30814

National Library of Canada | Bibliothèque nationale du Canada | CANADIAN THESES ON MICROFICHE | THÈSES CANADIENNES SUR MICROFICHE

NAME OF AUTHOR/*NOM DE L'AUTEUR* DANIEL JOSEPH SALOMON

TITLE OF THESIS/*TITRE DE LA THÈSE* A Sequential Language for Nanoprogramming the QM-1

UNIVERSITY/*UNIVERSITÉ* University of Alberta

DEGREE FOR WHICH THESIS WAS PRESENTED/ *GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE* Master of Science

YEAR THIS DEGREE CONFERRED/*ANNÉE D'OBTENTION DE CE GRADE* 1976

NAME OF SUPERVISOR/*NOM DU DIRECTEUR DE THÈSE* Barry J. Mailloux

DATED/*DATÉ* Sept. 22, 76 SIGNED/*SIGNÉ* Daniel Salomon

PERMANENT ADDRESS/*RÉSIDENCE FIXÉ* 8620 - 24th Ave.
Montreal, Que. H1Z 3Z6

NL-91 (3-74)

THE UNIVERSITY OF ALBERTA

A SEQUENTIAL LANGUAGE FOR NANOPROGRAMMING THE QM-1

by

© DANIEL J. SALOMON

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1976

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "A Sequential Language for Nanoprogramming the QM-1," submitted by Daniel J. Salomon in partial fulfilment of the requirements for the degree of Master of Science in Computing Science.

Barry J. Mailloux
Supervisor

[illegible]

W. K. Dawson

Date Sept. 21, 1976

ABSTRACT

The Nanodata QM-1 is a user-microprogrammable computer with up to three levels of program control. The lowest level is called nanoprogramming and uses a horizontal microinstruction format. A short description of the architecture of the QM-1 is given. A programming language is proposed to simplify the task of preparing nanoprograms. The principle contribution of Lizard, the proposed language, is that it eliminates the explicit specification of algorithm parallelism and the need for learning the hardware timing characteristics of the QM-1. The complete "Lizard Language Reference Manual" is included as an appendix. A Lizard translator would have to discover the parallelism possible in a source program; therefore, established techniques for doing this are discussed and adapted to work for Lizard. The result of the parallelism analysis is stored as a directed graph where the nodes represent elementary operations and the edges show the essential ordering of the nodes. The hardware timing constraints of the QM-1 are summarized in a table and systematized so that they can be easily stored in the parallelism graph. A method is proposed for scheduling the parallelism graph into nanoprogram form. The heuristics that it uses to approach an optimal schedule are explained. Not all of the obstacles to translating Lizard into nanocode are treated in this thesis but the major ones that have not been tackled are summarized.

ACKNOWLEDGEMENTS

Several people have provided me with valuable assistance in researching, writing and editing this thesis: S. Dasgupta, R. Lee, R. Liknaitsky, W.S. Luk, J. Opatrny, S. Sutphen and M.M. Sharon. I would also like to express special thanks to J.C. Demco for his careful criticisms of the first draft, and to B.J. Mailloux, my supervisor.

TABLE OF CONTENTS

* * *

TABLE OF CONTENTS (Cont'd)

LIST OF TABLES

LIST OF FIGURES

TABLE OF ABBREVIATIONS

Abbrev.		Defined in Section
ALU	- Arithmetic and Logic Unit	2.1
ALUP	- Arithmetic and Logic Unit for P-store	2.1
CIH	- Carry In Hold	2.1
COD	- Control Store Output Data	2.1
COH	- Carry Out Hold	2.1
INCP	- P-register Incrementor	2.1
MEUL	- Memory Element Usage List	4.1
MIR	- Micro-Instruction Register	2.1
MOD	- Main Store Output Data	2.1
MPC	- Micro-Program Counter	2.1
MTO	- Minimum Time from Origin	5.1
NIM	- Nano-Instruction Matrix	2.2
NOD	- Nanostore Output Data	2.1
NPC	- Nano-Program Counter	2.1
PPG	- Program-Parallelism Graph	4.1
RMI	- Rotate, Mask and Index	2.1
SP	- Scheduling Priority	5.2
STO	- Scheduled Time from Origin	5.1
TDE	- Timing Distance from End Node	5.2
TDO	- Timing Distance from Origin	5.1
UPC	- Unscheduled Predecessor Count	5.1

CHAPTER I

INTRODUCTION

The concept of microprogramming was proposed by Wilkes in 1951 [10] as a systematic and efficient way to design a computer. At that time, programmers were asking for more-powerful machine-language instructions and more-flexible addressing formats and it was becoming increasingly difficult to design and test the circuitry that was needed to decode and execute these instructions. Wilkes outlined a machine-language instruction format that required very little decoding and that could be implemented by very simple circuitry but was unsuitable for production programming. His proposal was that programs written in this easily-supported instruction format should be concealed in their own memory unit and used to decode and execute powerful conventional machine-language instructions stored in the main memory. The programs that are used to implement machine-language instructions in this way, have come to be called microprograms and the memory unit in which they are stored is called control store.

Wilkes had intended that only computer designers would write microprograms. Since each instruction works intimately with the computational circuitry of the computer, he considered microprogramming too difficult for the average programmer. In any case microprograms had to be stored in very-high-speed memories if microprogrammed computers were to have acceptable running speeds and the technology of the time could supply only read-only memories with adequate access times. Thus the contents of control store were defined at the factory and could not be modified after delivery.

The concept of a programmer being able to define the machine-language of the computer that he uses, had too much potential to remain out of reach for long. When advances in integrated-circuit technology decreased the cost of high-speed semiconductor memories, manufacturers began to design microprogrammed computers with writable control stores. Not all of the manufacturers adopted Wilkes' suggested format for micro-instructions but they kept the basic principle of a simple computer emulating a complex one. In general these computers were designed with a standard set of machine-language instructions implemented by microprograms in a read-only control store, and were provided with a writable control store that could be used to extend the standard set of instructions or to hold programs that would benefit from the very high execution speeds of a microprogram.

The Nanodata QM-1 is a user-microprogrammable[1] computer

[1] Liberties will be taken with the word "microprogram" and its derivatives until the Nanodata terminology is introduced later.

with several unique characteristics. It has no main-memory machine-language instructions defined and does not even have a preferred format for such instructions. It was designed to be used for computer architecture research [11] but its flexibility also makes it suitable for emulating a wide range of existing computers. A QM-1 has been used to replace an obsolete computer for which the manufacturer no longer provides hardware maintenance, and thus, for the cost of writing an emulator, saved a large investment in software development costs. Another QM-1 was used to assist in design testing and software development for a computer that had been designed but not yet built.

Nevertheless, as Wilkes predicted, microprogramming is a very demanding endeavor and the QM-1 must be the most difficult c[illegible]ter to microprogram. Chapter III discusses the distinc[illegible]ive [illegible]ficulties that are posed [illegible] the QM-1. Many of the [illegible]ulties seem to be strict[illegible]lerical in nature in tha[illegible]ey require ready access to a large data base of specifications on the requirements for performing each type of operation. This prompted an investigation into the feasibility of writing a translator that accepts a convenient programming language and manages all of the clerical tasks for the programmer.

This thesis proposes a language, called Lizard[1], that

[1] The name "Lizard" does not stand for anything: it was chosen for its imagery. All the acronyms that could be devised were judged to be unacceptable.

greatly simplifies the preparation of microprograms for the QM-1. A description of the Lizard language is presented in Appendix A. The important factors that governed the design of Lizard are discussed in Chapter III. Chapters IV and V outline the implementation of a Lizard translator. The mundane aspects of writing a programming-language translator are not discussed there; only the unique problems encountered with Lizard are treated. Many of these unique problems lie in areas of Computer Science for which adequate theory does not exist and some could form the basis for a complete thesis on their own. Therefore the author tries to present acceptable solutions to most of them rather than treating any single one in full detail. Chapter VI outlines some of the deficiencies of the methods presented in Chapters IV and V, and suggests a general direction for improvements.

The next chapter, Chapter II, will survey the architecture of the QM-1 emphasizing the characteristics important to this thesis and thus will provide a basis for the discussions in the chapters that follow.

CHAPTER II

THE ARCHITECTURE OF THE QM-1

The architecture of a microprogrammable computer can usually be classified as one of two types, vertical or horizontal [3]. A vertical machine has a relatively short word length for its micro-instructions and a high degree of encoding of fields in each word. A micro-instruction for a vertical machine is quite similar to an ordinary machine-language instruction except that the operation that it performs is usually more elementary. A horizontal machine, on the other hand, has a long word length for its micro-instruction and usually little encoding in the fields of that word.

Each of these two architectures has its own advantages and disadvantages, principally in the size of memory needed to hold a microprogram, the complexity of the circuitry needed to support the architecture, the speed of execution an the ease of programming. The QM-1 architecture is partly vertical and partly horizontal, thereby hopefully incorporating the advantages of both systems and the disadvantages of neither [1] (although one could argue convincingly that the opposite has resulted). These two microprogram formats are at different levels of control.

The lowest level of microprogram has a horizontal format and is used to define the instructions that will comprise the next higher level of microprogram, which has a vertical format. In Nanodata terminology, which shall be used from now on, the lowest level of microprogram is called a <u>nanoprogram</u> and only programs written for the second level of control are called microprograms. The QM-1 has a preferred format for its micro-instructions but no micro-instructions are permanently defined. Instead, circuitry is provided (Nano Address Select) that will rapidly convert a 7-bit micro-instruction opcode into the address of the nanoprogram that implements that micro-instruction.

2.1 <u>Functional Components</u>

The principal functional components of the QM-1, together with most of the data paths between them, are shown in Figure 1. The control lines from the Nano-Instruction Matrix to the computational and memory units are not shown. The 18 bit data paths are called busses and many of them have a three letter mnemonic label in italic. The meaning of these mnemonics and other abbreviations used in Figure 1 are given in Table 1 and Table 2.

As can be seen in Figure 1, the QM-1 has three bulk memory units: <u>Main Store</u>, <u>Control Store</u> and <u>Nanostore</u>. The design philosophy of the QM-1 was that Main Store would be the main memory of an emulated machine, Control Store would hold the microprograms that emulate the main-store machine,

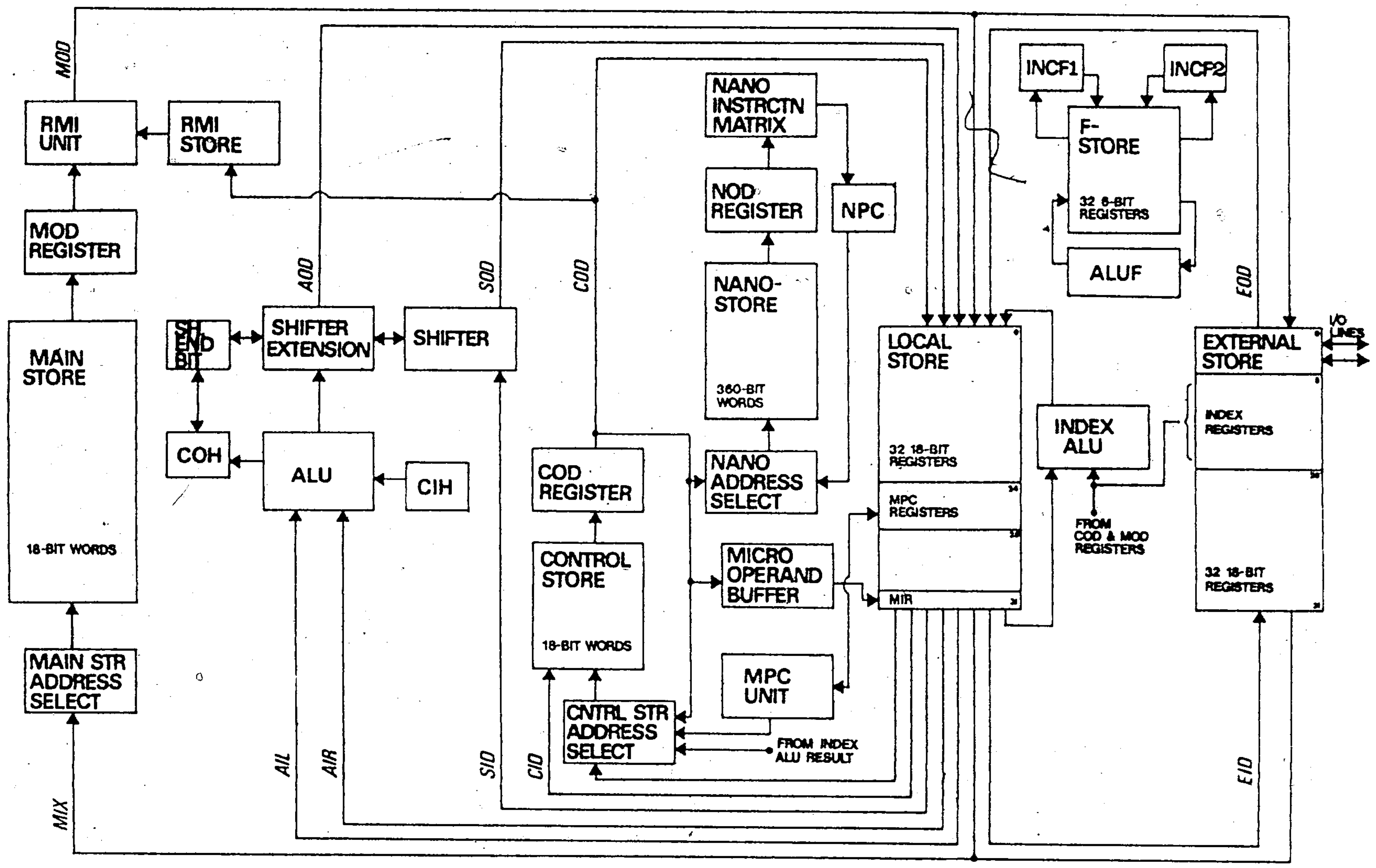


Fig. 1: QM-1 Functional Components and Data Paths

Table 1. Data Bus Mnemonics

AIL	- ALU Input Left
AIR	- ALU Input Right
AOD	- ALU Output Data
CIA	- Control Store Input Address
CID	- Control Store Input Data
COD	- Control Store Output Data
EID	- External Store Input Data
EOD	- External Store Output Data
MIX	- Main Store Input Multiplexed Carries input data words or addresses to Main Store.
MOD	- Main Store Output Data
SID	- Shifter Input Data
SOD	- Shifter Output Data

Table 2. Abbreviations in Figure 1

ALU	- Arithmetic and Logic Unit
ALUF	- Arithmetic and Logic Unit for F-Store
CIH	- Carry In Hold
COD	- Control Store Output Data
COH	- Carry Out Hold
INCF1 & INCF2	- F-register incrementors and decrementors
MIR	- Micro-Instruction Register
MOD	- Main Store Output Data
MPC	- Micro-Program Counter
NOD	- Nanostore Output Data
NPC	- Nano-Program Counter
RMI	- Rotate, Mask and Index
SH	- Shifter

and Nanostore would hold nanoprograms needed to implement the micro-instructions in Control Store. In practice, main-store machines are often emulated directly by nanoprograms in order to achieve higher running speeds. Because of the limited nature of the data paths out of Nanostore, Control Store is used to keep any constants and tables that may be needed by a nanoprogram for an emulation and it is also fast enough, that it can be used to hold the accumulators of an emulated machine.

Local Store is a bank of 32 eighteen-bit registers. These are the general-purpose registers used by a nanoprogram and there are direct data paths from Local Store to almost all the other units. Registers 24 to 27 have special connections to the MPC Unit that make them especially suitable for use as micro-program instruction counters. Their values can be easily incremented and used as addresses into Control Store. Local-store register 31 is set up to be used as a micro-instruction register, its special structure will be discussed in greater detail later.

External Store is a bank of 32 eighteen-bit registers. Some of these have special hardware functions as:

1) port registers to the eight I/O channels

2) parameter registers for the memory-segmentation unit

3) interrupt enable and program-check masks

4) interrupt pending and program-check flags

5) p[illegible] a table of interrupt-handling routine add[illegible]s.

Some are also connected to be the right input operand to the Index ALU comparison unit. All of the 32 registers are easily accessed and can be used for general-purpose storage when their hardware function does not interfere.

The Arithmetic and Logic Unit (ALU) is an asynchronous computational unit that takes two 18-bit inputs from the AIL and AIR data buses and yields one 18-bit result that passes through the shifter to the AOD bus. The ALU can perform all 16 possible logical functions of two inputs as well as many useful and curious operations involving combinations of addition, subtraction, incrementing, decrementing and logical functions. When the operation being performed requires a carry in, it comes from the CIH register and when the operation results in a carry out it appears in the COH register. The function to be performed is selected by control lines from the Nano-Instruction Matrix and fro F-store.

The Shifter can perform shift operations on single words (18 bits) or double words (36 bits). As can be seen in Figure 1, the low-order word of input is taken from the SID bus and the high-order word is taken from the output of the ALU. The low- and high-order words of output appear on the SOD and AOD busses respectively.

The Shifter and the ALU together also produce six bits of status information about the result that they have computed (overflow, zero result etc.) These six bits can be tested in a conditional branch or saved in a special register in F-store (FIST) for later testing.

The Index ALU is computationally equivalent to the main ALU but its inputs and outputs are selected in a manner that make it more suitable for certain applications and less suitable for others. Its direct connection to the Control Store Address Select unit makes it especially useful for doing computations on control-store addresses. Since the right input to the Index ALU can come only from External Store, the COD Register or the MOD Register, this unit is less suitable for general computation.

The <u>Rotate Mask and Index (RMI) Unit</u> provides a simple method of extracting subfields or operating on a word read from Main Store before it is transferred to Local Store. The data word first undergoes a right circular shift by an amount specified by a ROTATE parameter, it is then logically anded with a MASK parameter and finally added to an INDEX parameter. <u>RMI Store</u> contains three groups of the three parameters used by the RMI unit. Any of the three groups can be selected for use at any time and the contents of RMI Store can be changed as required.

<u>F-store</u> is a bank of 32 six-bit registers that are used mainly for bus control. As is shown in Figure 1 there are 12 main data buses in the QM-1 that link Local Store with most of the other computational and memory units. Each of these buses has a register in F-store associated with it. The value contained in the associated F-registers selects which of the 32 local-store registers is connected to the data bus. For example, the Memory Output Data Register is connected to Local Store by the MOD bus. The second F-store register is associated with the MOD bus and is called FMOD. The value in FMOD determines the local-store register to which the MOD bus is connected. When a nano-program does a Gate Main Store operation the contents of the MOD Register will be transferred to the local store register selected by

FMOD.

The EID and EOD busses are special cases in that there are two F-Registers associated with each of them. One of the F-registers selects the local-store register connected to one end of the bus; the other F-register selects the external-store register connected to the other end. Although the values in the F-registers select which local-store and external-store registers are connected to the data busses, actual transfer of data takes place only on command.

An F-register has six bits and therefore can store a number from 0 to 63. For most data busses if the value in the associated F-register is greater than 31 then the bus is disconnected from Local Store. However, the MIX and MOD busses to and from Main Store utilize some of the extra range so that when FMIX or FMOD has a value between 32 and 39 their associated bus will be connected to one of the first eight external-store registers. Since these are the port registers to the I/O channels, direct transfers from I/O devices to Main Store are facilitated.

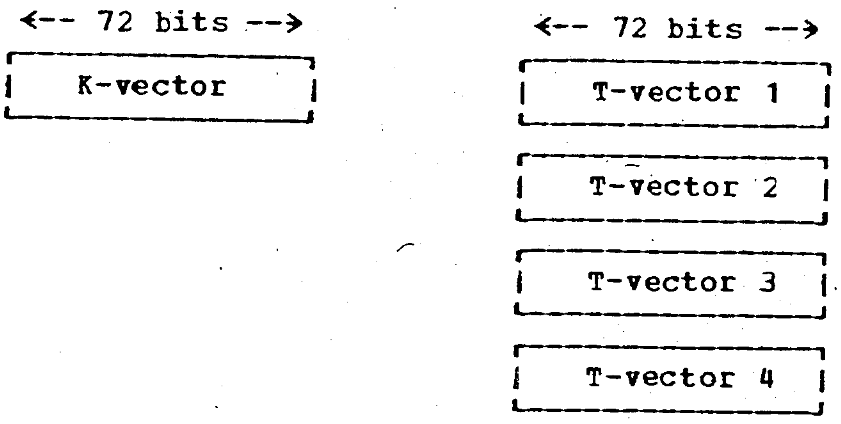
F-store has three computational units directly connected to it to perform six-bit arithmetic on its contents. There are two incrementers (INCF1 and INCF2), which can increment or decrement the contents of any F-register and there is a six-bit arithmetic and logic unit (the ALUF), which takes two F-register inputs and places the result back in an F-register.

The F-registers that are not directly involved in bus control are either used as 6-bit scratch registers or have "special" hardware functions. Note also that not all the 18-bit data paths in the QM-1 are controlled by F-registers. The inputs and outputs attached to the Index ALU are selected by fields in the active nanoword.

2.2 <u>Nanoprogram Control Structure</u>

All data transfers and computations in the QM-1 are initiated and controlled by the <u>active nanoword</u>, the one currently in the Nano-Instruction Matrix. A nanoword is 360 bits long and is divided into five fields: one <u>K-vector</u> and four <u>T-vectors</u> (see Figure 2,. Informally one could say

Figure 2. The structure of a Nanoword



that the T-vectors initiate all data transfers within the QM-1 and that the K-vector contains <u>working constants</u> used by the T-vectors. Only one T-vector in the active nanoword is in control at any time and the machine clock activates the next circularly sequential T-vector every 80 nanoseconds.

A T-vector is principally made up of single-bit fields most of which each control data flow along a particular data path. When the bit is one, data is allowed to flow from one end of the data path to the other, where it is latched into a register; when the bit is zero, no data flows. Other bits in the T-vector signal the bulk memory units to start fetch or restore cycles using the address and/or data present on their input lines. Some of the operations requested by single-bit fields in a T-vector begin as soon as the T-vector gains control (these are called "leading-edge events") and others take place just before the T-vector loses control ("trailing-edge events"). The results of a leading-edge event can often be used by a trailing-edge event in the same T-vector. There are also several multi-bit fields in a T-vector that modify some of the data transfers, usually by selecting the inputs and outputs of the data paths.

A nanoword will remain active with its four T-vectors circularly gaining control until one of its T-vectors requests that a new nanoword be transferred from the NOD

Register into the Nano-Instruction Matrix (NIM).[1] Typically the last T-vector in each nanoword will request that a new nanoword should be gated into the NIM, but since the command to gate a nanoword is a conditional command and since a T-vector can be conditionally skipped by the previous T-vector the request to activate a new nanoword can be made by any of the four T-vectors of the active nanoword.

A T-vector that requests an 18 bit operation or data transfer never explicitly specifies the registers involved. Instead they are chosen by the contents of the F-register associated with a data bus, or the contents of a K-field selected by the T-vector. However, when a T-vector requests a six-bit transfer to or from F-store, the F-register involved in the transfer is explicitly encoded in the T-vector. Up to three different F-registers can be selected by a T-vector to have values transfered into and/or out of them. There are several six-bit fields that can participate in a data transfer with an F-register:

1) A working constant field in the active K-vector (see below)
2) The A, B or C field of the Micro-Instruction Register (local-store register 31) (see below)
3) The ALUF six-bit computational unit
4) The F-store incrementers

[1] A new nanoword can also be loaded by a machine check error or by the operator from the console.

5) Some F-registers

6) The front panel switches (source only)

7) An I/O interrupting channel identification register (source only)

The first two of these seven items are the most important and deserve detailed description.

Table 3 lists the subfields of a K-vector and calls eight of them working constants. A T-vector can directly request a data transfer from one of these fields to F-store or from F-store to one of these fields. The initial values in the working constants are specified in the source nanoprogram and if they are reset by an F-store to K-field transfer then the new value is available only so long as the current nanoword remains active; Nanostore is not changed.

2.3 The Micro-Instruction Register

The Micro-Instruction Register (MIR), register 31 in Local Store, is divided into three six-bit fields called A, B and C as shown in Figure 3. These three fields roughly match the preferred micro-instruction format on the QM-1 which is: a 7 bit opcode, a 5 bit operand and a 6 bit operand. Special data paths exist to load the MIR with the operands of an emulated micro-instruction (see Figure 1) and set the upper seven bits to zero (the opcode field is removed and decoded into a nanostore address). A T-vector can then request a transfer of these operands to F-store to be used in bus control. Thus in a micro-instruction the A

Table 3. K-vector Subfields

Name	Length	Function
KN	10	Holds one nano-address used for branching.
KA	6	Working constant and I/O Port Selection field.
KB	6	Working constant.
KSHC	6	Working constant and shifter shift type control field.
KSHA	6	Working constant and shifter shift amount control field.
KALC	6	Working constant and ALU function selection field.
KS	6	Working constant and global-condition test mask.[1]
KT	6	Working constant and local-condition test mask.[1]
KX	6	Working constant and special-condition test mask.[1]
Misc.	12	Esoteric control functions.
Unassigned	2	

[1] A <u>test mask</u> is used to select the conditions under which a conditional branch will occur.

and B fields are usually used to specify local-store registers upon which the operation selected by the opcode field is to be performed. Since data can be transferred in either direction between F-store and the MIR, the MIR serves as a link between the six-bit and the 18-bit architectures.

Figure 3. Micro-Instruction Register Subfields

```
<----------------- 18 bits ---------------->
 ___________________________________________
|      C      | |        A      |      B     |
|_____________|_|_______________|____________|
|<-- 6 bits -->| |<-- 5 bits -->|<-- 6 bits -->|
               ^
               |
             1 bit
```

2.4 <u>Nanoprogram Timing</u>

All of the computational and memory units in the QM-1 are at least partly asynchronous; that is to say that their circuitry is continuously operating on the values presented on the input lines. When a Nanoprogram changes one or more of the inputs to a unit the new input values must remain stable long enough for the computed result to become valid. It is the responsibility of the Nanoprogrammer to keep track of the timing requirements of all the data trans[illegible]s and calculations that he initiates. The basic unit of time that he works with is called the <u>T-period</u> and is the amount of time that a T-vector normally stays in control (80 nanoseconds).

Consider for example a nanoprogram that uses the shifter unit to perform a single word shift operation. The steps required for this operation can be listed as follows:

1) Ensure that the KSHC and KSHA K-fields contain the correct values for the operation required (see Table 3).

2) Transfer a 6 bit value to F-register FSID to select the local-store register whose contents are to be shifted.

3) Ensure that the value in the local-store register selected by FSID is stable nd ready to be shifted.

4) Transfer a 6-bit value to F-register FSOD to select the local-store register that will receive the result.

5) Command a transfer of the shifter result to Local Store.

When any of the inputs to the shifter changes, the result that it has computed is not reliable until two T-periods have elapsed. Also, to ensure that the result is transfered to the correct local-store register, one T-period should be allowed between the setting of FSOD and the request to transfer the result. These conditions can be met in a nanoprogram by ensuring that the T-vector that performs operation 5 above is at least the second T-vector after those that perform operations 1, 2 and 3 and that operation 4 is requested in a T-vector preceeding that of operation Appendix B contains a list of all the basic operations on the QM-1, their inputs (sources) and the period of time that each input must remain stable for the result to be reliable.

Since a T-vector is 72 bits long and can theoretically request up to 28 elementary operations simultaneously, it is generally not desirable to waste T-vectors solely to meet timing requirements. Therefore, two or more computational or memory units will usually be operating in parallel and the steps required to initiate and conclude these operations will be mingled. There is also a provision for "stretching" the amount of time that a T-vector remains in control, making it last two T-periods (160 nanoseconds). Since most operations happen on the trailing edge of a clock cycle, the time between operations is doubled in this way, so that timing requirements can be met when there are not enough "housekeeping" operations to fill a T-vector.

There are three similar terms used frequently in this thesis whose subtle differences in meaning should be pointed out at this point, they are: T-vector, T-period and T-step. A T-vector is a 72-bit field in a nano-word, a T-period is the machine cycle time, 80 nanoseconds and a T-step is an event, the execution of a T-vector. A T-step can last one or two T-periods. The term T-step was invented so that one could talk of stretching T-steps since saying that one has stretched a T-vector suggests that it now contains more than 72 bits.

CHAPTER III

LIZARD LANGUAGE DESIGN OBJECTIVES

A programmer faces many impediments to rapid and efficient nanocoding in addition to the difficulties that a standard machine-language programmer would encounter.

1) Even basic operations such as addition or Main Store fetches must be programmed as many elementary instructions (nanoprimitives).
2) The elementary instructions that comprise the basic operations in a nanoprogram are arranged to be performed in parallel. It is difficult to express a parallel algorithm made up of dozens of intermingled elementary instructions so that the purpose of each instruction is clear and so that debugging is a rational process.
3) A complete table of the timing constraints that must be observed is long and not easily learned.
4) The purposes of the encoded fields in both the K-vector and T-vectors are seldom simple. Usually the encodings are indirect or doubly indirect selectors and sometimes the coded value in a field is used for two different purposes. This makes it difficult to write efficient nanocode without constant reference

to the QM-1 documentation.

How can these encumberances be eliminated by a translator without significantly reducing the power of a nanoprogram? The first problem considered was that of freeing the programmer from learning the timing constraints. Enabling a translator to manage the timing constraints is essentially a clerical matter that involves finding a systematic way to represent them; currently, they are scattered throughout the QM-1 documentation. Appendix B is a table of almost all the timing constraints that a programmer must follow. The remaining ones are context-dependent and are treated differently (see Section 4.4). The table could be made more systematic by disregarding some of the detailed information it contains or with extra programming effort could be used as it is presented. The use of Appendix B by a Lizard translator is discussed in Section 4.4.

In nanocode, timing constraints are met by placing interacting operations into T-vectors that are adequately displaced from each other. If a translator is to handle timing constraints for a programmer, it must be able to choose the T-vectors into which operations will be placed and hence must know which operations can be performed in parallel and which cannot. There are two ways that a translator can get this information: it can be supplied by the programmer, or, the translator can analyse the structure of the program to discover possible parallelism. Notations that allow a programmer to indicate parallelism in his program are cumbersome and error-prone. This awkwardness is the result of an attempt to express the two-dimensional nature of a parallel algorithm in a one-dimensional stream of instructions. The method used in nano-assembler, listing in the same statement the operations to be done in parallel, is workable only because it allows no flexibility; it does not show the parallelism possible in a program, it gives only one possible arrangement of the operations performed by the program.

Following this line of reasoning and with the aim of eliminating the second nanocoding impediment mentioned above, the Lizard language does not have syntactic constructions for expressing parallelism but relies on a translator that can detect parallelism on its own. Techniques for the automatic analysis of parallelism in a program are presented in Section 4.1.

In order to maintain as much of the power of nanocode as possible the Lizard language has statements to perform almost all of the elementary operations possible in a nanoprogram. Some could not be provided because they would limit a translator's ability to reorganize the program. Because the single operations specified in a nanoprogram are so elementary, a sequential list of them would be inordinately long. Lizard therefore allows the elementary instructions that make up a basic operation to be grouped into the same statement. Note the difference here from nanocode: in nanocode one groups operations into the same statement when they are to be performed simultaneously, in Lizard one groups operations that are logically related. Thus the five steps that are required for a shift operation, as listed in Section 2.2, could be specified in a single Lizard statement. In this way, the first impediment to nanocoding mentioned above is overcome or at least has its impact reduced.

The last impediment mentioned above, forces a programmer to use an uncomfortable amount of indirection. When nanocoding, a programmer can seldom specify the operands of an operation directly; instead he must usually specify a field whose contents will select the operands. The Lizard language allows a programmer to explicitly name his operands and for efficiency also allows him to use indirection if he cares to. Also, an alert nanoprogrammer will occasionally notice opportunities to use values encoded in indirect operand selectors, for more than one purpose. If he takes advantage of these opportunities he runs the risk of being forced to drastically modify the organization of his program should even minor changes be needed later. A Lizard translator could easily be made to recognize sharable fields with no inconvenience to the programmer.

3.1 The Language Level of Lizard

How high should be the level of the language accepted by a translator to nanocode? To answer this question one must consider the applications to which the translator will be put. If the translator is to be used for production microprogramming then efficiency is the utmos [illegible] consideration. An inefficient emulation will [illegible] the performance of all programs run on the emulated c[illegible]r. Efficient nanocoding is difficult unless the progra[illegible] as intimate control of every register and function in the machine. This type of control would be difficult to provide in an algorithmic language. The efficiency of the output code would depend on a translator that used highly sophisticated optimization techniques if the language was to bear more than a superficial resemblance to an algorithmic language. On the other hand, if the translator is to be used in computer-architecture research situations, where programming time is a more important consideration than running speed, then an algorithmic language would be the most desirable.

In any case the problem of translating a sequential assembly-like language to nanocode is a significant subset of the problem of translating an algorithmic language to nanocode and is not an unreasonable point at which to begin. Furthermore, the process of translating an algorithmic language to a sequential assembly-like language is a well developed topic in computer-science literature.

The main consideration in designing the format of a Lizard statement was regularity of syntax. A scheme whereby Lizard statements resembled English sentences was rejected because the relationship of keywords to operands became more intricate as the meaning of the statement became more explicit. A simpler format was chosen whereby each Lizard statement has a header part which is followed by a list of keyword parameters. The header part, in general names an operation that is commanded by the setting of a single bit in a T-vector. The keyword parameters indicate the settings of encoded fields in a T-vector or K-vector that modify the operation specified by the header. The parameters are also used to specify the transfer of values to F-registers. These F-register transfers could be specified in separate statements, but coding them in the same statement as the operation for which they are being performed, makes a program shorter, and possibly easier to read.

3.2 Some Specific Design Considerations

Branch Statements. Each nanoword in a nanoprogram must contain the operations necessary to load the Nano-Instruction Matrix with the next nanoword to be executed. Thus, in order to maintain normal instruction flow, a straight-line segment of a Lizard program that, when translated, occupies more than one nanoword, must have extra operations inserted into it that were not specified by the programmer. These operations would set up and use the contents of the Nano-Program Counter and the NOD Register. Because of this, Lizard is not able to give full control of these registers to the programmer but instead forces him to identify his branch operations in a single statement and expands these into nanoword fetch sequences.

Main-Store Operations. Main-store accesses are a fundamental part of most nanoprograms but are also one of the more complicated basic operations on the QM-1. Simple main-store reads and writes use two wait loops, one for the memory-fetch cycle and one for the memory-restore cycle. To simplify these operations for a programmer Lizard has single-statement main-store operations that expand into the appropriate wait loops. This is a violation of the principle that Lizard should be an assembly-like language but perhaps a justified one. It would be possible to supply a Lizard programmer with more elementary main-store statements but this would eliminate the possibility of certain types of optimization by the translator and would be more complex for a programmer to use.

Micro-Instruction Fetches. In a set of nanoprograms that implements a set of micro-instructions, the most commonly recurring sequence of operations will be the one needed to fetch the next micro-instruction and jump to its emulation. Because this sequence is not trivial, Lizard has statements that expand into the two common microfetch conventions: the simple-fetch convention and the prefetch convention. The form of these statements is flexible enough that a programmer can specify almost any combination of the operations that can be part of a microfetch, and thus, almost any microfetch convention could be invented.

The Preprocessor. The Lizard language definition includes preprocessor expressions that are evaluated into simple integers or strings and a provision for conditionally including or excluding a set of Lizard statements from a compilation. The familiar macro facility common in assembly languages, is not provided. In general, in order to use Nanostore economically, a sequence of steps that is needed more than once should be made into a nanocode subroutine or a common tail word. There are a few cases where this is not possible, such as micro-instruction fetches, but these are taken care of by special Lizard statements.

The Lizard language has hundreds of keywords but has no reserved words. This is a result of the fact that user-defined variable names are needed only in preprocessor expressions; the actual program must use register names. Since Lizard preprocessor expressions are adequately delimited from Lizard statements, a translator will have no trouble distinguishing variable names from keywords.

Translator Output. The Lizard language definition in Appendix A assumes that a Lizard translator will output nanocode, suitable for input into the nano-assembler, rather than a binary load module. There are several reasons why this seems to be the best strategy.

• Translating a Lizard program into a nanoprogram will be a slow and costly process. If the output is a readable nanoprogram then a programmer will be able to make minor modifications directly to the nanocode rather than to the Lizard source.

- When a major change to a Lizard program is made a programmer would want to be able to retranslate only those modules in which changes had been made. But at present there is no nano-linkage editor and all interacting parts of a nanoprogram must be nano-assembled together in order to get a nanoprogram load module.
- It seems highly unlikely that a Lizard translator could ever produce as efficient nanocode as a good human programmer. A programmer might therefore want the option of being able to optimize critical portions of the translator output and mix translator-produced nanocode with hand-written nanocode.

To facilitate the mixing of nanocode modules, the nano-assembler label syntax was adopted for Lizard. By declaring a label in a Lizard program to be an entry point (via the Lizard ENTRY statement) any label in a Lizard program may be used by an external nanocode module. Declaring a label in an ENTRY statement ensures that the label will fall on a nanoword boundary. Thi may not happen for an internal label if the branch operation that uses the label can be translated into a skip operation rather than a branch. The translator will also ensure that the label appears in the output nanocode even if there are no references to it. By declaring a label to be external (via the Lizard EXTERNAL statement) any label in an external nanocode module may be used by a Lizard program.

3.3 Steps in the Translation of Lizard

In the subsequent chapters, each aspect of translating a Lizard program into nanocode will be examined. The reasons for some of the steps will not be clear until the problems that arise in other steps have been examined. Ignoring these complications, the process of translating a Lizard program into nanocode can be summarized as having four major parts.

1) Expand each Lizard statement into its individual elementary operations.
2) Build a graph to represent the possible parallelism in the source program.
3) Insert into the graph, the timing constraints that must be observed
4) Schedule the operations in the graph into T-vectors and output the generated nanocode.

The first item in this list is fairly simple: Appendix A contains the details needed for doing this. Items 2 and 3 are discussed in Chapter IV and the last one is discussed in Chapter V.

CHAPTER IV

BUILDING THE PROGRAM-PARALLELISM GRAPH

4.1 Detecting Parallel Operations

A great deal of research has been done on the problem of recognizing possible parallelism in a program, most of it with the aim of implementing a high-level language compiler for a parallel-processor environment. Baer [6] presents the conditions which guarantee that two program statements can be performed in parallel. He classifies the memory elements[1] used by a statement S(i) into two sets:

1) I(i), which contains the memory elements used only as inputs by S(i), and
2) O(i), which contains the memory elements used as outputs by S(i).

Throughout this thesis the members of the set I(i) are called sources or inputs and those of O(i) are called sinks or outputs.

Two statements S(1) and S(2), coded sequentially in that order, may be executed in parallel (with any type of overlap) or have their order of execution reversed if three

[1] The term "memory element" refers to registers and bit flags as well as memory locations.

conditions are met:

$$O(1) \cap I(2) = \phi \qquad (4.1)$$

$$I(1) \cap O(2) = \phi \qquad (4.2)$$

$$O(1) \cap O(2) = \phi \qquad (4.3)$$

where: "$\cap$" represents intersection and
"ϕ" represents the empty set.

The first condition ensures that S(2) does not require results computed by S(1). The second condition ensures that S(2) will not destroy the contents of memory elements input by S(1) before they are used. The last condition ensures that the final state of all memory elements will be the same regardless of the order in which S(1) and S(2) are executed.

For a continuous segment of code containing no branches or loops these conditions can be tested between all pairs of statements to discover which statements must retain their coded order of execution. The result could be represented by a graph where the nodes are statements and a directed edge between two nodes indicates the order in which the statements must be executed. I shall call this graph the Program-Parallelism Graph (PPG).

For this method to be correctly applied to a Lizard program, it must be modified slightly. Baer's definition of the set I(i) excludes memory elements that are inputs to a statement if they are also outputs. This is done to ensure that I(i) and O(i) contain no common members, since otherwise some nodes in the graph would have two edges linking them. For instance, if statement S(1) used a memory element M as both a source and a sink, then conditions 4.2 and 4.3 would each generate an edge to statement S(2) if M was a member of O(2). Similarly if S(2) used a memory element N as both a source and a sink then conditions 4.1 and 4.3 would each generate an edge from statement S(1) if N was a member of O(1). For the type of application that Baer was concerned with, the condition which generated an edge between two nodes is irrelevant, since all edges have the same meaning. For analysing parallelism in a nanoprogram, however, the condition that generated an edge between two nodes must be known because each imposes its own type of timing constraints on the scheduling of two nodes (see section 4.4). Thus the definition of I(i) should be modified so that it includes all memory elements used as inputs by S(i) even if they are also used as outputs. Allowing two nodes to be linked by two different edges does not cause any serious problems, although it may slow down subsequent processing of the graph produced. The extra edges could be eliminated by a "patch" to a program building the graph, that allows only the edge with the most severe timing constraints, to be inserted.

For most applications, an analysis of parallelism must include a test that ensures that two statements to be executed simultaneously do make any conflicting usages of computational units. There are two characteristics of the QM-1 that make such a test unnecessary here. 1) Most computational units in the QM-1 can be used only by also using the associated F-registers. Two statements that both use such a unit will thus have an inherent memory-element usage conflict and will have their sequential order preserved. 2) Each computational unit can be used only by setting specific bits or fields in a T-vector; therefore, conflicts can be detected and eliminated at scheduling time and need not be discovered while building the PPG.

A preliminary algorithm to build the PPG will now be presented. The efficiency of the algorithm will not be analysed; it is presented principally to provide a basis for further discussion. Changes will be made to the algorithm as the discussion proceeds.

<u>Algorithm 4.1</u>

Maintain a Memory Element Usage List (MEUL), initially empty, with records of the form:

```
┌───┬───┬───┐
│ M │ N │ F │
└───┴───┴───┘
```

where:

M is a memory element code

N is the number of a node in the PPG that uses the location named by M

F is a flag indicating whether N uses M as an input, an output or both

Process the statements in the input program sequentially from first to last and perform the following steps at each one:

1) Create a node to represent the current statement in the Program Parallelism Graph.
2) If any memory element in the input set I(i) of the current statement appears as an output memory element in the MEUL then create an edge from the node that uses it as an output, to the current node. This applies condition 4.1.
3) If any memory element in the output set O(i) of the current statement appears as an input or output memory element in the MEUL then create an edge from the node that uses it, to the current node. This applies conditions 4.2 and 4.3.
4) For each memory element in the input set I(i) and the output set O(i) of the current statement add one record to the MEUL to indicate that they are used by the current statement.

By applying this algorithm, every usage of a memory element will be tested against every other usage for violation of the three conditions 4.1, 4.2 and 4.3. When this algorithm is used by a Lizard translator it should be applied to the elementary instructions generated by each Lizard statement, not to the Lizard statements themselves.

To simplify further processing and use of the PPG produced by algorithm 4.1, each independent program unit should be headed by a BEGIN node and trailed by an END node.

An edge will be added to the PPG from the BEGIN node to each node with no predecessors, and another from each node with no successors, to the END node.

4.2 QM-1 Memory Elements

The memory elements that can be referenced by an elementary operation in a nanoprogram are listed in Table 4.

Table 4. QM-1 Memory Elements

no.	Memory element
1 - 32	32 F-registers
33 - 64	31 local-store registers
65 - 67	A, B and C fields of LSR(31)
68 - 99	32 external-store registers
100 - 108	9 RMI registers
109	COD Register
110	MOD Register
111	COH
112	CIH
113	Micro-Operand Buffer
114	Main Store contents
115	Control Store contents
116	Nanostore contents
117	NOD Register
118	NPC
119	Super Direct MS Access bit
120 - 127	KA, KB, KSHC, KALC, KSHA, KS, KX, KT
128	ALU Status Enable
129	Shifter status enable
130	Direct MS Access bit
131	KN
132	IO ID
133 - 140	8 External Channel Registers
141	Interrupt bits

If a numeric code were assigned to each of these memory elements the code could be used in the M field of a MEUL record. Unfortunately there are certain complications that should be pointed out. Local-store register 31 is represented in the table by its three subfields A, B and C (see chapter 1). Each of them individually, A and B together or A, B and C together can be referenced as a source or sink by a statement. Thus the MEUL would have to be constructed so that a usage of LSR(31) as a source or sink would be recognized as a use of A, B and C, and a usage of A would imply the use of LSR(31) but not of B or C. A similar problem arises with local-store and external-store register references. As will be elaborated in section 4.3, it is often not possible to determine exactly which local-store or external-store register is being used by an operation. Thus an entry in the MEUL must be able to specify the set of registers that could be a source or sink to an operation. It is even possible for a main-store operation to use a register as a source or sink and not specify whether the register it is using is in Local Store or External Store.

For these reasons the M field of an entry in the MEUL can not be a simple code that names the memory element used by a node. One viable approach would be to have all local-store and external-store locations (including A,B and C) represented by the same code in an MEUL record, but when that code occurs, the record also contains a 66 bit mask that specifies exactly which of the registers and fields this entry could be naming. The mask could be easily compared with another mask when it was necessary to test whether the set of sources and sinks of two statements intersected. It is important to note that the mask would not name all the possible sources and sinks of a node but would instead be describing a single source or sink to a node.

The three bulk-storage units, Main Store, Control Store and Nanostore appear in Table 4 as a single memory element. This may seem to drastically reduce possible algorithm parallelism but, in fact, does not. Any reference to a bulk-storage unit must utilize specific memory elements (F-registers and output-data registers). Conditions 4.1, 4.2 and 4.3 will create edges to connect nodes that use these memory elements and will thereby always disallow the interchanging of the order of execution of two bulk storage accesses to the same unit.

Some of the memory elements listed in Table 4 cannot have their value changed under program control and thus would seem not to belong in the table. However, their functions are such that it is convenient to treat them as modifiable in order to simplify the structure of the Lizard language and the steps in the scheduling process.

The eight External Channel Registers (133 to 140 in table 3.1) are part of the channel controllers. There is no single physical register in the controllers that could be called an External Channel Register but for the purpose of Algorithm 4.1 this treatment is adequate. They were invented so that the I/O line transmission delay time (5 T-periods) could be handled uniformly with all other timing constraints.

4.3 Determining the Sources and Sinks of a Node

To apply Algorithm 4.1 to a Lizard program, one must be able to determine which of the memory elements in Table 4 are used as sources and sinks by each of its elementary operations. Appendix B lists all the possible sources and sinks of each type of elementary operation and the conditions under which a memory element will actua a source or sink for each operation.

Note however that usually when Local Store or External Store is the source or sink of an operation the actual register being referenced is selected by the contents of an F-register. Thus they are indirectly addressed registers (IA registers). Since the value in an F-register can be determined at run time as the result of a computation of arbitrary complexity, a compiler would not in general be able to determine which of the IA registers are actually used by an operation. Indeed, because of the power of the computational units attached to F-store, a compiler could not always determine whether the computations on F-Store even terminate. This is a fundamental problem with applying traditional parallel-analysis algorithms to nanoprograms.

The situation is not as grave as it would seem at first glance. It is actually seldom useful to select, by computation, the registers that one is using in a program. To take advantage of this fact, Lizard statements that use a local-store or external-store register as a source or sink are designed to allow the programmer to explicitly specify which register is being referenced. When this feature is used, then the compiler will know exactly what registers are sources and sinks and the application of Algorithm 4.1 is simple.

There are nevertheless some cases where a programmer cannot specify the registers that he is using in a statement. For example:

1) When a nanoprogram is used to implement a micro-instruction, the A and B fields of the micro-instruction usually select local-store registers that are to be operated on. In this case there are often some constraints on the possible values that A and B can contain, but it must be possible to handle a range of values.
2) Part of a nanoprogram could be a subroutine used in several places by the nanoprogram. In this case the subroutine could be expected to receive parameters (passed via F-store, HOLD or HOLD2) that selected the registers to be operated on by the subroutine.
3) There may exist algorithms whose implementation is greatly shortened if their register operands are selected by computation.

The easiest way to handle all three cases is to treat each imprecise reference to Local Store as a possible reference to any part of Local Store. If this were done, Algorithm 4.1 would retain the sequential ordering of references to IA registers that was coded in the source program. This is a rather drastic solution; remember that the local-store registers are the general-purpose registers of a nanoprogram and hence one does not want to overly restrict possible parallelism in their use.

Another approach is to expect the programmer to use the Lizard-language feature that allows him to declare the set of registers that contains each IA-register reference. For most applications this would not be an excessive hardship, especially if program data-flow analysis was used to assist him. Program data-flow analysis (see Allen and Cocke [15]) was devised to assist in the optimization of the code produced by algorithmic-language translators This procedure computes the range of statements that can be affected by each value-assignment statement in a program. It can be used by a Lizard translator to propagate through a program, information that a programmer supplies about a register selector that he is using.

4.4 Timing Constraints

The connectivity graph produced by Algorithm 3.1 shows the precedence relation of two nodes but does not contain the timing constraints placed on their scheduling. Because the timing constraints will be required by two independent passes over the graph it is convenient to compute them once and store them in the graph itself. Each directed edge in the precedence graph will have a number associated with it that will specify the minimum number of T-periods that must elapse between the execution of the nodes that the edge connects. This number is called the "timing distance" between two nodes. Each of the three conditions specified in equations 4.1, 4.2 and 4.3 generates a particular type of directed edge which shall be called type 1, 2 and 3 respectively. Each type   edge has its own reason for placing a timing restriction on the relative scheduling of its predecessor and successor nodes.

A type 1 edge connects two nodes if the predecessor node is preparing a value in a memory element for the successor node. As explained in Section 2.3, a new value placed in a register must remain stable long enough to allow it to propagate through the circuitry attached to that register. The timing distance for a type 1 edge depends on the type of memory element being prepared and the type of the successor node. These times are shown in Appendix B.

The notation used in Appendix B, works well for all cases except one. If the predecessor node is an operation in the 18 bit domain that sinks to LSR(31) and the successor is an operation in the 6 bit domain that sources from the A, B or C subfields of LSR(31), then the timing distance between the nodes must be two T-periods rather than one. This is the only case where the timing distance for a tvpe 1 edge depends on the type of the predecessor node.

A type 2 edge states that its successor node may not overwrite some memory element whose current value is needed by its predecessor. The timing distance for a type 2 edge is usually zero T-periods, that is, the successor node can be scheduled into the same T-period as the predecessor node. Again this is due to propagation delays through the circuitry; the change in value caused by the successor node will come too late to affect the operations that use the old value. Some elementary operations have sources whose value <u>cannot</u> be changed in the same T-step that they are used. For example the value of FMOD cannot be changed in the same T-step in which a GATE MS is performed. All such case are indicated in Appendix B.

A ty 3 edge states that the two connected nodes both output values into the same memory element and that the value prepared by the successor node is the one that must ultimately remain. The timing distance required by a type 3 edge is one T-period, which means that the two nodes must be scheduled into different T-steps.

With the addition of the timing constraints to the Program Parallelism Graph the course of further processing diverges drastically from other parallelism analysis algorithms. Some edges in the graph that would be redundant now contain essential information. And the inclusion of branch statements in the the PPG becomes more complicated, as shall be discussed next.

4.5 Branch Statements

Nanocode is probably the only machine language in which branch operations are not really needed. This results from the fact that the QM-1 has two levels of microprogram control (microcode and nanocode) and that a conditional jump micro-instruction can be implemented by a nanoprogram that has no branch or skip instructions (see appendix C). The set of micro-instructions that could be implemented would be somewhat restricted but could still be used to emulate any desired main-store machine. However it is not the purpose of Lizard to impose this type of constraint on nanoprogrammers, so let us now consider the problem of representing branch statements in the program connectivity graph.

The method of detecting parallel operations described in Section 4.1, was proposed only for branch-free code segments but it can be adapted to work for programs containing branches. Consider a program as being divided into sections where each branch statement or label statement in the program is a boundary between two sections. The algorithm used to build the PPG must meet several requirements. 1) It should find parallelism between statements in the same section but keep the sections independent and in the same order as in the source program. 2) Branch operations have no output values but they do have inputs. The algorithm must ensure that the timing constraints for these inputs are met. 3) Timing constraints must be observed between statements that are logically adjacent even if they are lexically separate, and between statements on either side of a branch statement. The following paragraphs will explain these requirements and discuss how they can be met.

When a nanoprogram is executing, several elementary operations and data transfers can be going on in parallel, but there is only one flow of control in the program. For this reason there is no advantage to representing two alternative control paths of a program as independent branches of the PPG. The two control flow paths must eventually be placed sequentially into logically contiguous streams of T-steps and the order in which they appeared in the source program is as good as any. Thus, when Algorithm 4.1 encounters a branch statement it should create an edge from all nodes that currently have no descendants, to the brar.h node. Also, all nodes generated by the program section that follows the branch should be made descendants of the branch node (not necessarily immediate descendants). The edges to and from a branch node that were nct generated by Conditions 4.1, 4.2 or 4.3 are called control-flow edges. The same should be done for labelled statements except that a label node should be created and the control-flow edges attached to this node rather than the nodes that might be generated by the labelled statement. A control-flow edge leading to a branch node <u>or</u> from a label node should specify a timing constraint of zero T-periods since the predecess and successor of this type of edge can be scheduled into the same T-vector. A control-flow edge leading from a branch node <u>or</u> to a label node should specify a timing constraint of one T-period since such edges connect nodes that must be scheduled into separate T-vectors.

The addition of control-flow edges will preserve the ordering that existed in the source program of control nodes with any other node and thus will strictly maintain the control structure of the program. However there is still the problem of maintaining adequate timing distances between nodes that are lexically separate but logically adjacent. Consider for instance the following Lizard program segment:

```
 1            ALU, OP=ADD, LEFT=LSR(5), RIGHT=LSR(6),  #
                   RESULT=LSR(4)
              ¢ TEST FOR RESULT NOT ZERO ¢
 2            GO TO PAST, COND=RNZ, COND TYPE=LOCAL
 3            ALU, OP=INCR LEFT, LEFT=LSR(4), RESU[illegible]LSR(4)
 4    PAST:   NO OP
 5            SHIFTER, OP=LEFT ARITHMETIC, AMOUNT=1,  #
                   INPUT=LSR(4), RESULT=LSR(4)
```

This p[illegible]

1) Ad[illegible]he contents of local-store registers 5 and 6 an[illegible] stores the result in local-store register 4.
2) Branches to the statement labelled PAST if the result computed is not zero.
3) Increments local-store register 4 by one.

4) Uses a NO OPeration statement to hold the label PAST in order to simplify the discussion of timing constraints.

5) Shifts the contents of local-store register 4 left, one position, arithmetically.

The character "#" is the statement continuation marker.

Statement 5 uses local-store register four as an input and hence, according to Appendix B, should be scheduled two T-periods after statement 3 and two T-periods after statement 1. To meet the second timing constraint the scheduler must insure that the timing distance from statements 1 to 2 plus the timing distance from statement 4 to 5 will be at least two T-periods. Timing constraints like these will not always be as difficult to violate as they are in this example, nor will they always be as easy to discover. The problem is especially complicated by label nodes that are the target of backward-looking branch statements and by shared code segments.

A general-purpose solution, that will correctly handle all possible uses and abuses of branch statements, would be to ensure that the timing distance from a label node to all its successors (not just immediate successors) is sufficient to meet all possible requirements of the successor node. This is not really a bad solution to the problem. Label nodes are considered to be leading-edge nodes and there are usually many housekeeping operations (6-bit transfers) to be performed between the label node and a major computation node, that will require only one T-period of timing distance from the label node. Thus few T-vectors are likely to be wasted. Note also that simply stretching a labeled T-step will provide enough timing distance to schedule almost any operation into it; only those operations requiring three or more T-periods since the setting of their sources (and there are very few) would be excluded.

4.6 Storing the Program-Parallelism Graph

A large graph such as the PPG, is usually stored as a boolean connectivity matrix, because this permits the processing algorithms to be fast and simple. In this method of storage, if there are n nodes in the graph then an n by n boolean matrix M is required. The entry in the i-th row and the j-th column of M, (i.e., M(i,j)), is set to one (true) if there is an edge from node j to node i, otherwise M(i,j) is zero (false). This method could be adapted for storing the PPG by using M(i,j) to hold the timing constraint imposed by the edge from node j to node i, rather than simply a zero or one. Since zero is a valid timing constraint for an edge in the PPG, some other value must be used in M to represent the fact that no edge exists between two nodes.

If a connectivity matrix for a PPG was stored as a full square array it would occupy a great deal of storage. A Lizard program that generates 250 nodes (approximately 250 nanoprimitives) would not be unusual for its size but it would require 250^2 (= 62.5 K) bytes of storage for its connectivity matrix alone. This would occupy almost half of the memory of the Nova emulation that supports the QM-1 software. Fortunately a connectivity matrix is usually very sparse. Suppose we represent the density of a connectivity matrix by a density factor f, defined as $f = e/n$, where: e is the number of edges in the graph that the matrix represents, and n is the number of nodes. At the end of the next section, I show that with certain proposed improvements to Algorithm 4.1, the lower and upper bounds on f will be 1 and 5 respectively, and that 3 can be taken as an acceptable expected value. This means that the connectivity matrix would have less than 5n non-null entries and thus would be less than $5n/n^2 = 5/n$ full. Therefore a 250-node Lizard program would generate a connectivity matrix that is less than 2% full. The use of sparse matrix storage techniques seems to be called for but would result in a trade-off: the size of the longest permissible Lizard program would be increased but the translation process would be slower.

Suppose a linked-list strategy was adopted for storing the sparse matrix that represents the PPG. The processing algorithms to be presented in Chapter V must be able to determine both the immediate predecessors of node k, which are indicated by the entries in row k of M, and its immediate successors which are indicated by the entries in column k. Therefore a two dimensional linked list strategy would be needed where entries in the matrix could be reached either by following down the column links or across on the row links. This type of storage for sparse matrices is described by Knuth [7, pp 299 - 301]. Each record in the linked lists would have five fields:

1) row number of this record (2 bytes),
2) column number of this record (2 bytes),
3) pointer to next record in this row (2 bytes),
4) pointer to next record in this column (2 bytes), and
5) the timing constraint of the edge represented by this record (1 byte).

Thus each edge in the PPG would require nine bytes of storage and there would be an additional overhead of 4 bytes per node for pointers to the first record in its column and its row. This would mean that a connectivity matrix with a density factor f and n nodes would benefit from this storage technique if $(9f+4)n$ was less than n^2; i.e., n must be greater than $9f+4$. Thus a connectivity matrix with a density factor of 3 and more than 31 nodes, could be stored more compactly as a linked list matrix than as a full matrix.

Another way to store the PPG would be as a linked graph structure. Since the processing algorithms to be presented in Chapter V traverse the graph both from top to bottom and from bottom to top, each node in the PPG would need two lists associated with it: one that listed all the immediate descendants of the node and the timing constraints of the edges to them, and the other a similar list of the immediate predecessors of the node. Each record in these lists would have three fields:

1) a pointer to the immediate-descendant (or immediate-predecessor) node (2 bytes),
2) the timing constraint of the edge (1 byte),
3) a pointer to the next record in the list (2 bytes).

Thus an edge could be represented by 5 bytes in one of these lists. Note, however, that every edge in the graph would appear in two lists: the descendant list of its predecessor and the predecessor list of its descendant. The entries would be unique and could not be shared. Thus every edge would require 10 bytes of storage and with the two list pointer fields in each node, the PPG would uire (10f+4)n bytes of storage. A linked graph would therefore not be as compact as a sparse connectivity matrix but would probably require the same amount of time for processing.

4.7 <u>Eliminating Redundant Edges</u>

Many algorithms exist for finding the transitive reduction of a boolean connectivity matrix (Baer [16], Simoes [17], Hsu [18], Aho et al. [22]). When used they will eliminate an edge between two nodes if the nodes are also connected by a path of length greater than one. (The connectivity representation of a node allows only one edge between two nodes hence there cannot be another path of length one.) The edges removed in this fashion are truly redundant and their only effect is to slow down further processing of the connectivity matrix. Nevertheless, the transitive reduction algorithms have a computational complexity of order n^2, whereas the algorithms that use the matrix later have complexities of order n. Thus it is doubtful whether time spent reducing the matrix would be regained in later processing. Furthermore, it would not be trivial to modify these algorithms so that they can be used to reduce a graph with weighted edges such as the PPG. In any case in the course of the reduction process, they produce the transitive closure of the connectivity matrix which is by no means a sparse matrix. This would eliminate the possibility of using sparse-matrix techniques for storing the PPG.

If one is using a sparse matrix for storing the PPG then it is more important to keep the number of its edges to a minimum from the very start, rather than eliminating them afterward. Algorithm 4.1 lends itself to some simple modifications that will both reduce the number of edges in the PPG and keep its Memory Element Usage List (MEUL) from growing excessively long. As it has been described, Algorithm 4.1 will compare each usage of a memory element by a node with every lexically preceding usage of the same memory element, and test for the violation of Conditions 4.1, 4.2 and 4.3. However this would mean a great deal of needless comparison if label nodes are treated as described in Section 4.5. The purpose of creating an edge between two nodes is to ensure: 1) that they are scheduled in the correct order and 2) that they are separated by an adequate timing distance. Since all nodes lexically preceding a label node will be a predecessor of the label node, and all nodes lexically following a label node will be successor with adequate timing distance from the label to meet all possible requirements; therefore there is no need for any edges joining nodes that appeared on different sides of a label node. Thus, when Algorithm 4.1 is sequentially processing the nodes generated by a program, and a label node is encounterred, it can discard the entire MEUL, since memory element usages that precede the label node are no longer important.

An interesting property of the three parallelism conditions can also be used to reduce the number of edges in the PPG. Conditions 4.2 and 4.3 taken together, ensure that if a node N(j) uses a memory element W as a sink, then it will have an edge to it from all lexically preceding nodes that use W as either a source or a sink. Furthermore, Conditions 4.1 and 4.3 taken together ensure that node N(j) will also have an edge from it to all lexically succeeding nodes that use W as either a source or a sink. Thus, considering only the precedence information that an edge implies, memory element W should not be allowed to cause the insertion of an edge from node N(i) to node N(k), if N(i) precedes N(j) and N(k) follows N(j).

Consider now the timing information contained in an edge. When W would have added a type 2 or 3 edge from N(i) to N(k) then a type 3 edge will exist between N(j) and N(k). Since a type 3 edge carries a more severe timing constraint than a type 2 edge, no edge is needed between N(i) and N(k). Similarly in cases where W would have added a type 1 edge from N(i) to N(k) then one of two conditions will result, either: 1) a type 1 edge with the same timing constraint will be created from N(j) to N(k), or 2) the sum of the timing distance of the edge from N(i) to N(j) with the timing distance of the edge from N(j) to N(k) will be greater than or equal to the timing distance of the edge that would have been created from N(i) to N(k). Thus no edge is needed from N(i) to N(k). This rather elaborate proof can be replaced by a simple intuitive argument. If a node replaces the contents of a memory element by sinking to it, then no node that follows it can have any interest in a previous value that the memory element may have contained.

Algorithm 4.1 can easily make use of this property of the three parallelism conditions. After Algorithm 4.1 has added all the required edges to node N(j) then all existing references in the MEUL to the sinks of N(j) should be deleted, then the sources and sinks of N(j) can be added to the MEUL. By deleting entries in the MEUL the total number of edges in the PPG will be reduced.

Let us now attempt to estimate the density factor f of the connectivity matrix that the improved Algorithm 4.1 would produce. Type 1 edges represent the fact that the predecessor node is preparing the value of a memory element for the successor node. Every node will have at least one type 1 edge leaving it and a few will have two or three. A PPG in which every value placed in a memory element was used later by two different nodes would be unusual. Thus for the total number [illegible] 1 edges per node we can estimate a lower bound o[illegible]d an upper bound of two.

Type 2 a[illegible]pe 3 edges can only arise for memory elements that have their values set more than once between label nodes. With the changes that have been made to Algorithm 4.1, a node cannot ha more than one type 2 edge or one type 3 edge leaving it, and there is a good probability that it will have none. Thus we can estimate the lower bound for the total number of each of these types of edges as being zero, and the upper bound as one.

Control flow edges are the edges added to divide a program up into sections delimited by branches and labels, and thus their number depends primarily on the number of branches and label nodes. A PPG with no control flow edges could still represent a useful program and thus the lower bound on the number of control flow edges is zero. It would be difficult to imagine a PPG in which every node had a control flow edge attached to it, thus one control flow edge per node would be a liberal upper bound. If we sum all of these lower and upper bounds on the number of each type of node then we get a lower bound of 1 and upper bound of 5 on the density factor **f**. (Five is not an absolute upper bound but it is a sensible upper bound for non-contrived programs.) An actual density factor of about 3 would not be unreasonable to expect.

CHAPTER V

SCHEDULING THE PROGRAM-PARALLELISM GRAPH

The previous chapter dealt with building the Program Parallelism Graph(PPG), this chapter is concerned with placing the operations represented by the nodes of the PPG into nanowords. The method proposed here is basically "demand-list scheduling" a standard technique used for scheduling tasks in multiprocessor environments. This method has been shown to produce non-optimal results (see for example Kohler [8]) but it may be that no cptimal algorithm for this type of problem exists [14].

5.1 Demand-List Scheduling

To apply demand-list scheduling to a Program-Parallelism Graph representing a nano-program, the following data structures and items are needed:

1) a list of nodes whose predecessors have all been scheduled, called the Schedulable-Node List (SNL),
2) a nanoword image consisting of a K-vector image and four T-vector images, and
3) a field in each node of the PPG, called the unscheduled-predecessor count (UPC).

Initially the begin node of the PPG is placed on the

Schedulable-Node List and the first of the four T-vector images is made the "current" T-vector image. Nodes are taken from the SNL one at a time and fields in the K-vector image and the current T-vector image have their values set according to the operation specified by each node. If the K-vector or T-vector fields needed by the node being scheduled, are already allocated to another previously scheduled node then a test is made to see if the values encoded in the disputed fields can be shared by both nodes. If they cannot then the unscheduled node will be blocked from placement in the current T-vector.

Since the begin node does not specify any operations it can be trivially scheduled into a T-vector without setting any bits or fields. When a node from the SNL is scheduled into a T-vector, the UPC field of each of its successors is decremented by one. When the UPC field of a node has been decremented to zero, indicating that it has no more unscheduled predecessors, then that node is placed on the Schedulable-Node List. When no more nodes from the SNL can be placed in the current T-vector image, it is retired and the next T-vector image is made current. When all four T-vector images have been retired the whole nanoword image is retired; the operations encoded in it are converted to nano-assembler format, and output.

This is the basic process required for scheduling the PPG into nanowords; however, no mention has been made of the problem of meeting the timing constraints specified by the edges in the PPG. Therefore some additions must be made to the above method. The current T-vector image should have a number associated with it called the timing distance from origin (TDO). This number would be zero for the first T-vector of the first nanoword filled. For the n-th T-vector the TDO could be computed as

$$TDO(n) = TDO(n-1) + S(n-1) \qquad (5.1)$$

where:

S is a function defined as S(n) = 2 if the n-th T-vector has its stretch bit set to one, and S(n)=1 otherwise.

In order to know the minimum timing distance from the origin at which a node can be scheduled, we must know the scheduled time from origin (STO) of each of its predecessors. A field in each node of the PPG should be reserved for holding this number. The STO field of a node that has been scheduled into the n-th T-vector can be computed as

$$STO = TDO(n)$$

for nodes representing leading-edge events, or as

$$STO = TDO(n) + S(n)$$

for nodes representing trailing-edge events. The function S(n) in this equation is the same one as in Equation 5.1.

The minimum time from origin (MTO) at which a node can be validly scheduled, depends on the STO of its predecessors and its timing distance from each one. If a node on the SNL has predecessors P(1), P(2), ..., P(m), and the edges from these nodes specify timing distances of T(1), T(2), ..., T(m), then the MTO of the node is computed as

$$MTO = Max \{ STO(P(1))+T(1), STO(P(2))+T(2), \ldots, STO(P(m))+T(m) \}$$

A node on the SNL can be placed in T-vector n if its MTO meets the condition

$$MTO \leq TDO(n)$$

for nodes representing leading-edge events, or the condition

$$MTO \leq TDO(n) + S(n) \qquad (5.2)$$

for nodes representing trailing-edge events.

Note, however, that the value of S(n) changes dynamically as the n-th T-vector is being filled. If a node on the SNL cannot be placed in the n-th T-vector because it does not meet condition 5.2, then, in many cases, which will be discussed later, the node can cause the stretch bit of the n-th T-vector to be set to 1 and thus change the value of S(n) from 1 to 2. S(n) can change value only if T-vector n is the current T-vector, but when it does change value, then all of the STO fields of trailing-edge nodes scheduled into the n-th T-vector must be updated.

5.2 Assigning Scheduling Priorities

The previous section suggests that the Schedulable Node List (SNL) should be searched sequentially for nodes that can be placed in the current T-vector image. However, scheduling a node from the beginning of the SNL may prevent the scheduling of a later node, whose early placement is more important. One way to handle this problem would be to maintain the node on the SNL sorted in decreasing order of some assigned sche g priority. A true scheduling priority would be a function of the current situation (which nodes had been scheduled and when), but a method similar to critical path analysis can be used to compute a constant value that will be a good approximation.

Critical path analysis (see for instance, Gear [9]) was developed to assist in the scheduling of industrial projects made up of a sequence of tasks some of which could be performed in parallel with others. The project is represented as an acyclic network (graph) whose nodes represent tasks and whose edges give the quired partial ordering of the tasks. Critical path analysis can be pplied to such a network to find the longest path through the network. The length of the longest path gives the minimum duration of the project and no task on this path can be delayed without delaying the completion of the project. This path is called the critical path and its nodes are called critical nodes. All other paths with a length equal to the longest path are also critical paths.

During the scheduling of a PPG, the critical node(s) among those that have not yet been scheduled, will be the one(s) with the greatest timing distance from the end node (TDE). If a node n in the PPG has descendants D(1), D(2), ..., D(m) and the edges to these descendants specify timing distances T(1), T(2), ..., T(m), then its TDE can be computed as

$$\mathrm{TDE}(n) = \mathrm{Max}\ \{\ \mathrm{TDE}(D(1))+T(1),\ \mathrm{TDE}(D(2))+T(2),\ \ldots,\ \mathrm{TDE}(D(m))+T(m)\ \} \qquad (5.3)$$

If the end node is given a TDE of zero then TDE's can be easily computed for all the nodes in the PPG by traversing it backwards from the end node to begin node.

The formula presented above would be acceptable if all T-steps lasted the same number of T-periods. But a T-step can be stretched to permit the scheduling of a node that would otherwise have to be postponed until the next T-step. When a T-step is stretched, the nodes scheduled into it will have a timing distance of two T-periods from their predecessors whether they need it or not. Thus edges that specify a timing distance of one T-period should in some circumstances be given an equal weight with those that specify two T-periods. The selection of which node is to be scheduled first is critical only among nodes that have large TDE's or have many nodes between them and the end node. Since a node in either of these two categories will always cause the stretching of a T-step (see next section) the calculation of scheduling priority of a node should always give equal weight to edges specifying one T-period with those that specify two. This can be done by computing the scheduling priority (SP) of node n as

$$SP(n) = \text{Max } ?\ SP(D(1)) + ?(T(1)+1) \div 2?,$$
$$SP(D(2)) + [(T(2)+1) \div 2],$$
$$\ldots, SP(D(m)) + [(T(m)+1) \div 2] \} \qquad (5.4)$$

where:

the operator "÷" represents integer division and D(1), D(2), ..., D(m) and T(1), T(2), ..., T(m) are defined as for Equation 5.3.

Sometimes it is possible that a node can be scheduled into the same T-step as one of its predecessors; hence nodes should be placed in the SNL immediately upon their unscheduled predecessor count going to zero. By the definition of scheduling priority above, a node cannot have a higher scheduling priority than any of its predecessors; therefore, when it is sorted into the SNL it can be placed in a position that is yet to be inspected.

Another small consideration that should be mentioned here is that some nodes can be scheduled into any of the three encoded fields in a T-vector that command F-register transfers. When it is decided that such a node is to be placed in a particular T-step the exact field into which it is placed should not be selected until after the placement of all other nodes that can go into only one of the fields and that dc not prevent the placement of the postponed node.

The concept of a scheduling priority has been invented here as an approximate solution to a complex problem that also arises in many other applications. It is usually called the problem of "task scheduling with limited resources;" whenever a number of tasks that compete for some limited resource, are all ready to be started, one must select a subset of them that can be performed together and that minimizes the cost of the project. Davis [12] has surveyed solutions to this problem that were developed for industrial-project scheduling. In general they use the trial-and-error technique of generating a number of promising schedules (not the complete set) and selecting the best one. The solutions developed handle fairly simple situations where usually only one resource is considered (manpower) and adapting them to the scheduling of a PPG would not be simple.

5.3 Stretching T-steps

As was mentioned in Chapter II there is a bit in each T-vector that can be turned on to make the T-vector remain active for two T-periods instead of one. This is done for two reasons: 1) some operations that can be requested (such as calculations with the ALUF) will not work at all unless the T-step in which they are placed is stretched, 2) often more operations can be scheduled into the same T-step if the T-step is stretched, since it will then provide more timing distance from previous T-steps.

The main reason for stretching a T-step in the second case, is to reduce the total number of T-vectors needed to hold a nanoprogram; if a T-step is stretched unnecessarily then the only effect will be to increase the running time of a program. Therefore a node should be permitted to cause the stretching of a T-step only if doing so reduces the total length of the nanoprogram. When Equation 5.4 is used to compute the scheduling priority of a node its value will be an estimate of the total number of T-vectors that will be needed to schedule all the descendants of a node. This means that the node at the head of the SNL is on the path that will require the most T-vectors to schedule; if its placement is delayed until the next T-vector, then the total length of the nanoprogram will be increased. The head node and any nodes with scheduling priorities equal to the head node (none can have a higher scheduling priority) should be allowed to cause the stretching of a T-step. The value of the scheduling priority of the head node is called the critical scheduling priority. Although none of the nodes on the SNL can have a scheduling priority greater than the critical scheduling priority some may have a total timing distance from the end node (TDE) greater than the TDE of the head node. These should also be allowed to cause the stretching of a T-step, since otherwise the paths on which they lie on will require more T-vectors for their scheduling than the critical path.

There are two more classes of nodes for which stretching is sometimes advantageous: 1) nodes whose TDE is greater than the critical scheduling priority but less than the TDE of the head node on the SNL, 2) nodes that have a scheduling priority less than but "close" to the critical scheduling priority. Liberal rules on allowing nodes from these two classes to cause the stretching of a T-step will result in a nanoprogram that requires fewer nanowords, whereas strict rules will result in one with a shorter running time. The programmer should therefore be able to pass a parameter to the scheduler that will specify whether he is more concerned with optimizing running time or the number of nanowords produced. The scheduler can use the value of this parameter to change its definition of "close" and to decide which nodes in the two classes can cause the stretching of a T-step.

Occasionally, for timing reasons, it will not be possible to schedule the head node on the SNL into the current T-vector even if the stretch bit is set. In this case, its scheduling priority should still be used as the critical scheduling priority but the T-step should be stretched only if: 1) some other node on the SNL causes it to be stretched or 2) the difference between the minimum time from origin (MTO) of the head node and the timing distance from origin (TDO) of the current T-vector image, is an even number. If the second condition is false (i.e., MTO minus TDO is odd) then there is room for one unstretched T-step before the node is scheduled.

CHAPTER VI

REMAINING WORK AND CONCLUSIONS

There are many aspects of writing a Lizard translator that have not been discussed in the preceeding chapters. Most of these are only details about storing the various data structures needed and about implementing algorithms to perform some of the operations described. Some of these details could be easily filled in by an imaginative programmer, whereas others require more thought and analysis. The main reason that they have not been discussed is that a sufficiently accurate description would make long and boring reading; the only useful way to supply them would be as a working translator. On the other hand, some important characteristics of the QM-1 that complicate a translator, have not been mentioned because the author could not devise satisfactory methods for dealing with them. In all cases, a simple procedure exists for circumventing the obstacles that they impose, but good solutions will have to be found if a translator is to produce acceptable results. This chapter will discuss these major considerations and why they are important.

6.1 K-field Allocation

The Lizard language does not require a programmer to specify which K-fields will hold the six bit constants used by his program (e.g. values to be moved into F-registers for bus control). This was done as a convenience for the programmer and because the K-field that should be used at a particular point in a program, can be effectively chosen only at scheduling time. Thus the translator should pick the K-fields that will be used by a program. There are very few firm restrictions on how K-fields can be chosen for this purpose but the method used will greatly affect the efficiency of the resulting nanocode. If, for instance, the same K-field is used to set up the left and right operands of an ALU operation, then the ALU operation would have to be scheduled into two separate nanowords, which would make the program longer and slower than it needs to be. Furthermore, a constant can be transferred to an F-register from other places than just K-fields. There may be another F-register that contains the desired constant and an F-register to F-register transfer could be performed instead of a K-field to F-register transfer. Or possibly the F-register incrementers and decrementers (INCF1 & INCF2) could be used to generate a value of a constant from a close value. Allocating K-fields or choosing a value transfer method, can be done efficiently only during the scheduling of the PPG, so that the K-fields used by an operation can be selected to produce the least conflict with the requirements of other operations. Even at scheduling time, K-field allocation would be difficult because a node being scheduled would have to yield a K-field to any higher priority node that could be placed into the same nanoword, not just those that could be placed into the same T-vector.

Unfortunately, the parallelism-analysis algorithm that has been presented, requires that all sources and sinks of an operation must be defined before the algorithm is applied. Therefore, it would hav [illegible] be modified [illegible] recognize parallelism in a program in which some [illegible]ations were specified as a choice of alternatives.

6.2 Restrictions Imposed by Component-Associated Registers

The architecture of the QM-1 has a characteristic that greatly restricts parallelism analysis. In order to use a computational or memory unit a program must use the F-registers and other memory elements associated with it (such as carry holds and output-data registers). Because of this the parallelism-analysis algorithm that has been presented will always find some interdependency of memory-element usage between two program segments that use the same QM-1 component. Thus, for instance, if a Lizard program performs two ALU addition operations, they will always be scheduled in the same order that they appeared in the source program. This leaves the programmer with the burden of predicting which order of the operations will produce the best nanoprogram and his choice could change its efficiency significantly. A translator would have dificulty discovering this type of interchangeability after a Lizard statement had been broken down into the elementary instructions that comprise it; therefore, parallelism analysis would probably have to be performed on entire Lizard statements and a way found to signify that two sequences of instructions in the PPG may be interchanged but not intermingled.

6.3 Optimization of Branches

When a PPG is being scheduled into nanowords and a label node is encountered, the current nanoword image must be retired, no matter how empty it is, and a new one begun. This results from the fact that nanobranches are made by loading a new nanoword into the Nano-Instruction Matrix and hence labels can appear only at the top of a nanoword. The method of handling branches and labels outlined in Chapter IV rigidly preserves their position relative to the other statements. However, there are many cases in a program where ordinary statements can change places with control statements without changing the computation. In an algorithmic language, some of these changes would be counter-intuitive in that they would seem to slcw down the program - for instance, moving a simple computation statement into the range of a loop - but, since a T-vector takes the same amount of time to execute regardless of how many operations it performs, moving operations around in a nanoprogram could prevent the generation of a nearly empty nanoword and thus shorten and speed up a program. Because of the multitude of trivial operations that infest an average nanoprogram, there is more opportunity for this kind of change than would exist in an ordinary program, but the detection of operations that can change places with a control statement is difficult to automate.

6.4 Converting Branches to Skips

Two methods exist for performing branch operations in a nanoprogram: the nanobranch and the skip operation. Skip operations can be used only to skip a group of elementary instructions that can be coded into a single T-vector, but when they can be used, they greatly reduce the number of wasted nanowords. If branch statements are handled as has been described in Chapter IV then very few of them could be changed into skip operations. For instance, if a Lizard program specified a branch around an ALU statement, then the branch could be converted to a skip operation only if the K-field to F-register transfers implied by the operands of the ALU statement, were moved so that they preceeded the branch statement. This is a result of the fact that the elementary instructions needed to set up the operands for an ALU operation can not be placed into the same T-vector as the elementary instruction that transfers the result into the result register. Even if a programmer recognized these situations and split his ALU statement into two parts, one that set up operands and another that transferred the result, a translator would still have difficulty recognizing that the branch around the seco[illegible] ALU statement could be performed as a skip operation. To recognize this possibility, the scheduling process would have to be interrupted whenever a branch node became available for scheduling, and a look-ahead operation performed, to predict whether the nodes branched around could be scheduled into one T-vector. A look-ahead operation of this type would be a minor scheduler itself, since it would also have to observe the timing constraints and take note of the current contents of the K-vector image. If the look-ahead operation concluded that a skip could not be used then there is still a possibility that delaying the scheduling of the branch node for one or more T-vectors would [illegible]it its transformation into a skip. A scheme like this would greatly complicate the scheduling process but would be worth the effort if it worked well.

6.5 Blinding the Programmer

When a programmer is preparing a nanoprogram, he often has a choice of computational units that he can use for an operation. For instance, he often can use the INDEX ALU rather than the ALU, for doing a computation, or he could do a local-store to local-store transfer using the shifter, the ALU or the INDEX ALU. He usually makes the final decision about which unit to use for an operation very late in the programming process because only then can he judge its effect on the compactness of his program. There are many other more subtle choices that a programmer can make that will have significant effects on the efficiency of his program. Unfortunately the Lizard language completely blinds the programmer to the effects that his choices will have on the resulting nanocode. He may be forced to do trial-and-error programming in an [illegible] to improve the efficiency of his programs. This [illegible] be an undesirable but unavoidable result of higher-level nanocoding, since it would be very difficult to devise a scheme [illegible]r doing parallelism analysis on a program that represented a computation in all its possible variations so that the final selection could be made at scheduling time. It may however, be possible to devise a heuristic that could make a reasonable prediction, early in the [illegible]anslation process, of the best meth[illegible] to use.

6.6 Conclusions

The preceding chapters seem to indicate that writing a translator for the Lizard language would be an extremely difficult, costly and time-consuming project. Many improvements would have to be made to the translation methods presented if the nanocode produced was to approach the efficiency of a hand-written nanop[illegible] [illegible]am. Once it was written, the translator would consume large amounts of execution time, so that a programmer could not use it freely unless he had time on his hands. Although some of the complications are intrinsic to horizontal microcode, most of them are the result of characteristics of the QM-1 architecture that impede automatic translation. Redesigning the QM-1 to make it a more suitable target for automatic translators might take less effort than writing an efficient translator for its present architecture.

BIBLIOGRAPHY

1. Nanodata Corporation, QM-1 Hardware Level User's Manual. Williamsville, N.Y., August 1974.

2. Robert F. Rosin, Gideon Frieden and Richard H. Eckhouse, "An Environment for Research in Microprogramming and Emulation." CACM, Vol. 15, No. 8, pp 748-760, August 1972.

3. Samir S. Husson, Microprogramming: Principles and Practice. Englewood Cliffs, N.J.: Prentice-Hall, 1970.

4. A.K. Agarawala, and T.G. Rausher, Foundations of Microprogramming: Architecture, Software and Applications. New York, N.Y.: Academic Press, 1976.

5. A.J. Bernstein, "Analysis of a Program for Parallel Processing." IEEE Transactions on Electronic Computers, Vol. EC-15, No. 5, pp 757-763, October 1966.

6. J.L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing." ACM Computing Surveys, Vol. 15, No. 1, pp 31-80, March 1973.

7. D.E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Reading, Mass.: Addison-Wesley, 1969.

8. Walter H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems." IEEE Transactions on Computers, Vol. C-24, No. 12, pp 1235-1238, December 1975.

9. C.W. Gear, Introduction to Computer Science. Chicago: Science Research Associates, Inc., 1973.

10. M.V. Wilkes, "The Best Way to Design an Automatic Calculating Machine." Manchester University Computer Inaugural Conference, pp 16-18, July 1951.

11. C.V. Ramamoorthy and M.J. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs." In Proceedings AFIPS 1969 Fall Joint Computer Conference, pp 1-15, Montvale, N.J.: AFIPS Press, 1969.

E.W. Davis, "Resource Allocation in Project Network Models - A Survey." Journal of Industrial Engineering. Vol. XVII, No. 4, pp 177-188, April 1966.

13. Jerome D. Wiest and Ferdinand K. Levy, A Managment Guide to Pert/CPM. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1969.

14. J.D. Ullman "Polynomial Complete Scheduling Problems." ACM - Operating Systems Review, Vol. 7, pp 96-101, October 1973.

15. F.E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure." CACM, Vol. 19, No. 3, pp 137-147, March 1976.

16. Jean-Loup Baer, "Matrice de Connexion Minimale d'une Matrice de Precedence Donnee." RIRO, Vol. 3, No. 16, pp 65-73, 1969.

17. J.M.S. Simoes Pereira, "On the Boolean Matrix Equation M'= ... " JACM, Vol. 12, No. 3, pp 376-382, July 1965.

18. Harry T. Hsu, "An Algorithm for Finding a Minimal Equivalent Graph of a Digraph." JACM, Vol. 22, No. 1, pp 11-16, January 1975.

19. L. Wayne Jackson and Subrata Dasgupta, "The Identification of Parallel Micro-operations." Information Processing Letters, Vol. 2, No. 6, pp 180-184, Amsterdam: North-Holland Publishing Company, April 1974.

20. C.V. Ramamoorthy and Masahiro Tsuchiya, "A High-Level Language for Horizontal Microprogramming." IEEE Transactions on Computers, Vol. C-23, No. 8, pp 791-801, August 1974.

21. C.V. Ramamoorthy and M.J. Gonzalez, "Recognition and Representation of Parallel Processable Streams in Computer Programs - II (Task/Process Parallelism)." In Proceedings ACM 24th. National Conference, pp 387-397, New York: ACM, 1969.

22. A.V. Aho, M.R. Garey and T.D. Ullman "The Transitive Reduction of a Directed Graph." SIAM Journal of Computing, Vol. 1, No. 2, pp 131-137, June 1972.

APPENDIX A

LIZARD LANGUAGE REFERENCE MANUAL

```
OOOO       OOCO COCCOOOOO    OOOOO    CCCCOOOOC   CCCCOOOO
 OO         OO  CCCCOOOO    CCCOOOO    CCOOOOOOC   CCOOOOOO
 OO         OO  O    OO    OO     OO   OO    CC   OO     OO
 OO         OO      OO     OO     OC   CCOOCOOOO  OO     OO
 OO         OO     OO      OCCCCOOOOO  COCOOOOC   OO     OO
 OO      O  CO    CO    O  OCCCCCOOOO  OO  OO     OO     OO
 OCCCCCCCOO OO   OCCCOOOO  OO     OO   OO   CC    CCOOOOOO
OOOCOCOCOO CCCC CCCCCOOCO OOOO   OOOO COOO    CCO COOOOOOO
```

L A N G U A G E   R   F E R E N C E   M A N U A L

September, 1976

Department of Computing Science
University of Alberta

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

SECTION I.

INTRODUCTION

Lizard is not a high-level language! The Lizard language was designed to make nanoprogramming for the QM-1 a less demanding endeavour; however, it still resembles an assembler language more than it resembles ALGOL or FORTRAN. Its main differences from conventional Nanocode are few but significant.

1) The programmer need have no knowledge of the speeds of the circuitry, T, F and R clocks and leading and trailing edges. The Lizard Scheduler handles all timing constraints.

2) The programmer describes his algorithm sequentially; the Lizard Scheduler is responsible for determining which operations can be performed in parallel.

3) The programmer need not know the numeric values of the operations he specifies; the Lizard Scheduler will share coded values for multiple purposes whenever possible. This makes a program easier to read and modify.

4) Associated steps are grouped together into one statement for a more transparent representation of an algorithm.

5) The programmer prepares a continuous stream of statements; he need not worry about nanoword boundaries.

6) The format and function of Lizard statements are more intuitive than those of nano-assembler.

This manual is written for someone with a good background in computer science and a good understanding of the QM-1. Without these presumptions the Lizard Language Reference Manual would be an enormous tome.

Some new terms for QM-1 components are used in this manual. They are: MOD Register, COD Register, NOD Register and Micro Operand Buffer (MOB). If any doubts about their meaning exist, consult Chapter II of the thesis "A Sequential Language for Nanoprogramming the QM-1," by Daniel Salomon. Throughout this manual, the "QM-1 Hardware Level User's Manual" will be called the "QM-1 Manual."

SECTION II.

METALANGUAGE DESCRIPTION

The metalanguage used in this manual is very simple and is used only where it shortens or simplifies a description. It uses the following symbols:

A-Z	Upper Case - must appear as shown
a-z	Lower Case - replace appropriately
<>	Angle Brackets - enclose a term Each term is defined either in the section "General Terms" or soon after its first use.
|	(OR) - separates alternative forms
{}	Braces - encloses alternative forms
[ ]	Brackets - enclose optional forms
_	Underscore - indicates default form
	All other symbols - must appear as shown

SECTION III.

LIZARD STATEMENT FORMAT

A Lizard statement is made up of three parts: an optional <Label>, a header, and a list of keyword parameters. The <Label> is separated from the header by a colon (:) and the parameters are separated from the header and from each other by comma's (,).

If a hash mark (#) is the last non-blank character on a line then the statement on that line is continued on the next line. The hash mark is treated as a blank. If one of the commas that occur normally in a statement, appears at the end of a line, then this will also indicate that the statement continues on the next line. Blank lines in a source program are ignored. More than one statement can be coded on the same line if they are separated by semicolons (;).

A comment is delimited by cent signs (¢) and is treated as a blank. A comment may not be continued across a line boundary; the end of the line will terminate the comment.

Blanks can be used liberally. A string of blanks is always treated as a single blank except in a preprocessor character string constant. Blanks may not appear in keywords nor between the characters of multicharacter preprocessor operators.

SECTION IV.

GENERAL TERMS

<Source> - is one of:
- A, B, C
- SWITCHES
- KA, KB, KT, KX
- DEVICE (IO ID of interrupting device)

<Dest> - is one of:
- A, B, C
- KA, KB, KS, KT, KX, KALC, KSHA, KSHC

<Constant> - is an integer i in the range $0 \le i \le 63$ (i.e. a 6 bit constant). A number is assumed to be in decimal notation unless it begins with a zero, in which case, it is interpreted as an octal number.

<F-register> - is one of:

Name	Number
FMIX	0
FMOD	1
FCIA	2
FAIL	3
FCID	4
FAIR	5
FCOD	6
FAOD	7
FSID	8
FSCD	9
FEID	10
FEOD	11
FEIA	12
FEOA	13
FACT	14
FUSR	15
FMPC	16
FIDX	17
FIST	18
FIPH	19
G(0) to G(11)	20 to 31

Each F-register was deliberately given only one name in order to avoid confusion in a program.

<G(GSPEC)> - is one of:
    G(0) to G(11), B, KS, KSHA, KX

<Six-Bit Value> - is one of:
    a <Constant>, a <Source>, a <G(GSPEC)>
    or an <F-register>

<Register Selector> - is a <Six-Bit Value>

<Label> - is a symbolic name acceptable to the nano-
    assembler as defined in section 6.2.3 of the QM-1
    Manual:

    "6.2.3 SYMBOLIC NAMES

    Symbolic names are strings of letters, digits
    periods, and single occurrences of the blank
    character... [However Lizard treats strings of blanks
    as a single blank.] A symbolic name may begin with a
    letter or period. [Leading and trailing blanks are
    ignored.] Only the first 10 characters are used, and
    if the 10-th character of a name is a blank it is also
    ignored."

SECTION V.

ACTIVE STATEMENTS

1. Value Transfer

Header: {COPY x TO y | x -> y}

There are two types of Value Transfer:

i) Six Bit Transfer -
x is a <Register Selector> and
y is a <Dest> or an <F-register>

ii) Eighteen Bit Transfer -
x and y take the form: LSR(<Register Selector>)

2. Six-Bit Swap

Purpose: The two operands have their contents exchanged.

Header: SWAP <Source-dest> , <F-register>

<Source-dest> - is one of:
A, B, C, KA, KB, KX, KS, KSHA

3. F-register Increment & Decrement

Purpose: Increment or Decrement the contents of an F-register b[illegible]e.

Header: [INCR [illegible]CR] <F-register>

4. <u>Control-Store Read</u>

Purpose: A word is read from Contro[illegible] Sto[illegible] into the COD
Register. If the GATE parame[illegible]er is [illegible]cified then the
word is also transferred to [illegible].

Header: READ CS

Parameters:

1) ADDR = <CS Address Selector>
Required Parameter.

2) DEST = LSR(<Register Selector>)
Loads FCOD with the <Register Selector> value.

3) [<u>GATE</u> | NO GATE]
This parameter specifies whether the COD Register is to be gated into the destination register selected.

<CS Address Selector> - is used to set the "CS ADDR SELECT" T-field and can be one of the following:
LS (<Register Selector>)
CIA
COD
MPC
MPC + [1 | 2 | B | AB]
INDEX(<Label>)

If the first form is chosen the "CS ADDR SELECT" T-field is set to CIA and FCIA is loaded with the <Register Selector> value.
If the last form is chosen then the <Label> is the label of the INDEX ALU statement whose result is to be used as a Local Store address.

5. <u>Control-Store Gate</u>

Purpose: The contents of the COD Register are copied into Local Store.

Header: GATE CS

Parameters

1) DEST = LSR(<Register Selector>)
Loads FCOD with the <Register Selector> value.

6. <u>Control-Store Write</u>

rpose: A word is written from Local Store into Control
Store.

Header: WRITE CS

Parameters:

1) ADDR = <CS Address Selector>
Required Parameter.

2) SOURCE = LSR(<Register Selector>)
Loads FCID with the <Register Selector> value.

7. <u>External-Store Read</u>

Purpose: A word is transferred from External Store to Local
Store.

Header: READ ES

Parameters:

1) SOURCE = ESR(<Register Selector>)
Loads FEOA with the <Register Selector> value.

2) DEST = LSR(<Register Selector>)
Loads FEOD with the <Register Selector> value.

8. <u>External-Store Write</u>

Purpose: A word is transferred from Local Store to External
Store.

Header: WRITE ES

Parameters:

1) SOURCE = LSR(<Register Selector>)
Loads FEID with the <Register Selector> value.

2) DEST = ESR (<Register Selector>)
Loads FEIA with the <Register Selector> value.

9. <u>External-Store - Local-Store Swap</u>

Purpose: The contents of a local-store register are swapped with the contents of an external-store register.

Header: SWAP ESR(<Register Selector>), LSR(<Register Selector>)

10. <u>Main-Store Read</u>

Purpose: A word is read from Main Store into the MOD Register. If the GATE parameter is specified then the word is also transferred to the register specified by the DEST parameter.

Header: READ MS

Parameters:

1) ADDR = <MS Register>
Loads FMIX.

2) DEST = <MS Register>
Loads FMOD.

3) DIRECT
Sets value of "DIRECT MS ACCESS" K-field.

4) [<u>GATE</u> | NO GATE]
This parameter specifies whether the MOD Register is to be gated into the destination register selected.

5) RMI = <RMI Group>
Sets value of "RMI SELECT" T-field. When this parameter is omitted, BYPASS is assumed. This parameter is used only if the GATE parameter is also specified.

<MS Register> - is one of:
LSR(<Five-Bit Register Selector>)
ESR(j)   where $0 \leq j \leq 7$
R(<Register Selector>)
ONES   (valid only after parameters ADDR or SOURCE)
NULL   (valid only after parameter DEST)

<Five-Bit Register Selector> - is one of:
an integer i in the range $0 \le i \le 31$
a <Source>, a <G(GSPEC)> or an <F-register> whose contents are in the range of i

<RMI Group> - is one of: BYPASS, A, B, C

11. Main-Store Write

Purpose: A word is transferred from the register specified by the SOURCE parameter to Main Store. The MOD Register is cleared to zero but will eventually be loaded with the old contents of the Main Store location being overwritten. (See the Await Stable MOD Register statement below.)

Header: WRITE MS

Parameters:

1) ADDR = <MS Register>
Loads FMIX for fetch cycle.

2) SOURCE = <MS Register>
Loads FMIX for restore cycle.

3) DIRECT

12. Main-Store Gate

Purpose: The contents of the MOD Register are copied into Local Store.

Header: GATE MS

Parameters:

1) DEST = <MS Register>
Loads FMOD.

2) RMI = <RMI Group>

13. Main-Store Split Read

Purpose: The first half of a read - modify - write Main Store operation is initiated. A word is read from Main Store into the MOD Register and the location read is set to zero. If the GATE parameter is specified then the word is also transferred to the register specified by the DEST parameter. The address of the location read is saved so that the next Main-Store Split Write operation can reset its value. No Main Store transfers may be performed between the two halves of a read - modify - write operation. If the second half is never performed, then the main-store location read, will remain zero.

Header: FETCH MS

Parameters:

1) ADDR = <MS Register>
Loads FMIX.

2) DEST = <MS Register>
Loads FMOD.

3) DIRECT

4) [<u>GATE</u> | NO GATE]
This parameter specifies whether the MOD Register is to be gated into the destination register selected.

5) RMI = <RMI Group>

14. Main-Store Split Write

Purpose: The second half of a read - modify - write Main Store operation is initiated. A word is transferred from the register specified by the SOURCE parameter to the Main Store location read by the latest Main Store Split Read operation.

Header: RESTORE MS

Parameters:

1) SOURCE = <MS Register>
Loads PMIX.

2) DIRECT

Note: To function properly a Main Store Split Write must be preceded by a Main Store Split Read and no other Main Store transfers may come between them.

15. Await Stable MOD Register

Purpose: This statement is used after a Main Store Write to wait until the old contents of the Main Store location just overwritten have become stable in the MOD Register.

Header: AWAIT MS DATA

16. ALU Calculation

Purpose: Any or all of the inputs for an ALU calculation are set up. If the GATE parameter is specified the result is transferred to Local Store.

Header: ALU

Parameters:

1) OP = <ALU Operation>
The <ALU Operation> specified is loaded into the "KALC" K-field.

2) LEFT = LSR(<Register Selector>)
Loads FAIL with the <Register Selector> value.

3) RIGHT = LSR(<Register Selector>)
Loads FAIR with the <Register Selector> value.

4) RESULT = LSR(<Register Selector>)
Loads FAOD with the <Register Selector> value. This parameter specifies the Local Store Register that will receive the result.

5) DCW = LSR(<register Selector>)
Used only with DECIMAL operations. This parameter is used to specify which LSR will recieve the decimal correction word. It loads FSOD with the <Register Selector> value.

6) CIH = <Carry Specifier>
Loads the value of the Carry In Hold register before the ALU GATE operation.

7) CARRY CTL = <Carry Control Code>
Specifies the value of the "CARRY CTL" T-field during the ALU GATE Operation.

8) STATUS ENABLE
Sets the "ALU STATUS ENABLE" K-field to one.

9) [GATE | NO GATE]
Specifies whether the result is to be gated into the RESULT register.

Note: If the OP and CIH parameters conflict, an error message will be produced.

<ALU Operation> - is one of:
    ADD, SUBTRACT, DOUBLE, INCR LEFT, DECR LEFT,
    PASS LEFT, PASS RIGHT, AND, NAND, OR, NOR, XOR,
    EQUIVALENCE, L IMPLIES R, R IMPLIES L, NOT LEFT,
    NOT RIGHT, ZERO, ONES
    plus all coded operations shown in table in the QM-1
    Manual section 5.6.2.
    Any of the above may be preceded by the word DECIMAL
    A <Six-Bit Value> can also be used as an <ALU
    Operation>

<Carry Specifier> - is one of: 0, 1 or * (residual)
    (possibly CLEAR and SET rather than 0 and 1)

<Carry Control Code> - is one of:

Name	Carry CTL
NONE	0
CLEAR CIH	1
SET CIH	2
ALU TO BOTH	3
ALU TO COH	4
SET COH	5
CLEAR COH	6
SH TO COH	7

17. Shifter Operation

Purpose: Any or all of the inputs for a Shifter operation are set up and if the GATE parameter is specified the result is transferred to Local Store.

Header: SHIFTER

Parameters:

1) OP = <Shifter Operation>
Loads the "KSHC" K-field with the <Shifter Operation> value.

2) OP TYPE = <KSHC Type>
This parameter is used only when the OP parameter is omitted. It is used to specify whether this is a double or single length shift.

3) AMCUNT = <Shifter Amount>
Loads "KSHA" K-field with the <Shifter Amcunt value.

4) INPUT = LSR(<Register Selector>)
Loads FSID with the <Register Selector> value.

5) RESULT = LSR(<Register Selector>)
Loads FSOD with the <Register Selector> value. This parameter specifies the Local Store Register that will receive the result.

6) COH = <Carry Specifier>
This parameter is required only when the LEFT CONTROL switch in KSHC is set. It loads the value of COH before the Shift Operation.

7) CARRY CTL = <Carry Control Code>
Specifies the value of the "CARRY CTL" T-field during the SHIFTER GATE operation.

8) STATUS ENABLE
Sets the "SH STATUS ENABLE" K-field to one.

9) [<u>GATE</u> | GATE BOTH | NO GATE]
Specifies whether the result is to be gated into the RESULT register.

Note: The simplest way to set up a double length shift operation, is to have an ALU statement with the NO GATE parameter, set up the ALU inputs and outputs, then have a SHIFTER statement with a GATE BOTH parameter, set up the shifter inputs and gate the two result words into Local Store.

<Shifter Operation> - is made up of any combination of the following five items (default is underlined):
1) [LEFT | RIGHT]
2) [SINGLE | DOUBLE]
3) [CIRCULAR | LOGICAL | ARITHMETIC]
4) RIGHT CTL
5) LEFT CTL

A <Register Specifier> can also be used as a <Shifter Operation>. Its value will be converted to a <Shifter Operation> according to the KSHC LAYOUT given in section 5.6.3 of the QM-1 manual.

<Shifter Amount> - is a <Six-Bit Value>

<KSHC Type> - is one of: DOUBLE or SINGLE

18. Carry Control

Header: {SET | CLEAR} {CIH | COH}

19. ALUF Operation

Header: ALUF

Parameters:

1) OP = <ALUF Operation>
The <ALUF Operation> value is loaded into the <ALUF Operation Source> given by the OP VIA parameter.

2) OP VIA = <ALUF Operation Source>
Sets the "AUX3" T-field value. This parameter selects the K-field or F-register that will hold the operation code.

3) LEFT = <ALUF Left-Input Selector>
Required parameter if the operation selected uses a left input. Sets the "FSEL1" T-field value.

4) RIGHT = <ALUF Right-Input Selector>
Required parameter if the operation selected uses a right input. Sets the "FSEL2" T-field value.

5) RESULT = <ALUF Result Selector>
Required parameter. Sets the "FSEL0" T-field value.

```
<ALUF Operation> - is one of:
     a) the functions given in Table 5.6.7B in the QM-1
     Manual
     b) a <Six-Bit Value>


<ALUF Operation Source> - is one of:
     A, B, KT, KB, G(8), G(9), G(10) or G(11)


<ALUF Left-Input Selector> - is one of:
     <F-register>
     A, B, C
     KA, KT
     <G(GSPEC)>


<ALUF Right-Input Selector> - is one of:
     <F-register>
     A, B
     KX, KA, KB
     <G(GSPEC)>


<ALUF Result Selector> - is an <F-register>
```

20. Index ALU Operation

Header: INDEX ALU

Parameters:

1) OP = {<Direct Index Function> | <Indirect Index Function>}
Sets "FSEL2" T-field with the <Direct Index Function> value or loads the <Index Operation Source> specified in the OP VIA parameter with the <Indirect Index Function> value.

2) OP VIA = <Index Operation Source>
Sets the "FSEL2" T-field.

3) LEFT = LSR(<Register Selector>)
Loads the <Index Left Selector> chosen by the LEFT VIA parameter with the <Register Selector> value.

4) LEFT VIA = <Index Left Selector>
Sets the "AUX2" T-field.

5) RIGHT = <Index Right Input>
Loads the <Index Right Selector) chosen by the RIGHT VIA parameter with the <Index Right Input> code value.

6) RIGHT VIA = <Index Right Select>
Sets the "AUX3" T-field.

7) RESULT = LSR(<Register Selector>)
Loads the <Index Result Selector> chosen by the RESULT VIA parameter with the <Register Selector> value.

8) RESULT VIA = <Index Result Selector>
Sets the "GSPEC" T-field.

Note: There are only three valid combinations of the OP and OP VIA parameters. They are:

i) OP = <Direct Index Function>

ii) OP = <Indirect Index Function>
OP VIA = < Index Operation Source>

iii) OP VIA = <Index Operation Source>

<Direct Index Function> - is one of:
L-1, L+1, L XOR R, ALL ONES, ZERO, NOT R, L-R, L AND R, L OR R, L+R, R, NOT L, L,

<I[illegible]direct Index Function> - is the same as an <ALU Operation> excep[illegible] that DECIMAL may not precede any of the operations listed. A <Register Selector> can also be used as an <Indirect Index Function>.

<Index Operation Source> - is one of:
A, B, KA, KB
FMPC, FIDX
G(0) to G(11)

<Index Left Selector> - is one of:
A, B, KX, KA, KB or
RESULT (Left Input Register = Result Register)

<Index Right Input> - is one of:

Source	Code	
ESR(8)	**0000	* = Don't Care
ESR(9)	**0001	
. . . .		
ESR	**1011	
AL[illegible]	**1100	
(a[illegible] one	**1101	
MO[illegible]	**1110	
CO[illegible]	**1111	

A <Re[illegible]er Selector> can also be used as an <Index Right Input>. Its value will be looked up in the Code table and a Source determined in that way.

<Index Right Selector> - is one of:
A, B, KT, KB, G(8), G(9), G(10) or G(11)

<Index Result Selector> - is one of:
G(0) to G(11), KSHA, B, KS or KX

21. Halt

Header: HALT

Parameters:

1) TYPE = {TXX | LOOP}
The first option sets the "TXX" T-field; program will halt if the Program Stop console switch is on. The second option sets up a hard loop halt.

22. Force Scheduling

Purpose: All statements preceeding this one are scheduled before any that follow it. This is useful only if one intends to single step through a program during debugging.

Header: TEST-POINT

23. Increment Microprogram Counter

Header: INC MPC

Parameters:

1) AMCUNT = <MPC Increment>
Sets lower two bits of "GSPEC" T-field.

<MPC Increment> - is one of: 1, 2, B or AB

24. Branch

Header: GO TO (<Label>)

Parameters:

1) COND = <Condition>
Loads one of "KS", "KT" or "KX" K-fields, depending on COND TYPE parameter, with the <Condition> value.

2) COND TYPE = <Condition Type>
Required parameter, unless branch is unconditional. Sets the "TEST SPECIFIER" T-field.

3) F = <F-register>
Sets the "FSEL1" T-field to the <F-register> value. This parameter chooses the <F-register> that is to be tested for zero. Therefore it is required only if F NOT ZERO is part of the condition to be tested.

4) INDEX = <Label>
This parameter chooses the INDEX ALU statement whose result is to be tested for zero. Therefore it is only required if IRNZ is part of the condition to be tested.

5) HCLD
This parameter specifies that if the branch is taken the values of the "KALC", "KSHC", "KSHA" AND "KS" K-fields will be retained for use at the location branched to.

6) HCLD2
This parameter specifies that if the branch is taken the values of the "KA" and "KB" K-fields will be retained for use at the location branched to.

Note: The COND parameter gives the conditions under which the branch is to be taken. When NEGATE is specified in the COND TYPE parameter confusion can arise as to the meaning of the COND parameter. The easiest way to remember the expected result is that if the branch would have been taken without NEGATE being specified then it will not be taken when NEGATE is specified and vice-versa. The NEGATE keyword does not appear in the COND parameter because it is not used to load the value of KS, KT, or KX. It is used to set the "TEST SPECIFIER" T-field.

<Condition> - takes the form:
{<Global Local Condition> | <Special Condition>}

<Global Local Condition> - is any number of the following tests separated by the word OR:

SLB	(Shifter Low Bit)
OVERFLOW	
RNZ	(Result Not Zero)
SIGN	
CARRY	
SHB	(Shifter High Bit)

<Special Condition> - is any number of the following tests separated by the word OR:

F NOT ZERO	
MS DNR	(MS Data Not Ready)
MS BUSY	
PROGRAM CHECK	
IRNZ	(Index Result Not Zero)

<Condition Type> - takes the form:
[NEGATE] {SPECIAL | GLOBAL | LOCAL}

25. <u>Nanostore Write</u>

Purpose: Write one 18 bit byte from External Store into the Nanostore location selected by LSR(31). The format of the Nanostore address is given in section 5.4.1.2 of the QM-1 Manual. If the byte address or Nanoword address is invalid Nanostore will not be changed.

Header: WRITE NS

Parameters:

1) SOURCE = ESR(<Register Selector>)
Loads FEOA with the <Register Selector> value. This parameter selects the External Store register whose contents will be written into Nanostore.

26. Computed Branch

Purpose: A branch is made to the Nanostore location selected by LSR(31). The format of the Nanostore address is given in section 5.4.1.2.cf the QM-1 Manual. This branch is implemented by a "WRITE NS" operation with an invalid byte address. To ensure that Nanostore is not modified the "B" field in LSR(31) will be set to 77 octal by this statement, unless specified otherwise.

Header: COMPUTED BRANCH

Parameters:

1) LEAVE B
This parameter requests that the "B" field of LSR(31) not be set to 77 (octal) before the "WRITE NS" operation. It should be specified only if the programmer is certain that the "B" field already contains an invalid byte address; otherwise, Nanostore will be changed in some unpredictable way.

2) HOLD
This parameter specifies that the values of the "KALC", "KSHC", "KSHA" and "KS" K-fields will be retained for use at the location branched to.

3) HOLD2
This parameter specifies that the values of the "KA" and "KB" K-fields will be retained for use at the location branched to.

27. <u>Auxiliary Action</u>

Header: AUX ACT

Parameters:

1) FACT = <Aux Action>
Loads F-register FACT with <Aux Action> value.

2) RMI = <RMI Group>
Sets the "RMI Select" T-field. This parameter is needed only if parameter FACT is omitted. The option BYPASS in <RMI Group> doesn't make much sense for an AUX ACT statement but is allowed for uniformity with the Main Store operations.

<Aux Action> - is one of the following or its code number:

Name	Code
NO OP	00
DISABLE INTS	77
ENABLE INTS	76
SET RELATIVE MS	75
SET DIRECT MS	74
LOAD RMI(i) ROTATE	57
LOAD RMI(i) MASK	56
LOAD RMI(i) INDEX	55

where i is an <RMI Group>

28. <u>Allow Interrupts</u>

Purpose: Set the allow interrupt K-fields.

Header: ALLOW {NO | ONLY NANO | ONLY MICRO | ALL} INTS

29. <u>Generate or Clear Interrupt</u>

Header: {GENERATE | CLEAR} INT

Parameters:

1) LEVEL = j where 2 <= j <= 31
Required Parameter. This parameter sets the "GSPEC" T-field.

30. Auto Or Operation

Header: AUTO OR (<Label List>)

The <Label List> contains the labels of statements of one of the following types:

Eighteen Bit Value Transfer
Control Store Read
External Store Read
Main Store Read
Main Store Gate
Main Store Split Read
ALU Calculation
Shifter Operation
Index ALU Operation
Increment Microprogram Counter
Load LSR(31)

All the statements in the list must use the same LSR as a result register. The logical OR of their results will be placed in the result register only one Main Store operation may appear in the same <Label List>.

<Label List> - is made up of any number of <Label>'s separated by commas.

31. Output Statement

Purpose: Send an XIO pulse to the selected port.

Header: XIO

Parameters:

1) PORT = <Port Selector>
Required Parameter. Loads low order 3 bits of the "KA" K-field. If the port selector is not a constant then all of "KA" is loaded.

2) DEVICE = <Device Selector>
Loads the K-field or G Register selected by the DEVICE VIA parameter with <Device Selector>.

3) DEVICE VIA = <Device-Selector Source>
Required Parameter. Sets the "GSPEC" T-field. This parameter chooses which K-field or G Register contains the <Device Selector>.

4) COMMAND = <Command Value>
Loads the <Command Value> into the <Command Value Source> selected by the COMMAND VIA parameter.

5) COMMAND VIA = <Command-Value Source>
Required parameter. Chooses which <Source> contains the <Command Value>.

<Port Selector> - is one of:
an integer i in the range $0 \leq i \leq 7$
a <Source> or a <G(GSPEC)>
(Possibly an <F-register> also later)

<Device Selector> - is a <Register Selector>whose value is used to select a device rather than an LSR.

<Command Value> - is a <Register Selector> whose value is used as an I/O Command rather than to select an LSR.

<Device-Selector Source> - is a <G(GSPEC)>

<Command-Value Source> - is a <Source>

32. <u>Input Statement</u>

Purpose: Send an RIO pulse to the selected port.

Header: RIO

Parameters:

1) PORT = <Port Selector>
Required parameter. Loads low order 3 bits of "KA" K-field. If the port selector is not a constant then all of "KA" is loaded.

SECTION VI.

OMITTED PARAMETERS

Most of the parameters for the Active statements are not marked as being required parameters. Often the function of these parameters is to load an F-register or a K-field with a value required for the operation being performed. When a parameter is omitted from a statement then either the F-register or K-field that it loads is not needed for this operation or the value currently there is the one desired. If a programmer wants to emphasize the fact that he is using a residual value, he may include on a statement a parameter that would otherwise be omitted and use an asterisk (*) as its operand. For example the following two program segments are equivalent:

```
SEGMENT.1: INCR FAIL
           ALU, OP = NOT LEFT, RESULT = LSR(6)

SEGMENT.2: INCR FAIL
           ALU, OP = NOT LEFT, LEFT = *, RESULT = LSR(
```

Use of the second form is strongly suggested if the listing control parameter NO PARS is chosen. (See the Listing Control Statement.)

There is seldom any advantage to deliberately omitting a parameter from a statement. If a parameter specifies a redundant operation, such as loading an F-register with a value that it already contains then that parameter will be ignored.

SECTION VII.

REGISTER-SELECTOR RANGE SPECIFICATION

Because the Lizard translator does parallelism analysis it must know all the possible inputs and outputs of each statement in a program. A problem arises from the fact that local-store and external-store registers are indirectly addressed (via the F-registers). The Lizard language allows a programmer to explicitly specify which local-store or external-store register he is using, but sometimes this is not possible. Whenever a program uses a <Register Selector> that is not a <Constant> the translator will assume that any register in Local Store or External Store (depending on the statement) could be the one referenced. Such an assumption restricts the parallelism possible in a program; therefore, Lizard provides a construction, called a <Register-Selector Range Clause>, that a programmer can use to specify the range of registers that could be referenced by a <Register Selector>.

<Register-Selector Range Clause> - takes the form:
IN (<Constant Range List>)

<Constant-Range List> - is a list of <Constant>'s and/or <Constant Range>'s separated by commas.

<Constant Range> - takes the form:
<Constant> - <Constant>
and it names all the <Constants>'s that fall in the range between the first <Constant> and the second.

Examples of <Register-Selector Range Clause>'s:

```
    IN (0-8)
    IN (0,2,4,6,8-31)
```

Whenever a programmer uses a <Register Selector> that is not a <Constant> and he knows some limits on the range of values that it could be representing, he should use a <Register-Selector Range Clause> immediately after the <Register Selector>, to inform the Lizard translator of those limits.

Example:
```
    ALU, OPP=ADD, LEFT=LSR(A), RIGHT=LSR(FAIR),
         RESULT=LSR(20)
```

becomes:

```
    ALU, OP=ADD, LEFT=LSR(A IN (0-7)),
         RIGHT=LSR(FAIR IN (0,2,4,6,8)),
         RESULT=LSR(20)
```

If a programmer uses the same <Register Selector> in several places, he may want to notify the translator of its range only once. He can do this by means of the Range Statement.

33. Range Statement

Purpose: Define the range of a <Register Selector> from this point up to its redefinition.

Header: RANGE OF <Register Selector> IS
        <Register-Selector Range Clause>

SECTION VIII.

MICROLINKAGE INSTRUCTIONS

34. Micro-instruction Fetch

Purpose: Read the next micro-instruction from Control Store into the COD Register, increment the MPC register, load the Micro Operand Buffer (MOB) and LSR(31) with the micro-operands, compute the Nano-address selected by the Micro-opcode, and jump to that address.

Header: MICRO FETCH

Parameters:

1) MPC INC = {0 | 1 | 2 | B | AB}
This parameter specifies the displacement from the MPC register of the Micro-instruction to be fetched. The MPC Register will be incremented by this amount. If the parameter is omitted 1 is assumed, <u>not</u> 0.

2) FOUND
This parameter specifies that the COD Register already contains the next micro-instruction and the MPC Register has already been incremented. Therefore the FOUND parameter and the MPC INC parameter are mutually exclusive.

3) NO OPERANDS
This parameter specifies that LSR(31) should not be loaded with the micro-instruction operand fields.

Note: This statement will result in a sequence of nano-operations similar to the following:

- Read Control Store from one past the MPC Register.
- Increment the MPC Register by one.
- Load the Nano Program Counter with the next nano-address.
- Load the Micro Operand Buffer (MOB) with the operands of the next micro-instruction.
- Load LSR(31) from the MOB.
- Jump to the nano-address in the NPC.

35. Micro-instruction Prefetch - Part 1

Purpose: This statement is used for fetching micro-instructions according to the prefetch convention described in section 7.1 of the QM-1 Manual. Instructions are fetched in two steps. This part of the fetch uses the op-field of the micro-instruction currently in the COD Register to determine the nano-address that will be jumped to in part 2 of the prefetch operation. At the same time the Micro Operand Buffer (MOB) is loaded with the operand fields of the micro-instruction currently in the COD Register.

Header: PREFETCH ONE

Parameters:

1) REREAD
If the COD Register, the MPC Register or PMPC have been changed since this micro-operation emulation was entered, so that the COD Register no longer contains the next micro-instruction to be executed then this parameter should be specified. It requests that the operation:

    READ CS, ADDR = MPC + 1

be performed before the Nano-address determination is made.

Note: This statement will result in a sequence of nano-operations similar to the following:

- Load the Nano Program Counter with an address computed from the opcode field of the micrc-instruction currently in the COD Register.
- Load the Micro Operand Buffer (MOB) with the operands of the micro-instruction currently in the COD Register.

36. <u>Micro-instruction Prefetch - Part 2</u>

Purpose: This part of the fetch reads the Control Store word <u>after</u> the one containing the <u>next</u> micro-instruction to be executed so that it will be available in the COD Register when execution of the next micro-instruction starts. The MPC Register is then incremented by one, LSR(31) is loaded with the operand fields of the next micro-instruction to be executed and a jump is made to the nano-address determined in part one of the prefetch operation.

Header: PREFETCH TWO

Parameters:

1) FOUND
This parameter specifies that the COD Register is already set as desired.

2) NO OPERANDS
This parameter specifies that LSR(31) should not be loaded with the operand fields.

Note: This statement will result in a sequence of nano-operations similar to the following:

- Read Control Store from two past the MPC Register.
- Increment the MPC Register by one.
- Load the operands of the next micro-instruction into LSR(31).
- Jump to the next emmulation.

Notes on the Prefetch-Type Microfetch

The prefetch-type microfetch convention (see section 7.1 in the QM-1 Manual) is in common use because it allows for faster nano-programs. With prefetch, the fetching of micro-instructions from Control Store is overlapped with the fetching of Nanowords from Nanostore. The increased running speed comes at the cost of a loss of simplicity and with certain constraints on the logic of the nanoprogram.

The nanoprogrammer should be reluctant to modify the COD Register before part one of the prefetch sequence has been executed since another control-store read will have to be performed to reload the COD Register with the next micro-instruction. After part one of the prefetch sequence, the type of jump that can be executed is restricted (since all Conditional Branch statements must be translated into SKIP operations).

- No backward looking conditional branches are allowed.
- No conditional branches to external labels are allowed.
- A forward looking conditional branch may jump only over statements that can be scheduled into one T-step.
- A forward looking conditional branch cannot jump over a conditional or unconditional branch.

These restrictions do not apply to Computed Branches or unconditional branches.

37. Load LSR(31)

Purpose: Load the "A" and "B" fields of LSR(31) with the micro-instruction operands currently stored in the Micro Operand Buffer (MOB) and zero the "C" field. The MOB was loaded from the COD Register during a Micro-instruction Fetch or Part 1 of a Micro-instruction Prefetch. Thus normally this statement will reload LSR(31) with the operands of the currently executing micro-instruction. But, if the prefetch convention is being used and this statement is coded after part 1 of the Micro-instruction Prefetch then the operands of the <u>next</u> micro-instruction to be executed will be loaded.

Header: LOAD LSR(31)

38. Micro Entry Point

Purpose: Mark a legal micro entry point.

Header: MICRO ENTRY

Parameters:

1) MNEMONIC = <Label>
Required parameter. The <Label> specified here will be the Mnemonic of the micro-instruction whose emulation starts at this entry point. The opcode of the micro-instruction will be the low order seven bits of the Nanostore address of this statement.

2) FORMAT = <Micro Instruction Format>
Required Parameter. This parameter is used to describe the source statement format and the Control Store format of the micro-instruction whose emulation starts at this entry point.

3) [SUPERVISOR | NO SUPERVISOR]
This parameter is used to specify whether the Nanocode produced by the Lizard statements that follow can be executed only while in Supervisor Mode (See description of F-register FIDX in section 4.3.2.3). If the parameter is omitted then the currently selected option remains in effect.

<Micro Instruction Format> - is any of the following coded messages:

(OP NULL)	(OP AR)
(OP A,B)	(OP B)
(OP A,BR)	(OP BR)
(OP AR,B)	(OP A,B,V)
(OP AR,BR)	(OP ABR,V)
(OP ABR)	(OP ABS,V)
(OP ABS)	(OP A,B,C,D,E)
(OP A)	(OP A,B,C,DER)

The symbols in the coded message have the following meanings:

OP	7 bit opcode
NULL	no operands
A	5 bit absolute A parameter
B	6 bit absolute B parameter
AR	5 bit positive relative A parameter
BR	6 bit signed relative B parameter
ABR	11 bit signed relative AB parameter
ABS	11 bit absolute value
V	18 bit absolute word
C	6 bit absolute C parameter
D	6 bit absolute A parameter
E	6 bit absolute B parameter
DER	11 bit signed Relative AB parameter

These formats are those accepted by the Version 1.3.9 Micro-assembler.

SECTION IX.

SUBROUTINE SUPPORT STATEMENTS

A nanoprogram subroutine is a sequence of statements that, upon completion, returns control to the point from which it was invoked. Because of the limitted capability for nano-address manipulation in a nanoprogram, a subroutine may not modify the Nano Program Counter. The subroutine support statements were devised to ensure that the Lizard translator follows this rule.

39. Subroutine Header Statement

Purpose: This statement delimits the start of a Lizard subroutine. The <Label> on this statement is the name that is used to invoke the subroutine.

Header: SUBROUTINE

Paramet s:

1) LOC=<Address Constant>
This parameter specifies the Nanostore address where the subroutine is to start.

2) HELD
This parameter specifies that the values in the "KALC", "KSHC", "KSHA" AND "KS" K-fields may have been held constant by a call to this subroutine.

3) HELD2
This parameter specifies that the values in the "KA" and "KB" K-fields may have been held constant by a call to this subroutine.

40. Subroutine Return

Purpose: This statement directs that a subroutine should return to the point from which it was invoked and, if the return is conditional, gives the conditions under which the return should be made.

Header: RETURN

Parameters:

1) COND = <Condition>
Loads one of "KS", "KT" or "KX" K-fields, depending on COND TYPE parameter, with the <Condition> value.

2) CON~~D~~ TYPE = <Condition Type>
Required parameter, unless branch is unconditional. Sets the "TEST SPECIFIER" T-field.

3) F = <F-register>
Sets the "FSEL1" T-field to the <F-register> value. This parameter chooses the <F-register> that is to be tested for zero. Therefore it is only required if F NOT ZERO is part of the condition to be tested.

4) INDEX = <Label>
This parameter chooses the INDEX ALU statement whose result is to be tested for zero. Therefore it is only required if IRNZ is part of the condition to be tested.

5) HOLD
This parameter specifies that the values of the "KALC", "KSHC", "KSHA" and "KS" K-fields will be retained for use at the point to which the subroutine returns.

6) HOLD 2
This parameter specifies that the values of the "KA" and "KB" K-fields will be retained for use at the point to which the subroutine returns.

41. Subroutine Call

Purpose: This statements directs that a subroutine be invoked from this point and gives the conditions under which the invokation will be made.

Header: CALL <Label>

Parameters:

1) COND = <Condition>
Loads one of "KS", "KT" or "KX" K-fields, depending on COND TYPE parameter, with the <Condition> value.

2) COND TYPE = <Condition Type>
Required parameter, unless branch is unconditional. Sets the "TEST SPECIFIER" T-field.

3) F = <F-register>
Sets the "FSEL1" T-field to the <F-register> value. This parameter chooses the <F-register> that is to be tested for zero. Therefore it is only required if F NOT ZERO is part of the condition to be tested.

4) INDEX = <Label>
This parameter chooses the INDEX ALU statement whose result is to be tested for zero. Therefore it is required only if IRNZ is part of the condition to be tested.

5) HOLD
This parameter specifies that the values of the values of the "KALC", "KSHC", "KSHA" and "KS" K-fields will be retained for use by the subroutine invoked.

6) HOLD2
This parameter specifies that the values of the "KA" and "KB" k-fields will be retained for use by the subroutine invoked.

Notes on Using Subroutines

A subroutine is a Lizard program unit that starts with a Subroutine Header Statement and ends with a Program-Unit Trailer statement (END statement). Lizard program units cannot be nested in each other. A subroutine may be given more than one entry point by using the External Entry Point statement. There are several restrictions on the type of branch that may be performed in a subroutine.

- No backward looking branches may be used.
- Forward looking unconditional branches must jump to a label internal to the subroutine and the branch statement must be immediately followed by an External Entry Point.
- Forward looking conditional branches may branch around a sequence of statements only if they will fit into one T-vector or if the last statement of the sequence jumped is a return statement.
- A forward looking conditional branch may not jump over a conditional return statement.
- A forward looking conditional branch may jump over only one unconditional return statement and the statements between them must be able to fit into three T-vectors.
- Interrupts cannot be enabled in a subroutine.
- A subroutine cannot contain a Subroutine-Call statement.

SECTION X.

CONTROL STATEMENTS

42. Program-Unit Header

Purpose: This statement delimits the sta of a Lizard program unit.

Header: BEGIN

Parameters:

1) LOC = <Address Constant>
This parameter specifies the Nanostore addr s here the output nanocode is to start.

2) SUPERVISOR
This Parameter specifies that the nanocode produced by this Lizard program unit can be executed only while in the Supervisor Mode (See description of F-register FIDX in section 4.3.2.3 of the QM-1 Manual).

3) HELD
This parameter specifies that the values in the "KALC", "KSHC", "KSHA" and "KS" K-fields may have been held constant by a jump to this statement.

4) HELD2
This parameter specifies that the values in the "KA" and "KB" K-fields may have been held constant by a jump to this statement.

<Address Constant> - is an integer i in the range
$0 \le i \le 1024$ (i.e. a ten-bit constant).
It is assumed to be in decimal notation unless it begins with a zero, then it is interpreted as octal.

43. Program-Unit Trailer

Purpose: This statement delimits the end of a Lizard program unit.

Header: END

44. Force Nanoword Boundary

Purpose: This statement is used to force a new nanoword boundary and to change scheduler directives part way through a Lizard program unit.

Header: LEDGE

Parameters:

1) [SUPERVISOR | NO SUPERVISOR]
This parameter is used to specify whether the Nanocode produced by the Lizard statements that follow can be executed only while in Supervisor Mode. (See description of F-register FIDX in section 4.3.2.3 of the QM-1 Manual.) If this parameter is omitted then the currently selected option remains in effect.

2) LOC = <Address Constant>
This parameter specifies the Nanostore address where the forced new nanoword is to be placed; the remaining nanocode generated will continue after it.

45. F-store and K-field Dump

Purpose: This statement requests that the scheduler dump its lists of the contents of F-store and the K-fields as it has determined them for this point in the program. If one is known only as a range of values, then the range will be printed. Any register or field whose contents are not known will be so marked. The scheduler maintains these lists so that it can determine, in normal programming situations, which Local Store Registers are being used to pass data between statements.

Header: DUMP

46. External Entry Point

Purpose: This statement marks points that can be branched to by jumps from outside this Lizard program unit. The <Label> attached to this statement is the one that the external jump will use.

Header: ENTRY

Parameters:

1) HELD
This parameter specifies that the values in the "KALC", "KSHC", "KSHA" and "KS" K-fields may have been held constant by any jump to this location.

2) HELD2
This parameter specifies that the values in the "KA" and "KB" K-fields may have been held constant by any jump to this location.

Note: This statement can also be used to mark alternate entry points in a subroutine.

47. External-Label List

Purpose: List all the <Labels> outside this Lizard program unit that are used by branch statements within this unit.

Header: EXTERNAL (<Label List>)

48. No Operation

Purpose: A clever programmer may find a use for this statement. It does nothing at all.

Header: NO OP

49. Listing Control

Purpose: To Control the form of the Lizard source program listing and scheduler output.

Header: LIST

Parameters:

1) [SOURCE | NO SOURCE]
A source listing is either requested or supressed.

2) [EXPAND | NO EXPAND]
Source statements with preprocessor expressions in them are relisted with the value of each expression replacing its occurrence. If the NO SOURCE parameter is also specified then even statements without preprocessor expressions are listed but statements with expressions are listed only once.

3) [PARS | NO PARS]
The assumed values of all parameters omitted from a statement will printed after each statement if they can be computed. For example, if a parameter that normally loads an F-register with a value is omitted then the current value in that F-register is printed if known.

4) [NANO | NO NANO]
The Nanccode produced from the Lizard source program is listed.

5) [MAP | NO MAP]
A listing is produced of all the <Labels> used in the Lizard source program along with their Nanocode addresses.

6) TITLE = <Title>
The <Title> specified will appear at the top o every page from this point on.

7) PAGE
The next statement will appear at the top of a new page.

Note: Listing Control statements may be inserted at any point in the source program to change the type of listing produced. The options chosen remain in effect until they are explicitly changed. Since the Nanocode listing and map are produced only at the end of each Lizard program unit only the ultimate selection of these parameters is significant.

<Title> - is the first sixty characters that follow the equal sign after the TITLE parameter. Preprocessor expressions are valid and are evaluated before the title is used. Therefore to represent an apostrophe in a <Title> use two adjacent apostrophes.

SECTION XI.

PREPROCESSOR EXPRESSIONS

A <Preprocessor Expression> is an expression evaluated at compile time whose result can be used to modify the source program. A preprocessor expression enclosed in apostrophes can be used anywhere in a Lizard program; it will be evaluated and its occurrence replaced by the result. Preprocessor expressions are also used in the Preprocessor Expression Statement and in the If Statement but without the delimiting apostrophes. Preprocessor expressions may be continued across line boundaries by the use of the continuation character (#).

There are only two preprocessor data types:
1) character strings and
2) integers in the range -65,536 to 65,535
(an arbitrary sixteen bit maximum)

Constants

Character string constants are enclosed in quotation marks ("). The occurrence of a quotation mark in a string is represented by two sequential quotation marks. An empty string is a valid character string and is represented by two sequential quotation marks (""). Preprocessor character string constants may not be continued across line boundaries but long strings can be formed by the use of the concatenation operator.

There are two types of integer constants: decimal and octal. An octal constants may have only valid octal digits (0 to 7) and must begin with a zero. Either type may be signed and their value will be held as a sixteen bit two's complement binary integer.

Variables

Preprocessor variable names must start with a letter and can contain only letters and digits. They can be any length but only the first ten characters are significant. The type of a variable is taken from the expression whose value is assigned to it.

Labels internal to the Lizard program unit may be used as constants in a preprocessor expression only as the argument to a LABEL function call. The result will have integer type and its value will be the Nanostore address of the statement that the label is attached to. Expressions containing <Labels> will not be fully evaluated until all scheduling has been completed, therefore if they affect the amount of nanocode generated in any way they will be rejected.

Operators

Operators have no precedence, they are evaluated strictly from left to right but the order of evaluation can be changed by parentheses.

There five types of preprocessor operators and functions:

i) Integer Operators

+	addition and positive sign
-	subtraction and negative sign
*	multiplication
/	integer division
REM(x,y)	integer remainder of x/y
ABS(x)	absolute value function
LOW(x)	lower six bits of a two's complement representation of x
HIGH(x)	two's complement representation of x arithmetically shifted right six places
LABEL(x)	returns the Nano-address of <Label> x

The functions LOW and HIGH are useful for preparing the address in LSR(31) for a Computed Branch or a Write Nanostore.

ii) Relational Operators

<	less than
>	greater than
=	equal
<=	less than or equal
>=	greater than or equal
¬=	not equal

These operators take integer operands and yield an integer result with the value one for true and zero for false.

iii) <u>Logical Operators</u>

```
&    logical and
++    logical or
¬    logical not
/+    exclusive or
SHL(x,y)    x is arithmetically shifted left y bits
SHR(x,y)    x is arithmetically shifted right y bits
```

These operators take integer operands and yield an integer result. The operands are treated as two's complement binary numbers and the result is computed for each bit position to form an integer result.

iv) <u>Character String Operators</u>

```
+   concatenation of strings
SUB(x,y,z)    take a substring from string x, starting
     at character y, that is z characters long
CHAR(x)    the integer x is converted to a character
     string in base 10 representation
CHAR8(x)    the integer x is converted to a character
     string in base 8 representation (without the
     leading zero: a leading zero is easier to add on
     later than it is to strip off)
```

v) <u>Assignment Operator</u>

```
->    value and type assignment
```

The variable on the right takes the value and type of the expression on the left. The value and type assigned also becomes the result of the assignment operation. Thus in the expression:

```
     1 + 1 -> X + 2 -> Y
```

variable X will take the value 2 and the type integer, and variable Y will take the value 4 and the type integer.

<u>Syntax of Expressions</u>

The syntax of a <Preprocessor Expression> should be fairly obvious but certain details need to be mentioned. Two operators may not appear together hence a negative sign or positive sign can only appear at the front of an expression or after a left parentheses. Blanks can appear anywhere except in a variable name, a function name or between the characters of a multi-character operator.

SECTION XII.

PREPROCESSOR STATEMENTS

50. Preprocessor Expression Statement

Purpose: Evaluate a preprocessor expression. If this statement is to have any effect on the program the <Preprocessor Expression> should contain at least one assignment operator and to be fully sensible the last operation in the expression should be assignment. This statement is mainly useful for grouping initializations of variables together or for emphasizing important assignment operations when burying them in source code could conceal their presence.

Header: P <Preprocessor Expression>

51. If Statement

Purpose: An If Statement is always paired with one End If Statement in the same way that an open bracket is paired with a close bracket; no matter how deeply nested they are, one can always find which two match up. An If Statement may also have one Else Statement associated with it. An Else Statement is associated with the most deeply nested If - End If pair that encloses it.

The <Preprocessor Expression> in the If Statement is evaluated. If the result is one all the statements up to the associated Else Statement (if one is present) or End If Statement will be included as part of the source program but those statements between the associated Else Statement and End If Statement (if there are any), will be ignored. If the result is zero the opposite happens. If the result is neither zero nor one a compile time error will be produced.

Header: IF (<Preprocessor Expression>) THEN OPEN [<Label>]

Note: If the optional <Label> is included in the If Statement then it is the label that will appear on both the associated Else Statement and End If Statement. This label cannot be used as the object of a branch. This method of delimiting statement groups is sometimes more convenient than indentation when programming on punched cards.

52. Else Statement

Purpose: Refer to the If Statement.

Header: ELSE

53. End If Statement

Purpose: Refer to the If Statement.

Header: CLOSE

SECTION XIII.

IMPROVEMENTS

There are two specific improvements to Lizard which the author had planned but did not take the initiative to make.

1) Lizard should contain a method for describing and maintaining register usage conventions for a group of nanoprograms that implement micro-instructions. For instance the MULTI microprogramming language observes the conventions that FCOD and FAIR contain the value 31 when entering and leaving each micro-instruction emulation.

2) The three microfetch statements could be combined into one. There are a limited number of operations that can take part in a microfetch sequence. The new statement could have a parameter to select and modify each of these operations, and also parameters that would specify common combinations of the basic operations. This format would make it easy to adopt the usual microfetch conventions, alter a convention to deal with an unusual circumstance or invent a new microfetch sequence.

APPENDIX B

PPG NODE TYPES

DIRECTORY

Notation Used

Each source to a node is followed by a timing constraint in braces, "{}". This states the number of T-periods that must elapse from the setting of the value of the source, to the use of the value. This time is measured from the edge of the clock cycle in which the value of the source is set to the edge in which it is used. The time from the trailing edge of one T-step to the leading edge of the next T-step is zero T-periods and thus some leading edge nodes have a timing constraint of {0T} on some of their sources.

Many leading-edge nodes require their sources to remain stable while they are being used. This constraint is specified as a number in angle brackets, "<>", which states the number of T-periods that the source must remain stable after the leading edge of the node that uses it. When a timing constraint of this type is followed by the abbreviation "o.k." then it is impossible to violate. Sources followed by a dollar sign in angle brackets, "<$>", may not have their values changed in the same T-step that they are used.

Some sources and sinks to a node are used only under certain conditions. These conditions are given in square brackets, "[ ]", after the source or sink. If the condition requires information that is not contained in the node, then the closing bracket is followed by a plus sign, "+".

Nodes that can use LSR(31) fields A, B or C as a source have LSR(31) {2T} included as a source. This means that if A, B or C were prepared by an 18-bit operation on LSR(31) then two T-periods must elapse before they can be used as a source; whereas, if they were prepared by a 6-bit operation one T-period will normally suffice. This is the only case where the timing constraint depends on the type of the sinking node.

Description of Nodes

Constant->K

Sources: N/A

Sinks: K selected

Parameters:
1) Constant
2) K

Edge: Leading

Source->F

Sources:
Source {1T} [Source ≠ INCF, DECF, SW, IOI[illegible]
LSR(31) {2T} [Source = A, B, C]

Note: Stretch T-step when Source = INCF, [illegible] or G.

Sinks: F selected

Parameters:
1) Source
2) F

Edge: Trailing

F->Dest

Sources: F {1T} [F ≠ FIPH]

Sinks:
Dest.
LSR(31) [Dest = A, B, C]

Parameters:
1) F
2) Dest

Edge: Trailing

G->F

Sources: G reg. {2T}

Sinks: F reg.

Parameters:
  1) G
  2) F

Edge: Trailing

Note: This node must stretch T-vector.

Swap(Source-dest, F)

Sources:
  Source-dest {1T}
  F {1T}
  LSR(31) {2T} [Source-dest = A, B, C]

  Note that legal Source-dests are A, B, C, KA, KB; KX, KS, KSHA and that problems may arise for those Source-dests after the semi-colon since they use more than one group.

Sinks:
  Source-dest
  F

Parameters:
  1) Source-dest
  2) F

Edge: Trailing

Source->Dest (via FIPH)

Sources:
  Source {2T} [Source ≠ INCF, DECF, SW, IOID]
  LSR(31) {2T} [Source = A, B, C]

Sinks: Dest

Parameters:
  1) Source
  2) Dest

Edge: Trailing

Note: Always stretch T-step for this node.

<u>ALUF</u>

Sources:
- FL {1T} [Operation Type]
- FR {1T} [Operation Type]
- AUX3 source {1T}
- LSR(31) {2T} [source = A, B, C]
- Left Source {1T} [FL = FIPH]
- Right Source {1T} [FR = FIPH]

Sinks: F result

Parameters:
1) Left F Input (FSEL1)
2) Right F Input (FSEL2)
3) Result F (FSEL0)
4) Operation Source Select (AUX3)
5) Operation Type
   [0=no input, 1=left input, 2=right input, 3=left and right input]

Edge: Trailing

<u>XIO</u>

Sources:
- FIPH {S.T.} <$> (I/O Command Bus)
- G(GSPEC) {S.T.} <$> (Device Selection Bus)
- KA {1T, 0T when ka is loaded with nanoword} (Port Selection)
- ESR(KA) {0T}

Note: {S.T.} means that these sources can have their values set in the same T-step that contains the XIO.

Sinks: External Channel Register(KA)

Parameters:
1) GSPEC
2) FIPH

Edge: Leading

Note: If 1T is added to all source timing we can call it trailing edge.

RIO

Sources:
KA {1T, 0T when ka is loaded with nanoword}
External Channel Register(KA) {5T} [When this RIO is preceeded by an RIO or XIO]

Sinks:
ESR(KA)
External Channel Register(KA) [When this RIO is the first of a pair doing data acquisit

Parameters:
1) GSPEC
2) FIPH

Edge: Leading

MSGO (FETCH MS)

Sources:
Direct M.S. Access {0T}
Super Direct M.S. Access {0T} - (set by AUX ACT)
MS BUSY {1T arbitrary}
FMIX {0T} <1T>o.k.
ESR(FMIX) {0T} <1T> [FMIX > 31 and FMIX < 40]+
LSR(FMIX) {0T} <1T>o.k. [FMIX < 31]+
ALL ONES {N/A} [FMIX > 40]+
ESR16 & ESR17 {0T} <1T> [DIRECT MS ACCESS Bits]
Mainstore Contents {0T} [LSR(FMIX) - illegal address leads to Mainstore not accessed]

Note: Error in timing table in QM-1 Quick Reference Card.

Sinks: MOD Register

Parameters: 1) DIRECT MS ACCESS

Edge: Leading

<u>READ MS</u>

Sources: All sources of the MSGO node

Sinks:
MOD Register
Main-Store Contents*

*Note: Since old contents of Mainstore are restored this sink may not be necessary.

Parameters: None

Edge: Leading

<u>MSRS (WRITE MS)</u>

Sources:
Direct M.S. Access {1T}
Super Direct M.S. Access (set by AUX ACT)
FMIX {1T} <1T>o.k.
ESR(FMIX) {1T} <1T> [FMIX > 31 and FMIX < 40]+
LSR(FMIX) {1T} <1T>o.k. [FMIX < 31]+
ALL ONES {N/A} [FMIX > 40]+

Sinks:
Main-Store Contents
MOD Register

Parameters: 1) RMI Select

Edge: Leading

<u>GATE MS</u>

Sources:
MS DATA {Special} (MOD Register)
FMOD {1T} <$>
RMI Store(RMI Select) {2t}* [RMI Select]

*Note: AUX ACT may actually prepare RMI Store sooner (P.9 of QM-1 Quick Reference Card)

Sinks:
ESR(FMOD) [FMOD > 31 and FMOD < 40]
LSR(FMOD) [FMOD < 31]

Parameters: 1) RMI Select

Edge: Trailing

<u>GATE ES</u>

Sources:
FEOD {1T} <$>
FEOA {1T}
ESR(FEOA) {1T}

Sinks: LSR(FEOD)

Parameters: none

Edge: Trailing

<u>LOAD ES</u>

Sources:
FEID {1T}
LSR(FEID) {1T}
FEIA {1T} <$>

Sinks: ESR(FEIA)

Parameters: none

Edge: Trailing

<u>TXX</u>

Sources: none

Sinks: none

Parameters: none

Edge: Trailing

Note: Unlike AUX ACT 40, TXX causes program to stop after next T step.

READ CS

Sources:
    FCIA {1T}, LSR(FCIA) {1T} [C.S. Address Select = CIA]
    COD Register {2T} [CSAS = COD]
    FMPC, LSR(FMPC) {1T} [CSAS = MPC, MPC+1 or MPC+2]
    FMPC, LSR(FMPC), LSR(31) {1T} [CSAS = MPC+B or MPC+AB]
    All INDEX ALU sources {3T} [CSAS = INDEX]
    Control Store Contents {1T}

Sinks: COD Register

Parameters: 1) C.S. Address Select

Edge: Leading

WRITE CS

Sources:
    FCID, LSR(FCID) {1T} <1T>
    FCIA, LSR(FCIA) {1T} [CSAS = CIA]
    COD Register {2T} [CSAS =COD]
    FMPC, LSR(FMPC) {1T} [CSAS = MPC, MPC+1, MPC+2]
    FMPC, LSR(FMPC), LSR(31) {1T} [CSAS = MPC+B, MPC+AB]
    All INDEX ALU sources {3T} [CSAS =INDEX]

Sinks:
    Control Store Contents
    COD Register

Parameters: 1) C.S. Address Select

Edge: Leading

Note: A READ CS executed simultaneously will be ignor

GATE CS

Sources:
    COD Register {2T}
    FCOD {1T} <$>

Sinks: LSR(FCOD)

Parameters: none

Edge: Trailing

GATE ALU

```
Sources:
     ALU Status Enable {0T}
     FAIL, LSR(FAIL) {2T}* [Type of KALC]
     FAIR, LSR(FAIR) {2T}* [Type of KALC]
     FAOD {1T} <$>
     KALC {2T}*
     KSHC {1T} (Only DOUBLE/SINGLE bit is important)
     CIH {2T} [KALC bit 4 = 0]
     -- AND --
     FSID, LSR(FSID), KSHC {2T} [KSHC includes DOUBLE & LEFT
          and KSHA ≠ 0]

     *Note: {3T} if KSHC includes double.

Sinks:
     LSR(FAOD)
     CIH [CARRY CTL > 4]
     COH [CARRY CTL > 2]
     FIST [ALU STATUS ENABLE]

Parameters:
     1) Type of KALC
          [0=no input, 1=left input, 2=right input
          4=CIH input, and all combinations of these]
     2) Type of KSHC
          [0=Single, 1=Double]
     3) Carry Control
          [0=no right control, 1=right control]
     4) ALU STATUS ENABLE

Edge: Trailing
```

GATE SHIFTER

Sources:
    SH Status Enable {0T}
    FSID, LSR(FSID) {2T}
    FSOD {1T} <$>
    FAOD {1T} <$> [KSHC includes DOUBLE and GATE BOTH is
        specified]
    KSHA, KSHC{2T}
    COH {1T} [Left Control = 1]
    All ALU sources {3T} [KSHC includes DOUBLE, i.e. bit
        1=1]
    All ALU sources {2T} [KSHC includes DOUBLE and KALC =
        PASS LEFT]
    -- OR --
    Only ALU sources {2T} [KALC includes DECIMAL, i.e. bit
        1 = 1]

Sinks:
    LSR(FSOD)
    FIST [SH STATUS ENABLE]
    COH

Parameters:
    1) Type of KSHC
    2) Type of KALC
    3) Carry Control
    4) GATE BOTH
    5) SH STATUS ENABLE

Edge: Trailing

CARRY CONTROL

Sources: none

Sinks:
    CIH [Carry Control = 1 or 2]
    COH [Carry Control = 5 or 6]

Parameters: 1) Carry Control value

Edge: Trailing

INDEX ALU

Sources:
- Left select: A, B, KX, KA, KB, G(GSPEC)* {2T}
- LSR(Left Select) {2T}
- Right Select: A, B, KT, KB, G8 to G11 {2T}
- LSR(Right Select) {2T} [Right Select < 11]+
- MOD Register {2T} [Right Select = 14]+
- MS DATA {Special}
- COD Register {2T} [Right Select = 15]+
- ALL ONES {N/A} [Right Select = 12 or 13]+
- Result Select {1T} <$>: G(0) to G(11), KSHA, B, KS, KX

*Note: If G(GSPEC) is used as a source it must be repeated in the previous T step. Only the GSPEC must be set up in the previous T not the selected K field nor the selected LSR.

Sinks: LSR(Result Select)

Parameters:
1) Left Select Select (AUX2)
2) Right Select Select (AUX3)
3) Operation Select or Operation Select Select (FSEL2)
4) Result Select Select (GSPEC)

Edge: Trailing

INC MPC

Sources:
- FMPC <$>, LSR(FMPC) {2T}
- LSR(31)(B) {2T} [GSPEC = 2 (+0, +4, +8 or +12)]
- LSR(31)(AB) {2T} [CGSPEC = 3 (+0 +4, +8 or +12)]

Sinks: LSR(FMPC)

Parameters: 1) GSPEC (modulo 4)

Edge: Trailing

LOAD NPC

Sources:
    KN {1T} [NPC Source = KN]
    COD Register {2T} [NPC Source = CS]
    Interrupt Bits {1T} [NPC Source = SEQ]
    FIDX {1T}

Sinks:
    NPC
    Micro Operand Buffer [NPC Source = CS]

Parameters:
    1) NPC Source
        [1=CS, 2=KN, 3=SEQ]

Edge: Trailing

READ NS

Sources:
    BRANCH Bit, ALTERNATE Bit {0T} <1T>
    NPC {0T} <1T> [BRANCH Bit = 0 & ALTERNATE Bit = 0]+
    Nanostore Contents {0T} <1T>
    Allow Interrupt Bits {1T}*

    *Note: This timing constraint means that if the allow
        interrupt bits are set then READ NS may not appear
        in T1.

Sinks: NOD Register

Parameters: none

Edge: Leading

<u>WRITE NS</u>

Sources:
    LSR(31) {1T} <2T>*
    FEOA, ESR(FEOA) {1T}

    *Note: All WRITE NS and READ NS operations initiated
        during this two T-periods are ignored.

Sinks:
    NOD Register
    Nanostore Contents

Parameters: none

Edge: Leading

<u>LOAD R31</u>

Sources: Micro Operand Buffer {1T}

Sinks:
    A, B, C (These are 6-bit domain transfers, see pages 82
        & 86)

Parameters: none

Edge: Trailing

Branch

Sources:
- FIST {2T} [TEST* = 2 or 3]
- F (selected) {2T} [TEST = 6 or 7 and KX contains FZERO]+
- All ALU sources except ALU STATUS ENABLE and FAOD {3T} [TEST = 4 or 5, Bits 4, 3, 2, or 1 are on in KT, plus all info source conditions of GATE ALU]+
- All Shifter sources except ALU & SH STATUS ENABLE, FSOD, FAOD {3T} [TEST = 4 or 5, any bit of KT is set, plus all info source conditions of gate SH]+
- KS {2T} [TEST = 2 or 3]
- KT {2T} [TEST = 4 or 5]
- KX {2T} [TEST = 6 or 7]
- All INDEX sources {2T} [TEST = 6 or 7, KX contains INDEX RESULT]+
- MOD Register status {1T, (logical problem not timing)} [TEST = 6 or 7, KX contains MS DATA]+
- NOD Register {2T} [This is the only source when TEST = ALWAYS]

*Note: Here TEST refers to the TEST SPECIFIER field of the T vector.

Sinks: N/A

Parameters:
1) TEST SPECIFIER
2) F Sel1
3) Type of Test

Edge: Trailing

Label

Sources: none

Sinks: none

Parameters: none

Edge: Leading

AUXILIARY ACTION

Sources:
    FACT {1T}
    COD Register {1T} [FACT = 55, 56, 57]+

Sinks:
    RMI Store(RMI SELECT) [FACT = 55, 56, 57]+
    ENABLE Interupt Bits
    Super Direct MS Access

Parameters: 1) RMI Select

Edge: Trailing

ALLOW INTS

Sources: None

Sinks: ALLCW INTS Bits

Parameters:
    1) Type of Interrupts Allowed
        [1=Nano, 2=Micro, 3=Both]

Edge: Leading

CLEAR (GEN.) INTERRUPTS

Sources: Interrupts pseudo source) [Mode = 1]

Sinks: Interrupts (pseudo sink) [Mode = 2]

Parameters:
    1) Level
    2) Mcde [1=Clear, 2=Generate]

Edge: Trailing

<u>BEGIN</u>

Sources: none

Sinks: none

Parameters: none

Edge: Leading

<u>END</u>

Sources: none

Sinks: none

Parameters: none

Edge: Trailing

APPENDIX C

SAMPLE PROGRAMS

Three sample programs are presented here: one in nano-assembler and two in Lizard. The first program is in nano-assembler and consists of the nanocode that implements the micro-instructions SRAI and SWMS from the MULTI Multiprogramming-Support System. The second program is the first program translated into Lizard. The last program shows how a conditional-jump micro-instruction could be implemented by a nanoprogram that uses no branches or skips (as promissed in Section 4.5). It is presented in Lizard but it could be easily coded in nano-assembler.

```
*
*  SAMPLE PROGRAM 1
*
        B.IR:=31.
SRAI:  MICRO = OP A.B, LEGAL MICRO OP
*      'SRAI A,B' PERFORMS AN ARITHMETIC RIGHT SHIFT,
*      [illegible]ES, ON R('A').  THE 'B' PARAMETER IS
*      [illegible]ED TO THE SHIFT AMOUNT FIELD, WHILE 'A'
*      [illegible]E SHIFTER INPUT AND OUTPUT BUSSES.
*      S[illegible] STATUS IS SET (SHIFTER HIGH BIT, SHIFTER
*      [illegible]
....   A[illegible]INTS, SH STATUS ENABLE,
       KSH[illegible]= SINGLE ! RIGHT ! ARITHMETIC
S...   A->FSID, A->FSOD     "SET SH BUSSES"
       B->KSHA              "SET SHIFT AMCUNT"
       LOAD NPC(CS)         "READ NEXT NANOWORD ADDRESS"
.X..   READ CS(MPC+2)       "PREPARE NEXT MICRO-INSTRUCTION"
       MPC PLUS 1, READ NS
..X.   GATE SH              "SHIFT R(A)"
       GATE NS, LOAD R31    "END INSTRUCTION"
*
*
SWMS:  MICRC = OP A.B, LEGAL MICRO OP
*      'SWMS A,B'  SWAP R('A') WITH MS(R('B')).
*      THE MAIN STORE OPERANC ADDRESS IS ACCESSED
*      DIRECTLY FROM R('B').  THE DATA REGISTER IS
*      SPECIFIED DIRECTLY AS R('A').  STATUS IS
*      UNAFFECTED.
....   BRANCH(SWAP MS), KX=MS BUSY, HOLD
       KALC=PASS RIGHT, KA=R.IR
S...   GATE NS(NOT X)         "AWAIT MAIN STORE NOT BUSY"
       KA->FMOD, B->FMIX      "SET UP MAIN STORE BUSSES"
       A->FAOD, CLEAR CIH     "READY ALU FOR PASS RIGHT"
       LOAD NPC(CS)           "FIND NEXT MICRO-EMULATION"
.S..   FETCH MS               "READ MAIN STORE"
       READ CS(MPC+2)         "PREFETCH MICRO-INSTRUCTION"
       MPC PLUS 1             "SEQUENCE R.MPC"
       READ NS, GATE NS       "FETCH NEXT NANOWORD"
*
SWAP MS:
....   ALLOW INTS, KX = MS DATA, HOLD
S...   A->FMIX                "PLACE DATA ON INPUT BUS"
       GATE NS(NOT X)         "AWAIT MAIN STORE DATA AVAIL."
       GATE MS                "TRANSFER WORD READ"
.S..   GATE ALU, WRITE MS     "SWAP DATA WORDS"
       READ NS, GATE NS, LOAD R31  "END INSTRUCTION"
```

```
¢
¢  SAMPLE PROGRAM 2
¢
        BEGIN
        P  31 -> R.IR      ¢ A PREPROCESSOR STATEMENT
¢
¢  SHIFT RIGHT ARITHMETIC IMMEDIATE
¢
SRAI:   MICRO ENTRY, MNEMONIC=SRAI, FORMAT=(OP A,B)
        PREFETCH ONE
        SHIFTER, OP=SINGLE RIGHT ARITHMETIC, AMOUNT=B
             INPUT=LSR(A), RESULT=LSR(A), STATUS ENABLE
        PREFETCH TWO
¢
¢  MAIN STORE SWAP
¢
SWMS:   MICRO ENTRY, MNEMONIC=SWMS, FORMAT=(OP A,B)
        PREFETCH ONE
        FETCH MS, ADDR=LSR(B), DEST=LSR(31)
        RESTORE MS, SOURCE=LSR(A)
        ALU, OP=PASS RIGHT, RIGHT=LSR('R.IR'),
             RESULT=LSR(A)
        PREFETCH TWO
        END
```

```
¢
¢  SAMPLE PROGRAM 3
¢
       BEGIN
¢
¢  BRANCHLESS JRZ   "JUMP ON ALU RESULT ZERO"
¢
¢  LIZARD PROGRAM TO IMPLEMENT A CONDITIONAL BRANCH
¢  MICRO-INSTRUCTION WITHOUT ANY NANOBRANCHES OR SKIPS
¢
       MICRO ENTRY, MNEMONIC=JRZ, FORMAT=(OP ABR)
¢
¢  MASK OUT ALL BITS IN FIST EXCEPT "ALU RESULT ZERO"
¢
       04 -> KA
       ALUF, OP=AND, LEFT=FIST, RIGHT=KA, RESULT=FIST
¢
¢  GENERATE A "PASS LEFT" OPERATION IN KALC IF
¢  FIST=0 AND A "L OR R" OPERATION IF FIST ¬= 0
¢
       013 -> G(1)
       ALUF, OP=ADD, LEFT=FIST, RIGHT=G(1), RESULT=G(2)
       G(2) -> KALC
¢
¢  ASSUME LSR(0) CONTAINS ZERO
¢  MOVE EITHER LSR(0) OR LSR(31) TO LSR(31)
¢
       ALU, LEFT=LSR(0), RIGHT=LSR(31), RESULT=LSR(31)
¢
¢  ADD LSR(31) TO LSR(FMPC)
¢
       ALU, OP=ADD, LEFT=LSR(31), RIGHT=LSR(FMPC),
            RESULT=LST(FMPC)
       PREFETCH ONE, REREAD  ¢ FETCH FROM NEW MICRO-ADDR.
       PREFETCH TWO
       END
```