coroutine.

The technique of addressing by offsets is also applied using the top of the stack as a base address. This provides invaluable compactness for referencing elements of structured variables which have been dynamically allocated and which use a pointer variable as the means of addressing.

3.1.4 Coroutines

Another characteristic of the Lilith virtual machine is its support for the concept of a *coroutine.* This concept is not supported in hardware by any machine of which the author is aware. Some machines provide coroutine usage in their handling of interrupts, but only to a limited degree. They fail to provide coroutine capablility for general use in a program. In the Lilith, a single instruction allows the processor to completely suspend one coroutine and activate another. Since this operation is identical to the manner in which interrupts are handled in the Lilith, the programmer finds an especially pleasing unification of the concepts of processes and interrupt handlers. Familiarity with this feature of the Lilith will provide the Lilith user with new dimensions to use in the solution of programming problems.

3.2 An Overview of the Structure of the Lilith Virtual Machine

Just as the Lilith *virtual machine* is considered to be a *virtual* construction, each of the elements which are part of it should also be regarded as *virtual* since they may in fact be simulated by the microprogrammed interpreter or may in reality be the composite effect from several sections of *real* hardware.

BUS is a 16-bit data path which connects all elements of the machine together.

The *main memory* is a standard random access memory operated in read and write cycles for addresses to 130,nnn sixteen-bit words.

The *global register (G)* and *local register (L)* provide the base addresses for address calculations needed for access to program data variables from main memory.

The *stack top register (S)* and the *heap register(H)* provided the address values used for creating stack behavior from sections of main memory.

The *code frame register (F)* and the *program counter offset register (PC)* are used to generate a byte address for each virtual machine instruction. These registers are part of the *instruction fetch unit* which utilizes the 64-bit read feature of the main memory to reduce memory cycle requirements for instruction fetches.

The *process register (P)* holds the pointer to the process descriptor area for the process in execution.

The *mask register (M)* holds the interrupt mask which determines which interrupts may be recognized.

The *evaluation stack* is a true last-in-first-out stack structure composed of sixteen levels of sixteen-bit words. Data paths connect the *evaluation stack* to the *arithmetic/logic unit* and the *BUS*, which leads to memory and peripheral devices.

The *arithmetic/logic unit* operates on data values stored in the *evaluation stack*. It performs the various arithmetic operations for various data types: *cardinal*, *integer*, and *real*. It also performs relational operations comparing magnitudes for these same types but yielding boolean results. It can perform the standard logic operations, and it can even perform directly a number of set operations. For address calculations the *arithmetic/logic unit* may reference the global and local variable base address register as well as *BUS* for access to constants and offsets.

A block diagram showing the interconnections of *virtual* components of the *virtual* machine is given in Figure 3.1.

There are additional elements of the Lilith virtual machine which need to be introduced at this time. These elements are "softer" in nature, being more a matter of program organization within memory than of structure.

> The *module frame table* is a table of pointers found in the memory of the Lilith. Each program module has a base address entered for it into the *module frame table* so that it may be addressed by other modules.
>
> The *data frame* is an area of memory assigned to a module holding its global variables and string constants. The address of the *data frame* is the value entered into the *module frame table.*
>
> The *code frame* is an area of memory assigned to a module for its compiled instructions. The address of the *code frame* is the first word of the *data frame.*
>
> The *procedure table* is a table of values giving the entry point address for each procedure of a module. The entry point address is an *offset* amount from the start of the *code frame.*
>
> A *coroutine* is a concept used to describe the situation when the execution of a program is divided into more or less independent parts. A coroutine is like a procedure in the sense that it can be activated as a procedure is activated and in the sense that it "owns" data variables which it may use. But it is unlike a procedure in the sense that it does not necessarily own unique sections of compiled instructions, nor is it entered at the same point each time it is activated. *Coroutines* do not execute concurrently; only one *coroutine* at a time can be in execution; the others must all be *suspended.*
>
> A *process descriptor* is a data structure in the memory which can hold the state of a coroutine during the times when it is suspended.
>
> The *interrupt vectors* are pointers to process descriptors which are assigned fixed memory locations. When an interrupt occurs, the Lilith can create a coroutine transfer by going to the fixed addresses to find process descriptors where it can save the state of the coroutine in execution and retrieve the state of the coroutine which needs to be activated to handle the interrupt.

The overall relationship of the elements discussed in this section is shown in Figure 3.2. The relationship of these elements will be discussed in greater detail in following sections.

3.3 The Lilith Virtual Machine in Operation

As was briefly discussed in the previous section, a computing task performed by the Lilith virtual machine is called a *coroutine*, and more than one coroutine can be operational in the machine at any given time; although only one coroutine is actually in execution (called *active*) at any give time, while the others are held in a *suspended* state. Changes which cause the active coroutine to be suspended and replaced by another suspended coroutine are called *coroutine transfers*. *Coroutine transfers* may be caused either by programmed instructions or by hardware interrupts.

Each *coroutine* "owns" a number of elements which make it unique and distinguish it from other coroutines active within the system: First, each coroutine is assigned a workspace in main memory, which it uses in part as a stack and in part as a heap. When the coroutine is created, the registers associated with the stack and heap are initialized appropriately to use the workspace. Figure 3.2 illustrates the address relationship of the S, L, and H registers to the workspace.

A second unique element of a coroutine is the *process descriptor*, which is a storage area capable of holding the entire state of the processor relative to the execution of the coroutine. When a coroutine is suspended, its state is saved into its *process descriptor*. Usually, the storage for the process descriptor is placed at the beginning of the workspace by the procedure which initializes the coroutine and sets it into operation. (Usually this is the procedure NEWPROCESS, except when the first coroutine is initialized by the microprogram in performing the "booting" procedure.) The register P of the Lilith virtual machine always contains a pointer to the process descriptor of the active coroutine.

A third unique element of each coroutine is the the program address value which determines where--from among the selection of program modules loaded in memory--the coroutine is to take its next instruction. It will be discussed later how this value is actually a combination of values from two of the machine's registers, F and PC. The program modules themselves are not of necessity uniquely assigned to any coroutine. They are simply resident in the machine and available to perform tasks for any coroutine which chooses to execute instructions from any of the procedures of a given module.

COMMENT: There may be some confusion about the *coroutine* aspect of the operation of the virtual machine. Programmers have a tendency to identify a "process" or a "task" with a section of program code. They speak of a "program running in a computer." In the Lilith, the concept of "running a program" should be understood to mean: "to cause the main coroutine in execution to begin executing instructions from the desired program module (or collection of modules) after it (they) has (have) been loaded into memory." The distinction may appear subtle, but it will help to clarify thinking when the processor is involved in an especially complex situation of coroutines and modules.

A fourth element unique to each coroutine is the local storage allocated for each procedure as the procedure is entered. This storage belongs to the coroutine which was active when the procedure was entered and is available for the use of that procedure only when that coroutine is active. If two coroutines enter and execute instructions from the same procedure, each will have its own separate copy of the local variable storage. This will not be the case for the global storage defined for each module and accessed by the procedures of the module. When such global variables are used by two coroutines executing from the same procedure, the storage referenced will be the same for both. This subject will be discussed further in the section describing module organization.

It may help the reader to clarify the conceptual relationship between coroutines and program modules as they exist in the Lilith if an analogy is presented: We may view each coroutine as a moving, dynamic entity to be compared with an automobile. A program module in this analogy would be comparable to a neighborhood of streets, buildings and passages. The collection of all program modules in the Lilith memory would be represented by an entire city. Just as cars move through the city constrained by the configuration of the streets and pass from one neighborhood to another, coroutines work their way through the structures of program modules performing functions and moving on to other modules. Just as several cars may travel in similar paths or even travel down the same streets, several coroutines may operate within the same program module sharing global data but each privately using its own local data.

NOTE: From the above analogy, I have come to the conclusion that the name "coroutine" is inappropriate to describe the concept which has been assigned this name. The term "coroutine" tends to connote a function similar to a subroutine, or in other words, a portion of a program--when, in fact, the important part of what we call a *coroutine* is actually the execution state of the computing task as contained in registers and stack workspace assigned to the task. A task does follow a pattern of execution established for it by the procedures which have been programmed, but in the Lilith as in many computers, these procedures are not uniquely owned by a single task. The confusion, I believe, comes from having incorrectly regarded the simple case of sequential programs as the actual "work-doers" rather than as patterns which channel the "execution state" into a desired performance. The problem with this conceptualization is that it breaks down when the programmer attempts to visualize multi-programming as an independent collection of programs performing separate functions when the programs themselves are allowed to use parts of each other. The programmer then begins to wonder where the coroutine is. In this manual, rather than fight the established pattern of nomenclature, I will continue to use the term "coroutine" but mean thereby a task-performing entity whose essence is uniquely captured in its process descriptor when it is in a suspended state or present in the registers of the machine when it is active, together with its work space--R. Ohran.

3.3.1 Instruction Fetch and Execution

Once it has a valid coroutine in an active state (there is always at least the initial coroutine set up by the microprogram when the "reset" button is pushed), the Lilith follows a repetitive pattern of fetching instructions and executing them. The basic cycle begins by fetching an M-code instruction from main memory. The address of an M-code is a so-called *byte address* because an instruction may consist of as little as 8 bits taken from either the right or left half of a normal 16-bit word of memory. Hence, a *byte address* requires one additional bit in the least significant position of the address to determine from which half of the word the address is taken. This *byte address* is computed from the sum of the contents of the code frame register (F register) and the program counter offset register (PC register). The F register value is shifted left 2 bits and the PC register value added to it; therefore, the resulting address has 18 bits. An 18-bit byte address allows addressing of 265K bytes of memory or 131K words. The so-computed *byte address* is passed to the memory address register over the *BUS* and the selected instruction byte is returned through the memory data register over the *BUS* to the control unit for decoding.

After the instruction byte has been fetched, the program counter register is incremented by one. If additional bytes are needed to complete the instruction, they will also be fetched in the above fashion with the possible exception that if the decoding of the first byte specifies that the second byte is to be fetched as a negative value, then the higher order eight bits of the 16-bit BUS are set to *ones.*

Once the instruction byte has been decoded, and additional instruction bytes have been fetched as need, the processor enters the execution wherein the specified operation is carried out according to the order code of the instruction.

3.3.2 The Lilith Virtual Machine Instruction Set

Eight bits constitute the basic atom from which instructions are composed. A complete Lilith instruction may use one, two, or three such groups of eight bits (bytes). (Refer to Figure 3.3.) The individual instructions are categorized by the value of their first byte. Subsequent bytes usually provide addressing or constant data values for use by the instruction. The exception to this rule is the "escape" code, which causes the second byte to be evaluated before the instruction type is determined. The instructions fall into 6 main categories:

1. Instructions which transfer data between main memory and a work area called the evaluation stack which is part of the arithmetic/logic unit

2. Instructions which manipulate data on the stack

3. Instructions which test the contents of data on the stack and modify, in some instances, the next selected instruction by storing new values in the F and PC registers

4. Instructions which implement transfer to procedures and in the process modify registers associated with the management of the free storage area known as the stack.

5. The coroutine transfer which suspends the execution of a process and activates another

6. Instructions which transfer data between the evaluation stack or main storage and input/output devices

Appendix B gives the complete instruction set listing. More will be said about some of the more important instructions in later sections.

3.4 The Organization of Programs in Memory

At any given instance, the Lilith will be found to have at least ten (or more likely 20) different modules in memory, which may be visited by the coroutines operating within the system. There is a master table of identifying addresses for each module which is known as the *module frame table.* The table begins at the absolute address of 20 (in hexadecimal) and has room for 96 entries. The position of a module's identifying address relative to the start of the table is important, for it is the value by which the module is selected within instructions which need to specify a module outside of the one they are in. This structure is not a matter of arbitrary program selection; it is fixed in the microprogram of the Lilith. The *module frame table* is shown in the upper left hand corner of Figure 3.2.

3.4.1 Modules

A *module* is a collection of procedures and data structures which typically have a commonality associated with the accomplishment of a given task or related set of tasks. It also has a structure which is a matter of definition in the microprogram which defines the instruction set. This structure consists of a global data frame and a code frame. Modules are loaded into memory in contiguous areas from a low address above certain reserved addresses to high addresses. A pointer to the start of a module is entered into the *module frame table.* All references to a module from external points use an index value into the *module frame table* to fetch the pointer to the module. This pointer points to the global data area where, in turn, the first word is another pointer which gives the F register value for the start of the code frame. At the start of the code frame are as many pointers as are necessary for all the procedures of the module. Figure 3.2 shows this relationship.

3.4.2 The Global Data Frame of a Module

As mentioned, the *module frame table* contains entries which point to the beginning of the memory allocated to a module. This pointer value does not point directly to the program segments of the module but to the memory area reserved to hold the variables declared globally with the module. The first three locations of this area are not, however, assigned to global data variables but are used for special functions. These functions are as follows:

```
Word 0:  This word will contain a value which, when placed in the
         F register, will be used by the instruction fetch unit to
         fetch instructions for execution.  This value will not be
         the actual starting address for the code frame because the
         F register has a 1-bit offset from the normal address
         so it will contain the address divided by two.
```

Word 1:	This address will initially be zero and will be set to a non-zero value with the first call of the main procedure of the module. This word is used as a flag to inhibit multiple execution of the main procedure.
Word 2:	String constants needed by the program will be appended to the memory assigned to global variables and referenced indirectly and post-indexed through the pointer found in this location.

After the first three words in the data frame, the next storage locations are allocated for use as global data variables.

Besides being addressable from all procedures of a module, global data variables have three important characteristics which affect their function in a program: First, unlike local variables, the storage allocation and "life" of the global data variable is permanent, as long as the program module defining the global variables is resident in memory. In Modula-2 global variables, this life of variables can even extend beyond the completion of the program which invokes the loading of the module. Second, global variables are the only variables which may be exported from one module to another. Third, global variables are equally accessible by all coroutines which execute with the module which has defined them or imported them. These features, universal scope and permanence beyond the time boundaries of procedure calls, extend the use of global variables beyond the normal functions seen in PASCAL programs.

Because of these characteristics, global variables assume a special role in Modula-2 programs which supercedes their use in PASCAL programs. This role is related to three different aspects of communications between modules, programs, and coroutines:

First , because of their ability to be exported, global variables play an important role communicating data between separately compiled modules which import and export them. This becomes an important consideration in the management of complex programming problems as the selection of which global variables to export and import affects significantly the efficacy of the design of the programs using them.

The second communication aspect relates to the lifetime of the global data variables which are associated with the residency of program modules in memory. In the Lilith, it is not necessary to purge a module from memory simply because the task which referenced this module and caused it to be loaded has terminated. It is possible to reuse a loaded module together with all accumlated information in its global variables by linking the module into a newly assembled selection of program modules to be used by a new coroutine. The new coroutine thereby benefits from the accumulated information which the previous task computed. In theory, it would be possible to even run the same program repeatedly with improved performance each time because of the accumulation of information stored in the global variables of a reused module.

The last communication function of global variables is associated with the nature of modules which are used by more than one coroutine within their overlapping lifetimes. Because global variables are linked with a program and not with a coroutine (as is the case with local variables which are allocated privately and separately to the stack of each coroutine), the global variables become a commonly shared resource to the coroutines which reference the same program module. This can be an important means of communication, but it can also become a source of failure if consideration is not given to the well known problem of synchronization and deadlock.

3.4.3 The Code Frame

As a module is compiled, each procedure encountered in the process is assigned a number in sequence by which it is referenced from other modules and from other procedures within the same module. An address for the entry point of each procedure relative to the address held in the F register is placed in a table at the start of the code frame. The sequence number assigned to the procedure is the index value through which the offset value is found. Therefore, the starting address for the code frame is also the location where the address for procedure 0 of the module is found. The main program section of a module is always *procedure 0*. There will be as many words of procedure addresses as there are procedures within the module.

The code frame is currently assigned by the operating system to the first even word address just above the global data and the string constants. However, there is no reason it could not be assigned to another memory area. If the amount of storage of the Lilith available for data variables becomes a limiting factor, the code frames of each module can still be relocated out of the prime addressing space.

3.4.4 Procedure 0 of a Code Frame

Procedure 0 of the code frame is always the main program procedure of a module. If a given program module references any external subordinate modules, their main procedures will be called for execution before the execution control is given to the main procedure of this program module. In this instance, the main program procedures of the subordinate modules may serve as initialization routines. This technique applies to any level of nested module references. However, for the case where a module is referenced by more than one other module, the main procedure of that module will still only be executed a single time because of special program instructions which detect this condition based on the state of a flag stored in the second word of the data frame as described in a preceding section.

3.4.5 Addressing of Procedures

In contrast to conventional methods of making a procedure call according to the starting memory address for the procedure code, a procedure call in the Lilith is referenced only by a simple number representing the sequential ordering of procedures within the module, i.e. 1, 2, 3..N. This number provides an index value from the start of the code frame to a single memory location which then provides the actual starting address of the procedure as an offset from the code frame. For internal procedure calls within the same module, only the relative procedure number is required. For procedures external to the module where the call is made, the procedure must be called through the use of its 8-bit module number and its 8-bit procedure number.

The addressing of procedures and modules by an index value into a table provides two advantages which may not be readily apparent. First, since all procedure entry addresses are located by index value rather than address, relocation of a code frame to another address is managed quite easily, requiring nothing more than the change of the pointer in the data frame. Second, procedure variables are nothing more than integer table indexes. This also simplifies relocation of code problems and renders the problem of loading and execution of new modules relatively simple.

3.5 The Stack, Procedure Activation Records, and Local Variables (Figure 3.4)

Initially, all available memory not in use for modules is assigned to the stack of the first coroutine. Three registers, S, L, and H, control the use of this memory. The S-register (S meaning "stack pointer") points to the lowest unused storage location or top of the stack. The H-register points to the highest free storage location for purposes of overflow checking, and the L-register is used as a register holding the base address of the procedure activation record and the local variables of the active procedure. From this address, offsets are computed to memory locations within the stack. As the memory of the stack is allocated for procedure activation records and local variable storage with each procedure call, a comparison is made to determine whether or not the S-register has been incremented beyond the bounds allocated for use by the stack. Each procedure call causes the S-register to be incremented by four plus the number of local variables defined for that procedure. At any given instant, the top memory locations of the stack will hold: a) the local variables used by the procedure currently in execution, and b) the activation record necessary to return the execution control back to the program segment which called the procedure currently in execution.

3.5.1 The Procedure Activation Record

Each time a procedure is called, it is necessary to place enough information on the stack so that the execution of the program making the procedure call can be resumed correctly, and so that the activated procedure may have access to information at lower nesting levels. Four words of information must be stored for this purpose. These words have the following use:

```
Word 0:  If calling a procedure in an external module, then this word
         will contain the pointer value from the module pointer table for
         the module making the call.

         If calling a procedure within the same module, then this word
         will contain a pointer value establishing a "static link" to
         variables on the stack which are considered at a lower nesting
         level, or global, to the called procedure.


Word 1:  This word will contain a pointer to the start of the next lower
         activation record.  This is the so-called "dynamic link" which
         is necessary to deallocate storage from the stack when the
         current procedure is terminated.


Word 2:  This word will contain the PC value for the next M-code instruction
         following the instruction which called this procedure.  If this
         procedure was called from within the same module, then there will
```

be a '0' in the most significant bit of this word. If the call was made from an external module, then the most significant bit of this word will be a '1', and the procedure return instruction will know to restore the G-register from word 0 of this record, as well as the F-register from the first word of the global data area pointed to by the G-register.

Word 3: This word will be used to save the state of the interrupt mask in the event that the module entered wishes to change it.

3.5.2 The L-register

Each time a procedure is called, the stack is expanded twice. First, as the procedure call is made, a procedure activation record is constructed on the stack, occupying four words of the stack. Then as the procedure is entered, more storage is allocated to accommodate local variables which are declared within the procedures. Parameters of the procedure call are also allocated in this manner. The newly invoked procedure will then begin to process information into and out of these memory locations. To facilitate this more expediently, the L-register is assigned an address pointing to the base of the procedure activation record. The invoked procedure may then address these memory locations efficiently as small offsets from the L-register base. Instructions are available which use either four or eight bits for the offset information. The efficiency of the program object code is improved greatly through this mechanism.

The use of the L-register in this manner implies an additional overhead for the operations of procedure call and return from procedure. Each of these operations must alter the contents of the L-register so that subsequent fetches of local variables will be correctly performed. In the case of the procedure call, it is necessary to save the value of the L-register into the dynamic link (word 1) of the new activation record so that it may be restored later. Then the base address for the new activation record must be loaded. It is held in the S-register at the time of the procedure call. The return from procedure instruction must restore that L-register value in use by the program segment which made the procedure call. It does this by loading the L-register from the dynamic link which was prepared by the procedure call instruction.

3.6 Addressing of Data Variables

The storage addressing schemes of the Lilith virtual machine directly reflect the needs of the Modula-2 language. In Modula-2, variables may be allocated in a multitude of ways, but finally they will be referenced through program instructions in one of four separate ways:

1. Offset from the G-register:

   The G-register points to a data area reserved for use with the module which is currently being used by the active coroutine. Data variables which are considered global to the entire module are allocated from memory in this area. There are special instructions giving access to these variables which minimize the number of bytes necessary to make the reference. (LG3..LG15, LGW, SG3..SG15, SGW, etc.)

2. Offset from the L-register:

   In a manner analogous to the use of the G-register, there are instructions which operate on data variables which have been allocated locally by a procedure to the stack of the active coroutine. This includes the variables which are declared as formal parameters of the procedure. The method of use again involves the use of special instructions which reduce the number of bytes necessary to make the reference. (LG4..LG15, LGW, SG4..SG15, SGW, etc.)

3. Offset from the G-register area of an External Module:

   This method of addressing gives a module access to global variables in other modules. The mechanism is similar to references to its own global variables but requires an additional memory cycle to fetch the external module's data frame table entry.

4. Offset from a Calculated Address

   Numerous references to data variables require either references to pointer values stored in other variables or references made with the help of calculated addresses as is necessary for indexed addressing into arrays. In such cases, the needed address will finally be found on top of the expression evaluation stack as the result of other variable fetches and arithmetic operations. To make references with such addresses there exists in the instruction set a number of instructions which provide efficient memory addressing based upon the value found in the top of the stack. (LSW0..LSW15, LSW, LSD, SSW0..SSW15, SSW, SSD, LXB, LXW, LXD, SXB, SXW, SXD)

As a final comment on addressing, it seems worthwhile to point out once again that data operations on local data variables affect only the local storage of the coroutine using the procedure. On the other hand, data operations affecting globally allocated variables affect all coroutines which use that particular module.

3.7 The Evaluation Stack

The evaluation stack of the Lilith is only 16 levels deep and it does not automatically overflow in the main memory stack as has been the case in other stack machines. There is also no possibility to check the stack for overflow. This poses an interesting question as to how the Lilith avoids undetected errors caused by accidentally overfilling the stack. The answer is contained in a simple rule: All programming in the Lilith is done using the Modula-2 compiler, and the compiler never uses the stack for any kind of an operation where it cannot determine the level of usage of the evaluation stack--as in the case of building arguments on the evaluation stack by recursive procedure calls.

This rule implies that each executed statement leaves the evaluation stack in an empty state at the end of its interpretation. Because function calls may occur in the middle of a statement, function procedure calls require that the evaluation stack be saved onto the main memory stack before calling the procedure. A return from function must restore the evaluation stack.

Coroutine transfers, obviously, must also save the state of the evaluation stack.

3.8 Coroutine Transfers and Interrupts

An explicitly programmed coroutine transfer has the form:

```
    TRANSFER(old,new);
```

where the parameters `old` and `new` are pointers to process descriptors. The process of executing a coroutine transfer begins with saving the contents of the evaluation stack onto the main memory stack of the coroutine being suspended. Then, the entire state of the machine is saved into the process descriptor whose address is found in the P register of the machine. When the entire state has been saved, the address of this process descriptor is saved in the pointer variable "old". Then, the address of another process descriptor is picked up from the variable new and the registers of the machine are restored to their last values for the coroutine resuming execution.

NOTE: The machine takes the value from the variable "new" at the beginning of the operation and saves it in an unused register internally during the saving of the coroutine state. Therefore, it is possible for "old" and "new" to actually be the same pointer variable.

3.8.1 Interrupts

A hardware interrupt requires that the state of the machine be saved and that a special driver program be activated to satisfy the source of the interrupt. When the need has been taken care of, the state of the machine must be restored and the interrupted coroutine allowed to continue processing. In the Lilith, this operation is identical in execution to the coroutine transfer which can be explicitly programmed. The interrupt handler becomes just another coroutine. The only difference is that instead of allowing *any* process descriptor pointer to be used as a parameter of the coroutine transfer instruction, an interrupt-caused transfer must use assigned locations for the parameters of the transfer operation. It is the responsibility of the programmer to see that the process descriptor pointer for an interrupt handler is stored into the correct memory location which is assigned to that interrupt.

This characteristic of the Lilith has some interesting properties. For example, once a coroutine has been written to perform a given function, the programmer can allow that function to be initiated either by a real interrupt or by an explicitly programmed coroutine transfer. The interrupt driver need not know the difference and also does not need to respond differently. This feature of Modula-2 programming is discussed more thoroughly in the Modula-2 reference report by N. Wirth.

3.9 Master Reset

When the master reset button of the system is pressed, the virtual machine must be initialized to a known beginning state and begin operation. It was also designed to aid the debugging of errant programs by saving the state of the machine before performing the initialization. The process consists of saving the machine state, loading memory with a boot program, restoring the machine with a new state taken from the freshly loaded boot program, and beginning execution. Although this operation takes place under control of the microprogram interpreter of the virtual machine, the same procedures which perform the interpretation of the coroutine transfer instructions are actually used.

4 Decisions Made in the Design of the Lilith

The overall goal of the project was to achieve an increase in performance while maintaining simplicity wherever possible. Efficiency was more the concern than brute computational power. From the start it was clear that the machine had to be limited by a goal of keeping it compatible a with simple office environment while providing substantial computational power. The limitations were in size, in complexity affecting reliablility, in power consumption, and in noise creation. There were no firm measurable guidelines to control the design decisions which had to be made. Only a generally intuitive feeling was used as the basis for the decision. This is not to say that the benefits of each design alternative were not thoroughly considered--only that the final decision as to whether or not an additional design feature would be worth the added cost was left to the intuition. In the final analysis, time will verify the correctness of these decisions. One thing is certain: the machine could have been either much bigger and more complex or much smaller and less complex; both of these possibilites remain to be tested.

4.1 Selection of the Memory Organization

The first major decision that was considered was the size and organization of the memory. A straight 16-bit by 131k organization was considered, but it was concluded that too much processing power would be lost in refreshing the display. A 64-bit organization for both read and write operations appeared too costly in terms of the extra overhead required for the ability to write individual 16-bit data words into each 64-bit memory cell. So, a compromise solution was chosen where the memory could be read in 64-bit words but written only in 16-bit words. Then, for the benefit of the processor data access operations and any eventual direct memory access devices, the memory was equipped with two read data buses--one having the full 64 bits and the other giving the 16-bit word selected by the full address. For 64-bit read cycles, the least significant two address bits were ignored.

A second major design decision regarding the memory was the decision to make the memory accessible from more that one port, which required the use of a memory-cycle-request arbitration circuit. The considered alternatives were either to synchronize the display and the processor or to provide the display with its own frame buffer and to remove it from the address space of the processor, causing it to be accessed as an input/output device. It was felt that the ability to efficiently use the processor power in the manipulation of the images was a prime consideration, so this eventually led to the design decision to make the memory an asynchronous device with respect to the processor, allowing multiple requests for service.

4.2 Memory Interaction with the Micro-Controller

The asynchronous memory system having been chosen, another design decision had to be made: how should the microcontrol unit handle the eventual possiblity that the memory interface might not be finished with a previous operation when it is ready to begin another? Two alternatives were considered.

First, the MCU could be so constructed that it could test the status of a memory interface before attempting to use it, Or second, it could blindly go ahead with any memory operation. In this case it would be oblivious to the state of the memory interface and the memory interface could, if not ready, signal to the master clock circuitry the need for a delay in the clock event until the interface was ready. In considering this problem, it was felt that a significant loss of processing power could come from the first approach. The reasoning was that if a memory interface were not ready, then no additional speed advantage would come from either method as both would be required to wait until the operation was completed. However, for the large number of times when the memory was available, then an additional cycle would be wasted to verify this fact. It was also felt that if the microprogram called for a memory operation and the memory were not ready, then it was not likely that anything useful could be done while waiting.

So, the second alternative was selected. In retrospect, it is not certain that this was necessarily the best choice. In measuring the performance of the system, it was observed that the minimum cycle time of the microinstructions was lengthened by the signal propagation necessary to respond correctly to the "not ready" signal from the memory interfaces. Initially, it was much greater that now, but substantial redesign of the memory interfaces and the clock generation circuitry of the CPU was necessary to achieve the goal of 150 nsec cycle time.

4.3 The Instruction Fetch Unit

The multiple port memory and 64-bit wide memory path having been chosen, the opportunity was naturally quite apparent to exploit these design features in some manner for the processor. A standard cache memory approach had been thought of, but it was considered to be too complex for the level of machine being designed. Still,it was felt that advantage of the wide memory bandwidth should be taken. Finally, the idea developed of building a separate instruction fetch unit which could fetch, through its own memory port, eight bytes of instructions at a time, and hold them until needed by the processor. Initially, it was thought that instructions could be fetched in 16-bit words, but this did not fit well with the 8-bit opcode structure of the Lilith virtual machine.

A great deal of thought was given to the problem of addressing bytes while the main store was actually addressed in words. A further complication was the fact that byte addressing would limit the size of code memory to 65k bytes, or only 32k words, because of the need to store a procedure return address in a 16-bit word of memory. It was considered a great limitation that only 32k words could be use for code, so thought was given to make all pointers and return addresses 32 bits long. This affected integer arithmetic, the size of passed parameters, and numerous other operations which made the machine into a much bigger machine than we had desired.

Finally, the problem congealed into a very usable solution through the organization of an instruction fetch unit having two registers, a base address register for the code frame, and an offset value which was called the program counter. The base address register, called the F-register, is added to the offset (PC) in the IFU through the use of MSI chips--an idea that was disliked at the beginning because of the thought of including an entire 19-bit adder in the IFU. The F-register is only 16 bits but is offset by three bits from the PC offset value--giving a total of 19 bits for the byte-wise addressing of the instructions in memory. This solution provided many advantages: an expanded address space for instructions, efficient 64-bit fetches of instruction bytes, transparent overlapped instruction fetches taking place automatically through the port assigned to the IFU each time the last instruction byte of the current eight byte group was taken, and no necessity to recompute addresses when loading a module into memory, thanks to the offset addressing.

A final concern after this scheme had been conceived was the possibility that an error could occur if an instruction code were updated in main memory after it had already been fetched for the current instruction stream. A check with the compiler writers verified that anyone caught performing any kind of instruction modification, as used to be common practice in von Neumann machines and assembly language programming, would be heartily flogged.

4.4 The Expression Evaluation Stack

The initial concept of the virtual machine for the language Modula-2 did not include the expression evaluation stack as described previously. The idea of a small hardware stack for expression evaluation purposes was considered in the sense that it had tradionally been implemented in Burroughs and HP machines, namely that the stack would automatically overflow in to the main stack in memory. It was Urs Ammann who first suggested that a small stack for use by the compiler need not be very large, yet would never need be tested for overflow since the compiler would use it only for operations where it could control the depth of use. His initial suggestion was that four levels would be more than sufficient, but at that point his statement was not readily believed.

After a study of the the 2901 architecture had been made, a mechanism was conceived whereby sixteen levels of stack were readily added to the ALU and in such a fashion that the PUSH, POP, and arithmetic operations on the top two levels could be accomplished entirely in one microcycle! The addition required 6 additional chips and some control bits in the microinstruction word. The utilization of the stack does require the ability to test when the stack is empty. This "stack empty" test is required because each time a procedure is invoked, the stack must be emptied into the procedure activation record which is part of the main stack . If this were not done, a recursive procedure could be the cause of increasing entries on the stack, which the compiler could not control at compile time. Furthermore, the same stack-save operation must be performed with each coroutine transfer, lest the evaluation stack become overfilled and confused because of its inability to test for overflow.

4.5 The Short Operand Instructions

The shifter on the ALU was deemed necessary from the beginning to handle the complex shifting operations associated with image manipulations on the display. The AMD 25S11 chips were chosen from the beginning by the project leader, N. Wirth, and no question was ever made of their appropriateness. However, in studying the virtual machine instruction set, it became apparent that a large number of instructions which would be very frequently used involved the separation of a 4-bit operand from the single eight bit opcode. It was initially intended that the decoding would take place by creating map entries to different locations of the control store where separate cycles would introduce these four bit constants with the micro instruction format which places 8 bit constants onto the cpu BUS. It was entirely coincidental that the already extant architecture lent itself so well to the separation of the 4 bit operand into a holding register at the same time that the eight bit opcode was mapped by the map ROMs, in the MCU, into the starting address of the microcode sequence for that M-code. Otherwise, an additional two memory locations, as well as two addition micro-instruction executions, would have been necessary for the interpretation of each of the short operand instructions.

5 Theory of Operation

In the following sections, an attempt will be made to explain the theory of operation involved for each major component of the Lilith processor. This will be done in a manner which we hope will relate the function of the hardware to the process of emulating the Lilith virtual machine. There will not be an attempt to exhaustively clarify the function of each integrated circuit in the system. This will be reserved for a later chapter (Chapter 6) as it would involve introducing too many details which would cloud the real subject for this chapter.

5.0.1 An Overview of the Real Machine

Beneath the outer shell of the "virtual machine," we find the inner machine which is responsible for creating the illusion of virtual machine. The programmer of the Lilith sees this machine very rarely. He uses the programming language Modula-2 which shields him from most encounters with the virtual machine. Only when he creates code procedures or uses the debugger does the programmer catch a glimpse of the virtual machine. The virtual machine, in turn, shields the programmer almost completely from the real machine. The programmer only becomes aware of it when he desires to add an additional instruction to the repetoire of the virtual machine.

The real machine consists of many subsystems tied together by a common data artery called the *BUS*. The *BUS* is 16 bits "wide" and operates using three-state drivers. There are no logical operations performed by simultaneously driving the bus from two or more sources as can be the case with buses constructed using "open-collector" technology. At any given time, there is only a single bus source. Alternatively, the *BUS* may not be in use and thus be in a high impedance state.

The *BUS* connects the following system components together:

MicroControl Unit:

This component, located on a single printed circuit board designated MCU, is the controlling entity which contains the interpreter program for emulating the virtual machine. To this unit, the M-codes of main memory are data which it must analyze and interpret through special *microprogram* instruction sequences assigned to each M-code. One of its important functions is providing the timing signal, CPUClk, which synchronizes the operation of all components of the system. Another important function is controlling the flow of information over *BUS* by emitting control signals selecting the bus source and destinations.

Arithmetic/Logic Unit:

This component, located on a single printed circuit board designated ALU, is the residence of most of the registers used for holding data and addresses. It also has the arithmetic circuitry which operates on these registers. The evaluation stack and a special barrel shifter are also found on this board.

CPU Data Port:

This component occupies a portion of the circuit board designated the CDP. Its function is to provide access to the memory subsystem for all memory operations other than the fetching of M-code instructions for interpretation.

Instruction Fetch Unit:

Instruction in this case refers to M-code instructions. This system component, located on a single printed circuit board designated IFU, handles the special function of fetch M-codes for interpretation by the microprogram of the real machine.

Memory:

The memory subsystem is resident on four printed circuit boards with a portion of the access control sharing the CDP board with the CPU Data Port. The memory has eight ports, of which four are used in the standard configuration.

I/O Address Decoder:

This subsystem is also resident on the CDP board. Its function is to decode the I/O address lines for the selection of an I/O device and to generate the I/OClk timing signal.

Display Processor:

This subsystem, resident on a single printed circuit board designated DSP, displays a memory-mapped bitmap on the raster of a CRT. The information for the display is taken from the main memory through one of the ports of the memory subsystem.

Disk Controller:

This subsystem is resident on a single printed circuit board designated DSK. It handles the transfer of sectors of data between the Winchester disk and the processor. It has an on-board sector buffer, and the transfer of information between the sector buffer and the processor memory is handled by a microprogrammed block transfer.

Miscellaneous I/O:

A number of simple I/O interfaces are also located on the CDP printed circuit board along with the CPU Data Port. They are: the keyboard interface, the mouse interface, the real time clock, and the RS232c UART interface.

A block diagram showing the interconnected relationships between the elements of the Lilith is found in Figure 1.1

5.1 Micro-Control Unit (MCU)

When we think of a computer, we think of a device which has a characteristic behavior typlified by several features. First, it accepts tasks to perform, as defined by a collection of instructions called a program, which it places in memory. Second, in the performance of these tasks, it operates in cycles, first fetching an instruction, then executing operations according to the content of the fetched instruction, and then fetching another instruction, ad infinitum. Third, in the selection of its instructions, it may take them either from a single sequence given numerically ascending addresses, or it may branch to a new sequence based on the results of previously performed operations. We have described a computing entity such as this in the previous sections and referred to it as the ***Lilith virtual machine.*** Now we wish to describe a sub-component of the Lilith computer which also exemplifies characteristics that qualify it as a computing element. Like a computer, it has instructions taken from a random access memory, it executes these instructions according to an order code, and it also may choose its next instruction according to the results of previously executed instructions. To avoid confusion with the main Lilith computer, we refer to this sub-component as a microcontroller, and we refer to the instructions which it executes as "microinstructions." The microinstructions together are referred to as the microprogram, and they are stored in the "micro-control store" to avoid confusion with the main storage or memory. This entire subsystem is referred to as the Micro-Control Unit or MCU.

5.1.1 The Function of the MCU

The function of MCU is to emulate the Lilith virtual machine. It can do this because all other elements in the system are under its control, more or less. Its method of accomplishing this task is to fetch one virtual machine instruction (M-code) after another and, depending on the instruction, to command the other elements of the system to perform the function defined by that M-code. To perform the function of each M-code, the MCU uses a sequence of microinstructions in its control store which have been programmed for that particular M-code. In the process of executing a sequence of microinstructions, the MCU will at times emit control signals which cause the arithmetic/logic unit (ALU) to perform arithmetic and logical manipulations of 16-bit data words stored in registers within the ALU. At other times, the MCU will also coordinate the transfer of data from one subsystem, viz the ALU, over a 16-bit bus, called BUS, to other subsystems, i.e. the memory, and it will synchronize the operations of these subsytems through the use of a special timing signal known as the clock, designated CPUClk.U. It will typically require four or five microinstruction execution cycles to identify and interpretively perform the operation defined for a single M-code. The total collection of all microinstruction segments for the complete repetoire of virtual machine instructions is called the ***virtual machine interpreter.***

5.1.2 Components of the MCU

The MCU has the following components which are involved in its operation:

Microinstruction Register (MIR):

This is a 40-bit register holding the instruction being executed.

Control Store:

The control store consists of a fast bipolar PROM memory with a 40-bit word size and 4096 possible words.

Next Address Logic:

The circuitry of the next address logic is based on the AMD 2911 parts which provide a choice of possible addresses for the next instruction including subroutine calls and returns, and table jump based upon a fetched M-code

Clock Generation Circuitry:

The clock generation circuitry is externally controllable circuitry that generates the clock timing pulse which synchronize the system

Bus Source and Destination Control Signal Generation Logic:

This logic decodes bits from the microinstruction and thereby generates signals which control other sub-systems and their use of the BUS for sending and receiving data.

Interrupt Priority Logic:

The interrupt priority logic interrupts the execution of microinstructions at the beginning of the interpretation of a new M-code and causes a coroutine transfer to an interrupt handling routine

The interconnection of these components is shown in Figure 5.1.

5.1.3 An MCU Microcycle

The fetch and execution of a microinstruction requires two periods of time called *cycles.* Each cycle is either 150 or 225 nanoseconds long. During the first cycle, the microinstruction is read from the control store memory and loaded into the microinstruction register (MIR). The loading of the MIR takes place at the point in time which is the end of the first cycle and the beginning of the second. During the second cycle, the contents of microinstruction loaded in the MIR assert control over the machine and "execute" the microinstruction. During the second cycle in the execution of the microinstruction, the first period of the next microinstruction (fetching the microinstruction from the control store) is simultaneously taking place in "pipeline" fashion. Because of this overlap in the execution of one instruction and the fetch of the next, the Lilith completes the execution of one microinstruction during every cycle.

We will begin our analysis of a microcycle at the point in time marked by the upward transition of the MCUClk signal, which is both the end and the start of a microinstruction cycle. Shortly after this point in time, the microinstruction address bus will stabilize with the address of the next microinstruction to be fetched from the control store. During the entire 150 nsec period until the next MCUClk upward transition, the control store will respond to this address, read out the microinstruction from its internal storage, and present it to the inputs of the microinstruction register. When the next MCUClk transition occurs, the microinstruction register is loaded with the microinstruction fetched from the control store.

Once loaded into the microinstruction register, the bits of the instruction are transmitted to the various processor components where they determine the transfer and manipulation of data during this cycle. During the entire clock period of 150 nsecs, until the next MCUClk transition occurs, the microinstruction register propagates control signals throughout the machine. When the next MCUClk pulse does occur, it causes registers throughout the machine to sample and hold data created by the microinstruction. This marks the end of a typical microinstruction cycle. It required 300 nsecs from beginning to end. However, because of the pipeline construction of the MCU, two instructions are in process at any given time--one being fetched from the control store, the other in the microinstruction register in execution. The net throughput is one microinstruction each 150 nsecs.

5.1.4 The MCU Microinstruction Format (Figure 5.2)

A microinstruction is composed of groups of bits which typically select one of several possible operations. The values of each bit in the microinstruction are connected to one or more components of the Lilith which operate according to a definition assigned to the instruction bit values. Each microinstruction is 40 bits wide. The bits of the instruction are used to control the following functions:

ALU Operation and Operand Specification:

Twenty bits of the microinstruction go directly to the ALU and provide a myriad of possible instructions.

BUS Source and Destination Selection:

Eight bits of the microinstruction select the source and destination of data to be carried on the bus during this cycle.

Next Microinstruction Address Selection:

Based on condition code testing and 3 next address control bits, the next microinstruction address can be selected from either an address specified in the current microinstruction or a M-code table lookup address or an incremented previous value.

Microcycle Timing Control

A single bit can slow the timing of a microcycle by 50%. This is necessary for some instructions which cannot operate correctly in the normal cycle period. Also, If the BUS source or destination selection is a memory interface which is not ready, it can stop the cycle indefinitely (until the memory is ready).

Constant Generation:

A single bit identifies the microinstruction to be of the format which causes the rightmost 8 bits to be used as a constant placed on the *BUS* instead of as *BUS* source and destination selection bits. The ALU is automatically selected as the destination in this case.

All of the above listed microinstruction options are not in each instruction. There are three microinstruction formats to choose from, which select the various uses of the bits of the instruction:

Format I

The first format has ALU control, BUS source and destination selection, and next address selection by incrementation or M-code table lookup.

Format II

The second format also has ALU control but the BUS source and destination selection bits are used as a constant which is placed on the *BUS*; the *BUS* destination is automatically selected as the ALU.

Format III

The third instruction format has neither ALU control nor *BUS* source and destination control. It uses the ALU control bits as a jump address and the condition code selection. This selection code determines whether or not the jump address in the instruction is used for the next microinstruction or whether the current microinstruction address is simply incremented.

Figure 5.3 shows these microinstruction formats.

5.1.5 Clock Generation Circuitry

There are two important clocks which are generated by the clock generation circuitry: the MCUClk and the CPUClk. The MCUClk is generated for every instruction cycle, while the CPUClk is only generated for the execution of the microinstructions which have valid control bits for the ALU, namely the first and second microinstruction formats. These clock signals are generated regularly, for the most part, at 150 nsec intervals, but there are a few special circumstances which can alter this:

1. **Slow Cycle**

   A special bit in the microinstruction word can be set to cause the interval to be extended to 225 nsec to accomodate instructions which cannot complete in the 150 nsec interval. Currently, the only instructions which requires additional time are those which look up a starting address in the M-code map ROM for interpreting M-codes.

2. **Memory Delayed Cycles**

   When a microinstruction selects as the "BUS source" or "BUS destination" one of the subsystems interfaced to the memory (the IFU or the CPU Data Port) and this interface has not completed a previous operation, then the clock generation must be delayed until the interface is free.

3. **Diagnostic Clock Control**

   For diagnostic purposes, there are control signals accepted by the MCU which can halt the generation of clock pulses or permit the generation of single clock pulses. The Diagnostic Processor (DPU) uses these signals to statically execute test instructions and verify their correct results after one or more clock pulses. For normal operation, these control signals are not used.

5.1.6 Microinstruction Address Generation

At the heart of the microinstruction address generation circuitry are three AMD 2911 sequencer chips, which contain the logic and registers to hold the current address, increment it, save it in a stack after incrementing it for subroutine returns, and replace it with an externally generated address. The selection of these operations is controlled by three bits which are part of each microinstruction. These control bits can also select other features of the 2911 which will not be discussed here because these features were not used in the machine implementation. Schematic MCU2/5 shows the next microinstruction address generation circuitry. Most often, of course, the facility to increment the current microinstruction address is used so that instructions can be taken in sequence. There are other instances where an externally generated address is required:

1. **Subroutine Calls**

   For subroutine calls to a maximum depth of 4 levels, the old microinstruction address incremented by one is pushed onto an internal stack while the next microinstruction address is taken from an external source. At the end of a subroutine execution, the saved microinstruction address is popped from the stack when appropriate.

2. **Conditional Branches**

   For all conditional branch operations which test condition codes of the ALU, the selected microinstruction address will be gated directly from the microinstruction register if the condition code selects the branch. If a branch is not selected then special circuitry will change the control signals of the 2911 to cause it to take the next microinstruction in sequence.

3. **Table Lookup for M-code Interpretation**

For the table lookup operation required to interpret an M-code, a special "map" ROM uses the M-code as its address to select a 12-bit microinstruction address where the instruction sequence to interpret that M-code begins.

4. Interrupt Vectors

When an interrupt occurs, a special reserved microinstruction address will be substituted for the M-code table lookup address. At the same time, the main storage program counter will not be incremented so that the current state of the machine can properly be saved.

5.1.7 Processor Bus Control

The common means of communication between all subunits of the processor and all peripheral devices is a 16-bit three-state bus called *BUS*. During any given instruction cycle, only one subsystem can be allowed to assert data on the BUS, and for most cases only one subsystem samples the signal values of the BUS. With 4 bits in the microinstruction word allocated to both the BUS source selection and the BUS destination selection, there are 16 possible candidates for each role. In the MCU circuitry, these four bits are decoded into 16 separate signals which control individually each subsystem. Each of these signals is in its active state when it is asserted low. The device controlled by this signal is designed to respond to a low level on this line by either gating information onto the BUS, or sampling it from the BUS into a register, depending on whether the decoded signal is a source or destination control signal (Figure 5.5). There are two anomalous conditions to this otherwise pure decoding function: First, for the third microinstruction format, the BUS source and destination decoders are completely disabled and the BUS is not used during this type of instruction. Second, in the second microinstruction format, the decoders are also disabled because the control bits are being used for constant generation. But the BUS selection control signal, Dst=ALU.L, is forced to a low or active level because it is always the recipient of the constant information gated from the microinstruction during the Format II microinstructions. The implication of this is that if you simply wanted to store a constant into a memory location, you would have to first transfer the constant from a microinstruction into the ALU, and then, with a Format I microinstruction, transfer the constant value from the ALU to the memory data register of the CPU DATA PORT Interface.

5.1.8 Interrupt Handling (Figure 5.5)

Interrupts are allowed by the hardware to occur only at the beginning of an M-code microinstruction sequence. This requires the interrupt logic to be tightly interconnected with the table lookup logic for beginning microinstruction address selection for an M-code. There are eight levels of interrupts which originate external to the MCU and which are asserted low when a request is active. The request is clocked into a register every cycle but will be looked at only when a M-code fetch instruction occurs. For each of the eight interrupt levels, there is a mask bit which can inhibit the processor's response to the interrupt if set true. The source of mask information sent to the mask register is the BUS. The loading of the mask register is controlled by one of the decoded BUS destination selection signals. The mask register can also be read onto the BUS when selected by a BUS source control signal. If an interrupt request is not masked, then it will reach the interrupt priority encoding circuit which will generate a microinstruction address vector according to the priority of the highest active unmasked interrupt. This address will be enabled onto the microinstruction address bus at the start of the next "jump map" instruction. The vector includes a base value of eight, which means interrupt seven causes a microinstruction address of 15 to be used for the vector. Having the offset of eight (i.e. not beginning at zero) is necessary because location 0 is reserved for the master reset function.

5.1.9 Master Reset Circuitry

When a master reset command is given, the virtual machine of the Lilith follows a sequence of operations which first load a bootfile from disk, then establish a legal coroutine state, and begin execution of a task found within the programs which are part of the boot file. Once the initial task is fully operational, other tasks are usually initiated to handle keyboard I/O, the real time clock, and execution exceptions. The boot file has a special format, simpler than normal binary code files, which can be loaded by a short microprogram segment. There are two boot files which may be loaded. The user selects which of the two boot files to load with his first typed character. A *control A* cause the microprogram to load from the file called PC.BootFile.Back on the disk. Any other key causes the standard boot file, PC.BootFile, to be used. A special feature of the Lilith incorporated in the microprogram to permit complete debug capability is its ability to first dump the state of the machine into a special disk file before reinitialization of the system. This feature, which is initiated by the typing of a *control D*, allows debugging of runaway programs that have escaped the normal means of halting and analyzing.

5.2 Arithmetic/Logic Unit (ALU)

The arithmetic/logic unit is the basic mechanism for data manipulation in the Lilith. It is capable of sophisticated shifting, arithmetic operations, and logical operations. It also has storage registers and flip-flops which can capture the results of these operations. Most of the registers defined as part of the virtual machine are kept in the registers of the ALU.

The description of the combinational operations of which the ALU is capable is very detailed. Fortunately, the sequential or time dependent aspects of the ALU operation are very simple, especially in comparison with the MCU circuitry.

5.2.1 Timing of ALU Operations

Operations in the ALU are divided into time periods synchronized with the execution of microinstructions in the MCU. The basic timing signal, CPUClk, which originates in the MCU is the mark in time for both the start and end of a microinstruction. Immediately after the occurrence of a CPUClk transition from low to high, some of the control signals coming from the microinstruction register stabilize to values which select registers and operations for the next microinstruction execution. Other control signals select registers which are to receive the results of the computations. At the end of the microcycle--at the time of the next upward transition of the CPUClk--the new data created from the contents of registers and modified by the combinational logic of the ALU is captured into the registers selected by these control signals. This data can then be used as operands in subsequent microcycles.

5.2.2 Subsections of the ALU (Figure 5.7)

The major components of the ALU are the shifter, the stack, and the four 2901 bit slice arithmetic units. Other components of the ALU are the shift control, the condition code register, condition code selector and gating units which control the transfer of data to and from the stack and the CPU BUS. These component sections of the ALU are composed of registers and combinational logic. They are controlled by signals which originate in the MCU.

Obviously, the 2901 bit slice arithmetic units are the heart of the ALU. However, their performance has necessarily been augmented by a specially constructed barrel shifter and an external bipolar stack in order to provide all the computing features desirable to support the Lilith requirements.

5.2.3 ALU Functions

The ALU is the main theatre for operations concerned with the execution of the Lilith M-code instructions. To illustrate how the ALU would be involved in a sequence of M-code operations, we will examine the execution of the hypothetical Modula 2 statement:

```
c:= b + a↑
```

where "b" is declared a local variable of type INTEGER, "c" is a global variable of the same type, and "a" is a global variable of type POINTER TO INTEGER. The sequence of M-codes emitted by the compiler for this statement would be:

```
LL4     (*load the fourth local variable which is b*)
LG3     (*load the third global variable which is a*)
LSW0    (*load word addressed by top of stack which is
        pointer value from a*)
ADD     (*add the two operands*)
SG5     (*store top of stack into fifth global variable
        which is c*)
```

The sequence of ALU operations for the execution of the above instructions is as follows:

During the instruction fetch, the ALU receives the 4-bit constant "4" from the BUS, and stores it into a temporary holding register. In the next cycle, the ALU adds the constant to the local variable base address register (L-register) in the 2901 and transfers the resulting address to the data port for a memory read cycle referencing variable "b".

The contents of the memory cell assigned to variable "b" are transferred from the memory data port over the CPU BUS to the top of the stack register in the 2901 portion of the ALU.

The 4-bit constant "3" is similarly used together with the global variable base register (G-register) to fetch the contents of the variable "a". This time, as the value of "a" is loaded into the top of the stack register, the old contents of the top of stack, which have the value of variable "b", are simultaneously pushed onto the bipolar stack in the ALU.

The 4-bit constant "0" (zero) transferred from the IFU during the M-code fetch and the address in the top of stack are added together and sent to the memory data port to fetch the data value referenced by the pointer variable "a". This fetched data is stored on the top-of-stack register into 2901.

The top-of-stack register in the 2901 and the second level of the stack (which is located outside the 2901 in the bipolar stack) are added together, and the result is stored in the top-of-stack register.

The contents of the top-of-stack register are sent to the memory data port to be stored in a location. The address for this memory write cycle is constructed from the 4-bit constant "5" and the global variable base address register (G-register) added together.

In the execution of all the above instructions except the fourth, the ALU receives a 4-bit constant from the CPU BUS at the same time that the IFU places the instruction opcode onto the CPU BUS for decoding by the MCU. A special 4-bit masking operation strips this constant out of the opcode. The constant is stored temporarily into the Q register of the 2901 until it is used in the next microcycle with one of the other registers in the 2901, such as the local variable base address register (L-register).

5.2.4 2901 Chip Operations

The capabilities of the 2901 bit slice arithmetic units are thoroughly documented in the product manuals from Advanced Memory Devices (AMD), the manufacturer of the chip. An attempt will not be made here to exhaustively describe their capabilities. Instead, the intent here is to simply list the operations possible and the operands to which they may be applied (Figure 5.8).

In the microinstruction register (formats I and II only), seventeen bits directly control the 2901 arithmetic unit. They are MIR21..MIR28 and MIR31..MIR39. Bits MIR21 through MIR28 select the A and B operand choices from the 16 register file, while MIR31 through MIR39 are the actual control signals which select operations and operands. In the AMD documents, MIR31 through MIR39 correspond to signals I0..I8. The combinations of operations and operands in the 2901 far exceed that which can be encoded in 9 control signals, so the designers of the 2901 chips have provided a selection of 512 of the most usable combinations which may be specified with 9 control bits. The control bits are broken into three fields: source operand selection, operation, and destination operand selection.

The source operand selection bits are bits MIR31..MIR33. They select eight possible combinations of operands to be used by the arithmetic unit inside the 2901. These eight combinations are taken from five possible sources:

1. The register selected by the A field in the microinstruction
2. The register selected by the B field in the microinstruction
3. The Q register in the 2901
4. The direct data inputs to the 2901, which may come either from the stack or from the CPU BUS via the shifter
5. No register is selected, resulting in a word of zeros used as the operand

The eight combinations of these source operands available for use are given in Figure 5.9.

The operation selection field bits, MIR34, MIR35, and MIR37, select operations between the two source operands selected by the field consisting of bits MIR31, MIR32, and MIR33. Since the operations are not symmetric, it is necessary to specify which operand is the first. The symbols R and S are used and refer to combinations of operands as given in Figure 5.9. The list of the eight possible operations are given in Figure 5.10.

The destination control bits are MIR37, MIR38, and MIR39. These bits control the transfer of the arithmetic unit result to the register file, the Q register, and the gating to the output drivers of the 2901. The possibilities here are also increased and complicated by the single bit shift operations which are possible through the register file shifter logic and the Q register shifter logic, both of which are internal to the 2901. The eight possible destination operations constitute a subset of possible operations chosen according to the functions which are most commonly used. The total matrix of destination selections and their effect on the register file, the Q register, and the gating to the output drivers is given if Figure 5.10 taken from the AMD 2901 specifications. Since this format seems to mask the fundamental simplicity of the operations, the following explanation of each opcode may be more enlightening:

MIR37..MIR39	Operation
000	The computed function is transferred to the Q-register and to the output buffer.
001	The computed function is gated to the output buffers only.
010	The computed function is transferred to the register addressed by the B-field. The output buffers gets the contents of the register addressed by the A-field.
011	The computed function is transferred to the register addressed by the B-field and the same data is gated to the output buffers.
100	The computed function is loaded into the B-field-addressed register, shifted one bit to the right, and the Q-register is shifted one bit to the right. The unshifted computed result is gated to the output buffers.

101	The computed function is loaded into the B-field-addressed register shifted to the right, and the Q-register is unmodified. The unshifted result is gated to the output buffers.
110	The computed function is loaded into the B-field-addressed register shifted to the left, and the Q-register is shifted left as well. The unshifted computed result is gated the output buffers. This instruction is used in division.
111	The computed function is loaded into the B-field-addressed register, shifted to the left, but the unshifted result is gate to the output buffers.

5.2.5 Condition Testing in the ALU

Each microinstruction in Format I or Format II (see Fig. 5.2 for microinstruction formats) results in the setting of condition codes which are then testable using a Format III jump instruction. Most of the condition codes are generated in the 2901 bit slices as the result of an arithmetic operation, but some are developed in other parts of the processor. The condition codes resulting from 2901 operations are:

Z	The Z condition asserts high when the computed value from the arithmetic unit of the 2901 had a value of zero.
OV	The OV condition asserts high when a two's complement operation results in overflow (logically the exclusive-OR of the carry bits of the most significant two bit positions).
F15	The F15 condition asserts high when the most significant bit from the arithmetic unit.

F8	The F8 condition asserts high when the most significant bit from the lower half of the arithmetic unit.
S	The S condition asserts high when the XOR of the overflow and the sign bit.
C	The C condition asserts high when the carry bit from position 15 of the arithmetic unit asserts high.
StE	The StE condition asserts high when the last pop operation emptied the stack.
REQ	The REQ condition asserts high when an interrupt request is pending.

These condition codes allow testing of the arithmetic and logical operation results at the by the microinstructions.

5.2.6. Multiply Operations

"Multiply" operations take place in the conventional add and shift fashion where the add operation is optional, depending on whether or not the appropriate bit in the multiplier is a zero or one. At the beginning of the operation, a register to be used as the accumulator is set to zero and the multiplier is transferred to the Q-register. Then sixteen cycles of add and shift are performed with the Q-register shifted as well receiving the least significant bit of the accumulator into its most significant position. The optional addition of the multiplicand is accomplished by special logic which tests the least significant position of the Q-register and appropriately alters MIR32 of the microinstruction so that the addition will take place if that bit is a "1" but will be omitted if that bit is a "0."

5.2.7. The Evaluation Stack

The evaluation stack of the Lilith is located on the ALU board and has the uppermost two levels of the stack accessible as operands of a microinstruction. The organization of the stack is such that it is possible to perform all normal stack instructions in a single microcycle, These stack instructions include push, pop, and arithmetic operations between the top two levels, with the result being replaced onto the top of the stack. The stack may be tested to determine whether or not it is empty, but there is no possibility to test whether or not the stack overflows. Consequently, it is vital to design the system so that the stack is used only for operations where the depth of the evaluation stack usage can be controlled by the compiler.

The evaluation stack is 16 levels deep and is constructed from bipolar random access memory. Although it is made from a conventional random access memory, a special addressing mechanism gives it the appearance of being a stack. The top element of this stack is kept in a register of the register file in the 2901, with the lower entries kept in the bipolar random access memory. Because of this organization, microinstructions utilizing the stack have the capability to simultaneously reference, in a single microinstruction, both the top of the stack in the register file and the second level of the stack in the external bipolar memory through the direct inputs of the 2901.

Although it is not certain that the designers of the 2901 actually planned it, a feature of the 2901 is singularly useful in connecting an external stack. This feature is the bypass path from the A register to the 2901 buffer outputs. Because of this feature it is possible, as shown in Figure 5.11, to handle both push and pop operations in a single cycle. In the push operation, for example, the top of stack is addressed as the A-register in the 2901 and gated to the output buffers of the 2901 and to the input of the outboard stack. Meanwhile and simultaneously, the incoming data is gated from the cpu BUS through the shifters and into the direct inputs of the 2901. From this point, the new data for the stack passes through the arithmetic unit and is written into the top of stack register addressed by the B register address. All this takes place in a single cycle. A similar mechanism is used for a pop operation. If, in fact, the designers of the 2901 intentionally included this mechanism, they obviously felt it was of little value, since they left it out of their later improved versions.

Topics to be added:
- divide operation description
- operand masking during instruction fetch
- shifter usage
- mask generation

5.3 CPU Data Port (Figure 5.12)

The CPU Data Port (CDP) is an exchange center for data transfers between main storage and the processor. It is connected on one side to the 16-bit processor BUS and on the other side to the memory address and memory data bus. It has the capability to read data from a memory address in main storage and to pass the data from that location onto the processor BUS. Conversely, it can receive a piece of memory data (data to be stored in main memory) from the processor BUS and store it into main storage. The CDP is used for all memory cycles where data values are read for use in program variables and for transfers of disk data to and from buffer areas in main storage. The data port is not used for display refresh or the fetching of instructions.

5.3.1 Processor Control of Memory Data Port

All memory transfers between the CDP and the memory subsystem are initiated by the processor. The data port is connected to the processor data BUS and responds to commands from the MCU bus control logic to take or give data to the BUS. There are four signals which come to the data port from the MCU bus control logic: Src=MD.L, Dst=MD.L, Dst=HMAR.L, and Dst=MAR.L.BUS. The CDP responds to the Src=MD.L signal by placing the contents of the *read data register* on the BUS. The other three signals cause the data on the processor BUS to be loaded into one of the three other registers in the data port interface. When the lower part of the memory address is loaded, the control logic of the data port is also triggered to start a memory cyce. If the *memory data register* has been loaded since the last cycle, then a write cycle will take place, otherwise a read cycle.

5.3.2 18-Bit Memory Address Register

Two transfers between the processor BUS and the data port are necessary to give a complete 18-bit memory address (for access to high memory) to the *memory address register* of the data port. Two are required because the processor only transfers 16 bits at a time. Consequently, when the high portion of the *memory address register* is being loaded, it receives the least significant two address bits from the BUS, but no memory cycle is initiated. (Actually four bits, but at this time only two are usable in the current Lilith implementation.) After the high portion has been loaded, then a full sixteen bits are loaded into the low portion, which also causes the memory cycle to begin. After the completion of the memory cycle, the high portion of the register is automatically cleared. This means that memory cycles into the lower 64k words of memory do not need to load or clear the high memory address register. The current Lilith virtual machine interpreter uses this feature.

5.3.3 Memory Busy Condition

Since the memory operates asynchronously from the processor and can service memory cycle requests from a number of sources, it is possible for the processor to attempt an operation with the data port before the previously initated operation is complete. The most common such situation occurs when the processor attempts to initiate a memory read cycle and then attempt to transfer the data from the *memory data register* before memory has serviced the port. Similarly, this situation occurs when the processor attempts to initiate a new read or write operation while a previously initiated write operation is still uncompleted. For such events the data port has a control signal, DataPortRdy, which is asserted true whenever the data port is available for use. If a microinstruction references the data port when this signal is unasserted, then the cycle time of that instruction will be delayed until the data port is free and the microinstruction can execute correctly.

5.3.4 Memory to Data Port Interface

When the low memory address register of the data port is loaded from the processor BUS, a flip-flop is set, asserting the DataPortReq.L signal. This tells the memory cycle control that a memory cycle is needed. After the cycle has been allocated, the ClrReq signal from the memory control logic will clear this flip-flop. The complement side of the flip-flop generates the DataPortRdy signal, for the MCU inhibiting use of the data port during the period between a requested cycle and the completion of the cycle. The mechanism of data transfer between the port and the memory is described in the memory theory of operation.

5.4 The Instruction Fetch Unit

The only function of the Instruction Fetch Unit (IFU) is to fetch instructions bytes from main memory and to transfer these instruction opcodes to the MCU and the ALU. The microprogram stored in the control store of the MCU uses the instruction opcodes for interpretive execution of the M-code programs. Since these opcodes are for the most part from adjacent locations of memory, the IFU takes advantage of its 64-bit memory port to fetch 8 instruction bytes at a time, holding them in registers until the processor requests them. Whenever the last of eight fetched bytes have been used, or a jump instruction is executed, the IFU must fetch a new block of instructions.

5.4.1 Next Instruction Address Calculation by the IFU

The IFU fetches instructions from code portions (called *code frames*) of modules which are resident in the main memory of the Lilith, and it delivers them upon demand to the MCU for interpretation. The base address for the *code frames* of any module is to be found in the first word of the global data area for that module. The address of the global data area, in turn, is taken from the module pointer table in the lower part of memory. The beginning address of the global data area is referred to as the data frame pointer; the beginning address for the *code frames* of the module is referred to as the *code frame pointer.* A new code frame pointer is fetched from global data area in memory each time the processor begins execution in a new module. For subsequent instruction fetches from within the same module, the address found in the code frame pointer is kept available in a register of the IFU. It serves as the base for computing addresses which are used in memory read cycles bringing instruction codes, eight at a time, to the IFU. This register is denoted the "F register."

Any time a procedure of a different module is entered, the F (frame) register of the IFU must be loaded with the code frame pointer from the new global data area. The actual instruction address from within the module which is used for fetching instruction codes is formed from the value in F register plus a value loaded into a counter register called the *offset register.* The *offset register* most closely corresponds to what would normally be referred to as the program counter because each time an instruction is fetched, the value in the offset register is incremented to the next byte in memory. For this reason, the offset register is often referred to as the *program counter register* (PC), although, in truth, the actual program counter value is the sum of the F register and the offset register. The value of the offset register is limited to a maximum value of 32,767. Because of this, the size of any module is fundamentally limited to a maximum length of 32,768 bytes. This limitation has not been a problem to this date, and it is not expected to be a significant limitation at any time in the future.

The structure which allows the IFU to perform the address calculation as described above is shown in Figure 5.13. We note that a complete 20-bit adder is used to generate the proper memory address from the F register value and the PC offset counter value. The address generated is 18 bits in length, although both register values are only 16 bits in length. This results from adding the F register value, shifted left two places, to the offset register (PC register) value. Because of this addition of the shifted F register value to the PC register value, instruction addresses may be generated in areas beyond the first 65K of memory. This is an important extension of the Lilith memory addressing capabilities.

5.4.2 Instruction Byte Delivery to the MCU

The IFU transfers information to the MCU and the ALU via the BUS under control of the BUS source and destination control signals which are decoded from the microinstruction register in the MCU. There are four source control signals to which the IFU responds. Three of them cause the IFU to gate the next instruction byte onto the BUS with a zero value in the high order byte. The fourth causes the IFU to get the next instruction byte onto the BUS with a byte of "ones" in the high order position, thus creating a negative value. Although similar in this respect, the four source control signals have other very different effects.

Src=IR4.L:

This signal is used when the MCU is fetching the first byte of an instruction. The assertion of this signal causes the "jump map" ROM in the MCU to be enabled to give the starting address for the microcode program segment which interprets the M-code. This signal also causes the ALU to extract the least significant four bits of the byte and to store it in a temporary register in case the M-code happens to be one of the special format M-codes with imbedded 4-bit constants.

Src=IR8+.L:

This is signal is used whenever a second or third byte is expected in the interpretation of an M-code, typically as an operand or constant. The IFU gates the value "positively" by giving zeros in the high order byte.

Src=IR8-.L:

Whenever a second or third byte is needed with a negative value, this control signal is used because it causes the value to be gated "negatively" with ones in the high order byte.

Src=IR8*.L:

Whenever a branch condition is detected and the microprogram must fetch a another byte from the IFU before loading a new address into the offset register, this control signal will be used in fetching the last byte needed. This control signal performs exactly as does Src=IR8+.L, with the exception that it will not cause the control circuitry of the IFU to automatically fetch a new instruction block in the case that the byte fetched was the last of such a block. This prevents the needless fetch of an additional block of instructions when a reload of the offset register is impending anyway.

5.4.3 MCU Loading of the Frame and Offset Registers

As shown in Figure 5.13, the F and Offset registers are both readable and writable via the BUS. The necessity to write them has already been explained as a part of the function of giving the IFU the necessary address values for the correct computation of the program counter address (see Section 5.4.1). The BUS destination signals which control the loading of these registers are Dst=F.L and Dst=PC.L.

5.4.4 F Register Values and Byte Selection in the IFU

As mentioned previously, the F register is shifted by 2 bits when added to the PC. This gives an 18-bit address as the program counter when selecting the memory in byte increments. Since the word size of the memory read operation used by the IFU is 64 bits or 8 bytes, the least significant 3 bits of the effective instruction address are ignored by the memory. These three bits of the program counter are used by the IFU to the select the correct instruction by from the block of eight fetched bytes which have been loaded into registers from memory. A decoder circuit generates enabling signals for the output drivers of the registers as shown in Figure 5.5. These signals enable one of the eight registers to gate their contents to an internal IFU bus for instruction bytes. An additional three-state bus driver circuit gates the information from this bus onto the main processor bus (called BUS) when the IFU is selected during a microinstruction cycle.

As an effect of being shifted by two bits, the low order two bits of the F register are forced to be zero. This requires that a code frame begin at a *byte* address evenly divisible by four. The program which loads modules into memory is affected by this hardware quirk in that it must at times skip a word of memory in the loading process to reach an even word address for the start of a module.

5.4.5 MCU Interaction with the F and Offset Registers

As shown in figure 5.5, the F register and the offset register are both readable and writable from the BUS. The necessity to load these registers from the BUS has already been discussed in the explanation of how the IFU receives the values which it uses to calculate the program counter value for instruction fetches. The necessity to be able to read information from these registers is based upon the necessity to be able to completely save the state of a program in execution whenever a co-routine transfer takes place, as in the case of an interrupt response. In such a circumstance, the processor gates information from the F register and the offset register onto the BUS, using the BUS source control signals, Src=F.L and Src=PC.L. These signals are the complement signals to those used for loading the registers, namely: Dst=F.L and Dst=PC.L.

5.4.6 Instruction Fetches in Tight Loops

The IFU control logic loads its 8 instruction byte registers whenever it is necessary because all the bytes of the previous group of eight have been used, or whenever a program branch has taken place. It is possible that a very tight loop or a conditional statement could create a condition where an instruction sequence of a program might branch backward or forward but remain within the current block of instructions. Thus, detection of such a condition could save an unnecessary memory reference. Our design of the IFU does not, however, detect such a condition although we gave consideration to the possibility. It proved to be too difficult, or else we were not sufficiently clever, to discover a practicable mechanism to detect this and thus save an unnecessary memory cycle. Consequently, the simple design of this IFU initiates a new instruction fetch memory operation whenever the last byte is taken and whenever a program branch occurs.

5.4.7 Expansion of the Usable Addressing Space for Program Storage

The current implementation of the IFU has the F register shifted only 2 bits with respect to the PC offset. This gives only an 18-bit byte address. As a consequence, the current IFU can address no more than 256 K bytes or 128 K words. This limitation could easily be extended, however, by increasing the amount of shift in the relation at the adder of the F register and the offset (PC) register. This would have a relatively minor effect on the software; it would only require an adjustment in the loader to generate a proper value for the F register, based on a greater number of low order zeros in the code frame base value. It would also demand that each module begin in memory on a quadruple or octal word boundary, rather than merely a double word boundary as is now the case.

5.4.8 Calculation of the Actual Memory Address for an Instruction

When using the debugger program, a programmer may wish to inspect the actual M-codes as they are in memory for a given section of Modula-2 programming. In the compiler output listing for a module, the programmer will find offset register values given next to each compiled line. In the process descriptor, or else in the global data area, the programmer can find the base value to be loaded into the F register. How, using these two values, can the programmer determine the actual 16-bit word address for the M-codes compiled for a line of program? The answer is relatively simple. Since the F register value is shifted by two bits, one should begin by multiplying the hexadecimal F register value by 4, which yields the byte address for the start of the code frame. Then, this value can be added to the offset value given for a line of program to determine the real byte address for the M-code program segment for that line of code. Finally, the byte address must be divided by two to give the actual word address for the location of the M-codes in memory.

5.5 Memory Subsystem

The standard Lilith memory is organized as a 64-bit wide by 32k memory for read operations and as a 16-bit by 128k memory for write operations. Through additional multiplexing of the 64-bit read cycle output, the memory can be made to appear as 16 bits wide for the CPU Data Port or external peripheral devices. The addressing space of the memory will allow twice that amount of memory, but the current hardware implementation does not have room for the additional memory cards.

The various units of the computer access the memory through interface ports, and they compete for memory cycles according to a priority assigned each port. The highest priority is the display processor, followed by the CPU Data Port, the IFU, and lastly, the port assigned to the memory refresh circuitry which assures that read cycles are given as needed to addresses in order to assure that the dynamic memories retain their data.

The memory operates from its own control logic continuously and independently from the processor control. This includes the operation of the port priority arbitration circuitry which is enabled by the memory control logic that also generates the timing signals for the dynamic memories. This memory cycle control also handles the transfer of data between the port interfaces and the memory unit coordinated according to a special data transfer protocol which all ports follow.

5.5.1 Memory Busses (Figure 5.15)

The memory has three busses which are used in the process of completing a memory cycle for a port. The first bus is the 18-bit memory address bus, which is used solely for the transfer of the desired address from the port to the memory. The second bus is the memory data bus, which is 16 bits wide and is used for the data associated with a 16-bit read or write cycle. For read cycles, the data bus is driven by the memory; for write cycles, the port drives the bus. The third bus is used for read cycles only. This bus is 64 bits wide and carries four words of memory data corresponding to the four words addressed when the two least significant address lines are ignored.

5.5.2 Memory to Interface Transfer Protocol (Figure 5.16)

Data transfers between the memory and ports are synchronized by the use of seven control signals. When a port initiates a memory cycle it asserts a low logic level on a request line. This line is separate for each port, i.e. IFUReq.L. When the priority arbitration circuitry selects a port to receive the next cycle, it does so by asserting high a selection line. There are separate selection lines for each port, i.e. IFUSel. The high signal on the selection line causes the port to give the address for its memory cycle onto the memory address bus, and it also gives two other control signals which are bussed in a tristate manner: R/W' and

64/16'. R/W' tells the memory cycle control whether a read operation or write operation is being requested. 64/16' tells the cycle control whether the operation is a 16-bit read or a 64-bit read. This signal is meaningless for write cycles because the memory can write only in 16-bit words. If a write operation has been selected, the memory cycle control also asserts at a high level the signal *GateReq*, which causes the port to place its data for writing onto the memory data bus. If a read operation has been selected, then the memory will send a signal ***DataStrobe*** which will change to a low level at the moment when the port should sample either from the 16-bit memory data bus or from the 64-bit memory data bus, depending on which port is active. Sometime during the middle of the cycle, the memory cycle control will assert the signal *ClrReq* at a high level in order to reset the request line for the port which has been allocated the current cycle. This signal comes before the actual cycle is completed so that the request will be cleared within the cycle request priority arbitration circuitry, allowing another request too be acknowledged as soon as possible. The port selection signal will not be affected by the request line changing back to a high level.

5.5.3 Memory Chips

The memory chips in the Lilith memory are 16k dynamic MOS memory chips. These integrated circuits operate based on a simple scheme of receiving 14 address signals time multiplexed through seven pins and strobed by two separate strobe lines: RAS (row address strobe) and CAS(column address strobe). RAS occurs before CAS (Figure 5.15). At the time of CAS, the R/W' signal is also sampled which determines whether the single bit cell addressed within each chip either delivers its data for reading, or accepts new data in a write operation. Because these are dynamic memory chips, special circuitry is included in the memory to cause at least one read cycle to each of the possible row address combinations every two milliseconds. This prevents the loss of data through charge leakage.

5.6 Input/Output Decoding Logic

5.7 Display Processor

The display processor has the function of creating the necessary signals to drive a monochromatic display in a raster scan mode. It displays an image constructed from black and white points whose values are taken from a contiguous area of main memory with a direct correspondence between the bits of the words in memory and the points on the face of the screen. The display processor block diagram is presented in Figure 5.18. The display processor timing diagram is shown in Figures 5.19A and 5.19B.

5.7.1 Display Dimensions

The standard monitor used with a Lilith is 15 inches diagonally which, with the display controller of the Lilith, can display as many as 768 points horizontally on 594 lines. The frequency of the horizontal and vertical scan rates are the same as for the European CCITT standard for black and white television. The image is scanned in an interlaced fashion, meaning the odd lines are painted on the screen during one vertical scan and the even lines are painted in the next vertical scanned. The vertical scan rate is 50 times per second with a total image scan rate of 25 times per second. The horizontal scan rate is 15,000 scans per second. A relatively slow green phosphor is used for the display to prevent annoying flicker in the image.

5.7.2 Memory Refresh of Display

As many as twelve 64-bit memory words (48 sixteen-bit data words) per line can be used from memory to paint as many as 594 lines in the display image. It is not, however, necessary to use that much memory as the portion of the image containing memory data can be reduced from twelve 64-bit words per horizontal scan to as few as one or even none. Likewise, the number of vertical lines on the screen actually refreshed from memory can be reduced from 594 to any lower number including none. A total of 28,816 sixteen-bit words of memory can be used to refresh the display image at the maximum. Every 5 usecs during a horizontal scan line the display processor fetches from the main storage a single 64-bit word which is then painted on to the screen, allocating 70 nsecs to each of the 64 bits. Double buffering is used so that a word can be fetched while another is being displayed. The 64 bits, representing four 16-bit processor data words, are painted from left to right in the following sequence: word 0, bits 15,14,13....1,0; word 1, bits 15,14.....etc. Each adjacent group of 64 bits has a memory address one location higher than the one previously painted, to the left of it on the screen. The last 64-bit word painted on the right of a screen line has a memory address one location lower than the first 64-bit word on the left (or beginning) of the next screen line.

The final correspondence between bit values in the memory word and bright points on the screen is subject to two optional levels of inversion; one of them controlled by a switch on the display processor card, and the other is programmable and stored in a flip-flop. For the normal mode of viewing on the

Lilith, which is dark text on light background, a "1" bit in the memory corresponds to a dark point on the screen. Figure 5.20 shows the screen characteristics.

5.7.3 Bitmap Descriptor

When an image is to be displayed on the screen, the program must, of course, prepare the actual image in an area of memory by creating patterns of ones and zeros which are not numbers or vectors, but rather one-to-one mappings of the actual image which will be displayed. In order to instruct the display processor how to display the image (meaning how large and from what area of memory), the program must prepare a data structure know as a bitmap descriptor. The bitmap descriptor consists of four 16-bit words and has the following format:

First word : starting memory address for the image

Second word: number of 16-bit words displayed per horizontal scan

Third word : number of vertical lines displayed in image

Fourth word: three items of information packed into this word:

- a) image reversal bit
- b) horizontal offset of image
- c) vertical offset of image

Finally, to cause the actual display operation to begin, the program must pass the address of the bitmap descriptor to the display processor. At the level of programming in Modula-2, this will be seen as a PUT operation to the input/output device which has the address 0.

```
PUT(DisplayAddress,ADR(BitMapDescriptor);
```

At the microcode level, the address of the display processor, which is zero, will be loaded into the *I/O address register* and the data (the address of the bitmap descriptor) will be placed on the BUS from the top of the stack, and the BUS destination will be selected by the signal Dst=IOData.L. This will allow the display processor to sample the address from the bus into a register. Once the display processor has the address of the bitmap descriptor, it will fetch the descriptor during every vertical retrace and use the information to determine the image which is displayed.

5.8 Disk Interface

A secondary storage device in the context of a workstation system such as the Lilith has several functions to perform. Primarily, it provides the permanent residence for all programs, especially during periods when the machine is not in use and power is turned off. It also provides an interim storage for a program's data when the data cannot be held in memory. In addition, it is used to create duplicate files of both program and data for backup and sharing. Because of these capabilities, a disk quickly becomes a very important part of the machine to a user. The Winchester has platters which are not removable, which means that if the computer ceases to function, the user is paralyzed until his machine is once again operational. Therefore a floppy disk back-up is provided in the Lilith work station to enable the user to remove his floppy disk and continue his work at another staation

5.8.1 Performance Specifications of Winchester WD1001-05 Drive

To be added

5.16 Diagnostic Processor Unit

The Lilith is delivered with a circuit board which is not installed in the machine during normal use. This board is called the diagnostic processor unit (DPU), and its function is to isolate faults in the circuitry or in the microprogram. It does this by asserting control over the machine and causing the controlled execution of Lilith operations followed by immediate analysis of the operation to determine the correctness of each microprogram step.

In earlier computers a panel of lights and switches was built into the machine for such a purpose, i.e. to monitor the execution of instructions in the machine. As this unit has been used in the development of the Lilith, we, the designers generally referred to this card as the "monitor" card. One segment of the internal software was referred to as the "panel" software because our usage of this card to simulate the front panel of older computers. The name Diagnostic Processor Unit is new and was adopted to more correctly designate the function of the card. The DPU has no switches or lights, but rather consists of a Motorola 6802 microprocessor, associated circuitry, and a good deal of sophisticated test software which exercises the machine. This microprocessor communicates with the maintenance engineer via a standard RS-232 serial interface. By means of the serial interface, the maintenance engineer can conduct a dialog with the processor and pass binary files to be loaded into the microprocessor RAM memory. There is a selection of programs which can be loaded depending on which part of the machine one would like to test.

In order for the DPU to adequately test the Lilith, it was necessary to build a good deal of circuitry onto the DPU board for the porpose of sampling and controlling the key portions of the machine. Furthermore, the Lilith design itself required the inclusion of special control signals to give the DPU the capacity to disable certain Lilith functions and replace them with functions provided by the DPU.

The capabilities of the DPU are as follows:

It can read the microinstruction register of the Lilith processor and assert its own microinstructions if it desires.

It can read data from the CPU bus and be selected to drive data onto the BUS.

It can read addresses and assert addresses on the microprogram address bus controlling the microprogram memory.

It can control the generation of clock pulses in the CPU, i.e. it can stop them altogether; it can allow a clock pulse to take place; it can allow a certain number of them to occur; or it can release control, allowing full speed generation of clock pulses. A block diagram for the DPU is given in Figure 5.22.

With a little imagination, one can envision how the foregoing list of capabilities enables the DPU to exercise and test the performance of the Lilith quite thoroughly. Most of the tests are considered to be "static"--meaning not at full speed. But it is even possible to test the dynamic as well as the static characteristics of the machine by using the DPU's capability to generate a specific number of clock pulses at full speed. The full utility of the DPU will become apparent as examples of its uses are presented.

6 Circuit Description

6.1 MCU Circuit Descriptions

In Section 1 of Chapter 5, the nature of the microprogrammed computing engine, which is the heart of the Lilith, was described. Details were given concerning execution of the microinstructions, selection of the next microinstruction as determined during the execution of each instruction, and timing signal generation for each microinstruction cycle. The decoding of the microinstruction bits which control the transfer of data over the BUS was also discussed. Now each of these functions will be discussed again in relation to the actual circuitry performing that function.

6.1.1 Clock Generation Circuitry (Schematic MCU 3/5)

Looking at the block diagram for the MCU (Fig. 5.1) we see that the sole function of the clock generation circuitry is to generate two clock pulses: MCUClk.U and CPUClk.U. The MCUClk.U is the synchronizing timing pulse for every microcycle. The CPUClk.U, which has the same timing relationship as the MCUClk.U, is the synchronizing timing pulse for each microcycle which controls ALU operations in Format I and Format II microinstructions. The signal MIR14 from the microinstruction register is true (asserted high) for such ALU instructions and is used to enable the CPUClk.U signal through NAND gate U54b (schematic MCU3/5). The MCUClk.U and CPUClk.U clock pulses originate from pin 8 of flip-flop U55b. This flip-flop, together with the other flip-flop (u55a) in the same package, is part of a state machine which operates at a master clock frequency of 13.33 Mhz. This is twice the fundamental frequency of the MCU and CPUClk.U signals. Most of the time, this state machine creates an MCUClk and a CPUClk pulse every 150 nsecs. Sometimes, however, the state machine must delay the clock pulses one or more of the master clock cycles. There are several conditions which control these delays:

First, certain instructions found to be slower in execution are marked in the microinstruction word through setting bit MIR15 for execution times of 225 nsecs or three master clock periods.

Second, microinstructions which reference either the CPU Data Port or the Instruction Fetch Unit may find one of these two units unready to respond to the microinstruction, and the clock pulse must therefore be delayed until the selected unit is ready.

In the third case, when the machine is running with the Diagnostic Processor Board (DPU), the DPU may exert its control over the MCU to stop the clock altogether, or, by using the signal CPUClkDis.L, the DPU may arbitrarily selectively disable the CPUClk, regardless of the microinstruction in execution.

The state diagram for the clock generation state machine is given in Fig. 6.1. Chip U56a, a 74S153 dual 4-bit multiplexer, is used to create the next state for the state machine. Normally, the state machine cycles between states 00 (first half of clock) and 10 (second half of the clock), thereby giving the signals MCUClk and CPUClk a period of two master clock cycles. When the signal MIR15 is true, the multiplexer, U56, together with gates U16, pin 6 and pin 11, alters the state machine sequence to: 00 to 01 to 10. This sequence expands the period of the MCUClk and CPUClk signals to three master clock cycles.

(Note: The complement outputs of the flip-flop, U55, are used as state variables. Therefore, the D inputs to the flip-flops should be regarded as complement inputs.)

The signal MemRdy, which also controls gates U16, pin 6 and 11, was considered to be high for the previous analysis. This signal is high whenever the microinstruction has not caused selection of one of the memory interfaces before it is ready to respond. This signal originates from the circuits U22 and U21 in schematic MCU 3/5. These circuits logically create MemRdy from the two status signals, DataPortRdy and IFURdy, (which are high or low according to the state of the respective memory interface) and from the following microinstruction select signals: Src=IR8-.L, Src=IR8+.L, Src=IR4.L, Dst=PC.L, Dst=IR8*.L, Dst=F.L, Dst=MO.L, Dst=MAR.L, and Src=MD.L. One of the preceding signals will be asserted low whenever the microinstruction selects either the CPU Data Port or the IFU. The combination of any memory port selected when it is not ready causes MemRdy to be asserted low.

It is worthwhile to discuss the critical timing relationship of the MemRdy signal to the operation of the clock generation circuitry. An especially long path of gate delays is involved in this circuitry, which affects the maximum operational frequency of the computer. This path is critical when the executing microinstruction selects a memory port which is unready. In this case, the proper value for the MemRdy must propagate to the next-state generation circuitry for the clock generating state machine in time to alter the normal clock sequence. The circuitry involved in this chain is:

1. The master clock resets state flip-flop Q1 of the clock generating state machine.
2. The output of this flip-flop transitions to a one after a normal flip-flop propagation delay.
3. MCUClk.U changes to a high after the delay of the 74S140 power buffer driver, U54c.
4. The microinstruction register clocked by the MCUClk.U switches to the configuration of the new instruction after a flip-flop delay.
5. The BUS source (or destination) decoder selects one of the memory

ports after the time required by a 74S138 decoder circuit to select the correct output based on the MIR bus control logic.

6. The clock delay circuitry propagates the signal after the delay of two logic levels in the 74S64, U22, yielding a low asserted MemRdy signal.
7. The next state for the state variable flip-flop is selected low after the delay of a gate from U16, pin 6 and 11, and after the delay of the 74S153 multiplexer, U56.

If the total of these delays is longer than required to provide adequate setup time at the input to state variable flip-flop Q1 before the next master clock, then the processor will proceed to send (or receive) new data to (from) the memory port before it is ready to function.

The DPU asserts its control over the clock generation circuitry through the circuits U18e, U17b, and U17c, as shown in the schematic MCU 3/5. Three signals, Run, SSClk.U, and Reset.L, cause the output of U13b, pin 6, to appropriately control the asynchronous preset and reset inputs of the state machine flip-flops in U55. Reset.L is the strongest of these signals. If it is low, the clock will be stopped immediately and held in the asserted low state, which is the state for the second half of the clock cycle. When Reset.L goes high, the state machine also changes, giving a clock pulse in unison with the end of the Reset.L low state. This clock pulse initializes the microinstruction address sequencer to a value of zero.

With the Reset.L signal in the high state, the signals Run and SSClk.U control the functioning of the clock circuitry. Of these two, Run is the superior. If it is in the high state, then the clock generator will operate at full speed. When it is in the low state, then SSClk.U can operate, causing a single clock cycle for each upward transition on SSClk.U.

6.1.2 Microinstruction Register and Control Store ROMs (schematic MCU 1/5)

The MCU circuit board has positions for ten ROMs for the control store. So far, two varieties of ROMs have been used in the Lilith. During the development phase, 2k by 8-bit MOS EPROMs--erasable ROMs--were used to test the microprogram. In the production version, bipolar fusible link ROMs have been used both in 1k by eight and 2k by eight sizes. All of the ROMs used for the control store have similar pinouts, but some pins require different signals according to the ROM type. In order to accommodate the various types of ROMs, the MCU has selectable jumpers to route the correct signals to these pins. The proper jumper configuration for the 1k by eight and 2k by eight type ROMs is provided in

Appendix 1 of this manual.

The ten onboard control store ROMs are organized as a 5 by 2 array, meaning that a bank of five of the ten ROMs are enabled during any given microcycle, and a total of forty bits are read in parallel from the control store to the microinstruction register (MIR). The enable signal for each of the groups of five ROMs is one of the signals programmable by jumpers. When 1k ROMs are used, signal A10 from the microinstruction address lines is used; for 2k ROMs, signal A11 is used. ROMs U6 through U10 are selected if the enable signal is low. If the enable signal is high, then the inverter U18a, pin 2, generates a low enable signal which selects ROMs U1 through U5.

Each leading edge of the MCUClk transfers the next microinstruction from the outputs of the ROMS to the microinstruction register. The microinstruction register is composed of type 74S374 octal flip-flops having three-state outputs (U25..U31). The enabling of the three-state flip-flops is handled by a special circuit consisting of flip-flop U15b, pins 8 and 9, and NAND gate U14c, pin 8. The function of this circuit is to disable the onboard MIR in two special situations. The first situation is for the use of writable control store. The second situation occurs when the DPU is used to intervene in the operation of the processor by disabling its MIR and substituting a microinstruction from a similar register in the DPU.

In the first situation, the value of the microinstruction address signal A11 is used to indicate whether or not the onboard ROM and MIR will be used. Since this signal is valid one cycle before the microinstruction will execute (during the period when the instruction is being fetched from the control store ROM), it is necessary to capture and delay this signal by use of the flip-flop, U15b, pin 12. A similar circuit is designed into the writable control store. The final result is that the microinstruction register loaded from the control store unit selected by microinstruction address during the fetch portion of the cycle will be the one enabled during the subsequent execution portion of the microcycle. The reader has probably noticed that a complete microinstruction register is duplicated on the writable control store board. He may have asked why the microinstruction address was not simply used to disable the control store ROMs ahead of the microinstruction register by connecting the offboard instruction to the inputs of the onboard microinstruction register. The answer lies in the need to conserve the number of signals running off the circuit board through the card edge connector.

The second situation in which the microinstruction register is disabled only occurs when the processor is stopped and the Diagnostic Processor Unit has assumed control over the elements of the Lilith by providing the microinstructions for execution from its own microinstruction register. In such a case, the DPU asserts a low value onto the signal, DisMIR.L, which enters at the AND gate, U14c, and, via pin 8, disables the onboard microinstruction register irrespective of the microinstruction address signals. This signal has a pullup resistor to assure a noise free high value when the DPU is not inserted into the machine.

6.1.3 Microinstruction Address Generation (schematic MCU 2/5)

During the execution of every microinstruction, a background operation is taking place to read from control store the microinstruction which will be executed during the following cycle. The selection of this microinstruction is determined by the value of the microinstruction address bus (UA0--UA11). This bus is driven at all times by the outputs of three AMD 2911 circuits, U49 through U51, as shown in schematic MCU 2/5, except when the Lilith is being controlled from the Diagnostic Processor. In this circumstance, the signal Disable2911.L is asserted low by the DPU, which results in a disabling signal at pin 16 of all 2911s. The DPU then drives the microinstruction address bus arbitrarily from its own register for diagnostic purposes.

In the normal mode of operation, the microinstruction address may be generated in many different ways for a variety of purposes. A discussion of why the different ways are necessary was given in the MCU theory of operation section (Section 1, Chapter 5). Now we wish to describe how circuitry selects the microinstruction address according to the various situations.

First situation: microinstruction address equals last address + 1

This is an internal 2911 function. In every cycle, the 2911 computes the value of the microinstruction address incremented by one and stores it into an internal register. This happens in every cycle irrespective of how the microinstruction address is developed because even though the microinstruction address may originate externally, it must pass through the 2911 circuits in order to reach the control store. An instruction which will not change the sequence of microinstructions with a branch specifies the use of an incremented address equal to the last address plus 1. It does this by giving the appropriate code to the 2911 control field in the microinstruction. The 2911 control field bits are MIR16, MIR17, and MIR18. They pass through multiplexer U37. Pins 4, 7, 9, and 12 of multiplexer U37 reach the 2911 input control ports at pins 20, 19, 10, and 11. The signal which orginates from MIR17 and passes through pin 11 of multiplexer U37 is used for two input control signals of the 2911, pins 10 and 20. According to the document for the 2911 from AMD, one may see that this control configuration selects the incremented value of the previous address.

Second situation: microinstruction address taken from previous instruction

This is one of three situations in which an address originates in parallel from a source external to the 2911. In this case, the three-state octal buffers, U28 and U29, gate a 12-bit address to the parallel input lines of the 2911s, pins 4, 5, 6, and 7 of chips U49, U50, and U51. The address is gated from the most significant 12 bits of the microinstruction register (MIR 28 .. 39). The control field of the microinstruction for the 2911s, bits MIR 16 through MIR18, are given the value: 111. This value propagates to the 2911 in the same manner described in situation one and results in a command to the 2911 which gates the incoming parallel address directly to the 2911 outputs and to the control store. The microinstructions which use this mode operate conditionally according to tested values of condition codes. The selection of condition codes to control the jump are determined by bits in the microinstruction, MIR 16 .. MIR18. When a tested condition does not select the jump, the 2911 control signals are switched in the 74S157 multiplexer, U37, to the alternate inputs, pins 3, 6, 10, and 13, which are hardwired to give the "continue" control code described in situation one. As a final comment, please note that this situation is used by both jump and jump subroutine intructions. The only difference between the two types of instruction is that the JSR instruction specifies, in the 2911 control field, both the jump operation and the saving of the current "next microinstruction address" on the stack internal to the 2911. Whereas the JUMP instruction merely specifies a jump operation.

Third situation: microinstruction address taken from the M-code map ROM

This is the second of the situations in which the address originates external to the 2911 and is gated through the 2911 to the control store. In this case, the desire is to quickly determine which microinstruction address is correct for the starting point of an M-code interpretation. The address originates with a Format II microinstruction selecting an opcode from the IFU to be gated onto the BUS by the signal Src=IR4.L. The BUS data in turn provide the 8-bit M-code to a 256 by 12 bitmap ROM constructed from three 82S129 circuits, U38, U39, and U40. A signal, MapEnable.L, which references the IFU in this case (see schematic IFU1/3), is used to control the gating of the map ROM outputs to the 2911 inputs in a similar fashion to that described in the above situation for the normal branch microinstructions. From this point everything is as described in situation two.

Fourth situation: microinstruction address taken from interrupt vector address

Each time the Lilith begins the execution of a new M-code, the interrupt logic has an opportunity to subvert the table jump for the M-code and to substitute in its place a direct jump to an absolute address which is determined by the highest priority unmasked interrupt. This only occurs on microinstructions which are attempting to start the interpretation of a new M-code using the signal Src=IR4.L. In this situation, the interrupt signal coming from the MCU board is gated into holding register U45 (see schematic MCU 5/5) and passes through masking gates in either U36 or U46. If the corresponding mask bit of the mask register, U35, has not been set, the interrupt request will reach the priority encoder, U47, of the MCU, which creates a binary address corresponding to the highest active interrupt. The signal Req, from pin 15 of priority encoder U47, operates through gate U24a, pin 6 (schematic MCU 4/5), to abort the transfer of an opcode from the IFU by disabling MapEnable.L. Three-state octal buffers U48 and U29 (schematic MCU 2/5) gate the vector address from the priority encoder plus 8 (instead of a map address from the map ROMs) to the inputs of the 2911 and, in the same manner as described before, to the control store.

Fifth situation: microintruction address taken from the internal stack of the 2911

When a jump-to-subroutine microinstruction is executed, the address of the following instruction is stored onto an internal stack of the 2911. Later, when it is time to return from the subroutine, it is only necessary to give the 2911 control field of the microinstruction the value "001," which will cause the 2911 to select its next address from the value stored on its internal stack.

6.1.4 BUS Control Decoded from MIR (schematic MCU 4/5)

Eight bits (MIR 0-7) of the Format I microinstruction word are used to specifiy how information will be transferred over the 16 data lines known as the BUS. Four bits (MIR0-3) provide the possibility to select one of 16 devices as the source of data on the BUS, and four bits (MIR4-7) select one of 16 devices to receive the data from the BUS. Rather that have each unit look for its own address on four binary lines, 74S138 decoder circuits U33, U43, U32, and U41 are used to decode the two 4-bit selection codes into a total of 32 device selection signals. Each of these device selection signals has a value of zero when selecting a device; otherwise, they remain high. Sixteen of these signals select BUS source devices and sixteen select BUS destination devices. The source versions of these signals simply command the addressed device to gate the contents of its register onto the BUS for as long as the period of a cycle. The destination versions of these signals work cooperatively with the CPUClk causing the data sampled from the BUS to be transferred to a register whose selection and strobing are created from a combination of the BUS destination selection signal and the clock signal, CPUClk.U.

The microinstruction bit which enables the functioning of the decoders is MIR12. When this signal has a low value, it enables the four 74S138 decoders of schematic MCU 4/5 through pin 4 of each of them. Additional special circuitry in the form of U13c, output pin 8, is used to handle the special case of Dst=ALU.L. This circuitry forces the Dst=ALU.L signal low so that the ALU is implicitly selected as the destination for the BUS data when the microinstruction has Format II. As explained in Section 1.4 of Chapter 5, Format II is the format used when constants are gated to the BUS from the microinstruction register.

It should also be noted at this time that the microinstruction bits controlling the BUS source and destination decoders can have completely arbitrary values when a microinstruction of Format III is executed. This can lead to spurious outputs on these selection lines. However, since no CPUClk signal is given during the execution of a Format III instruction, this should not create any erroneous operation. Any augmentation of the Lilith circuitry should take this fact into account and should be designed to be insensitive to transitions on these signals except when CPUClk is present.

6.1.5 Generation and Use of BUS Data by the MCU

There are several instances where circuitry of the MCU either gates data onto the BUS or transfers data from the BUS into a register. One of these instances occurs when a constant from the MIR, bits MIR0..MIR7, is gated onto the BUS (see schematic MCU4/5). MIR12 is the signal which controls this operation . Since the signal is needed in an inverted form, it first passes through U18f, whose output at pin 12 becomes MIR12.L. In this form, MIR12 directly controls the operation of the octal buffer, U44. This buffer transfers the 8-bit constant, MIR0..MIR7, to the lower half of the BUS, thus providing the signals BUS0..BUS7.

When MIR12 forces constants to be gated onto the lower half of the BUS, the upper half of the BUS must be forced to zero. Circuitry consisting of inverting octal buffer U42 and gates U24b and U17a perform this function. There are four other conditions (see Section 4.2, Chapter 5) in which this circuitry is used. In each of these conditions, which are associated with the operation of the IFU, eight bits are provided for the lower half of the BUS.

In three of these conditions (indicated by IFU selection signals Src=IR4.L, Src= IR8+.L, Src=IR8-.L and SrcIR8*.L), chips U42, U24b and U17a provide zeros to the upper half of the BUS . Chips U17a and U24b together function as a five input inverted logic OR operation, enabling the inverted buffer, U42, for any of the three IFU selection signals or for MIR12.L. The fourth IFU selection signal, Src=IR8.L, reads a negative value from the IFU ,and consequently ones must be gated to the upper half of the BUS. This is accomplished by the connection of the Src=IR8.L signal to the buffer input pins of U42. This function actually belongs to the operation of the IFU, but for reasons involving simplification, it was included in the circuitry of the MCU.

Schematic MCU 5/5 shows how the interrupt mask register, U35, can be loaded and read from the BUS, using the BUS source and destination signals taken from the decoders of schematic MCU4/5. The BUS control signal, Dst=IRM.L, connected to pin 1 of U35 enables the loading of the register from the BUS. The CPUClk.U, which is connected to pin 11 of U35, actually strobes the data into the register. The outputs of U35 are gated to the BUS by octal buffer U34 under control of the signal, Src=IRM.L, entering via pins 1 and 19.

6.2 Arithmetic/Logic Unit

The complexity of the ALU circuitry probably exceeds that of any other board in the Lilith. It has an enormous number of different operations and a large number of control signals coming from the MCU to determine its operation. However, the sequential timing of the ALU follows a very simple cyclic pattern, which eases the burden of describing the functioning of its circuitry. The cyclic pattern begins with the stabilization of control signals from the MCU shortly after the beginning of a microcycle. During the period of time allocated to the cycle, the control signals cause the manipulation and alteration of data from the registers in the ALU and from the BUS to which it is connected. At the end of the cycle, data created during the cycle is saved into registers within the ALU and on other boards for use in the next cycle. Every section of circuitry in the ALU can therefore be described completely by considering only the possibilities for a single microcycle.

The block diagram of the ALU is given in Fig. 5.7. The major components which compose the ALU are the shifter, the stack, the 2901 bit slices, and the condition code logic. Each component will be discussed in turn.

6.2.1 The Shifter (schematic ALU 1/4)

The first component we will discuss is the barrel shifter. The barrel shifter is connected on one side to the BUS and on the other side to a second bus which is totally contained on the ALU circuit board. This second bus, which is called the D-bus or Direct Operand BUS, connects to the direct inputs of the 2901 arithmetic/logic units. The outputs of the evaluation stack also connects to the D-bus.

The shifter has the prime responsibility to pass information from the BUS to the inputs of the 2901 bit slices. Normally, the shifter will pass the data directly to the 2901 ALU, but under special command from the MCU, it can shift the data from the BUS in an "end-around" fashion so that the data can be rotated to any of the 15 other positions within a word. The shift is actually a true rotate, so that zeros are not brought into any portion of the word, and none of the bits are lost. The shifter is constructed from special AMD circuits (part number 25S10) designed to perform shifting operations. Each of these circuits accepts seven bits as input but outputs only four. The four selected bits are taken in a shifted fashion according to two shift control signals entering pins 9 and 10. In the ALU, the direction of shift is towards the least significant bit of the word, although the circuits could be used equally well in the opposite direction. Because each circuit can select at most four shift possibilities, two levels of logic constructed from these circuits are required to give the total of 16 shift possibilities. The organization is such that the first level, comprised of circuits U33, U34, U35, and U36, performs a shift of 0, 1, 2, or 3 bits. The second level is connected such that shifts are always in groups of four, with this level offering the possibilities of 0, 4, 8, or 12 shifts. The circuits comprising the second level are U15, U16, U17, and U18.

In operation, the control of the shifter is straightforward. A four-bit shift amount is presented by multiplexer u38 to the shifter. The least significant two bits of this count are connected to pins S1 and S0 of the first level of shift circuits, and the most significant two bits are connected to pins S1 and S0 of the second level of shift circuits. For any shift operation, the portion handled by each level depends on the value of the two-bit fields given to each level. For a shift of 9 bits, as an example, the first level shifts one bit and the second level uses its second shift selection which is 8.

The source of the shift count can come in two ways. If the shift amount can be selected before execution, then the amount is programmed into the microinstruction word, bits MIR8 through MIR11. If the shift amount is the result of a calculation, then the amount must be transferred from a register or from a variable location into the shift count register, U37. This register is loaded by selecting it as the destination for data on the BUS. The selection control for these two signals is a function of the value of MIR20. If it is high, then the shift count will be taken from the Shift Count Register by a 74LS157 multiplexer, U38. If it is low, the multiplexer selects the bits from the microinstruction register.

The shifter has a feature useful for the generation of masks. This feature is under the control of microinstruction bit MIR19. If this bit is true, then, from the most significant end of the word, a number of bits equal to the shift count will be set true. The OR gates of circuits U24, U25, U26, and U27 perform this function. As an example, if a zero from the BUS passes through the shifter shifted 7 bits with the mask feature enabled, then the mask generated will be 1111111000000000.

The shifter has one other feature useful for executing M-codes having four-bit operand fields right justified in the instruction byte. When this byte is sent from the IFU to the MCU for execution, the shifter responds to control signal Dst=ALU.L and selects these four bits alone for storage in a register (usually the Q register) in the 2901 by forcing the top four bits of the opcode to a zero. Shifter circuit U35 has special logic to disable its output when this signal is active. With the drive circuitry of U35 disabled, zeros are gated to the outputs of this shifter by octal buffer U28. Note that only half the outputs of chip u28 are used. For a better perspective of the value of this operation, the reader is directed to the section discussing M-codes in the chapter on the Lilith virtual machine.

6.2.2 The Evaluation Stack (schematic ALU 3/4)

The evaluation stack is an alternate source of input to the 2901 arithmetic unit D-bus (see Fig.5.7).

Selecting the stack as the input source to the 2901 requires disabling the shifter. The signal selecting between the shifter and the stack is the BUS destination selector for the ALU, Dst=ALU.L. When the ALU is selected as the destination of the BUS, the shifter is enabled and the stack is disabled as input to the 2901. The stack can, however, receive the output of the 2901. When the ALU is not selected as the BUS destination, the stack is automatically selected as the source of data coming into the 2901 from the D-bus.

The evaluation stack consists of four 16-word bipolar RAM circuits which are each 4 bits wide. This gives a total of 16 registers of 16 bits for stack storage plus one register inside the 2901 which is used as the top of the stack, giving a total of 17 levels. The circuits from which the evaluation stack is constructed are 74S189 tristate output random access memories and are in positions U1, U2, U3, and U4. These circuits have separate inputs for write operations which are connected directly to the outputs of the 2901. The outputs of these chips cannot be gated directly to the 2901 because they would conflict with the shifter. Consequently, two inverting octal buffers, U13 and U14, are used to control when the RAM outputs are gated to the D-bus.

In order to unburden the microprogram from selecting which of the registers in the evaluation stack is to be used, a special addressing circuit has been constructed to address the locations of the stack RAM in push/pop fashion. The push instruction requires the address to be decremented to the next unused location before writing the memory. The pop instruction requires the address to remain unchanged until after the data at the current location has been read, and then it must be incremented. (Note: the stack grows downward in the address space of the small ram.) These operations cannot be achieved in a single cycle using a counter alone. The design developed to meet these constraints uses a counter, U6, and an adder, U5, as described below.

6.2.3 The Pop Operation

The pop operation of the evaluation stack is relatively simple. In a microinstruction performing a pop, the first requirement is that the BUS destination does not select the ALU. If that were the case, it would imply that the shifter is active, which precludes use of the stack since the shifter gates its data to the same inputs of the 2901 as the stack. With the BUS control signal, DST=ALU.L in an inactive high state, the stack buffers, U13 and U14, are enabled by this signal inverted by U11, pin 12. The value of the stack address counter, which points to the highest level of the stack in the bipolar memory, is gated to the memory chips unaltered by the adder (U5) E-inputs because the output of the NOR gate U12, pin 1, is zero as caused by the signal DST=ALU.L. Microinstruction bit MIR13 is zero for this pop instruction, which enables the counter to increment at the end of the cycle as controlled by the up/down control input at pin 5 of U6. At the end of the microcycle, the incrementing of the counter causes it to select the next lower element of the stack.

6.2.4 The Push Operation

We will now consider the address generation for a push operation. In this operation, the address generated needs to select the next unused location of the stack memory rather than the location currently addressed by the counter. Since it would require an entire cycle to decrement the stack address counter, U6, it is necessary to perform the decrement using the adder, U5, to momentarily subract one from the value of the counter. At the end of the cycle, the counter is decremented, so the counter ends up pointing to the new top of stack in the RAM memory. The control signals for the counter and the adder are bit MIR 13 and DST=ALU.L. Both of these signals must be low for a push operation. A NOR gate, U12, pin 1, gives a high output in this situation which is used for three functions:

a. Passing through U11, pin 2, it enables U12, pin 10 to generate a write pulse for the 74S189 memories.

b. Added, as 1111, to the stack address counter by the adder U5, it effectively decrements the stack address value to the next unused location.

c. Controlling the stack address counter, U6, through the up/down count countrol, it causes the counter to decrement at the end of the microcycle. The counter after this document will assume the same value that comes from the adder outputs at the very beginning of the cycle.

In this fashion, the stack counter address logic accomplishes the required addressing for the push operation.

6.2.5 Stack Empty Detection

As indicated in the theory of operation section, it is not necessary to test the stack for overflow, but it is necessary to detect the empty condition so that the stack may be unloaded and saved for interrupts, coroutine transfers, and procedure calls. The carry output from the adder serves this function. In every circumstance except the push operation, the "E" inputs of the adder will be zero, hence with the incoming carry wired true, the carry out can only have a true value when the count is "1111." This signal can be tested as one of the condition codes by the jump format of microinstruction. However, it will have valid information only when it is tested in a microinstruction without stack operations.

6.4 Instruction Fetch Unit

The circuitry of the IFU partitions nicely into three components: the instruction address generation circuitry, the instruction holding registers and gating circuitry, and the memory cycle request timing circuitry.

6.4.1 The Instruction Address Generation Circuitry (schematic IFU 2/4)

Two registers, the code frame register (F) and offset register (PC), are key participants in the generation of the actual memory address for the fetching of a block of eight instruction bytes. These registers are loaded from the BUS during a standard microinstruction cycle using the signals Dst=F.L and Dst=Offset.L. Two 74LS377 octal registers,(U8, U11), are used for the code frame register. Four 74LS163 counter circuits (U16, U17, U18, and U19) are used for the offset register. Each of these registers receives its data directly from the BUS at the end of the microinstruction cycle in which it is referenced.

It is possible to read the contents of both the code frame register and the offset register. This feature is used by the processor in co-routine transfers and procedure calls. The 74LS244 Octal buffer circuits, U7 with U10 and U9 with U12, are activated respectively by the BUS control signals: Src=F.L and Src=Offset.L. These circuits gate the information directly to the BUS.

For instruction address generation, the contents of the code frame register and the offset register are summed together by the adder circuitry composed from four 74LS283s (U1, U2, U3, and U4). As described in the section covering the IFU theory of operation, the code frame register is shifted left two bits and added to the offset register for the purpose of extending the addressing range available to be use for program storage.

The summed result generated by the adder just described is gated to the memory address bus whenever the IFU requests a memory cycle. The 74LS244 octal buffers (U5 and U6, schematic 1/4), together with two of a four-buffer package (U23, pin 3 and pin 6 from a 74LS125), are enabled by the output of inverter U15f, pin 12 (schematic 1/4). This circuit gives a low enabling output only when its input, IFUSel, is asserted high, which occurs only when a memory cycle for the IFU is taking place.

6.4.2 Instruction Byte Holding Registers (schematic IFU 3/4)

Eight 74LS374 octal registers (U24, U26, U27, U28, U29, U30, U31,and U32) capture eight instruction bytes with each IFU memory cycle. The timing signal which loads these registers comes from pin 3 of nand gate u14a. This gate gives a low output when its inputs, IFUSel and DataStrobe, are asserted high. The rising edge of this signal causes the data to be loaded into the registers from the 64-bit memory read data bus.

The outputs of these eight registers are bussed together to form the effect of a multiplexer, with the active register selected by a decoder output from the decoder circuit, U20 (schematic 1/4). This decoder circuit is always enabled and decodes continuously the least significant three bits of the program address value, which is generated by the adder circuit discussed in the previous section. Consequently, the next-to-be-selected instruction byte is always available on the internal IFU bus which terminates at the 74LS244 octal buffer, U25. When it is required to gate this instruction byte to the BUS for use by the MCU or ALU, the signal GateIFUData.L is asserted low, thus enabling buffer U25. The logic controlling the generation of GateIFUData.L is described in the next section.

6.4.3 The IFU Memory Cycle Request Circuitry (schematic IFU 1/4)

The source and destination control signals for BUS which reference the IFU are all involved in the special timing circuitry which generates memory cycle requests and delays the MCU microinstruction execution when necessary. Logically, the circuitry's function is relatively simple; the actual implementation obscures this somewhat in order to minimize hardware and timing delays. There are three important signals generated:

GateIFUData.L:

This signal is asserted low any time the microinstruction register selects the IFU as the source to BUS. The logic implemented by U21c and U21b is a negative logic OR.

IncPC:

This signal is the inversion of GateIFUDate.L. It increments the value in the offset register to the next instruction byte address at the end of a microcycle. (The offset register is really a 16-bit counter.)

IFUReq.L:

This signal is generated whenever a new block of instructions is needed. There are two conditions which create such a need: 1) whenever the lastof the eight instruction bytes is taken (except when it is taken by the Src=IR8* signal), and 2) whenever the offset register is loaded by a microinstruction asserting the Dst=Offset.L. Both conditions assert the IFUReq.L signal (u22a ,pin 3) via the IFU memory cycle request flip-flop, because it is in these conditions that a new block of instructions is needed. The signal Sel7.L, connected to u22d, pin 13, originates from decoder u20. It indicates when the last byte of the block is being taken.

The three signals listed above initiate the necessary response of the IFU to the sequences of requests which will come from the processor. One of these responses is the request for a memory cycle from the processor. The timing for this operation is according to the protocol established for memory interaction. This subject has already been discussed in the section describing the operation of the CPU Data Port. The reader is referred to Section 6.3.4 for an explanation of this protocol. He will find a parallel situation described using signal names of uniform similarity, i.e. DataPortRdy, IFURdy, DataPortReq, IFUReq, DataPortSel, IFUSel.

6.5 Memory Subsystem

In the CPU Data Port theory of operation description, the protocol for interaction with the memory was described including the mechanisms for initiating memory cycles for both 16 and 64-bit data operations. In this section, we will first discuss the circuitry which must control the interaction with the memory interface ports and the generation of the necessary timing signals for the array of 16k memory chips. Then, we will discuss the organization of the memory array detailing the mechanism for expansion which allows the control mechanism on the master memory board (64R) to control cycles on the slave boards.

6.5.1 Priority Allocation Circuitry

The priority allocation circuitry, consisting of chips u30-u33 in schematic CDP 1/5, handles the incoming requests for memory cycles from the eight possible ports in the system. It also generates select signals, e.g. DataPortSel, which tell the selected port that the current memory cycle is allocated to that port. The control protocol for this circuitry is very simple. The memory cycle control asserts the GateReq signal high and waits for the priority allocation circuitry to assert the AnyReq signal high. As soon as the cycle control detects that the AnyReq signal is asserted high, it sets GateReq to a low value and initiates the timing of a cycle. When the priority allocation circuitry receives the low value on GateReq, it locks out any further requests and sets the selection signal for the highest priority port request.

The eight memory port request signals, such as DispReq.L, are asserted low when a cycle is being requested. They pass initially through an octal inverting buffer circuit (U30, as shown in schematic CDP 1/5) and are asserted high at the 74278 priority allocation registers, U31 and U32. The signal GateReq from the memory cycle control enters at pin 1 of the two priority allocation registers. The signal AnyReq returns from pin 5 of U31 and goes to the memory cycle control circuitry in schematic MEM 1/4 asserted high when the registers have detected a cycle request.

The selection signals which allocate the cycle request to the highest priority port memory request come from pins 10, 9, 8, and 6 respectively from the two priority registers as shown in schematic CDP 1/5. Because of the connection from pin 5 of U32 to pin 4 of U31, the priority of the four selection signals from U32 is higher than that of the four signals from U31. Four of the selection signals pass through AND gates and are enabled only when the memory cycle is underway as determined by the inverted value of GateReq passing through U21, pin 6. The purpose of these AND gates was initially to eliminate the possibility of driver clash on the Memory Address Bus, which was thought to be causing unreliability problems. At this time they are considered unnecessary, but are still in the printed circuit artwork.

6.5.2 Memory Cycle Control (schematic MEM 1/4)

The timing and control signals for the memory subsystem are generated by a 4-bit 74S195 shift register, U44 on the master memory board, 64R. This shift register cycles through a complex sequence of patterns utilizing both the serial shift mode of operation as well as the parallel entry mode of operation. The idle state has Q3 Q2 Q1 Q0 = 0111 with the register operating in the parallel mode as controlled by Q3 in the low state connected to the parallel entry control, pin 9. When AnyReq is true, indicating the presence of a memory cycle request, Q3 (pin 12) loads to a high state, and the shift register mode switches to serial operation. In subsequent clock periods, zeros are successively shifted in through the serial input port until Q3 becomes zero again and the mode reverts back to parallel entry. At this point the register has a value of 0000, and in successive cycles passes through the state 0000 and 0010 to the inactive state of 0111. The state machine diagram for this sequence is given in Fig. 6.2.

In the process of cycling through these seven states, the memory cycle control creates the two signals which interact with the selected port: DataStrobe, which is taken from the inverted Q0, and ClrReq which comes from Q2. The important aspect of DataStrobe is that the downward transition of this signal should be used by the selected memory port to latch the output of the 16-bit Memory Data Bus (or the 64 bit Memory Bus) into a holding register during read memory cycles. The ClearReq signal is set to a high state before the completion of a cycle so the logic of the requesting port can reset the memory cycle request ahead of the cycle completion. This is necessary so that it will not interfere with the operation of the priority allocation circuitry in arbitrating the allocation of the next cycle.

The memory cycle control also generates three other signals which are used by the array of 16k memory chips in the process of reading and writing memory into addressed locations. The signals are: RAS (Row Address Strobe), CAS (Column Address Strobe), and AX (Address Exchange). Two of these signals are synonymous with signals already mentioned: RAS and DataStrobe are the same, and CAS and ClrReq are also the same. The function of these signals will be discussed in Section 6.5.3.

6.5.3 Memory Arrays (schematic MEM 3/4)

The physical organization of the Memory Subsystem consists normally of four circuit boards providing a total of 128K, sixteen-bit words of memory. It may be optionally configured with as many as eight boards. The boards operate in pairs, each providing half of a word. The circuit boards have identical artwork but are configured differently in use by jumpers and the depopulation of certain sections of the board. Each board has 32 dynamic MOS memory chips with 16,384 bits per chip (see Fig. 6.12). The driver circuitry for the control signals necessary for these chips is included on each board, as are the multiplexer chip for the address lines and the buffers for the data outputs. Each memory board is organized into 32-bit words for read operations, but only 8-bit words for write operations. Two such boards operate together, creating a memory with 64-bit wide read operations and 16-bit wide write operations. The artwork of each board is also complete with tristate buffer gates to give the appearance of a 16-bit word size for read operations. This circuitry is only installed on one set of two boards and used for all boards to give the capability of 16-bit read operations.

6.5.4 Configuring the Memory Boards for Timing Signals

The Memory Array sections of all four memory boards are identical. The memory cycle control is populated and connected by jumper only on the board designated 64R, meaning the right half-word of the first 64k words of memory. The chips belonging to the cycle control are U44, U13, and U1 as shown in Fig. 6.11. On this board, these chips are installed and their operation is connected to the Memory Array section of the board by connecting jumpers J1, J2, J3, and J4.

6.5.5 Configuring Memory Boards for 16-Bit Read Multiplexers (schematic MEM 4/4)

The 64R board also has the read data multiplexer circuitry populated consisting of 74LS244 octal buffers U49, U50, U53, and U54 (fig. 6.13). The board designated 64L also has the read data multiplexer circuitry populated; none of the other memory boards (normally two, but possible as many as six others) have this circuitry installed. There are no jumpers to enable this circuitry. If the circuitry is installed, then it will function.

6.5.6 Page Selection (schematic MEM 2/4)

The most significant address lines, MAD16 and MAD17, determine which pair of memory boards will function. Each board has a 74S138 decoder providing switch selection of a range of addresses depending on the values of these two lines. The signal generated by the 74S138, U46, is named BoardSelect.L, and it is found on pin 9 through 16 of U47, a dual in-line switch package. There is also an overriding selection line labeled RefSel which enables the signal RAS of all boards for the special read memory cycles used to refresh the dynamic memory cells of the 16k memory chips. The special selection line for RAS generated from BoardSel.L and RefSel comes from the output of a NAND gate, U25, pin 8.

6.5.7 Address Bus (schematic MEM 2/4)

Address lines MAD2 through MAD15 are used in the selection of a single word from the memory array. The chips used by this array require the address to be given over seven signal lines at two separate times. A special purpose chip from Intel, the Intel 3242 in U35, performs this function. It selects values from seven of the address lines and sends them to the address inputs of the chips in the memory array, A0..A6, at the time when the RAS is asserted. The other seven address values are sent at the time when CAS is asserted. The control signal selecting the first or second group of addresses is AM connected to pin 3 of U35. This signal changes midway between the occurrance of the RAS and the CAS.

6.5.8 Word Selection by the least two significant address lines (schematic MEM 2

The least significant two address lines, MAD0 and MAD1, are not used in 64-bit read cycles as four words are simultaneously read out. In the 16-bit write cycles and in the 16-bit read cycles, these two bits must be used to select the correct word from the group of four addressed by the other address bits. A 74S138 decoder circuit, U26, is used for both read and write functions. In the read mode. The "S4" input of the decoder is connected toR/W'. In the read made it is asserted high and one of the upper half of the decoder outputs, U26, pins 7, 9, 10, and 11, is used to control the octal buffer which gates the proper group of eight bits to the Memory Data Bus for each of the pair of selected cards. The octal buffers controlled in this manner are circuits U49, U50, U53, and U54 (schematic MEM 4/4). These selection signals, originating at the decoder U26, are labeled GateData0.L through GateData3.L. In write mode, because R/W' has a low value, this decoder asserts one of the lower four selection lines, U26, pins 12 through 15, which are connected directly to the signals enabling memory write operations in the 16k memory chips. These signals are labeled R/W'0...R/W'3. Resistors are also used with these signals as pullups and resistive loads to prevent ringing and overshoot.

6.5.9 Row Address Strobe Signal (schematic MEM 2/4)

The signal RAS from the Memory Cycle Control is described in the manufacturers documentation under the name RAS. It is used to cause the memory chips to sample the address signals selected by the multiplexor, U35. The reader is referred to this document for a better understanding of the function of this signal. This signal eventually reaches the array of memory chips after passing through a buffering circuit, (U15), and signal conditioning resistors (Fig. 6.14) which help suppress overshoot. Because of the drive requirements for 32 chips, the signal is generated in parallel by four separate circuits under the names: RAS0.L, RAS1.L, RAS2.L, and RAS3.L.

6.5.10 Column Address Strobe (schematic MEM 2/4)

CAS is used to strobe the second group of seven address lines into the memory chip, and it is also the signal selecting which of the four words of memory chips is to be used for write and sixteen bit read operations. The logic gating this signal is generated by the decoder U2 and the NAND gate U24. U24 is low when a 64 bit read operation is being performed. In this case, CAS is gated through all four circuits of U3 and U4 giving a strobe to all words. For 16 bit operations, U24 is inactive, and the selected word of memory chips is given the CAS through the 74LS138 decoder U2, and one of the four circuits in U3 and U4.

6.5.11 Read Data Drivers (schematic MEM 4/4)

Two sets of data drivers are used for the output data of the read cycles. The first set consisting of 74LS244 octal buffers are in positions U48, U51, U52, and U55 of all memory boards. The enabling of these circuits is directly driven from the BoardSel.L signal. The second group of drivers are in reality operated as a multiplexer selecting a word of data from the 64 bit wide Memory Read Bus. The circuits involved are also 74LS244 octal buffers and they are in positions U49, U50, U53, and U54 of the lowest addressed pair of memory boards. Since these circuits function regardless of which pair of memory boards is selected, they are depopulated on higher addressed memory boards.

6.5.12 Memory Clock Circuitry (schematic MEM 1/4)

The memory can operate from its own independent clock which is a 74S124 oscillator in position U13, or it can operate from an externally supplied clock signal entering the master board through pin DE1. The nominal period for this clock is 80 nsecs, giving a cycle time of 560 nanoseconds.

6.5.13 Refresh Circuitry (schematic MEM 1/4)

The 16k memory chips of the Lilith memory are the dynamic MOS variety made by many different manufacturers under many different part numbers. One of the many available chips is the MOSTEK 4116. All of these parts require an operation called "refreshing" which causes the information in a row of the internal memory array to be renewed. A normal read cycle will perform this. In order to guarantee that each of the 128 rows in the 16,384 bit chip is addressed at least once every 2 milliseconds for a read cycle, the memory control logic includes a section of logic which acts as if it were a data port. This logic is basically an oscillator, U13, pin 7, which sets a flip-flop, U1, pin 5,to request a memory cycle once every 16 microseconds. When the request is granted by the memory port priority allocation unit, the memory performs a read operation from an address conveniently supplied by a counter circuit internal to the INTEL 3242 chip, U35, which also performs the address multiplexing for the memory chip array (schematic MEM 2/4). The value of this counter is incremented after each refresh operation by a signal from NAND circuit, U25, pin 11, which supplies a pulse during the refresh memory cycle to pin 2 of the

INTEL 3242. The signal RefSel is also connected to the INTEL 3242 at pin 1 in order to select the internal counter value as the source for the address output during the refresh cycle.

6.5.14 Memory Parity

At the time of this writing, memory parity has not been used with the Lilith memory. The operation of the machine has not visibly suffered because of this. However, some pressure in the form of expressed opinions from outside observers has caused the designers to consider adding this feature if for no other reason than to see if it possibly is needed.

6.7 Disk Controller

SEE :

WD1001

WINCHESTER DISK CONTROLLER

OEM MANUAL

80 - 031003 - 00 A1

7 Diagnostic Procedures

7.1 Power On and Reset Diagnostic Interpretation

7.2 Modula-2 Test Programs

7.4 Test Programs Using the DPU

7.4.1 Terminal Connection to the DPU

In order to be able to use the DPU in diagnosing Lilith problems, it is necessary to have an additional computer or terminal. The purpose of this extra equipment is to provide terminal interaction between the Motorola processor and the maintenance engineer and to provide the capability to load programs. The DPU uses none of the Standard Lilith peripherals for this function as any of these parts could be nonfunctional.

Originally, the DPU was developed with an HP terminal. As a consequence, all DPU software is compatible with the operations of the HP terminal. At this time, we no longer use such terminals for DPU operations, but the protocol is still the same. Now we use another Lilith as the terminal for the DPU whenever possible. The functions needed by the DPU which were originally satisfied by the HP terminal are as follows:

Display on a screen ASCII characters received from the DPU.

Send characters typed on a keyboard to the DPU.

Send upon request an entire file of characters constituting a binary load file in the commonly known as Motorola 'S' format. The DPU initiates this operation by sending an *escape* character followed by the lower case 'e'.

Send on request the next line of a load file terminated by a *carrage return.*' file. The request sequence from the DPU is constituted from am *escape* followed by 5 characters: <esc>, 46C, 160C, 60C, 122C, 21C

Send on request the message 'HELLO'. The request sequence from the DPU is <esc>, '↑', 21C.

The third and fourth items of the above list were originally used to transfer binary files in the Motorola 'S' format stored on the HP cassette tape. The special control sequences are the defined HP protocols for causing the transfer of either an entire file from the cassette or a single line of the file. Under such operations the position of the cassette determined which file was to be read. When another Lilith or some other computer is used. These files are kept in disk storage. Consequently, the program emulating the HP terminal needs to interrogate the operator at this point for the name of the file to be transfered from the computer disk.

The fifth item of the above list, the 'HELLO' response, allows the DPU to determine if in fact a terminal is connected. It uses this when the reset button of the DPU is pressed. If there in no response within five seconds, the DPU assumes that a standard Lilith 'cold boot' operation is needed, and it initiates that.

7.4.2 Preparing a HP terminal for Use with the DPU.

If you have an HP terminal then the problem is relatively simple. You need only make the proper RS-232 connection to the Lilith and transfer the Motorola diagnostic programs to an HP cassette. You may use the Lilith program 'hpcopy' for this function. When performing this transfer you will need to connect the HP terminal to the Lilith I/O connector on the back of the machine. It is a standard subminiature D female with following signal connections:

pin	signal
2	Incoming signal to the Lilith. (RS-232c voltage levels)
3	Outgoing signal from the Lilith. (RS-232c voltage levels)
7	Ground.

The baud rate is switch selectable in a range of 9600 baud down to 75 baud. The switch for selecting the baud rate is at the back of the CPU Data Port card (CDP).

The files to be transferred are: ex.MOT, ifu.MOT, alu.MOT, hwtest.MOT, dsktst.MOT. When you have your files on the HP cassette, you can connect the HP terminal to the DPU connector on the Lilith. It is wired identically to the I/O connector.

7.4.3 Emulating the HP Terminal

If you have more than one Lilith then it is only necessary to connect the two Liliths using an RS-232 cable with a '2 to 3' crossover and run the program 'HPTerm'. The Lilith to be tested should have its end of the cable installed in the DPU connector. At the other end, it should go in the I/O connector of the functioning Lilith which is emulating the HP terminal. Incidentally, at this time the DPU board should be installed with the baud rate switch of the DPU board matching that of the other Lilith's CDP board. 9600 baud allows transfer of binary files at maximum speed.

If you do not have an HP terminal or a second Lilith, then it is necessary to prepare an HP terminal emulator using whatever computer system you have available. A program must be prepared for this computer having the characteristics of the HP terminal. This program is best prepared using the program 'HPTerm' of the Lilith as a model and definition. The ASCII files listed in the preceeding paragraph should be transferred to your computer system. It would be best to do this when your Lilith is still functioning. You will find it more difficult to type these binary files from their printed listings.

7.4.4 Operating with the DPU Resident Software

At this point you should have an HP terminal, a second Lilith, or some other computer system (which can successfully operate in HP form) connected to the DPU port of the malfunctioning Lilith. You should also have available the files ex.MOT, alu.MOT, ifu.MOT, hwtest.MOT, and dsktst.MOT. Now press the upper of the two switches on the DPU card. If everything is working, the Motorola will respond:

```
HELLO
LOAD MOTOROLA
```

If you do not get this response, try the lower button. You may get the response:

```
nn  nn  nnnn   (*nnnn represents hexadecimal numbers *)
MOTOROLA
```

If you get the second response, but not the first, then your HP terminal emulator has either a bad connection on its input channel, or it is not responding properly to the Motorola s ID request.

7.4.4.1 The Main Command Level

When you have the above response, you are at the main command level found in the ROM resident software of the DPU. The software structure of the DPU diagnostic test programs is centered around the concept of tree of command levels. At each level one has a menu of one letter commands available which either invoke self-contained test routines or provide a descent to another command level. Organizationally, each command level is considered to offer commands associated with the testing of a section of the machine. At each sublevel, the control C may be typed to return to the main command level. If a 'question mark' is keyed, the menu of available commands at the active sub-level is printed. An escape interrupts any test in progress and returns the program to the command interpreter for the last active sub-level.

At the main command level, the following menu of commands are available at this level in the Motorola software:

```
U  RAM LADEN    (* german for load ram*)
O  BOOT & GO
L  LOAD MOTOROLA
P  MEM DUMP
I  INSPECT
G  GO
Y  TYPE CTRL
S  MAIN STORE
X  EXEC
V  DEVICES
T  TEST
M  MICRO MEMORY
```

Each of these commands is invoked by typing the single character at the start of the menu line. It is possible to enter a hexadecimal number before typing the command character. Execution of the subprogram begins immediately after typing the command character. A carriage return is not necessary. Some of the subprograms expect an additional constant to be typed after the command. This may give the illusion of not having begun the execution of the subprogram as there will be not indication to prompt the user, but this is not so. The second hexadecimal number will actually be requested by the subprogram.

7.4.4.2 Functions at the Main Command Level

The commands U,O,L,P,I,G,Y invoke functions found in the DPU ROM. The functions performed by each of these commands are:

U RAM LADEN

The subprogram will issue commands over the serial interface to read a binary file from the terminal. It expects to receive a binary file in the special, modified Motorola S-format which is used for microprograms. The file will be loaded into the writeable control store (if one is installed).

O BOOT & GO

This command will cause the DPU to emit a master reset pulse to the Lilith system and start the processor in execution at microinstruction address 000 which is the normal starting address for a 'boot' from the disk.

L LOAD MOTOROLA

This subprogram will issue commands over the serial interface to read a binary file from the terminal. It expects to receive a binary file in standard Motorola S-format which it will load into the RAM storage of the DPU. The file should contain an executable Motorola 6800 program.

P MEM DUMP format: startaddress 'P'

The subprogram will give a hexadecimal listing of the contents of the Motorola memory beginning at the given address. This command is included in the resident software to help in the debugging of Motorola test programs.

I INSPECT format: address 'I'

This command must be preceded by an address or the value zero is presumed. The command allows the inspection of the Motorola memory contents at the given address. The value will be printed on the same line immediately after the command. If the user wishes to change the value of this location, he must first type a ':' followed by a byte value in hexadecimal. If the user wishes to terminate the inspection sequence, he ends the number with a carriage return or space. If he wishes to inspect the next memory location, then he types a comma.

G GO format: startaddress 'G' stopaddress

This command causes the Motorola to begin execution at the hexadecimal address typed prior to the command. The program will set a breakpoint at the second address. If the start address is omitted, then the last breakpoint address is taken as the start address. If no second address is given, then a carriage return must be typed after the command to cause the transfer.

CAUTION: the start address and the stop address must always be different. It is impossible to execute a loop n-times by omitting the start address and giving the same stop address each time.

Note: This command is especially useful to stop a test program at a point in execution where the error condition can be checked with a probe or scope. The stopping point of a test program after it has detected an error does not always occur at such a point. Sometimes, in the process of detecting an error, the program must execute instructions beyond the actual error point.

Y TYPE CTRL **format: number 'Y'**

The number given before the command selects printing options for the execution of test programs. The options are:

```
0:  no messages

1:  type error and end of test messages

2:  type error messages only

3:  type error and end of test message; stop on error messages

4:  type error and end of test messages;
    on error, give a pulse on pin nnn
```

The commands listed above all perform a function and return to the same command level for the next command.

7.4.4.3 Sublevel Entry Commands at the Main Level

The commands: S,X,V,T,M are used to descend to lower command levels. These commands are intended to invoke programs which will be found in RAM memory. Consequently, these commands will not function until after the programs have been loaded. You may do this by typing 'L' (for load). At this point you will see the cassette drive begin to read if you have an HP terminal or the program HPTerm will prompt: file>. You may type the name of a valid Motorola binary file. At the time of writing of this manual, the files ex.MOT, alu.MOT, and ifu.MOT are the only ones which have been prepared for the 6802.

(*comment: There are also two other programs which have been prepared for use with the DPU. These two files, hwtest.MOT and dsktest.MOT, are in Motorola 'S' format, but the files are in Lilith native code.

They can be loaded through the Motorola, but are executed directly by a functioning Lilith processor. The current version of the program HPTerm on a Lilith will load all of these files into core automatically as the program begins and thereafter they may be loaded from core each time rather than disk. To do this, use the names: ex, alu, ifu, hw, or dsk. The HB disk may be turned off after the program HPTerm is operating*)

When a test program is loaded, jump addresses are inserted to allow the use of the remaining five commands at the main level command menu. These commands do not perform any testing functions directly. They are the entry point to a new level of commands. These new levels of commands are called sub-monitors.

S MAIN STORE

This command causes descension to a sub-monitor supporting a commands which can inspect and load the main memory of the Lilith. This collection of programs is found in the file 'ex.MOT' ('ex' in HPTerm).

X EXEC

This command causes descension to a sub-monitor which supports commands allowing execution and observation of individual Lilith microinstructions found in the normal control store. The use of these commands are most beneficial in the debugging of new microprograms as well as for use with the diagnostic programs written in Lilith microinstructions. This command becomes active when the file 'ex.MOT' is loaded ('ex' in HPTerm).

V DEVICES

This command causes descension to a sub-monitor supporting test programs which exercise the various I/O devices of the Lilith. The programs are only use for testing the devices for proper functioning. These programs are loaded with the file 'ifu.MOT' ('ifu' in HPTerm).

T TEST

This command causes descension to one of two sub-monitors supporting test programs to exercise and check the Lilith processor. The sub-monitor activated is that of the last file loaded, either 'alu.MOT' or 'ifu.MOT' ('alu' or 'ifu' in HPTerm). Because of memory size limitations, it is not possible to have both sets of test programs in memory at the same time. There are a set of commands common to both files. In general, the commands of 'alu.MOT' are associated with testing the MCU, ALU, and main memory. The file 'ifu.MOT' has the commands to test the IFU, the writeable control store, and the I/O devices.

M MICRO MEMORY

This command at the main level causes descension to sub-monitor supporting commands to load, examine, and error check the control store--ROM or writeable. This section of the test software is loaded with the file 'ex.MOT' ('ex' in HPTerm).

7.4.5 Library Primitives Resident in the DPU ROM

To enable test programs to be quickly written and to reduce debug time, a set of primitive operations for DPU testing of the Lilith was selected and included as part of the ROM memory of the DPU. These primitives provide the basic operations used in all the testing programs that have been written up until the time this manual is being written. These primitives have proved so useful that the actual test programs themselves are almost nothing more that a sequence of procedure calls and data definitions for the arguments of these procedure calls. It will be of value to describe this set of primitives in this manual for the benefit of the maintenance engineer who has found a fault in his machine. The error message from the DPU test program will direct him to the procedure in the listing of the test program which is detecting the error. There the maintenance engineer will find a commentary in the form of a Modula-2 program which describes the function of the procedure, and he will find the 6802 assembly language program for that procedure which will consist, for the large part, almost entirely of procedure calls to these primitives. As an illustration, some of the more important primitives are:

```
STUPC       EQU   0C1C2
*****       STORE MICRO PROGRAM COUNTER
*           PREPARES MICROCONTROLLER FOR EXECUTION AT ADDRESS IN X
*           UPC:=X

EXX         EQU   0C117
***         EXECUTES MICRO INSTRUCTION POINTED TO BY X
*           X POINTS TO MOST SIGNIFICANT BYTE
*           MOST SIGNIFICANT BYTE HAS LOWEST ADDRESS
*           X IS INCREMENTED BY 5

RCPU        EQU   0C1F0
****        READ CPU DATA BUS
*           B GETS TOP HALF
*           A GETS BOTTOM HALF

WCPU        EQU   0C1F7
****        WRITE CPU DATA REGISTER
*           CPU BUS REGISTER := B*256 + A
*           DATA IS GATED TO BUS WHEN SOURCE 13 IS ADDRESSED

EPMIR       EQU   0C15F
*****       ENABLE PERSONAL COMPUTER MIR REGISTER

EDMIR       EQU   0C158
*****       DIAGNOSTIC PANEL MIR REGISTER IS ENABLED
```

```
LDMIR        EQU   0C11D
*****        DIAGNOSTIC PANEL MIR REGISTER IS LOADED WITH
*            DATA POINTED TO BY X
*            MOST SIGNIFICANT BYTE HAS THE LOWEST ADDRESS
*            X IS RETURNED INCREMENTED BY 5
*            RETURNS WITH DIAGNOSTIC MIR REGISTER ENABLED
*            NO CPU CLOCK PULSE IS GIVEN
*            USEFUL TO GATE DATE NON-DESTRUCTIVELY TO CPU BUS

MCLK         EQU   0C1AC
****         SEND ONE MICRO CLOCK PULSE (DISABLE CPU CLOCK)
```

The above is an exerpt from the definition listing placed at the start of each test program. The complete repetoire of commands is contained in the first few pages of each test program. The complete listings for these test programs are given in Appendix F at the end of this manual.

7.4.5.1 The Error Printout Procedure of the DPU Resident Software

A common error reporting procedure is available in the DPU ROM for the use by all test programs. The procedure is programmed to be called by the software interrupt feature of the 6800 (SWI command). The program prints out a message utilizing as error source identifiers the strings stored in the program variable MAIN.ID and SUB.ID. It prints out as well the values in hexadecimal for the registers B,A,X, and PC in that order. A typical error message would appear as follows:

```
ERROR IN ALU STACK 00 01 0003 2647
```

which is to be understood as

```
ERROR IN MAIN.ID=ALU, SUB.ID=STACK, B=00, A=01, X=0003, PC=2647
```

The interpretation is dependent on the error routine which discovers the error but the normal usage of the routine has the expected value in the B and A registers with the actual value in the index register. The program counter value will contain the address immediately following the SWI instruction in the program used to invoke the error printing procedure. The diagnostic procedure to follow upon encountering such and error is to study the listing and the operations being tested. Then, use either the stop-on-error typeout control or the GO command with breakpoint to get the processor into a state where the error cause can be determined with a logic probe or scope.

For greater convenience, an appendix will at some time be generated for this manual which will suggest possible faulty chips or areas to test for each error message which can possibly be generated by the test programs.

7.4.6 Getting Started Quickly with the alu.MOT and ifu.MOT Test Programs

Before going into detail about the many capabilities of the test programs in the alu.MOT and ifu.MOT files, an abbreviated version of the necessary commands will be presented which will allow the maintenance engineer to quickly get the tests into operation without unnecessary and time-consuming study of this manual.

Once the ALU or the IFU test program (alu.MOT or ifu.MOT) is loaded, the next step is to move to the test program command level by typing a 'T' (for hwtest) at the monitor command level. If all has worked correctly, typing a '?' will display an entirely different selection of commands from those available at the main command level. (*comment:If you need to return to the main command level, you may type a 'control c' at any time, except when using the program HPTerm. Since 'control c' is used in the Lilith to abort a program and dump the contents of memory, 'control z' is used instead in the HPTerm program.*)

The quickest way to determine if any errors exist in the machine (errors which can be detected by this program) is to type the command 'X', which executes continuously a chain of the most important test subprograms in this program. If there are any detected errors you will encounter error messages. Otherwise, you will see only the 'END OF TEST' messages. After a while the messages will repeat themselves indicating that an entire cycle has been completed. If there are any error messages then the program listing must be consulted to determine further details about the nature of the error message. Appendix 3 written by Jirka Hoppe gives information about the use of each test program and the interpretation of the error messages. This appendix should be consulted when errors are encountered.

Using the DPU to Load Lilith Programs and Execute Them.

The executer program provides a number of commands for manual control of Lilith execution, i.e. single step of micro and m-code instructions, as well as the capability to view the state of the Lilith registers in between each operation. An example of the usefulness of this program is the loading and execution of programs in the Lilith entirely through the Motorola and without the use of the operating system or the disk.

As an example of how this may be used, we will discuss the use of the executer program in running the program disktest, which is written in Modula-2. If the Lilith disk or disk interface is not functional, then this program can provide greater diagnostic capability in determining the error. The reason for its greater capability is, of course, to be attributed to the fact that the program could be written in Modula-2 rather than 6802 assembly language. Also, it operates at the speed of the Lilith processor rather than the 6802.

To load a Lilith program, through the DPU, one must first load the executer file (ex.MOT or ex) which includes the subprogram--'STORE', the subprogram STORE is reached by typing an 'S' after loading the file 'ex'. The menu for the STORE program (type '?') also includes a 'load' command which will load a file in Motorola 'S' record format. This command loads programs into the Lilith memory, instead of into the DPU memory. After loading the file; dsktest.MOT, the user returns to the main command level with a 'control c' ('control z' in HPTerm).

Now that a Lilith program has been loaded into the Lilith memory with the 'store' subprogram module, the next step is to start the program properly using the 'executer'. We reach the 'executer' command level from the main command level using an 'x' command.

Now to start the program in main memory, the following sequence of instructions is used:

`*R` *to reset the entire Lilith system*

`*1P` *to set the microcode program counter to address 001*

`*G` *to start the Lilith processor at full speed (go)*

At this point the Lilith processor should be operating and looking for instructions coming in through the I/O connector on the back of the machine. To operate this program, reconnect the terminal cable from the DPU connector to the I/O connector and type a question mark. If everything is operating correctly, the Lilith should respond with a menu of instructions which will be described in the description of the program 'dsktst'. If there is no response, first repeat the load procedure to assure no mistake has been made, then retest the processor with the DPU programs to see if some error exists in the Lilith processor itself.

Debugging Microcode with the Executer

The executer program is also usable to control and analyze the execution of micro- instructions on a step by step basis. An example which illustrates its use is that of adding a new special purpose M-code to the system.

If the reader is not familiar with the mechanism for invoking special M-codes in Modula-2, he is directed to the explanation of this topic in chapter 12 of the Lilith Software Manual. See, for example, the method to use GET and PUT.

In order to provide a new microinstruction, the Lilith must have the entry point for the instruction programmed into the map ROMs of the MCU and the necessary micro-program for the M-code included in the microinstruction ROMs for the micro control unit. The MCU board is equipped with jumpers so that EPROMs can be used for control store while new micro code is being debugged. However, the clock of the machine must be set to a slower rate when EPROMs are used.

The following technique is suggested for testing a new M-code instruction:

1. Prepare test software in Modula-2 for the new instruction using the facilities of the Lilith operating system, and compiler, and utilities.

2. Add a single micro instruction at the start of the micro instruction sequence for the new M-code which causes the machine to loop with a 'jump to here' instruction.

3. Install the DPU and connect the external HP terminal or a computer system with software.

4. Load the executer program and give the boot command, 'o', loading the disk operating system.

5. Run the prepared test program until the machine executes the new M-code

and ends up in the infinite loop at the start of the new M-code instructi

6. Using the executer program in the DPU, halt the processor with a 'H' comm

7. Use the commands of the executer to advance the microprogram address coun past the infinite loop and single cycle through the microprogram of the n instruction.

8. Use the features of the executer to verify the correct performance of eac microinstruction as it is executed.

The full set of operations provided by the executer program is given in the following table:

R	RESET1	give reset signal to entire Lilith processor
0	BOOT + GO	reset processor and start boot from disk
nP	SET uPC	set 2911 microcontrol unit to 'n'
U	DISPLAY uADR+uINSTR	displays state of microcontrol unit
nN	NEXT uINSTR	gives 'n' clock pulses, executes 'n' microinstructions
T	TRACE REG	prints out 2901 register values
nX	SINGLE STEP	executes next 'n' M-codes (multiple microinstructions)
nS	EXEC UNTIL STOP ADR	executes microinstructions until uADR = nnnn
G	GO	places processor in RUN mode
H	HALT	takes processor out of RUN mode
nI	INSPECT	displaces contents of main store memory location 'n'
nY	BYTE INSPECT	displace contents of main store memory addressed by byte address 'n'

The above commands constitute a very complete set of control steps for activating and examining the operations of the microcontrol unit. For a more in-depth explanation of the functions controlled by the executer commands, the reader is directed to the chapter which discusses the circuitry and theory of the MCU.

Appendix A Input/Output Interfacing Guide

The purpose of this section is to describe the input/output address decoding logic and provide the LILITH computer owner adequate information for the design, construction, and debug of an interface between the Lilith and an external piece of equipment. The function of such an interface is the reliable transfer of data into and out of the LILITH.

A.1 Types of Interfaces

The LILITH architecture is designed to accomodate two types of interfaces: Programmed I/O (PIO) and Direct Memory Access (DMA). PIO interfaces are in general less complicated to design and construct, but offer less performance with respect to transfer rates. They do, however, offer more versatility in their use and are generally the better choice for device interfaces which do not demand high rates of data transfer. DMA interfaces are used for high speed devices as disks, networks, or laser printers.

A.2 Selection of Interface Type

PIO Interfaces are characterized by data transfers which occur whenever a program explicitly specifies the transfer. In contrast, a DMA interface, once initialized, will transfer data whenever it can without respect of the current state of program execution. Such a transfer represents a direct read or write cycle request to the LILITH memory. The LILITH memory request arbitrator can initiate such requests every 500 nsecs. In the case of the display processor, a 64 bit transfer is requested at an average time interval of A usecs which is equivalent to a transfer rate of 1,425,600 bytes/second. A PIO interface could clearly not be used for such a function because full processor dedication to data transfer would yield transfer rates in the order of 200,000 bytes/second.

A.3 Software for Interface Control

For the support of PIO interfaces, the Lilith has two special M-code procedures, GET and PUT. In order to include these procedures into a program, the programmer must define them as code procedures. This topic is covered in chapter 12 of the Lilith Software Manual. Such a definition has the form:

```
PROCEDURE GET(channel:CARDINAL; VAR info:WORD);
  CODE 240B;
END GET;

 PROCEDURE PUT(channel:CARDINAL; info:WORD);
   CODE 241B;
 END PUT;
```

When this definition has been included in a beginning of a module, the procedures GET and PUT will be recognized within the body of that module.

Use of GET and PUT

To use these procedures, one must include within the body of the program a procedure call including the necessary parameters. Two parameters are required for either procedure specifying the source (GET) or destination (PUT) input/output device and the data variable to provide (PUT) or receive (PUT) the transferred data. A instance of GET or PUT usage would have an appearance similar to the following:

```
GET(deviceaddress,destinationvariable);

PUT(deviceaddress,sourcevariable);
```

Execution of GET

For the GET procedure, the symbolic parameter 'deviceaddress' (shown above) must be either a constant or variable having a value equal to the assigned address of the I/O device. During execution of the procedure, this value will be fetched from the parameter stack, and transmitted over the bus to the I/O selection register. The device assigned this address will then respond with a word of data to be stored in the destination variable.

Execution of PUT

For the PUT procedure, the device address is processed in the same manner as for GET. Then, the selected device receives the contents of 'sourcevariable' given as a parameter in the procedure call.

A.3 Example I/O Program

The following example program module will illustrate the usage of GET and PUT within a program:

```
MODULE UARTDriver;
FROM SYSTEM IMPORT GET,PUT
EXPORT QUALIFIED GetUARTChar, SendUARTChar;
PROCEDURE GetUARTChar(var ch:CHAR);
CONST uartstatusaddr = 5
      uartdataaddr = 6;
      uartreceiveready = 0;
VAR status:BITSET;
BEGIN
  REPEAT
    GET(uartstatusaddr,status)
  UNTIL uartready in status;
  GET(uartdataaddr,ch);
END GetUARTChar;
PROCEDURE SendUARTChar(ch:CHAR);
CONST uartstatusaddr = 5;
      uartdataaddr = 6;
      uarttransmitready = 1;
VAR status:BITSET;
```

```
20          BEGIN
21            REPEAT
22              GET(uartstatusaddr,status)
23            UNTIL uarttransmitready in status;
24            PUT(uartdataaddr,ch);
25          END SendUARTChar;
26          END UARTDriver.
```

Example Description

From a functional standpoint, the above example program module provides procedures which receive and send a character through the UART (RS-232C) serial interface. For control of the UART, the UART interface has two input/output addresses which respond to the procedures GET and PUT. One of the addresses, which is referenced in program lines 5 and 16, returns status information of the receiver and transmit portions of the UART. Notice the usage of the BITSET construct to determine the "ready" state of each section of the UART. The second input/output address allocated to the UART interface reads the data from the receiver if a GET operation is specified, or transfers a byte to the transmit portion of the UART if a PUT operation is specified. By way of explanation, GET and PUT operations using the data address also alter the state of the ready bits within the status of the UART. This is a characteristic of the UART used in the interface.

A.5 Implementation of GET in firmware

The microcontrol firmware implements the transfer of data specified in the GET procedure. The mechanism is straightforward. First the address of a data variable is 'popped' from the stack. Then the I/O device selection code is taken from the next level of the stack and transmitted to an I/O selection register. With the correct device selected and responding to a command for data to be placed on the CPU bus, the controller directs the data to the memory data register of the memory port. In the next microcycle, the controller sends the address of the data variable to the memory port initiating a write cycle to store the device data into the data variable given as a parameter in the procedure call.

Implementation of PUT

The mechanism for the handling of the I/O Device Selection Code is identical to that for GET. Only the transfer of data is different since it must go the opposite direction. The data variable (or possibly the constant) given as second parameter then captures the correct data from the bus which received it from the selected device.

A.6 Slow Data Transfers

The data transfer for GET and PUT takes place over the 16 bit BUS. Normally a 16 bit word is transferred over this bus in a single 150 nanosecond cycle. For GET and PUT I/O transfers, the actual transfer uses 4 microprogram cycles instead of just a single cycle. This provides a total time of approximately 600 microseconds for the data transfer. The purpose in extending the time the data occupies bus is to accommodate peripheral devices which cannot adequately respond in a single cycle. Both the disk interface and the RS-232 UART interface require the longer period of time for a data transfer.

Timing of the Slow Data Transfer for GET and PUT

The timing for a GET and a PUT data transfer relative to the CPUClk.U cycle are given in figure A.1. Five clock cycles are necessary to complete the transfer. In the first cycle, the I/O device selection code is placed on the bus and transferred to the I/O address register on the CPU Data Port circuit board. Then the actual data is transferred during the next four CPUClk.U cycles. The I/O device selected by the address given during the first cycle receives or provides the data of the transfer operation during the last 4 cycles.

Control Signals from the MCU

Three signals from the Microprogram Control Unit (MCU) control the source and destination selections for the CPU data bus during an I/O operation. They are: Dst=IOA.L, Dst=IOData.L, and Src=IOData.L. Dst=IOA.L (unabreviated: Destination = Input/Output Address asserted low) cause the bus data to be loaded into a register for use with the other signals. This signal is logically combined with Dst=IOData.L and Src=IOData.L to create individual command signals for the each device separately. In this manner, an I/O device can be unambiguously directed to place or sample data on the CPU bus.

IOClock

The timing signal, CPUClk.U, has a period of 150 nsecs normally, and it occurs 4 times during a GET or PUT M-code execution. An I/O interface needs a clock signal, but it should be a single pulse and for many devices it needs to be longer than 150 nsecs. For this reason, a circuit on the DATA PORT card creates a signal called IOClock which can be used by an I/O interface to synchronize the transfer of data between the interface and the computer. The timing relationship of IOClock.U to a GET or PUT operation is shown in the figure A.1.

IOClock During GET

The signal, IOClock.U, is generated during the GET operation as well as the PUT operation, but it is not used to capture data from the BUS into one of the ALU registers (this function is handled by CPUClk.U as part of a normal microcycle). IOClock.U can be used, however, by the selected device during a GET if a further action in the interface needs to be initiated by a data read operation. In the case of the UART interface, the four cycle GET operation for the incoming data clears the "ready" status of the interface as well as gathers the received character. IOClock.U is used for this function because it a signal without ambiguity in contrast to the use of Src=IOData.L which is subject to transient fluctuationswith the start of each microcycle.

I/O Address Register

Selection of the I/O device to be used as the object of a GET or PUT operation can be handled in two ways: by direct decoding of the selection address given with each PUT or GET operation, or by response to decoded signals generated from the CPU Data Port card. In either case, the I/O selection code begins as a value passed as a parameter in either a GET or PUT procedure call. The value is located in the evaluation stack and is transferred onto the BUS and into a holding register in a single and normal microcyle. In this cycle, the bus destination is given by asserting the low level of Dst=IOA.L. Devices which are to be selected in this manner must constantly "monitor" for a transfer of this kind and thereby capture in an I/O Address Register the newest selection code from the bus with the assertion of each Dst=IOA.L signal. The normal CPUClk.U provides the properly timed transition to load the data into a register. With the selection code captured in a register, the device, if selected, can then respond to the four cycle data transfers controlled by the signals: Src=IOData.L and Dst=IOData.L (figure A.2).

Decode of I/O Address

Capture of the I/O selection codes as described in the preceding paragraph is done precisely in this fashion by circuitry located on the CPU Data Port card. Figure A.2 shows the logic for address decoding on this card. These selection codes are further decoded into bank selection signals creating a single selection signal for groups of eight I/O addresses. These signals together with the three least significant bits of the address selection code are available on the backplane and may be utilized by IO interfaces thereby eliminating the need for extra decoding logic. The location of these signals is given in the following table.

Signal	Pin	Signal	Pin
Bnk0.L	BS2	IOA0	BR1
Bnk1.L	AD1	IOA1	BH2

Bnk2.L	AE1	IOA2	BS1
Bnk3.L	AE2	CPUClk.U	BH2
Bnk4.L	AF1	IOClock.U	AT2
Bnk5.L	AF2	Dst=IOA.L	BA1
Bnk6.L	AH1	Dst=IOData.L	BB1
Bnk7.L	AH2	Src=IOData.L	BC1

Table 1: Back Panel Pin Assignments of I/O Signals (CDP card)

Preassigned I/O Selection Addresses

Certain I/O selection addresses have already been assigned at the writing of this document. The following table lists the allocation of these selection codes.

Address	Input Device	Output Device
00	reserved for CDP	Display
01	Keyboard Status	reserved for CDP
02	Keyboard Data	reserved for CDP
03	Mouse Buttons	Real Time Clock
04	UART Xmit Data	UART Receive Data
05	UART Status	reserved for CDP
06	Mouse X Coordinates	Mouse Reset
07	Mouse Y Coordinates	reserved for CDP
10	Disk Buffer Data	Disk Buffer Data
11	Disk Status	Disk Command
12	reserved for DSK	Disk High Track
13	reserved for DSK	Disk Low Track
14	reserved for DSK	Disk Sector
15	reserved for DSK	reserved for DSK
16	reserved for DSK	reserved for DSK
17	reserved for DSK	reserved for DSK

Table 2: Reserved I/O Selection Codes

A.7 Direct Memory Access

Some IO devices require data transfers at rates above those which can be managed by program usage of the procedures GET and PUT. Or, at least, the rates are of such a magnitude that using GET and PUT for such a purpose would absorb all of the available machine cycles and leave little extra for any sort of computation. For such cases, it is desireable to manage the device with the programmed mechanisms of GET and PUT, but then give the device 'direct access' to the memory of the computer so that information may flow through the interface to and from the main memory with as little computational overhead as possible. In such cases, the philosophy behind the interface design is to allow the transfer of control signals through the programmmed input/output procedures, and to allow the the transfer of data through the 'direct memory interface' so that as little disturbance as possible is created by the passage of a data word between an IO device and the memory of the computer. In the case of the display processor, a PUT operation is used to pass a memory address of the bitmap descriptor to the interface which subsequently uses direct memory data transfers to fetch the data displayed on the screen.

An unusual feature of the LILITH is the optional choice of memory word size for data transfers from the memory. For write operations to the memory one can transfer only 16 bits with each cycle, but with "read" operations it is possible to fetch in one cycle 64 bits if a greater transfer rate of data is desired. This feature is in addition to the normal capability of transferring a single word of 16 bits per cycle. Obviously, interfaces not requiring the greater bandwidth should be designed to use the single word data transfers so that the number of data registers and back panel connections are minimized.

A DMA interface design is more complicated that a simple programmed input/output interface using GET and PUT because of the complex timing relationships between the memory and the interface. There are many timing and control signals which pass between the memory port and the interface. The manner in which these signals must interact requires careful design to avoid malfunctions attributable to timing and signal delays. The discussion of the CPU Data Port describes the design of such a memory port interface. The timing circuitry found in schematic CDP 1/5 is a good example of a correctly designed interface which should be studied and emulated wherever possible.

There are two choices for memory word size if a read DMA operation is being performed. The interface may either receive 64 data bits, or only 16 bits. The selection of the mode is controlled by a signal line which the interface must assert when it is selected for a memory cycle. The capability to read either 64 bits wide or 16 bits is quite useful and versatile. For the display controller, it was naturally selected to read 64 bits wide so that fewer display refresh accesses were necessary. For a disk interface, one would choose the 16 bit read option and reduce the component count of the interface even though four times as many memory cycles are necessary for a transfer. In this case, a value judgement determines that the data rates are sufficiently slow that the reduced component count is of greater interest. On the other hand, the laser printer interface was very complicated and could have benefited from a reduced component count, but the extreme rate of data transfer demanded the use of the 64 bit data width merely to keep up with the voracious appetite of the printer.

Appendix B
M-code Instructions

When this document is prepared, it will explain in tutorial fashion the M-code instructions of the Lilith virtual machine. The reference document which defines the characteristics of each instruction is found in Appendix C. This document is written in Modula-2 and is the basis for interface between the compiler writer and the microprogrammer. The microprogram listing found in Appendix D is the implementation of the interpreter defined in Modula-2. It is written in 40 bit microcoded instructions executable by the Lilith hardware. Should there be any error in the performance of the Lilith (other than a hardware error), it must be first proven whether or not the error is in the microprogram or in the M-code definition. If there is an error in the M-code definition, then the compiler writers, Christian Jakobi and Leo Geissman, must be consulted in the correction. Microprogram errors should be brought to the attention of Werner Winegar.