A Description of the MATHILDA Processor

by

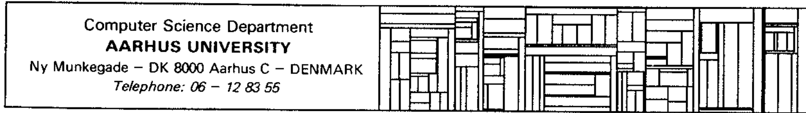
Bruce D. Shriver
and
Peter Kornerup

DAIMI PB-52

September 1975
(revised July 1980)

This work has been supported by the Danish Natural Science Research Council Grant No. 511-1546 and NATO Grant No. 755.

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



A Description of the MATHILDA Processor

by

Bruce D. Shriver

and

Peter Kornerup

Abstract
A dynamically microprogrammable processor called MATHILDA is described. MATHILDA has been designed to be used as a tool in interpreter and processor design research. It has a very general microinstruction sequencing scheme, sophisticated masking and shifting capability, high speed local storage, a 64-bit wide main data path, a partially encoded microinstruction, and other features which make it reasonably well suited for this purpose. Also, hardware modification is relatively easily undertaken to enhance the experimental nature of the machine.

This work has been supported by the Danish Natural Science Research Council Grant No. 511-1546 and NATO Grant No. 755.

TABLE OF CONTENTS

Foreword 2

1.0 INTRODUCTION

1.1 Historical Notes 3
1.2 General Design Criteria and Constraints 4

2.0 THE MATHILDA PROCESSOR

2.1 The Register Group and Standard Group 6
2.2 Counter A 8
2.3 Main Data Path 10
2.4 Working Registers 11
2.4.1 Microinstruction Format and a Few Examples 12
2.5 The Bus Shifter 16
2.6 Bus Masks 19
2.7 Postshift Masks 21
2.8 The Arithmetical and Logical Unit 24
2.9 The Local Registers 26
2.10 The Accumulator Shifter 27
2.11 The Variable Width Shifter 31
2.12 Double Shifter 32
2.12.1 Two examples using the shifters 32
2.13 The Accumulator/Variable/Double Shifter Standard Group 34
2.14 Loading Masks 34
2.15 The BUS Parity Generator 36
2.16 The Bit Encoder 36
2.16.1 Bit Encoder Conditions 39
2.17 The Status Port 40
2.18 Input Facility 41
2.19 Output Facility 42
2.20 The MDP Structure 43
2.20.1 The Bus Latch and the Shifted Bus Latch 45
2.21 The Control Unit 46
2.21.1 Microinstruction Sequencing 46
2.21.2 The Control Unit Arithmetical Logical Unit 47
2.21.3 Return Jump Stack Facilities A and B 51
2.21.4 The Save Address Register 53
2.21.5 The External Register 53

2.21.6 The Force 0 Address Capability 54
2.21.7 The Microinstruction Address Bus 54
2.21.8 Control Store Loading 54
2.22 The Conditions, Condition Selector, and Condition Registers 57
2.22.1 Short and Long Cycle 58
2.23 Auxiliary Control Facilities 60
2.23.1 Counter B 60
2.23.2 The Snooper Facility 60
2.24 An Alternative View of the Working Registers 61
2.25 An Alternative View of the Postshift Masks 62
2.26 Wide Store Address 63

3.0 MICROINSTRUCTION SPECIFICATION AND EXECUTION

3.1 Microinstruction Format 64
3.2 Microinstruction Execution 67
3.2.1 Clock Pulse 1 and Clock Pulse 2 69
3.3 Comprehensive Tables of Microoperations for Individual Functional Units 70
3.4 Assembler Notation 83

4.0 INDEX TO FIGURES, TABLES AND SYMBOLS

4.1 List of Figures 84
4.2 List of Tables 86
4.3 Table of First Occurrence or Definitions of Selected Abbreviations and Symbols 88

References 91

Foreword

It is the purpose of this document to give an introductory (yet reasonably detailed) description of the MATHILDA Processor. The main data path, the registers and functional units attached to it, and the control which can be exercised on these resources are discussed. The document is not a reference manual. Rather, it is written entirely from the pedagogical point of view, with the system described in a modular fashion. Examples are introduced after each component is added to the basic data path. The examples are written in the MARIA microassembler language [3]. The examples are deliberately kept simple so the reader will not spend time learning a complicated or clever algorithm but will learn the control mechanisms of the particular resources involved. Thus, many of the examples are "contrived" and do not perform any particular "useful" data transformations. It is hoped that this approach enhances the reader's understanding and underlines the overall simplicity and homogeneity of the structure and its components. This document is a fully revised version of an earlier report entitled, "A Description of the MATHILDA System", by B.D. Shriver, DAIMI PB-13.

September 1975

In this second printing a number of misprints in the text and in the examples have been corrected. The notation of a few microoperations has been changed. The editing was done by Jens Kristian Kjærgaard and Flemming Wibroe.

July 1980

A Description of the MATHILDA Processor

1.0 Introduction
MATHILDA is a dynamically microprogrammable processor which has been designed to be used as a tool in interpreter-oriented and processor design research. For the sake of completeness we will discuss briefly a short history of the unit and then some of the criteria which served as a basis for its design.

1.1 Historical Notes
In the spring of 1971 the Department of Computer Science of the University of Aarhus was considering the purchase of a standard minicomputer to act as a controller for a variety of peripherals and to simulate a medium speed batch terminal to the Computer Center's large system. A group of people were, at this time, working on the design of an integrated software and hardware description language called BPL [4]. To support this group and to make the use of such a minicomputer more flexible, it was decided to design and construct a microprogrammable minicomputer within the department itself.

The design was started and completed during the summer of 1971. The resulting machine, RIKKE-0 [5], was constructed and began running in early 1972. In the meantime a number of projects were proposed which were considered not to be compatible with that design. Among these were various projects in numerical analysis [7, 8] in which it was found that the word size and bus width of the RIKKE-0 (16-bit) was too short to obtain an efficient implementation of even standard arithmetic operations on numbers. It was then suggested that a microprogrammed functional unit with a wider data path and special features could be attached to RIKKE-0 as an I/O device, or "functional unit", together with a wider memory, for use with these projects. A proposal was made to the Danish Research Council to obtain a grant to design and construct such a functional unit. A grant was made in June, 1972 in which funds were awarded for hardware and a memory (32 K, 64-bit wide, 1.4 microsecond cycle time). The manpower for the construction of the unit was, in part, granted by the Research Council; two staff engineers and one staff technician were provided by the department.

The motivation for building the MATHILDA instead of purchasing a commercially

available machine can be summarized as follows. First, at the time we started our efforts, there were (to the authors' knowledge) no commercially available dynamically microprogrammable processors efforts which: (a) were in the price range we could afford, (b) were designed for or supported user written microcode or (c) offered a reasonable experimental and growth oriented structure. We felt that we had the in-house capability to design and construct the machine. The availability of LSI circuits and convenient mounting techniques and our experience with RIKKE-0 supported this view.

1.2 General Design Criteria and Constraints

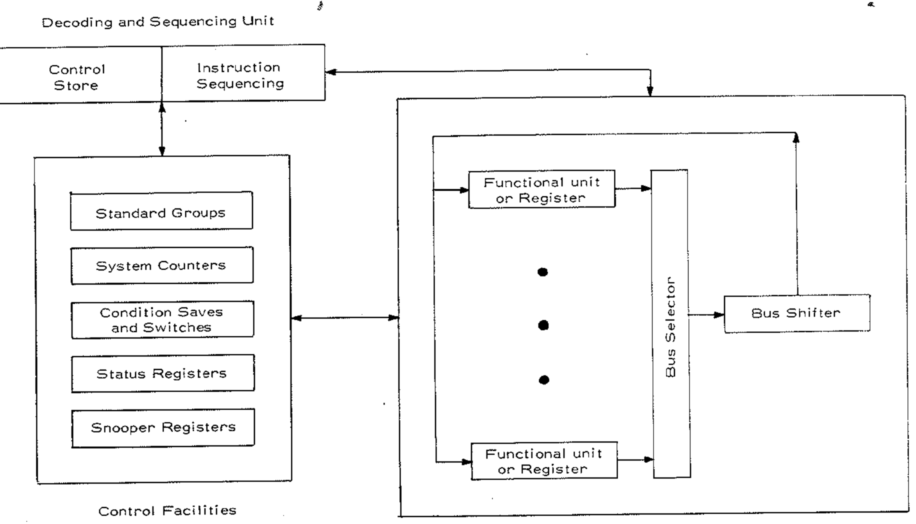
The MATHILDA machine is intended to be a research oriented machine. Its main design criterion then, within the money and timing constraints on the project, was to provide a machine on which a large variety of experiments related to processor and interpreter design and evaluation could be performed. We attempted to use the "top-down" design approach which quite frequently was tempered by the "forces from below", see Rosin [6]. We, therefore, tried to have various software and application-oriented ideas reflected in the design.

Two general software concepts had a reasonable impact on design. The one being the ability to multiprogram virtual machines and the other being the concept that virtual machines would be defined through several layers (e.g., R. Dorin [1]). The effect of these ideas is apparent in the design of the control unit, especially with respect to the capabilities of addressing. Many addressing features known on the virtual level are present here on the micro level.

Another criterion was to have a clean and consistent way of dealing with timing problems. We decided not to force the speed; rather we would have a slower machine than obtainable with the componentry at hand, and thus one, hopefully, with a reduced set of timing idiosynchrasies. It was also decided to be able to control all elements of the system from an immediate control or a residual control capability, or some combination of both. The residual control was made homogeneous to the user by having a reasonably "standard control register group" where ever such a control was provided.

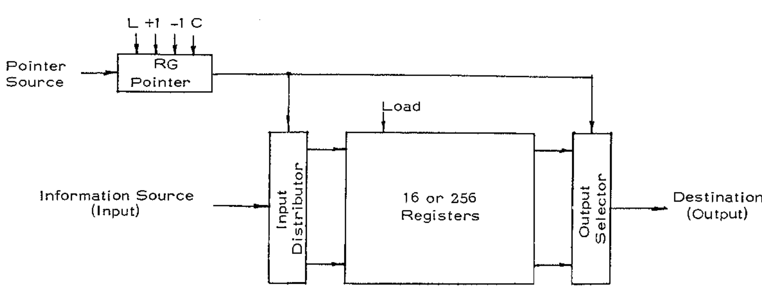
Another design criterion dealt with the actual construction of the unit. It had been decided, prior to the obtaining of the grant from the Danish Research Council, to construct additional RIKKE's by other funding. It became apparent, during the design phase of MATHILDA, that the machine would be reasonably complex and

that several features of MATHILDA included or extended similar features on RIKKE-0. Because of the complexity of the design, the limited funds and manpower available, and the fact that we wished to design, construct, and test the machine within one year, it was decided that the additional RIKKE's (now called RIKKE-1's [10] should be modeled after the MATHILDA system. Thus, one design criterion was to ensure a modularity in the hardware design. This would enable an economy in print-lay out and construction to be achieved. As an example, the main data path is laid out on one print board, 8-bits wide. Two of these boards interconnected, comprise one RIKKE-1 main data path with all registers, shifters, etc. Four of these RIKKE-1 boards interconnected, give the MATHILDA main data path.



The MATHILDA Processor

Figure 2. 1



Typical Register Group

Figure 2. 2

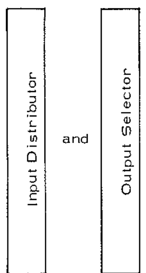
2.0 The MATHILDA Processor

MATHILDA, as has been stated earlier, has a microprogrammed controlled bus structure. The major elements of the system are shown in Figure 2.1 and are the: main data path , 2) decoding and sequencing unit, and 3) control facilities. In the following sections we will describe each of these elements independently and give examples of their utilization.

2.1 The Register Group and Standard Group

We begin by introducing a fundamental building block which is used in the various control mechanisms of the system, viz, a Register Group, RG, as shown in Figure 2.2. (After a particular system element is first introduced, an abbreviation for its name is given which, for the sake of brevity, is then used in the text; see the "Tables of First Occurrence of Abbreviations and Symbols", Section 4.0.) A RG is a set of 16 or 256 registers. The width of the registers and the number of registers in a specific RG will be stated when it is introduced. The element of a particular RG, which is to be used as a source or destination for the transfer of data, is pointed to by the RG address register. This register is called the Register Group Pointer, RGP, as shown in Figure 2.2.

The fact that the RGP is used to access one of the registers in the RG is indicated in Figure 2.2 by the output of the RGP going to the boxes.

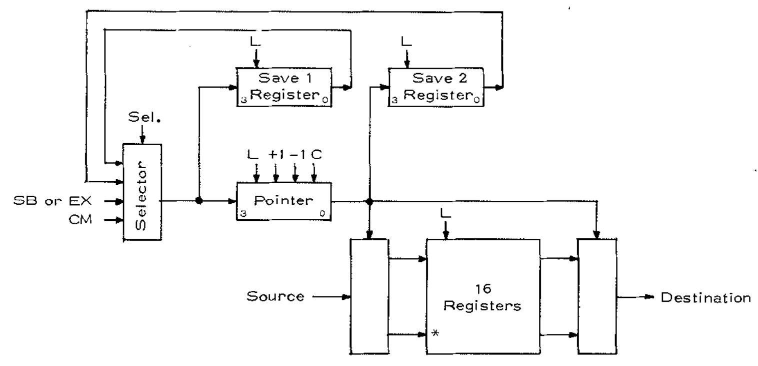


These are logical representations of the electronics which actually acccess the specified register. The phrases "Input Distributor" and "Output Selector" *will not* be included in any of the drawings which follow.

	Symbolic Notation	Microoperation
L	RGP := Pointer Source	Load the RGP from the Pointer Source
+1	RGP + 1	Increment RGP by 1
-1	RGP - 1	Decrement RGP by 1
C	RGPC	Clear (i.e., set to zero) RGP

Microoperations for the control of an RG

Table 2. 1



* The width of the registers depends on the particular selector involved.

Typical Standard Group

Figure 2. 3

There are four microoperations associated with an RGP. They are marked L, +1, -1, and C in Figure 2.2 and all subsequent figures and are explained in Table 2.1.

The symbolic notation RGP+1, RGP-1, etc. which is used in the microassembler, is used also in all of our examples. The abbreviation "RG" will be replaced by the abbreviation of the name of the functional unit with which that particular RG is associated. Most of the RGP's will have the microoperation

RGP := Pointer Source

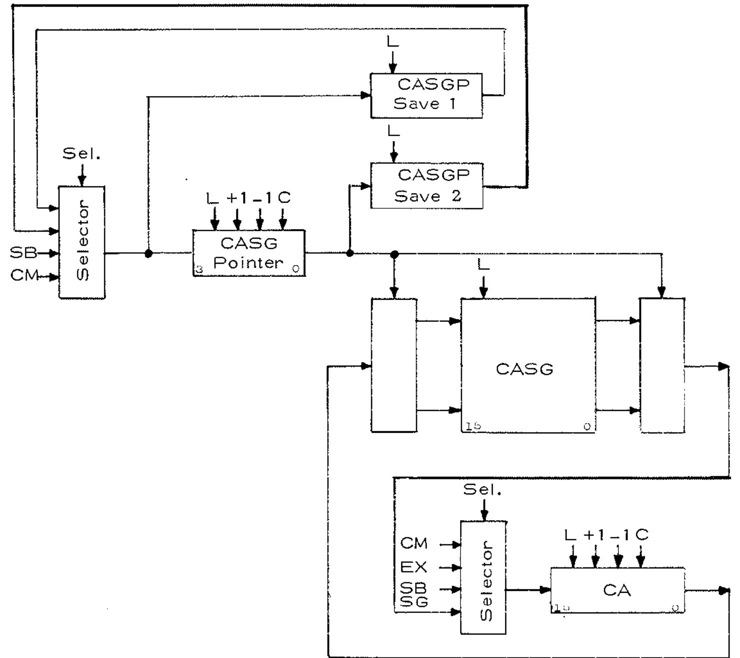
associated with them. The Pointer Source data itself can usually be selected to come from any of four different sources. There is one additional microoperation required for the control of an RG, namely the function labelled "Load" in Figure 2.2. If the loading of an RG can be initiated by a microoperation it will be indicated by an "L" on such a diagram. The *output of* a register group will be symbolically represented as RG, i.e., the name of the register group will be taken to be the value presented at its output. More completely, however, we could have written RG[RGP], which specifies the contents of the RG pointed to by RGP. The "[ ]", then, are used to represent the indexing operation. If a particular element of an RG is to be specified, we will write RG[j], $0 \leq j \leq 15$, e.g. RG[4].

A 16 element RG with the two Save registers and Pointer as shown in Figure 2.3 is a fundamental control element in the system and will be used with many devices in the subsequent sections. It will be referred to as a Standard Group, SG, and will be noted on drawings as such. It will not be explicitly drawn each time. Each SG will, however, be given a name closely associated with the particular functional unit to which it is connected.

Note, there are 2 new microoperations which are introduced to utilize these new facilities, the Save 1 register and the Save 2 register load:

S1 := CM|EX|S1|S2
S2 := RGP.

The machine timing is such that it is possible to "save" the contents of the RGP in its S2 register and then load the RGP from CM|EX|S1 in one microoperation. Use of this facility will be seen in later sections. Note that the RGP and S1 can be loaded from any one of four different sources. Which particular source is being selected is determined by the value of the "Sel." lines shown in Figure 2.3. The value associated



Counter A, CA

Figure 2.4

Symbolic Notation		Microoperation
L	CA:=CM|EX|SB|SG	Load CA from either CM, EX, SB, or CASG. Note the use of "|" to mean "or" in the symbolic notation for this microoperation.
+1	CA + 1	Increment CA by 1
-1	CA - 1	Decrement CA by 1
C	CAC	Clear (i.e., set to zero) CA

Table 2.2

Microoperations for control of CA

with the "Sel." lines is specified by a data field within the microinstruction executing the Load microoperation. This will be explained in Section 3.0.

2.2 Counter A

We will, from time to time, give small segments of microcode to illustrate the use of a device and its control. In order to make these examples clearer, and also to give a more realistic view of how such a code is actually written, we introduce the system counter, Counter A, CA. CA is a 16-bit wide counter as shown in Figure 2.4. CA has four microoperations associated with it as shown in the box labelled "CA" in this figure. These microoperations are given in Table 2.2.

Both the box labelled "Selector" in Figure 2.4 and the explanation of the microoperation "L" in Table 2.2 state that CA can be loaded from one of four possible sources:

1) immediate data within the Current Microinstruction, CM,
2) a 16-bit External Register, EX (discussed in Section 2.20.5),
3) bits 0 through 15 of the Shifted Bus (discussed in Section 2.5), SB(15:0). Here, "(i:j)" denotes the contiguous string of bits labelled i, i-1,...,j+1, j,i≥j. If i=j, "(i:j)" is written "(i)", and
4) from a 16-bit wide, 16 element SG called the Counter A Standard Group, CASG.

Thus the microoperation

CA := 37

loads CA with the constant 37 from a data field within the CM. While the microoperation

CA := SG

loads CA with the contents of the element of CASG which is pointed to by the CASG Pointer, CASGP. Notice that the CASG can be loaded with the contents of CA thus allowing one to save the current value of CA. The microoperations associated with the CASG, CASGP, and the Save registers are in Table 2.3.A.

However, the symbolic notation associated with the CASGP microoperations will

Preliminary (extended) notation

Symbolic Notation		Microoperation
L	CASG:=CA	Load the element of CASG pointed to by CASGP with CA
+1	CASGP + 1	Increment the CASGP by 1
-1	CASGP - 1	Decrement the CASGP by 1
C	CASGPC	Clear (i.e., set to zero) CASGP
L	CASGP:=CM|SB|S1|S2	Load the CASGP from CM, SB(3:0) CASGPS1, or CASGPS2. The notation SB(3:0) specifies the least significant (rightmost) bits of the Shifted Bus
L	CASGPS1:=CM|SB|S1|S2	Load the CASGPS1 from CM, SB(3:0), CASGPS1, or CASGPS2
L	CASGPS2:=CASGP	Load the CASGPS2 from the CASGP

Table 2.3A
Microoperations for control of CASG and CASGP

Symbolic Notation		Microoperation
L	CASG:=CA	Load the element of CASG pointed to by CAP with CA
+1	CAP + 1	Increment the CAP by 1
-1	CAP - 1	Decrement the CAP by 1
C	CAPC	Clear (i.e., set to zero) CASGP
L	CAP:=CM|SB|S1|S2	Load the CAP from CM, SB(3:0) CAPS1 or CAPS2. The notation SB(3:0) specifies the least significant (rightmost) bits of the Shifted Bus
L	CAPS1:=CM|SB|S1|S2	Load the CAPS1 from CM, SB(3:0), CAPS1, or CAPS2
L	CAPS2:=CAP	Load the CAPS2 from the CAP

Table 2.3B
Microoperations for Control of CASG and CAP

not be written as shown in Table 2.3.A. The notation introduced therein was given so that the reader could establish a firm understanding of the operation of CA and its associated control. Since the "pointer" microoperations are concerned only with the CASG and its associated control, the SG notation will be dropped in the symbolic notation for these particular microoperations as well as in the text. This causes no ambiguities since the CA itself does not have a pointer associated with it, but only with its SG. We will use this abbreviated notation throughout this report, and Table 2.3.B has been given so that the reader can compare both notations. Duplicate tables will not, however, be given for other resources.

We see in Figure 2.4 that the data which can be loaded into the CAP can also be loaded into an additional register called the CAPSave-1 register, CAPS1. If, for example, we know in advance the address of a particular register of the CASG, which we will want to use as counter data (e.g., some highly used constant), we can store this pointer in CAPS1 by loading CAPS1 from the CM,

CAPS1 := <constant>.

Here the "<>" are meta brackets enclosing meta symbols. Whenever we wish to use this stored pointer we can load it into the CAP by executing

CAP := CAPS1

The CAP not only points to the element of the CASG which can be chosen as data input to CA, but also can be stored in a register called the CAPSave2, CAPS2. Suppose we are pointing to a particular element of the CASG and in the next microinstruction we wish to have register 9 of the CAS to be used as counter data, *but* we do not wish to loose the pointer to our current data. The following microinstruction achieves this,

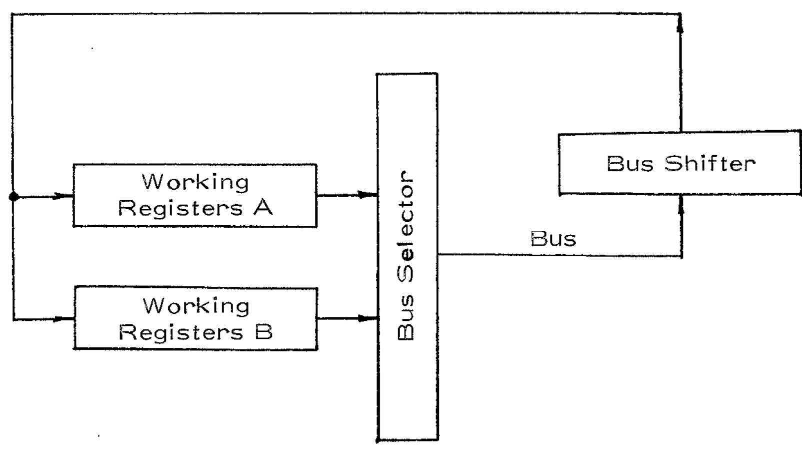
CAPS2 := CAP, CAP := 9

Thus at some later time if we execute

CAP := CAPS2

the pointer information which had been saved in CAPS2 would be restored.

We can test to see if CA contains zero. We will demonstrate the use of this condition and the microoperations in Tables 2.2 and 2.3B in subsequent examples.



Sub-system of the MDP

Figure 2. 5

2.3 Main Data Path Transport

Having introduced some elementary notions we will now examine in some detail the Main Data Path, MDP, the registers and functional units attached to it, and the control which can be exercised on these components. We will construct the MDP in a modular fashion - hopefully to enhance the reader's understanding and to underscore the overall simplicity and homogeneity of the structure and its components.

Let us introduce the concept of a MDP transport by considering a sub-system of the MDP consisting of the Working Registers A, WA, Working Registers B, WB, and the Bus Shifter, BS, as shown in Figure 2.5. The exact nature of WA, WB, and BS is not important to us here.

The BUS is a 64-bit wide data path. The input to the BUS (its SOURCE) is obtained from a bus selector which has eight inputs, two of which are shown here, i.e., WA and WB. The particular input which is selected as the SOURCE for MDP transport may be shifted a specified amount in the BS. The output of the BS, called the Shifted Bus, BS, can then be stored in one of seven possible 64-bit destinations (called Shifted Bus Destinations, SBD). Two such SBD's are shown in Figure 2.5., i.e., WA and WB. We will in this report specify bus transport information as we do in our microassembler, viz,

<DESTINATION> :=<SOURCE>,<BS Specification>

If the BS Specification field is empty, i.e., the BS is not to be used (no shift occurs) then the bus transport is given by

<DESTINATION> := <SOURCE>.

As an example, the MDP transport WB := WA has the obvious meaning of a register to register transfer from WA to WB. If a SOURCE is chosen to be transported but not stored in any of the SBD's, the bus transport information is written

<SOURCE>, <BS Specification>

or

<SOURCE>

as is appropriate. The SOURCE may be stored in destinations, D, other than SBD's during a MDP transport. We will learn what functional units or registers can serve as these "other destinations" as this report develops. If the SOURCE is to be stored in more than one destination, the DESTINATION portion of the MDP transport specification is written as a list of destinations separated by commas, i.e.,

< LIST > := <SOURCE> , < BS Specification>

or

< LIST > := <SOURCE>

where

<LIST>:: = SBD{,D}^{n} | D{,D}^{n}

where the " { } " are meta brackets used for grouping symbols.

The value of n and the units which can serve as destinations will be discussed later.

2.4 Working Registers

WA and WB, introduced in the previous section, are not single registers but each is a 64-bit wide, 256 element RG. Figure 2.6 shows WA; WB, not shown, is identical.

The first thing we wish to point out in this figure is that the WA Pointer, WAP, is a mechanism identical to CA except that it is 8-bits wide and not 16-bits wide. (Note thedashed-line box in Figure 2.6.) Therefore, WAP not only points to which element of WA can be used as a SOURCE for bus transport but also can be stored in a RG called the WAP Save registers, WAPS. This is identical to CA being saved. Also, as indicated in the box labelled "Selector" in Figure 2.6 the WAP can be loaded from any of four sources:

1) immediate data from the CM,
2) the least significant 8-bits from EX,
3) the least significant 8-bits of the SB, and
4) an element of WAPS.

This is identical to the loading of CA. Thus the microoperations WAP := 37 and WAP := WAPS have well defined analogous meanings.

Working Registers, A, WA

Figure 2.6

WAP := CM|EX|SB|SG	WBP := CM|EX|SB|SG
WAP + 1	WBP + 1
WAP -1	WBP - 1
WAPC	WBPC
WAPS := WAP	WBPS := WBP
WAPSP + 1	WBPSP + 1
WAPSP -1	WBPSP - 1
WAPSPC	WBPSPC
WAPSP:=CM|SB|S1|S2	WBPSP:=CM|SB|S1|S2
WAPSPS1:=CM|SB|S1|S2	WBPSPS1:=CM|SB|S1|S2
WAPSPS2:=WAPSB	WBPSPS2:=WBPS2

Table 2.4
Microoperations for control of WA and WB

The WA (and WB) registers are not loaded by a microoperation, but rather as a result of being chosen as a SBD in a bus transport specification; thus the loading of these registers is shown by the function "SBD Load" on Figure 2.6. This notation will be used in all subsequent drawings. The microoperations associated with the WA and WB are given in Table 2.4. The actual microoperation descriptions can be extracted from the previous tables and are not repeated here.

2.4.1 Microinstruction Format and a Few Examples

In order to present a few examples we will introduce the microinstruction format which we use in the microassembler. The format of a microinstruction is:

<LABEL> : <MDP transport>; <microoperations and data>; <instruction sequencing>. <comment>

where

a) <LABEL> is a symbolic name for the control store address of the microinstruction,
b) <MDP transport> is a field giving the MDP transport information as explained previously in Section 2.3,
c) <microoperations and data> is a field of up to 7 microoperations and immediate data to be executed or used during this microinstruction (the exact combination of microinstructions and data which can be included in this field and precise details of the timing of microoperations are given in Section 3.0),
d) <instruction sequencing> information will be written in the form

$$\text{if } c \text{ then } A_t \text{ else } A_f$$

which is to mean: if a particular condition is true then choose address A_t as the address of the next microinstruction else choose A_f.

e) <comment>: each instruction may be terminated with a "." after which comments may be inserted.

It is not necessary or appropriate at this point to list all of the conditions which are testable by the system nor how A_t and A_f are functions of the address of the current microinstruction, n. These matters will be dealt with in Section 2.20. However, conditions and address functions will be introduced as needed for

examples. If no condition is to be considered, i.e., if $A_{\mathfrak{t}} = A_{\mathfrak{f}}$, the sequencing information will merely be written $A_{\mathfrak{t}}$ (and not "if c then $A_{\mathfrak{t}}$ else $A_{\mathfrak{t}}$" where c is an arbitrary condition).

One microinstruction execution of the machine may be considered as consisting of four major sequentially executed steps:

A: Microinstruction fetch
B: MDP transport
C: Execute microoperations
D: Execute instruction sequencing.

Steps B, C, and D are controlled by fields within the microinstruction. Logically, to the user, these steps and substeps within these may be considered sequentially within the microinstruction execution, although of course many of the activities take place in parallel in the physical implementation.

Thus, the microinstruction located at Control Store location n,

WA := WB; WBP+1; n+1.

means: load the element of WA pointed to by WAP from the element of WB which is pointed to by WBP without shifting it during the bus transport; then increment WBP by 1; then obtain the next microinstruction from location n+1. The action associated with *every* microoperation specified in a microinstruction is completed *before* the next microinstruction is executed. For example, in the above microinstruction if WBP had been set to 9 before the beginning of the execution of this instruction, then WB[9] would be the SOURCE for the bus transport. At the end of execution of the instruction, the WBP would be set to 10. If, in the next microinstruction WB were again selected as the SOURCE, then the contents of WB[10] would be gated onto the BUS.

In order to give an example of a microinstruction using conditional branching, we establish the following convention for the testing of conditions which will be used in all of our examples (unless stated explicitly otherwise): *all* conditions which arise asa result of MDP transport and microoperation execution specified by a particular microinstruction, M, are testable in the *next* microinstruction to be executed after M is executed. This means that all the conditions available or changed during the execution of microinstruction M are "saved". These "saved" conditions are those

tested in the next instruction to be executed. Therefore, our microinstruction can be thought of being executed in the following sequential way:

A1: Save the conditions of the previous microinstruction
A2: Microinstruction fetch
B: MDP transport
C: Execute microoperations
D: Execute microinstruction sequencing based on saved conditions.

Let us introduce the notion that WA(63) is testable. (WA(63) is, in conjunction with the notation introduced in Sections 2.1 and 2.2, synonymous with WA[WAP](63:63).) If we wish, for example, to test WA[7](63), and if it is set to 1, jump to the microinstruction labelled BITON, else continue with the next microinstruction, we could write,

```
; WAP := 7.
; if WA(63) then BITON.
```

We could not write

```
; WAP := 7; if WA(63) then BITON.
```

according to our current convention.

Let us give an example which shows that it is possible to execute the same instruction conditionally. Assume there is at least one register of WA which contains bit 63 set to 1. The following microinstructions will: search WA, starting with register 0 and transfer the first register of WA encountered with bit 63 set to 1 to register WB[0]; then, store the address of the WA register which was transferred in register WAPS[0]; and then continue with the next microinstruction.

```
                ; WAPC, WAPSPC, WBPC.
LOOP:           ; WAP+1,                      ; if ¬ WA(63) then LOOP.
                ; WAP-1.
          WB:=WA ; WAPS:=WAP. |||||
```

We have introduced some standard defaults in these examples:

a) If the bus transport field is empty it means that an unspecified source is selected for bus transport but is not stored anywhere.
b) If the microoperations field is empty it means that no microoperations are to be executed during this particular microinstruction.
c) An empty <instruction sequencing> field or <else part> implies the next microinstruction to be executed is that in n+1 (if the address of the current microinstruction is n). If the microinstruction sequencing field is empty the specification ";<instruction sequencing>." is replaced by ".".
d) The instruction sequence shown is assumed to be located sequentially in control store and the address name is used only when needed in the microinstruction sequencing field.
e) The symbol ||||| will be used to indicate the end of the group of microinstructions in an example.

The notation HERE-1, HERE, and HERE+1 are used often in the microinstruction sequencing field to mean A-1, A, and A+1 assuming the address of the current microinstruction is A. As an example the instruction labelled LOOP above could have been written

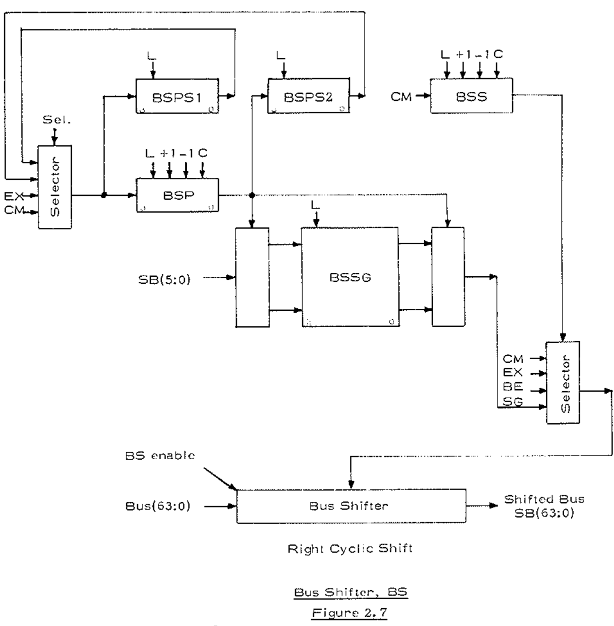
```
            ;WAP + 1                ; if ¬WA(63) then HERE.|||||
```

Through the use of CA the assumption that at least one register of WA contains bit 63 set to 1 is not required. CA can be used to control the number of elements of WA we will search. If we establish a routine labelled NONE which handles the situation when no element of WA contains bit 63 set to 1, then the code to perform the same task as related above is,

```
            ;WAPC,WAPSPC,WBPC.
            ;CA: = 255              ;LOOP.
            ;WAP + 1,CA-1           ;if CA then NONE.
LOOP:       ;                       ;if ¬ WA(63) then HERE-1.
     WB: = WA;WAPS: = WAP.|||||
```

The final example in this section uses the capability of loading CA from the SB. In the previous example CA was loaded with N-1 where N ($2 \leq N \leq 256$) is the number of registers of WA to be searched. Let us suppose that this number is in WB[0] and furthermore that we wish to save it in register CASG[0] because it may be overwritten if a transfer is made to WB. A possible code segment is,

```
                ; WAPC, WAPSPC, WBPC.
       WB       ; CAPC, CA:=SB.
                ; CASG:=CA                  ; LOOP.
                ; WAP+1                     ;if CA then NONE.
LOOP:           ; CA-1                      ;if WA(63) then HERE+1 else HERE-1
       WB:=WA   ; WAPS:=WAP.|||||
```



Bus Shifter, BS
Figure 2.7

BSSG:=SB
BSP:=CM|EX|S1|S2
BSP + 1
BSP – 1
BSPC
BSPS1:=CM|EX|S1|S2
BSPS2:=BSP
BSS:='CM'|'EX'|'BE'|'SG'|
BSSC

Table 2. 5
Microoperations for control of the BS

2.5 The Bus Shifter

The Bus Shifter, BS, introduced in Figure 2.5 and shown in more detail in Figure 2.7, is a 64-bit wide right cyclic shifter which can be set to shift n bit positions, $0 \leq n \leq 63$. There exists a dedicated bit in each microinstruction to control the BS indicating whether or not the BS should be used (enabled) during the current bus transport. If the BS is not enabled, no shift will occur.

If we wish to use the BS, the amount of shift can be selected from one of four possible sources as shown in Figure 2.7, i.e., from

1) a data field in the CM,
2) the least significant 6 bits of the EX register,
3) the output of the Bit Encoder, BE, (discussed in Section 2.16), and
4) an element of a 6-bit wide 16 element RG called the BSSG.

Which of these sources is to be used is determined by the contents of the Bus Shifter Selector, BSS. As shown in Figure 2.7, the BSS can be loaded from a data field in the CM. Once the BSS has been loaded the BS will be controlled as described above. The loading of the BSS will be symbolically represented by

BSS := 'CM'|'EX'|'BE'|'SG'.

Thus if BSS := 'SG' is executed, this means that the BSS will be loaded with a value which will select the BS shifter control to take its control data from the BSSG. The BSSG will be the BS control source until a different BSS := 'CM'|'EX'|'BE'|'SG' microoperation is executed.

Note that the quotes surrounding resource names here and in Table 2.5 are meant to indicate that *a specific encoding* associated with each particular resource name is loaded into the BSS register. This is to be contrasted with the semantics of previous assignment type microoperations such as

CA := SG

which means that CA is loaded with *the contents of* the CASG register pointed to by the CAP. The use of '<resource name>' throughout this report in conjunction with assignment type of microoperations will mean that a specific set of encodings has been bound to particular resource names at machine fabrication time and this binding is reflected in the microassembler.

In the following we will assume as a programming convention that the BSS contains the value 'CM', i.e., shiftinformation will be taken as data from the microinstruction. Whenever the user establishes another data source, he is assumed to reestablish the standard situation after the usage. The BSS register is actually a 2-bit counter, the value 0 corresponds to the encoding for 'CM'. This implies that the reestablishing of the standard situation can also be accomplished by the microoperation BSSC.

The MDP transport specification

WA := WB

means: take the element WB[WBP] and store it in the element WA[WAP] without shifting it. While the bus transport specification

WA := WB, → 3

means: take the element WB[WBP], shift it 3 bits right cyclic and then store it in the element WA[WAP].

A 64-bit left cyclic shifter and a 64-bit right cyclic shifter are related by the expression

lcs = 64 - rcs

where

lcs is the amount of left cyclic shift and
rcs is the amount of right cyclic shift.

We can therefore write as a notational convenience

WB := WA, ← 24

to mean the same thing as

WB := WA, → 40

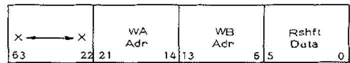
thus using ← (left shift) or → (right shift), whichever makes the understanding of the processing clearer. The microassembler will make the necessary computation.

The <BS specification> in the MDP transport field of the microinstruction is given by

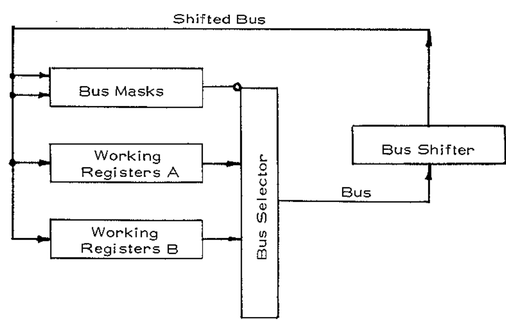
← | → | ← <constant> | → <constant>

Table 2.5 lists the microoperations associated with the BS in their symbolic form; their meanings should be obvious from previous tables and the text. Note that the BSSG is loaded with the least significant 6 bits of the SB, i.e., SB(5:0).

Example: Let us assume the following information to be in the register of WB to which we are currently pointing:

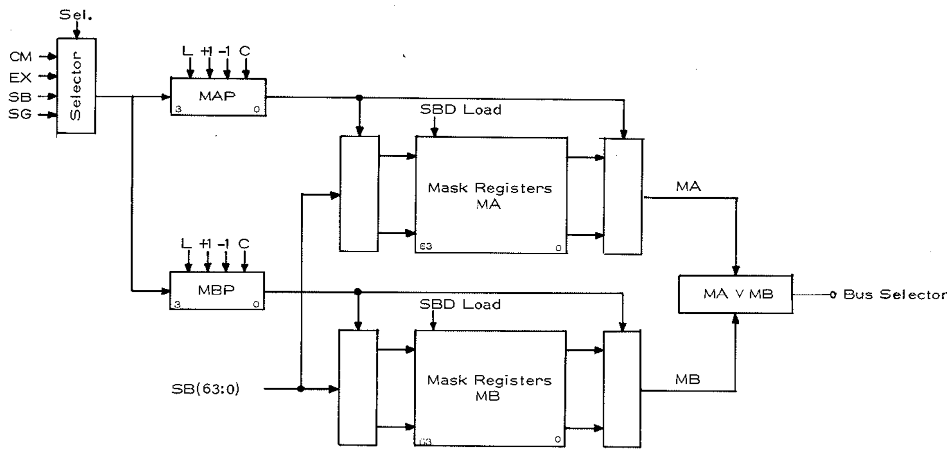


We wish to take a given WB[WB Adr], shift it a given amount (Rshft Data), and store it in a given WA[WA Adr]. The following code will: load the BSSG with the Rshft Data, save the current WBP, load WBP with the WB Adr, load WAP with the WA Adr, transfer the WB[WB Adr] to WA[WA Adr] shifting it right cyclic by the amount Rshft Data during transport, restore the old WBP, and then continue.



MDP Sub-system of Figure 2.4 expanded with BM

Figure 2.8



Bus Masks, MA and MB

Figure 2.9

```
WB, →14      ;WAP: = SB.
WB           ;BSSG: = SB,WBPS: = WBP.
WB, →6       ;WBP: = SB.
             ;BSS: = 'SG'.
WA: = WB, →;WBP: = SG.
             ;BSSC.|||||
```

In accordance with the programming convention expressed earlier in this section, the BSS has been set back to CM control by use of the BSSC microoperation.

2.6 Bus Masks

Let us now expand the initial MDP structure given in Figure 2.5 by adding the Bus Masks, BM, as shown in Figure 2.8.

The BM allows one to specify which bits of the SOURCE (i.e., the particular input to the bus selector which has been selected for bus transport) are actually to be transported. A mask is a string of 64-bits. If bit i ($63 \geq i \geq 0$) of a mask is a 1, then bit i of the SOURCE is to be transmitted; if bit i of the mask is a 0, then the value 0 is to be transmitted. Since the BM is not an input to the bus selector, but effects the transmission of the SOURCE, they are shown connected to the bus selector with the symbol —o (which we will interpret to mean "mask") and not by the symbol —► (which means "input").

The SOURCE is masked during *every* MDP transport by the mask which is specified to be

$$MA \vee MB$$

where

MA = an element of a 64-bit wide, 16 element RG called the Mask A registers,
MB = an element of a 64-bit wide, 16 element RG called the Mask B registers,
$\vee$ = logical "inclusive or".

MA and MB are shown in Figure 2.9. Again, by convention, the system is such that the "no mask", i.e., 64 1's, is in MA[0] and the "bus clear mask", i.e., 64 0's, is in MA[1]. We will assume this to be the case throughout all of the examples. One can

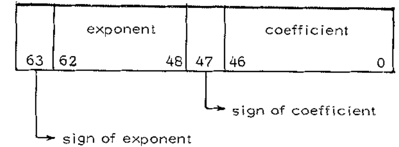
MAP + 1	MBP + 1
MAP − 1	MBP − 1
MAPC	MBPC
MAP:=CM|EX|SB|SG	MBP:=CM|EX|SB|SG
BMSG:=SB	
BMP := CM|EX|S1|S2	
BMP + 1	
BMP − 1	
BMPC	
BMPS 1:= CM|EX|S1|S2	
BMPS2:=BMP	

Table 2.6

Microoperations for control of the BM

then look upon the pointer MAP as a switch for the use of the bus mask: if MAP=0 then the BUS is not masked, if MAP=1 then the BUS is masked by the mask specified by MB. This is not, of course, the only interpretation of the use of the BM.

As an example, assume we are representing floating point numbers in the following sign magnitude format,



Suppose the following 4 masks are available in the first 4 registers of MB.

	63	62 ... 48	47	46 ... 0
MB[0]	1	0 ←→		→ 0
MB[1]	0	1 ←→ 1	0	←→ 0
MB[2]	0	←→ 0	1	0 ←→ 0
MB[3]	0	←→	0	1 ←→ 1

The following code will decompose a floating point number found in the register WA[WAP] and store the information as follows,

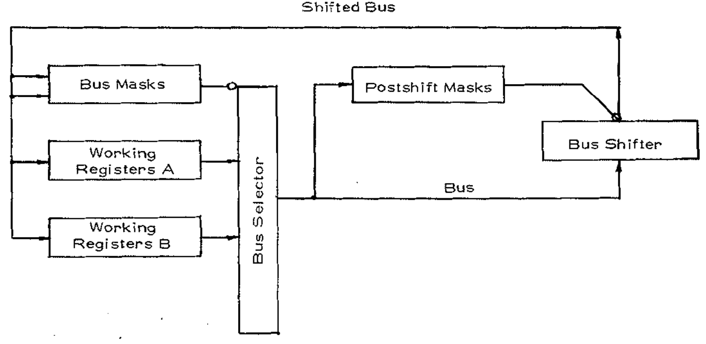
a) sign of the exponent in bit 63 of WB[0]
b) magnitude of the exponent shifted 1 in WB[1]
c) sign of coefficient in bit 63 of WB[2]
d) magnitude of the coefficient shifted 16 in WB[3].

```
                     ;MAP+1, MBPC, WBPC.
WB: = WA             ; MBP+1, WBP+1.
WB: = WA, ←1         ; MBP+1, WBP+1.
WB: = WA, ←16        ; MBP+1, WBP+1.
WB: = WA, ←17        ;MAPC|||||
```

It is suggested by this example that when formatted information is being decomposed

(e.g., a virtual machine instruction) one may wish to coordinate the use of the BS with the use of the BM. Let us therefore suppose the shift constants, 0, 63, 48, and 47 to be stored in the first 4 registers of the BSSG. The above decomposition and storage could be written as the following microoperations.

```
            ; CA:=3, MBPC.
            ; BSPC, WBPC, BSS:='SG'.
WB:=WA,← ; BSP+1, WBP+1, MBP+1, CA-1        ; if ¬CA then HERE.
            ; BSSC.|||||
```



MDP Sub-system of Figure 2.8 expanded with PM

Figure 2.10

Data to be used as a pointer to elements of either MA or MB can be stored in an SG and then loaded into either MAP or MBP when needed. Since this SG (shown as an input to the selector going to MAP and MBP on Figure 2.9) is used as a form of residual control for *both* MA and MB, it is called the BMSG (Bus Masks SG). The pointer which selects an element of the BMSG is called the BMP. Table 2.6 lists the microoperations associated with MA, MB, and BMP.

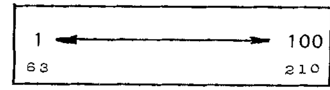
2.7 Postshift Masks

The Bus Masks, as described in the previous section, are applied to the SOURCE as it is gated onto the BUS and thus before the SOURCE is shifted in the BS. There is also a possibility of masking the SOURCE after it has been shifted by using the postshift masks, as shown in Figure 2.10.

One of the purposes of the postshift masks is to apply a mask to the output of the BS which will mask off the unwanted "cyclic" bits and replace them with 0's thereby realizing a logical shift. As an example, if the MDP transport

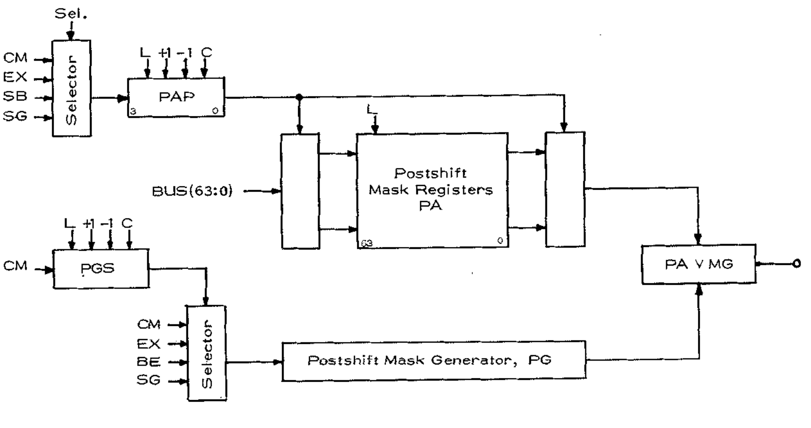
WB := WA, ← 2

is executed with the postshift mask



applied to the output of the BS, then we have taken a WA register, shifted it 2 bits left logical, and stored it in a WB register. Similarly, the MDP transport

WB := WA, → 6



Postshift Masks, PA and PG

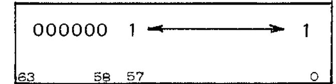
Figure 2.11

mask generator data n		PG
decimal	binary	
0	0000000	1 1
1	0000001	0 all 1's .
.	.	. .
.	.	. .
.	.	.
63	0111111	. all 0's 1
64	1000000	0 0
65	1000001	1 .
.	.	. .
.	.	. .
.	.	.
126	1111110	. all 1's .
127	1111111	1 . . . . . . 1 0

Table 2.7

PG Output

with the mask



applied to the output of the BS means a WA register is shifted 6 bits right logical and then stored in a WB register. The output of the BS is masked during *every* bus transport by the mask which is specified to be

PA $\vee$ PG

where,

PA = an element of a 64-bit wide, 16 element RG called the Postshift mask A register,
PG = a functional unit called the Postshift mask Generator,
$\vee$ = logical "inclusive or".

PA and PG are shown in Figure 2.11. This is quite similar to the BM where PG now takes the place of MB.

The PG is a functional unit which can generate a string of j 0's ($64 \geq j \geq 1$) starting from either the least significant bit (b_0) position or the most significant bit (b_{63}) position. The remaining k bits, j+k = 64, are set to 1. The PG can generate the 128 masks required to view the BS as both a logical and cyclic shifter. As is seen from Figure 2.11 the postshift mask generation data can come from one of four sources, CM|EX|BE|SG. Which particular source is to be used as data for the mask generation is determined by the contents of a 2-bit Postshift mask Generator Selection register, PGS, as shown in Figure 2.11.

Table 2.7 shows the relationship between the PG mask generation data as specified by the selected source (CM|EX|BE|SG) and the actual mask which will enter into the computation PA$\vee$PG.

If, in some previous microinstruction, the PGS has been set to point to the CM as the data source, then the PG data is specified in the "microoperations and data" field of the CM in the following symbolic way,

Operations associated with PA
PA := BUS
PAP := CM|EX|SB|SG
PAP + 1
PAP - 1
PAPC
Operations associated with PGS
PGS := 'CM'|'EX'|'BE'|'SG'|
PGS + 1
PGS - 1
PGSC (≡ PGS := 'CM')
Operations associated with PGSG
PGSG := SB
PGP := CM|EX|S1|S2
PGP + 1
PGP - 1
PGPC
PGPS1:= CM|EX|S1|S2
PGPS2:= PGP

Table 2.8

Microoperations for the control of the PM

PG "arrow" n

where,

n = the number of 0's to be generated and the "arrow" (←|→) indicates from which direction they should be generated; $0 \leq n \leq 64$.

When → is used, the value of n is inserted as the mask specification data directly. When ← is used, the microassembler will insert the value of 128-n as the mask specification data.

Thus the previous two examples could have been written (assuming PGS points to the CM as the data source)

WB:=WA,← 2; PG← 2 and
WB:=WA,→ 6; PG→ 6.

The programming convention is such that the mask of all 1's is in PA[0] and the mask of all 0's is in PA[1]. This is identical to the situation in MA. We will assume this to be the case throughout all of the examples. One can then look upon the pointer PAP as a switch for the use of the Postshift mask Generator: if PAP = 0 then the mask generator is not used, if PAP = 1 then the postshift mask which is to be applied will be that generated by the mask generator.

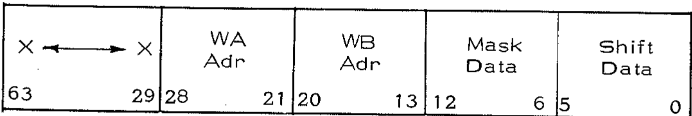
Table 2.8 provides a list of the microoperations associated with the postshift masks. The first half of this table deals with PA, the second half deals with the PG. The name of the SG associated with the PG control is the Postshift Mask Generator SG, PGSG. Note, the name of the SG associated with the PA pointer is the Postshift Mask SG, PMSG, and its pointer is called PMP. It is not discussed here but in Section 2.25.

When the EX is used as the *common source of control* for the BS and PG, the bits, EX(5:0), will specify the amount, n, of the right cyclic shift in the BS. The seventh bit, EX(6), will specify whether a logical right shift of n places or a logical left shift of 64-n places will be the result. A similar statement can be made for BE as will be seen in Section 2.16.

Arithmetical Logical Unit, AL

Figure 2.12

Let us extend the example of Section 2.5 in which we interpret a virtual machine instruction which performed a register to register transfer combined with shifting and masking. As shown below, if we use the PG we can execute an instruction which will take the contents of WB[WB Adr], shift it and mask it as described by the Shift & Mask Data, and then store it in WA[WA Adr]. If the data for the instruction is in the form



a possible code sequence could be,

```
WB, →21      ;WAP:=SB.
WB           ;BSSG:=SB, WBPS:=WBP.
WB, →6       ;PGSG:=SB.
WB, →13      ;WBP:=SB,PAP+1.
             ;PGS:='SG',BSS:='SG'.
WA:=WB, ←;WBP:='SG',PAPC.
             ;BSSC,PGSC.|||||
```

Note well, there are important assumptions in this example. The first is that PGS is assumed set to 'CM' upon entry to this code, i.e., the PG will be controlled by a data field in the CM, and the second is that PAP = 0 upon entry to this code, i.e., no postshift masking occurs. Indeed, we will make these assumptions in all examples which follow (unless stated explicitly otherwise). They can be summarized as follows: MDP transport normally occurs in an unmasked fashion; if a particular code segment requires the use of any masking facility, it is responsible for leaving the system in the state dictated by the established programming conventions.

2.8 The Arithmetical and Logical Unit

We will now add additional capability to the MDP in addition to the shifting and masking already encountered by introducing the Arithmetical and Logical unit AL. The AL, shown in figure 2.12, is a functional unit with 2 inputs which, for the moment we will call A and B.

AL FUNCTION F	
ARITHMETIC *	LOGICAL
A	¬A
A ∨ B	¬A ∧ ¬B
A ∨ ¬ B	¬A ∧ B
minus 1	all 0's
A + (A ∧ ¬B)	¬A ∨ ¬B
(A ∨ B)+(A ∧ ¬B)	¬B
A–B–1	A ≡ B
(A ∧ ¬B)–1	A ∧ ¬B
A + (A ∧ B)	¬A ∨ B
A + B	A ≢ B
A ∨ ¬B + (A ∧ B)	B
(A ∧ B)–1	A ∧ B
A + A	all 1's
(A ∨ B) + A	A ∨ ¬B
(A ∨ ¬B) + A	A ∨ B
A–1	A

*) in 2's complement; the arithmetic operations are shown with the carry-in set to 0. If the carry-in is to be 1, then the AL Function is specified by writing 'F+1' where F is the specified arithmetic function. The logical functions are not affected by the carry-in.

Table 2.9

AL Functions

ALF := CM|EX|SB|SG
SET ALF + (i, e., SET TO LR+AS)
SET ALF - (i. e., SET TO LR-AS)
SET ALF B (i. e., SET TO AS)
SET ALF A (i. e., SET TO LR)
SET ALF + 1 (i. e., SET TO LR+1)
SETALFALL0S
SETALFALL1S
ALP:=CM|EX|S1|S2
ALP + 1
ALP - 1
ALPC
ALPS1 := CM|EX|S1|S2
ALPS2 := ALP

Table 2. 10

Microoperations for control of the AL

6 bits are required to control the AL: 5 bits to select one of the 32 operations listed in Table 2.9 which this unit can execute on A and B and 1 bit which specifies the carry-in bit into the AL for any arithmetic operations.

The 6 control bits which specify the current operation for the AL are the contents of the AL Function specification register, ALF, which can be loaded, ALF := CM|EX|SB|SG, or set to particular arithmetic functions as shown. The SG associated with the ALF is called the AL Standard Group ALSG. The microoperations associated with the AL are given in Table 2.10.

If the ALF is to be loaded with an operation specification from the CM, we will note this symbolically merely by writing the required function in the symbolic form which appears in Table 2.9 in the ALF assignment statement, i.e.,

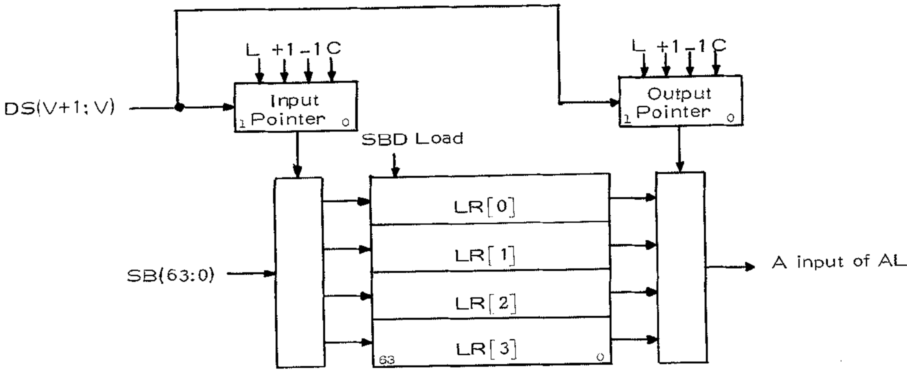
ALF := 'A+B'
ALF := 'A∧B'
etc.

A group of highly used function encodings can be directly loaded into the ALF by use of the specially provided SET microoperations. For example, ALF := 'A+B' can equivalently be specified by writing SET ALF+. The difference between these two microoperations is that ALF := 'A+B' uses a data field of the microinstruction to specify the encoding of A+B, whereas the SET ALF+ microoperation contains the encoding within itself and does not require the additional data field. The SET function potentially allows for more microoperations to be executed in parallel in a microinstruction as will be seen in Section 3.0.

The AL is always running. If the ALF is changed in a microinstruction, then the result of the newly computed function is available for MDP transport in the very next microoperation. Thus the microinstructions

```
         ; ALF:='all 1s', PAP+1.
WA:=AL   ; PG→ 48, PAPC.|||||
```

will put a string of 16 1's in WA[WAP]. The 1's will be least significant bit, b_0, justified.



Local Registers, LR

Figure 2. 13

LRIPC
LRIP + 1
LRIP - 1
LRIP := DS(V+1:V)
LROPC
LROP + 1
LROP - 1
LROP := DS(V+1:V)
LRPC
LRP + 1
LRP - 1
LRP := DS(V+1:V)

Table 2. 11

Microoperations for control of the LR

There are many testable conditions concerning the operation of the AL. A few of these are

Symbolic notation	*Condition*
AL	result of current AL computation is the bit pattern 11...11.
AL(0)	bit 0 of the result of the AL computation
AL(63)	bit 63 of the result of the AL computation
ALOV	AL overflow (equivalent to a carry-out during addition and a borrow-in during subtraction)

Before giving examples of the control of the AL let us first discuss the nature of its inputs, A and B.

2.9 The Local Registers

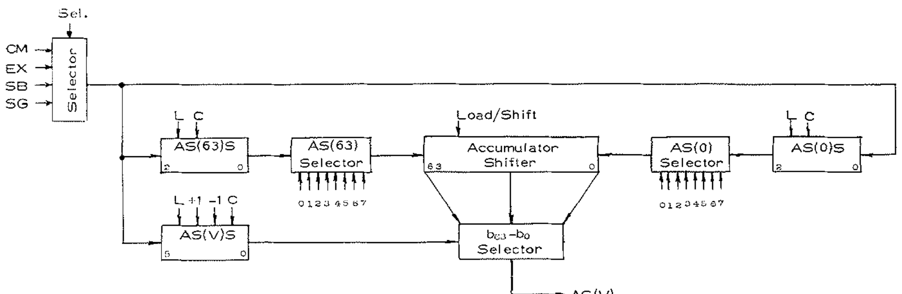
The Local Registers, LR, serve as the A input to the AL in the context of the AL Functions shown in Table 2.9. The LR, shown in Figure 2.13, consists of four 64-bit wide registers which have independent input and output pointers. The input pointer, LRIP, points to a LR register which can be used as a SBD for the current MDP transport. The output pointer, LROP, points to a LR register which can be used as the A input to the AL. The contents of this register can be gated onto the BUS by setting the ALF to A (i.e., SET ALF A) and then choosing the AL as the source for a MDP transport.

The LR input to the AL, i.e., LR[LROP], will simply be referred to either as "A" or "LR". When LR is used as a SBD, i.e., LR[LRIP] is loaded from the SB, we will also use the name "LR" as no ambiguity should arise.

Both the LR input pointer, LRIP, and the LR output pointer, LROP, are incrementable, decrementable, clearable, and loadable with two bits from the Double Shifter, DS(V+1:V), see Section 2.12. The utility of this last feature will be demonstrated with examples when the Double Shifter is introduced. Table 2.11 gives the microoperations associated with the control of the LR. The last four microoperations allow for the clearing, incrementing, decrementing, and loading of both the IP and the OP simultaneously.

2.10 The Accumulator Shifter

The Accumulator Shifter, AS, serves as the B input to the AL in the context of the AL functions shown in Table 2.9. The reason one is called the Accumulator Shifter is that not only does it serve as an input to the AL, but also it will serve as the accumulator required in the realization of the basic arithmetic operations (e.g. multiplication). The AS can serve as a SBD; but to be read, its contents must be gated through the AL with the ALF set to B. The AS, shown in Figure 2.14, can shift left or right one bit position, be loaded, or remain idle during the execution of any given microinstruction.



Source no.	AS(63) Input	AS(0) Input
0	0	0
1	1	1
2	AS(0)	AS(63)
3	AS(63)	BUS(63)
4	CR	SB(63)
5	DS(V+1)	DS(V+1)
6	AS(V)	AS(V)
7	VS(V)	VS(V)

Accumulator Shifter, AS

Figure 2.14

There are two interesting features of this shifter: a) its variable width characteristic and b) its connection to other elements of the system. The features are discussed in the following:

a) Although the shifter is 64-bits wide it may, in conjunction with either the BM or PM, be viewed as being m-bits wide ($1 \leq m \leq 64$). This is accomplished by having each of the 64 bits of the AS as input to a selector (labelled the b_{63}-b_0 selector in Figure 2.14). The output of this selector (called the variable bit, V) can then be a possible input into either the left or right end of the shifter, depending upon what particular type of shift one requires. When the AS is selected as a source for MDP transport by gating it through the AL, after the desired shift has occurred, the bits not considered to be a part of the shifter must be masked off. This can be done either by using the BM or the PM. The width of the shifter is then determined by the contents of the AS(V) Selection register, AS(V)S, as shown in Figure 2.14 and the use of an appropriate mask.

The AS(V)S can be loaded by the following microoperation

AS(V)S := CM|EX|SB|SG.

Thus, for example, if we wish to consider the AS as a 48 bit left cyclic shifter, we would execute the microoperation

AS(V)S := 47

while making sure that AS(V) will be used as the input to bit AS(0) during the shift operation. Subsequent use of the AS as a source could be accompanied by use of the PG masking off bits b_{63}-b_{48}, e.g.,

```
          ; SET ALF B
WA:=AL    ; PG→ 16.|||||
```

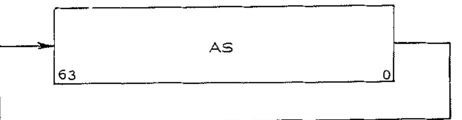
AS(0)S := CM|EX|SB|SG
AS(63)S := CM|EX|SB|SG
AS(V)S := CM|EX|SB|SG
ASLL
ASLR
AS(V)SC
AS(V)S+1
AS(V)S-1

Table 2.12
Microoperations for control of the AS

b) In Figure 2.14 it is seen that bits AS(0) and AS(63) can be filled with 1 of a variety of sources during a shift operation. Which source is to be used to fill the vacated bit position is determined by the contents of the AS(0) and AS(63) Source selection registers, AS(0)S and AS(63)S respectively. As an example, the execution of the microoperation

AS(63)S := 'AS(0)'

connects the shifter as shown below



An examination of the table in Figure 2.14 shows that the AS can be considered a logical shifter, a 1's fill shifter, a cyclic shifter, and a right arithmetic shifter. It can also be connected to another 1 bit shifter, called the variable width shifter, VS, to yield a long variable width shifter. It can be connected to a 2-bit shifter called the Double Shifter, DS, so it can be used in the merging of 2 bit streams into 1 or the diverging of 1 bit stream into 2. It can also be connected to the BUS, SB, and an entry in a condition register, CR.

Thus to use the AS, one must load the AS(V)S to set the width of the shifter and must load either the AS(0)S or AS(63)S to point to the source to be used as the input into the vacated bit position, i.e., one must set what the type of shift is, e.g., logical, 1's fill, long, etc. It is obvious that both of these operations need not be done each time the shifter is used, but only when one is "changing" the width or type of shifter. Table 2.12 lists the microoperations associated with the control of the AS. Note the AS can be set to a logical left, or logical right shift by use of the special microoperations ASLL and ASLR.

There are 2 bits in each microinstruction which control the operation of the AS: shift left, AS←, shift right, AS→, load, i.e., AS := SB(63:0), or be idle. When the AS is to be shifted, the operation is put in the "<microoperation and data>" field of the microinstruction; when the AS is to be loaded, it is specified as a DESTINATION in the "<MDP transport>" field of the microinstruction. As an example, the microinstruction

MDP Sub-system of Figure 2.10 expanded with LR, AL, and AS

Figure 2.15

WA:=AL; AS←.

stores the output of the AL in a WA register and then shifts the AS left, while the microinstruction

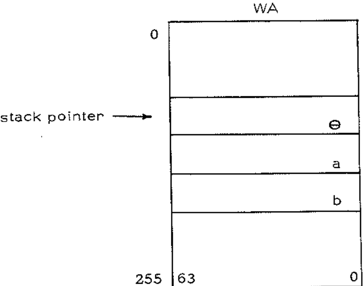
LR, AS:=WB; WBP+1.

stores a WB in both the AS and an LR and then increments the WB pointer. The AS can be used as input to AL (and subsequently as a SOURCE for MDP transport if the ALF is set to B) *and then* be either loaded or shifted *in the same* microinstruction.

Having introduced the AL and its inputs, LR and AS, we now have knowledge of the expanded MDP as shown in Figure 2.15.

Let us now give a few examples using these resources to demonstrate the use of their associated microoperations.

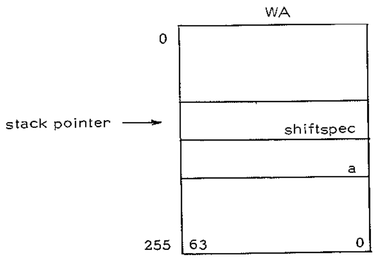
Example 1) Let us consider WA as a stack as shown below.



We wish to pop two operands, a and b, and an operator, ⊖, represented as an AL function from the stack and push a⊖b on the new top of stack. The following microinstruction sequence does this.

WA ; ALF:=SB, WAP+1, LRPC.
LR:=WA ; WAP+1.
AS:=WA .
WA:=AL .

Example 2) Let us again consider WA as a stack.



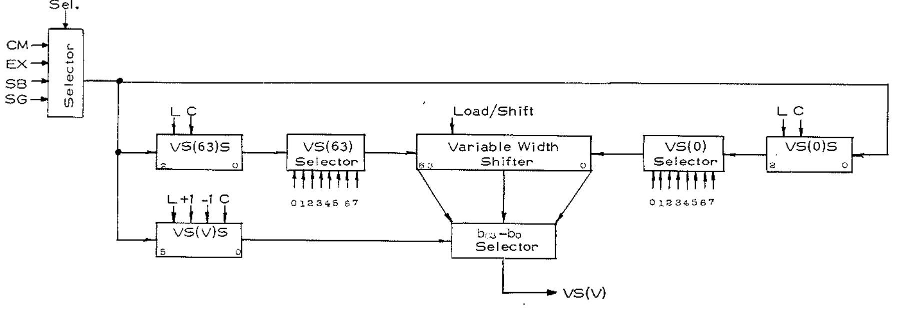
We wish to treat the AS as a left shifter whose characteristics are given by shiftspec. We wish to shift a n-times and return the result to the new top of stack after removing shiftspec and a. Let us assume shiftspec to have the following format:

0 ←→ 0	n	pgmsk	width	type
63 22	21 16	15 9	8 3	2 0

where

type = encoding found in the table of Figure 2.14 for logical, cyclic, etc. shift,
width = width of shifter-1, 1≤ width of shifter≤64
pgmsk = PG mask specification,
n = number of shifts-1, 1≤ number of shifts≤64

The following microinstructions execute the desired operation.



Source no.	VS(63) Input	VS(0) Input
0	0	0
1	1	1
2	VS(0)	VS(63)
3	VS(63)	BUS(62)
4	CR	SB(62)
5	DS(V)	DS(V)
6	VS(V)	VS(V)
7	AS(V)	AS(V)

Variable Width Shifter, VS

Figure 2.16

VS(0)S := CM|EX|SB|SG
VS(63)S := CM|EX|SB|SG
VS(V)S := CM|EX|SB|SG
VSLL
VSLR
VS(V)SC
VS(V)S +1
VS(V)S -1

Table 2.13

Microoperations for control of the VS

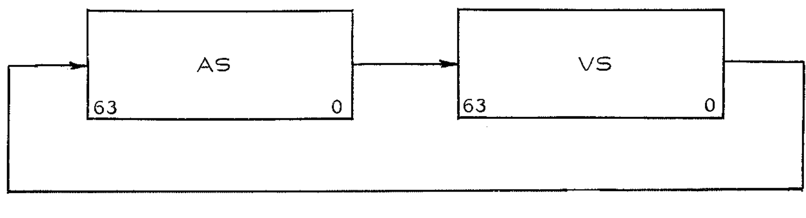

```
WA          ;AS(0)S:=SB.
WA, →3      ;AS(V)S:=SB.
WA, →9      ;PGSG:=SB.
WA, →16     ;CA:=SB,WAP+1.
AS:=WA      ;PGS:='SG',PAP+1,SETALFB.
            ;CA-1,AS ←                       ;if¬ CA then HERE.
WA:=AL      ;PAP-1,PGS:='CM'.|||||
```

2.11 The Variable Width Shifter

The Variable Width Shifter, VS, is a shifter functionally identical to the AS. The VS can be used as a SOURCE for MDP transport *and then* be either loaded or shifted *in the same* microinstruction. It is shown in Figure 2.16. The microoperations associated with the VS are identical to those associated with the AS and are listed in Table 2.13. One of the important features of the AS and VS, as seen from the tables in Figures 2.14 and 2.16, is that they can be connected together. This allows, for example, the AS and VS to be viewed as a "long" shifter when coupled together. The microinstructions,

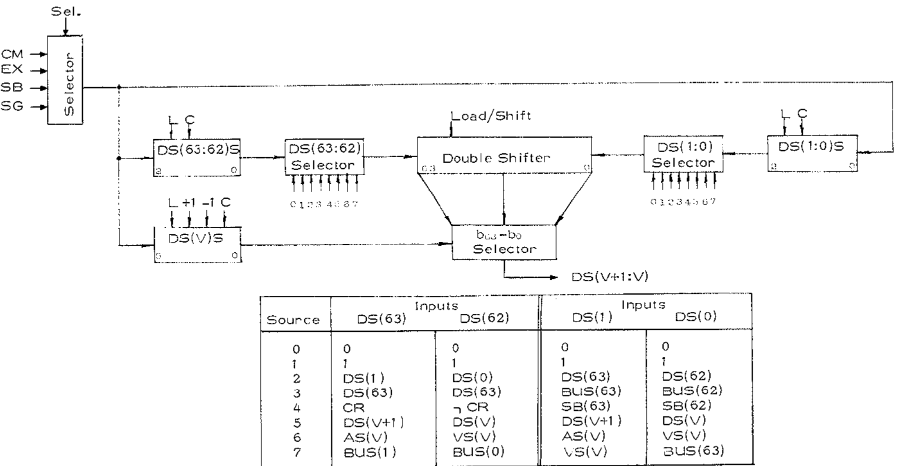
```
; AS(63)S:='VS(V)', VS(63)S:='AS(V)'.
; AS(V)SC, VS(V)SC.
```

connect the AS and VS together so that they can be viewed as a right cyclic 128-bit shifter as shown below.



Just as with the AS, there are 2 bits in each microinstruction which control the operation of the VS: shift left, VS←, shift right, VS→, load, i.e., VS:=SB(63:0), or remain idle.

Assuming the previous AS/VS connection has been made, subsequent execution of



Source	Inputs DS(63)	DS(62)	Inputs DS(1)	DS(0)
0	0	0	0	0
1	1	1	1	1
2	DS(1)	DS(0)	DS(63)	DS(62)
3	DS(63)	DS(63)	BUS(63)	BUS(62)
4	CR	¬ CR	SB(63)	SB(62)
5	DS(V+1)	DS(V)	DS(V+1)	DS(V)
6	AS(V)	VS(V)	AS(V)	VS(V)
7	BUS(1)	BUS(0)	VS(V)	BUS(63)

Double Shifter, DS

Figure 2.17

DS(1:0)S := CM|EX|SB|SG
DS(63:62)S := CM|EX|SB|SG
DS(V)S := CM|EX|SB|SG|
DSLL
DSLR
DS(V)SC
DS(V)S +1
DS(V)S -1

Table 2.14

Microoperations for control of the DS

the microoperations

AS→, VS→

shifts this 128-bit shifter 1 bit right cyclic. Other "long shifters", e.g. left logical, right logical, right arithmetic, etc., result from appropriate set up sequences.

2.12 Double Shifter

The Double Shifter, DS, is a shifter with functional characteristics similar to those of the AS and VS, except that it shifts 2 bit positions at a time and not 1. Bits DS(0) and DS(1) require input during a left shift and DS(62) and DS(63) require input during a right shift. The DS is shown in Figure 2.17. The DS can be used as a SOURCE for MDP transport *and then* be either loaded or shifted *in the same* microinstruction.

The microoperations which are associated with the DS are directly comparable to those for the AS or VS and are shown in Table 2.14. There are 2 bits in each microinstruction which control the operation of the DS: shift left, DS←, shift right, DS→, load, i.e., DS:=SB(63:0), or remain idle.

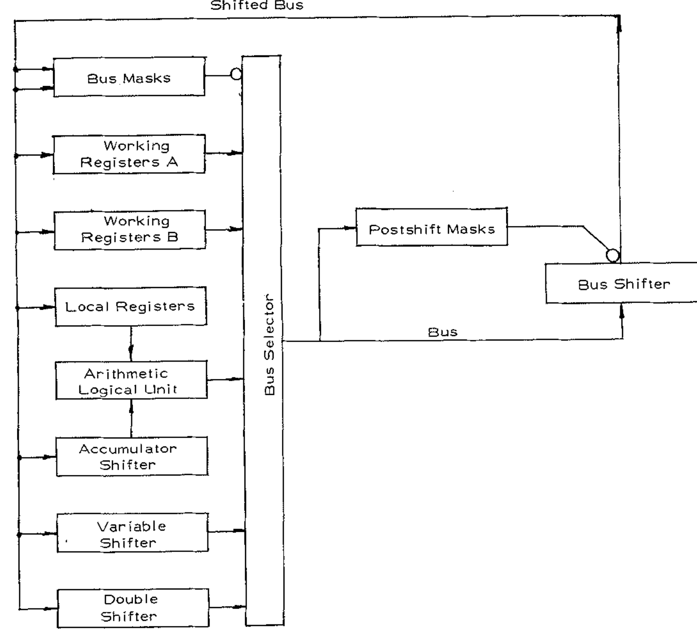
2.12.1 Two examples using the shifters

The AS, VS, and DS are collectively referred to as the "Shifters" whereas the Bus Shifters are not included in this term. The expanded MDP is shown in Figure 2.18.

Example 1)

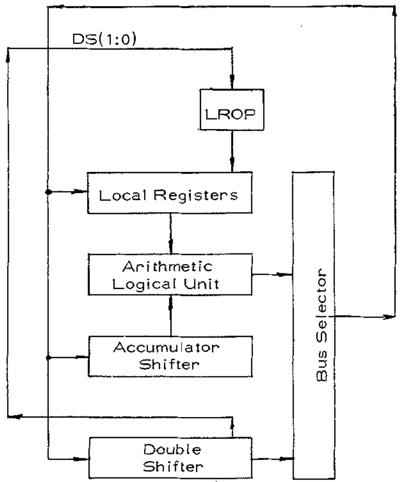
Suppose we wish to count the number of bits which are set to 1 in WA[WAP] and leave this number in the same cell. The following algorithm will do this.

a) Load the LR with the following constants
LR[0]:=0
LR[1]:=1
LR[2]:=1
LR[3]:=2
b) Clear the AS (considered here as an accumulator)
c) Set the AL to addition
d) Transfer the data to the DS



MDP Sub-system of Figure 2.15 expanded with VS and DS

Figure 2.18



Counting Loop for Counting Number of Bits set to 1 in a Word

Figure 2.19

e) Do the following 32 times and then do (f)
 i) if DS(1:0)=00 then accumulate LR[0]+AS in AS
  if DS(1:0)=01 then accumulate LR[1]+AS in AS
  if DS(1:0)=10 then accumulate LR[2]+AS in AS
  if DS(1:0)=11 then accumulate LR[3]+AS in AS
 ii) shift DS→
f) Store the accumulated result into WA[WAP].

The following microinstruction sequence accomplishes this. It is assumed the PG data source is the CM.

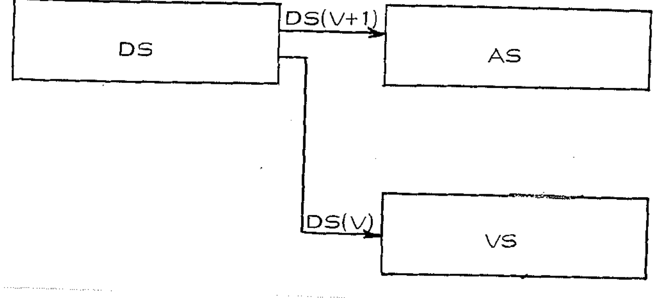
```
   DS := WA   ;ALF:='all 0's',LRPC           ;
AS,LR := AL   ;LRIP+1,SETALF+1               ;
VS,LR := AL   ;LRIP+1,VSLL,DS(V)SC           ;
   LR := AL   ;VS<,LRIP+1,SETALF+            ;
   LR := VS   ;CA :=30,LROP:=DS              ;
   AS := AL   ;DS>,CA-1,LROP:=DS             ;if ¬CA then HERE
   WA := AL   .|||||
```

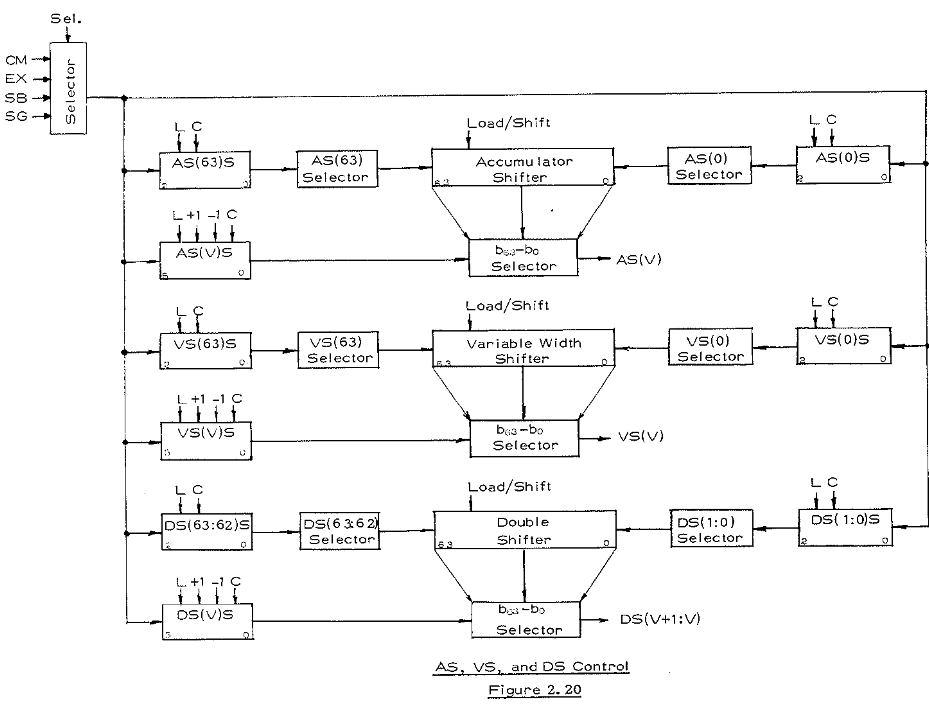
The subset of the MDP which is used during the counting loop instruction (AS:=AL) is shown in Figure 2.19. This may help in understanding the algorithm and code.

Example 2)
Consider the contents of the current WA register as a string of 64 bits. It is desired to pack all of the even numbered bits (b_0, b_2, etc.) in the right 32 bits of the current WB register and then odd numbered bits (b_1, b_3, etc.) in the left 32 bits of this register so that the result appears as

$$b_{63} \ldots b_5\ b_3\ b_1\quad b_{62} \ldots b_4\ b_2\ b_0$$

Because the DS, AS, and VS can be connected as shown below





AS, VS, and DS Control

Figure 2. 20

AVDSG := SB
AVDP := CM|EX|S1|S2
AVDP + 1
AVDP - 1
AVDPC
AVDPS1 := CM|EX|S1|S2
AVDPS1 := AVDP

Table 2. 15

Microoperations for the control of the AVD SG

Notation	Microoperation
AVDLL	Set AS, VS, DS to logical left shift
AVDLR	Set AS, VS, DS to logical right shift
AVD(0)S:=CM|EX|SB|SG	Load AS(0), VS(0), and DS(1:0) Source register from CM|EX|SB|SG
AVD(63)S:=CM|EX|SB|SG	Load AS(63), VS(63), and DS(63:62) Source register from CM|EX|SB|SG
AVD(V)S:=CM|EX|SB|SG	Load AS(V), VS(V), and DS(V) Selection register from CM|EX|SB|SG
AVD(V)SC	Clear AS(V), VS(V), and DS(V) Selector register

Table 2.16

Parallel AVD Microoperations

one can accomplish the stated requirement in the following way:

```
                 ; ALF:='all 0's', LRPC.
AS, VS:=AL       ; AS(63)S:='DS(V+1)',VS(63)S:='DS(V)', DS(V)SC.
DS:=WA           ; CA:=31.
                 ; CA-1, AS→, VS→, DS→; if ¬CA then HERE.
LR:=VS, →32      ; ALF:='A∨B'.
WB:=AL           .||||
```

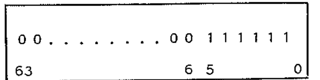
2.13 The Accumulator/Variable/Double Shifter Standard Group

The Shifter Control Selector shown in Figures 2.14, 2.16, and 2.17 is the same selector. This is, perhaps, made a bit clearer in Figure 2.20. The SG which is associated with this selector is called the Accumulator/Variable/Double Shifter SG, AVDSG. Shifter control data can be stored in the AVDSG for various shifter interconnections and then used in environment prologues. The microoperations associated with the AVDSG are shown in Table 2.15.

In addition there are several microoperations which allow control of the AS, VS, and DS to be executed in parallel. These are shown in Table 2.16.

2.14 Loading Masks

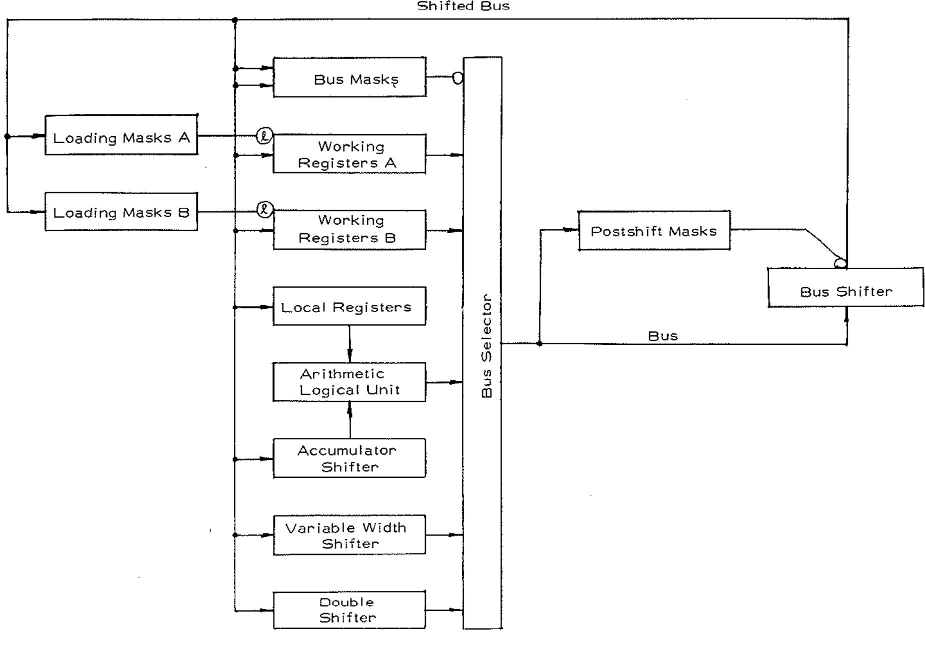
Associated with WA there is a SG of masks called Loading Masks A, LA. Associated with WB there is a SG of masks called Loading Masks B, LB. In what follows we will describe only LA; LB is identical in function. The purpose of LA is to be able to specify which bit positions in a working register WA will be loaded as the result of WA being chosen as the DESTINATION of a MDP transport, *while* leaving the nonspecified bits unchanged. As an example, if the loading mask



were contained in LA[LAP] when the bus transport

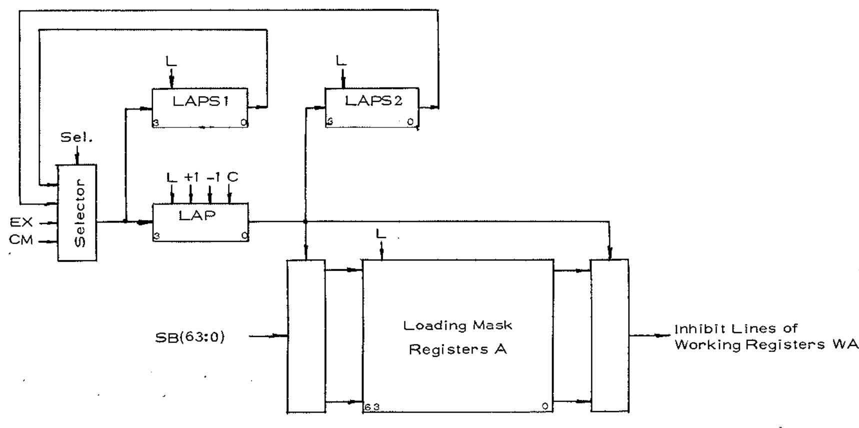
WA:=AL

is executed, bits SB(5:0) would be gated into WA[WAP] in bit positions b_0 through b_5 respectively, while bits b_6 through b_{63} *would not* change their value.



MDP Sub-system of Figure 2.18 expanded with LA and LB

Figure 2.21



Loading Mask Registers A, LA

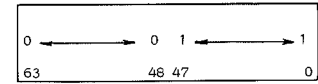
Figure 2.22

When WA is selected as a SOURCE for MDP transport, the mask LA acts in the following fashion: if bit i ($63 \geq i \geq 0$) of the mask is a 1, then bit i of WA is transmitted. If bit i of the mask is 0, then bit i which is transmitted is *indeterminate*. The relationship between the loading masks and the working registers is represented by the symbol —ℓ where the script ℓ in the mask notation —ℓ indicates the special nature of these masks. Figure 2.21 shows the expanded MDP with the loading masks added.

Figure 2.22 shows a more detailed sketch of LA; LB, not shown, is identical. There are 7 microoperations shown in Figure 2.22 associated with the use of LA. These are listed along with the corresponding microoperations for LB in symbolic form in Table 2.17.

The programming convention is such that the "full load" or "full read out" mask, i.e., 64 1's is in LA[0] and LB[0]. We will assume this to be the case throughout all of the examples which follow. One can then look upon the pointers LAP and LBP as selection switches for the use of the loading masks. If LAP = 0 then no loading mask is applied to WA, if LAP≠0 then WA is masked by the mask specified by LAP; a similar statement can be made for LBP. This is, of course, not the only interpretation of the use of the loading masks.

As an example, suppose we wish to place the high order 48 bits of the output of the DS into the least 48 bits of WB[0] leaving the high order 16 bits the same. If the mask



is in LB[9], the following microinstruction sequence accomplishes this:

```
                    ; LBP:=9, WBPC.
WB:=DS,→16          ; LBPC.|||||
```

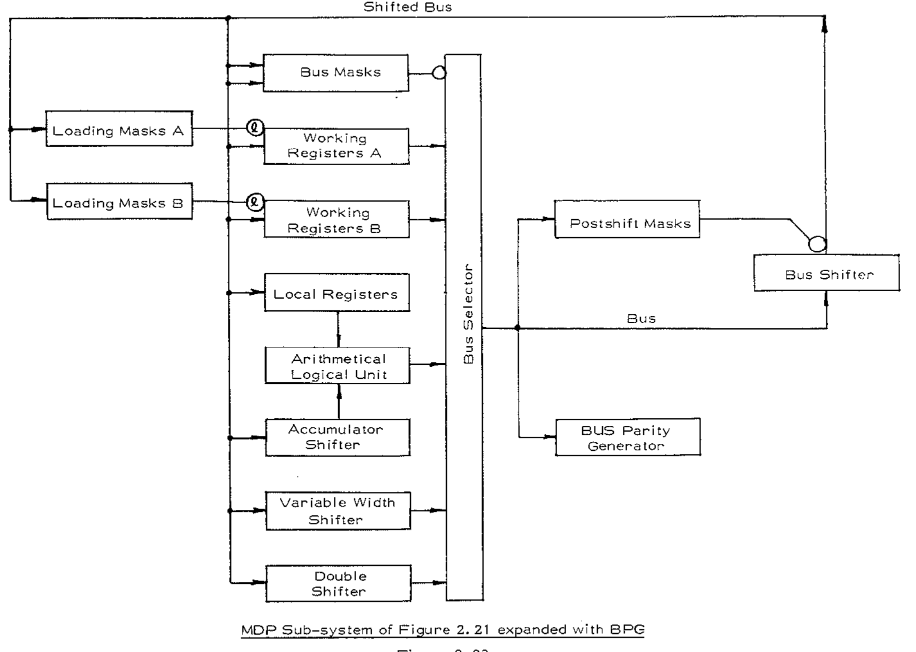
This mask could have been generated by use of the PG and AL. The code,

```
          ; ALF:='all 1s', LBP:=9.
          ; PAP+1.
AL        ; PG→16, LB:=SB, PAP-1.|||||
```

LA := SB(63:0)	LB := SB(63:0)
LAP := CM|EX|S1|S2	LBP := CM|EX|S1|S2
LAP +1	LBP +1
LAP -1	LBP -1
LAPC	LBPC
LAPS1:=CM|EX|S1|S2	LBPS2:=CM|EX|S1|S2
LAPS2:=LAP	LBPS2:=LBP

Table 2.17

Microoperations for control of LA and LB



MDP Sub-system of Figure 2.21 expanded with BPG

Figure 2.23

generates the mask and stores it in LB[9]. It should be reasonably obvious now how the loading masks can be used to store the result of various data transformations as they are determined, e.g., in the implementation of signed-magnitude arithmetic, the magnitude of the exponent, its sign, the magnitude of the coefficient and its sign can be stored in a given word as they are obtained.

We will henceforth assume in all examples (unless explicitly stated otherwise) that LAP = 0 and LBP = 0, i.e., that no loading masks are applied to either set of working registers. If a particular code segment uses the loading mask facility it is responsible for leaving the system operating in this fashion. The treatment of the loading masks then becomes quite identical with that of the bus masks and postshift masks as stated in Section 2.7.

2.15 The BUS Parity Generator,

The BUS Parity Generator, BPG, is a circuit which determines the parity of the 64 bits which compose the bus transport. It posts the result of this evaluation as a testable condition, the bus parity, BP, condition. If BP = 1, the BUS is odd parity; if BP = 0, the BUS is of even parity. This condition can be used, obviously, in any processingwherein parity information is viable, e.g., in communicating with devices which transmit words of a particular parity. The parity generator functions during each bus transport and has no microoperations associated with it. Since its input is the BUS, we show it attached to the bus structure as shown in Figure 2.23. Note, however, no output is shown as its only output is the BP condition.

2.16 The Bit Encoder

Let us label the bits of the BUS in the following way:

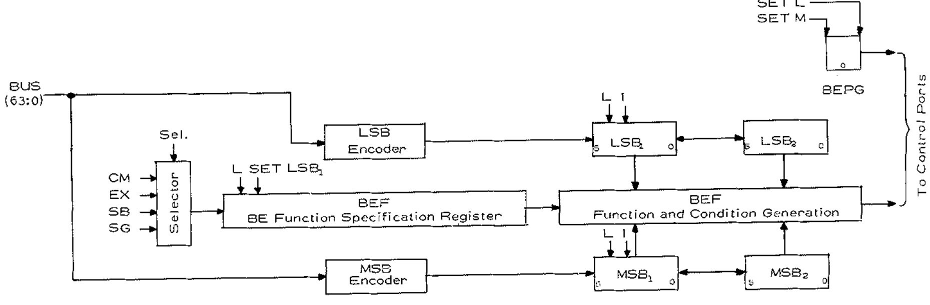
$$b_{63}\, b_{62}\, \dots b_1\, b_0\ .$$

Let us scan this string of bits from the right to the left, i.e., starting with bit b_0 and finishing with bit b_{63}. LSB will denote the value of the subscript of the first, nonzero bit encountered, while MSB will denote the value of the subscript of the last nonzero bit encountered in this string. This can be shown as

0 ⟷ 0 1 X...X 1 0 ⟷ 0

b_{63} ... b_k ... b_j ... b_0

MSB=k LSB=j



Bit Encoder, BE
Figure 2.24

	BE Functions F&G
F	LSB_1
	LSB_1-1
	MSB_1
	MSB_1+1
	L_1
	$\Delta L = L_2-L_1$
	LSB_2-LSB_1
	MSB_2-MSB_1
G	$\left[\frac{F}{2}\right]+1$ [ ] ::= integer part of

Table 2.18
Bit Encoder Functions

where $k \geq j$. If $k = j$ there are, of course, no bits between b_k and b_j ; if $k>j$, the $k-j-1$ bits, denoted by X's, between b_k and b_j may be any arbitrary string of $(k-j-1)$ 0's and 1's. If the bit pattern is identically zero then LSB and MSB are defined to be 63, respectively 0.

There is, on the MATHILDA System, a functional unit called the Bit Encoder, BE, which, during every MDP transport, encodes the MSB and LSB associated with data on the BUS. The BE, shown in Figure 2.24, can also manipulate these quantities. During each bus transport an "LSB encoder" and an "MSB encoder" determines the LSB and MSB associated with data on the BUS. The result of these encodings can be loaded into the LSB_1 and MSB_1 registers shown in Figure 2.24. A load of the LSB_1 register causes the old contents of the LSB_1 register to be moved to the LSB_2 register. Similarly, a load of the MSB_1 register causes the old contents of the MSB_1 register to be moved to the MSB_2 register. The contents of the LSB_1 and LSB_2 registers can be interchanged and the contents of the MSB_1 and MSB_2 registers can be interchanged.

The BE can compute various functions with the variables LSB_1 , LSB_2 , MSB_1 , and MSB_2 . These functions, F and G, are given in Table 2.18 where $L_i = MSB_i - LSB_i$, $i=1,2$. Which particular function is to be the output of the BE is determined by the contents of the BE Function Specification register, BEF,

BEF:=CM|EX|SB|SG.

When the BEF is loaded from the CM we will note this symbolically merely by writing the required function in the symbolic form in Table 2.18, e.g.,

BEF:='LSB1'.

The output of the BE can be used to control many devices in the system. It may, for example, be used to control the BS (see Section 2.5), it may be loaded into Counter B to control a process (see Section 2.23.1), or it may be used to generate a Postshift mask using the PG (see Section 2.7). There are only 6 bits of output from the BE. When it is used to generate a postshift mask using the PG, the direction from which the mask is to be generated must be specified in advance by use of either of the microoperations

BEPGL or BEPGM.

The first microoperation will cause a mask to be generated from b_0 (the least significant end of the SB) whereas the second microoperation will cause a mask to

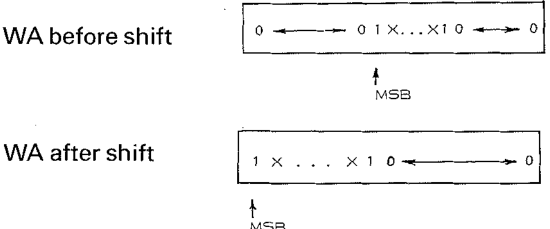
Notation	Microoperation
BELLOAD	$LSB_2 := LSB_1$ and then $LSB_1 :=$ (LSB Encoder output)
BEMLOAD	$MSB_2 := MSB_1$ and then $MSB_1 :=$ (MSB Encoder input)
BELMLOAD	BEL Load and BEM Load
BELI	Interchange LSB_1 and LSB_2
BEMI	Interchange MSB_1 and MSB_2
BELMI	BELI and BEMI
BEF := CM|EX|SB|SG	Load BE Function Specification Registers from CM|EX|SB|SG
SET BEF LSB1	Set BEF to LSB_1
BEPG L	Allows the semantics of the ← in PG←BE to be realized when the PG is controlled by the BE
BEPG M	Allows the semantics of the → in PG→BE to be realized when the PG is controlled by the BE
BESG := SB	
BEP := CM|EX|S1|S2	
BEP + 1	
BEP - 1	
BEPC	
BEPS1 := CM|EX|S1|S2	
BEPS2 := BEP	

Table 2. 19
Microoperations for control of BE

be generated from b_{63} (the most significant end of the SB).

The microoperations which control the BE are given in Table 2.19. Note the SG associated with the BEF is called the BESG.

Example 1)
We wish to take the contents of WA[WAP] and shift it left so that its MSB before the shift is shifted to bit position b_{63}. The result of this operation is to be placed back in WA. The contents of WA is shown below.



The following microinstructions accomplish this.

```
                     ;BEF:='MSB1+1'
  DS := WA           ;BEMLOAD,BSS:='BE'
  WA := DS,←         ;BSSC.|||||
```

Note in this example that the DS is merely used as temporary storage.

Example 2)
Consider the example of Section 2.12.1 in which we counted the number of bits which were set to 1 in a given 64-bit WA register. Instead of doing the counting 2-bits at a time in a loop which is exercised 32 times, we could still count 2-bits at a time, but only count

$$\left\lceil \frac{(MSB_1 - LSB_1)}{2} \right\rceil + 1$$

times, provided we shift the data LSB_1 places to the right before counting. The

Notation	Predicate
LSB1	$LSB_1 \equiv 0$
MSB1	$MSB_1 \equiv 63$
L1	$L_1 \equiv 0$ (i.e., $LSB_1 = MSB_1$)
L2	$L_2 \equiv 0$ (i.e., $LSB_2 = MSB_2$)
LD	$L_1 = L_2$
SGLD	sign $(L_2 - L_1)$
LSBD	$LSB_1 = LSB_2$
SGNLSBD	sign $(LSB_2 - LSB_1)$
MSBD	$MSB_1 = MSB_2$
SGNMSBD	sign $(MSB_2 - MSB_1)$

Table 2.20A

Bit Encoder Conditions

Notation	Predicate
BUS	$BUS(63:0) \equiv 0$
BE(0)	$BE(0) \equiv 1$
BEPGD	BEPG ≡ '←'

Table 2.20B

Conditions Related to BE Usage

following microoperations accomplish this,

```
DS=WA       ; BELMLOAD, SETBEFLSB1, BSS:='BE'.
DS:=DS, ⇸   ; BEF:='[L1/2]+1', SETALFALLOS, LRPC.
LR, AS:=AL  ; CB:=BE, SETALF+1,  LRIP+1              .LR[0]:=0
LR:=AL      ; CB-1, BSSC, LRIP+1                     .LR[1]:=1
LR:=AL      ; DS(V)SC, LRP+1                         .LR[2]:=1
LR:=AL      ; SETALF+, LROP:=DS(V+1:V)               .LR[3]:=2
AS:=AL      ; CB-1, DS⇸ , LROP:=DS(V+1:V)            ;if ¬ CB then HERE
WA:=AL      .|||||
```

Note that this code is only 1 instruction longer than the code in Section 2.12.1, example 1. This is caused by one additional MDP transport. Counter B, CB, used in this example can be loaded from the BE (see Section 2.23.1).

2.16.1 Bit Encoder Conditions

The conditions associated with the BE are listed in Table 2.20A. The important thing about the conditions is that *all* of them are available for testing irrespective of which particular BE function is specified.

There are two additional conditions which are related to this resource. The LSB and MSB encoding process yields a testable condition which indicates if bits b_0 through b_{63} of the BUS are all zero. This condition, i.e., BUS(63:0) ≡0, is denoted by BUS. Thus, there is no ambiguity in the example,

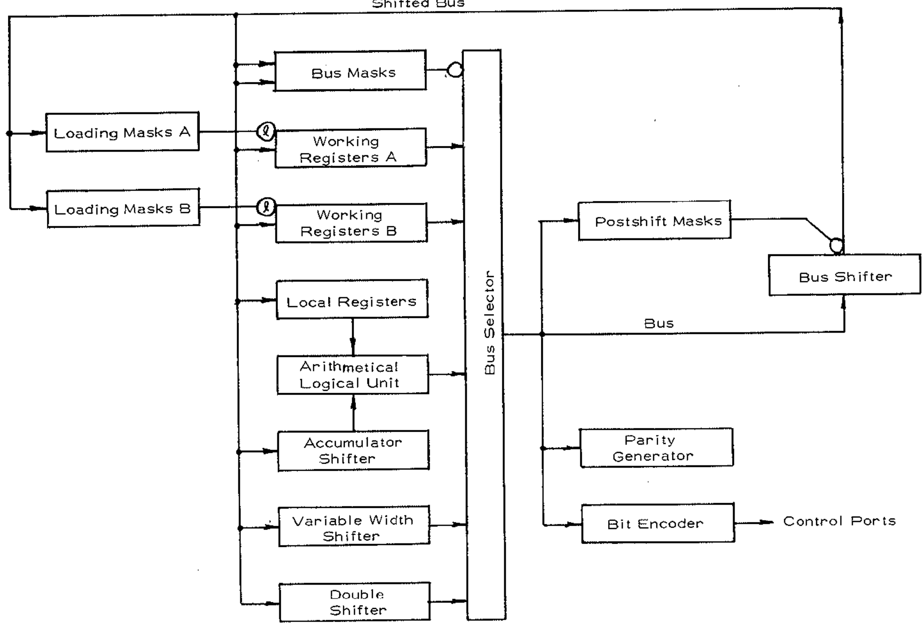
if BUS then A_t else A_f .

We can also test if the output of the BE is odd or even. This condition is, of course, written BE(0). These conditions, along with the ability to test the state of the BEPG register, are given in Table 2.20B.

Example

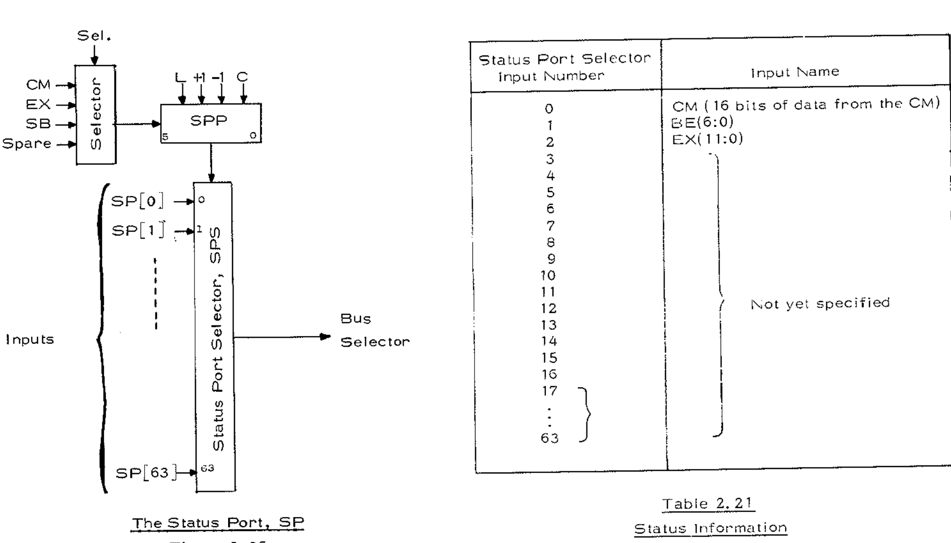
Suppose we wish to test if there is exactly one bit set to 1 in a particular bit string, say the contents of the VS, we could write

```
VS    ; BELM LOAD.
                    ; if L1 then ONEBIT.|||||
```



MDP Sub-system of Figure 2.23 extended with BE

Figure 2.25



The Status Port, SP

Figure 2.26

Status Port Selector Input Number	Input Name
0	CM (16 bits of data from the CM)
1	BE(6:0)
2	EX(11:0)
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	Not yet specified
13	
14	
15	
16	
17	
:	
63	

Table 2.21

Status Information

where ONEBIT is the address of the microinstruction to execute if one bit is set to 1.

Since the BE has as its input encodings from information on the BUS, we show it attached to the bus structure as shown in Figure 2.25. Note that the output of the BE is shown going to various "control ports" in accordance with the prior discussion.

2.17 The Status Port

The Status Port, SP, allows data sources other than those directly connected to the Bus Selector to be used as a SOURCE in a MDP transport. In this sense, then, it provides a Bus Selector expansion facility. The outputs of various resources are connected to a 64 input, 16-bit wide selector called the Status Port Selector, SPS as shown in Figure 2.26.

The output of the SPS is the particular status port input selected by the contents of the Status Port Pointer, SPP, and is symbolically written SP, i.e., SP≡SPS[SPP]. Table 2.21 shows the correspondence between particular inputs and the input number of the SPS.

All status port inputs are zero filled in the most significant bit positions. It should be pointed out that a constant, contained as a literal within a microinstruction, can be gated onto the BUS when SPP is zero. This is shown in the following code sequence:

```
                            ;SPPC. Comment: This could have been written as SPP:='CM'.
<DEST>:=constant            ;
```

The microassembler will treat this microinstruction sequence in the equivalent way

```
                            ; SPPC
<DEST>:=SP                  ; <constant in "mops and data"field>.
```

The programming convention for use of the SP will be that SPP=0 so that the first microinstruction in each of the above sequences need not be there. This convention will be assumed in all the examples which follow.

SPP := CM|EX|SB
SPPC
SPP +1
SPP −1

Table 2.22
Microoperations for control of SPP

The MDP transport specification

< DEST >:=SP

is to mean that the Status Port Selector input specified by the SPP contents is to be the SOURCE for the MDP transport. *Except* for the CM input, this data is information which has been set *prior* to execution of the MDP transport specified in the current microinstruction (and not data resulting from the execution of the microoperation specified in the current microinstruction). Table 2.22 lists the microoperations associated with the use of the SPP.

As an example, suppose we wish to interchange the WA and WB Pointers, i.e., WAP↔WBP. The following microinstruction sequence will accomplish this,

```
        ; SPP:='WAP'.
DS:=SP  ; SPP+1. Note this effectively executes SPP:='WBP'.
SP      ; WAP:=SB.
DS      ; WBP:=SB, SPPC.|||||
```

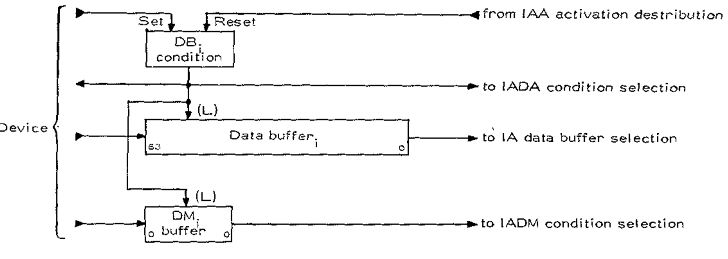
As a final example, assume we require the result of a BE computation to be used as data during a computation. In particular, the LSB encoding of the bit string contained in the DS is to be put into the AS. A possible microinstruction sequence is:

```
DS      ; SPP:='BE', BELML, SET BEF LSB1.
AS:=SP  ; SPPC.|||||
```

2.18 Input Facility

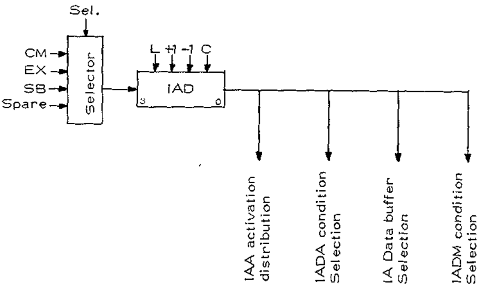
There are two input ports through which external devices may be connected to the bus selector. They are called Input Port A, IA, and Input Port B, IB. Up to 16 devices can be connected to each of these input ports in their fully expanded configuration. The basic input port consists of only a single device buffer and its associated Busy register and Data Mark buffer, as shown in Figure 2.27.

The particular device which is selected to be read in an expanded input port is pointed to by a Device Register, as shown in Figure 2.28. The contents of IAD



Device Interfacing on Input Port

Figure 2.27



Device Selection on a fully expanded IA

Figure 2.28

Notation	Microoperation
IAA	Activate selected device on IA
IAD := CM|EX|SB	Load IA Device Register from CM|EX|SB
IADC	Clear IA Device Register
IAD +1	Increment IA Device Register
IAD -1	Decrement IA Device Register
IBA	Activate selected device on IB
IBD := CM|EX|SB	Load IB Device Register from CM|EX|SB
IBDC	Clear IB Device Register
IBD +1	Increment IB Device Register
IBD - 1	Decrement IB Device Register

Table 2.23

Microoperations for control of IA and IB

determines at any given time, in an expanded configuration, which device interface is accessible.

There are two conditions associated with a selected device: a) data available, IADA, (identical to the busy condition), and b) data condition , IADM (the contents of the Data Mark buffer). All devices must be able to set the first condition. When a device is activated by the IAA microoperation, the condition IADA is reset (i.e., to false). The device is assumed to respond when it has data ready by setting of the condition. This implicitly loads the device buffer (the L-pulse), and the state of the IADA condition will now be true, informing the availability of data.

The second condition can be set by devices which can transmit two different sorts of information, for example control information and data. The IADM condition is loaded when data is loaded into a particular device buffer. The microoperations associated with the control of IA and IB are given in Table 2.23.

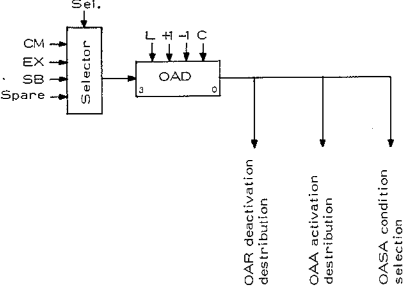
As an example, if we wish to read data from device 9 on IA and store it in AS, we can write the following wait loop:

```
            ;IAD: = 9,IAA.
            ;                         ;if ¬ IADA then HERE.
AS := IA .|||||
```

2.19 Output Facility

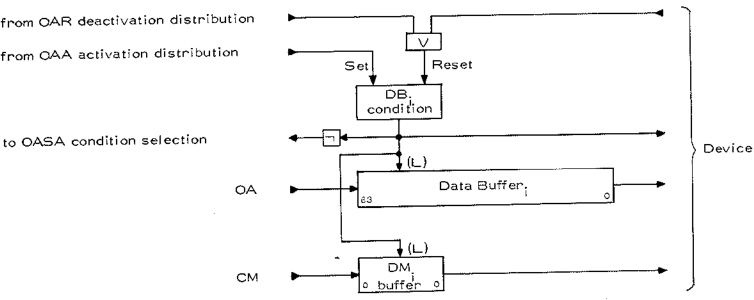
There are four output ports through which output to external devices may occur. They are called Output Ports A, B, C, and D; OA, OB, OC, and OD respectively. They are identical in operation with the exception that OA and OB are loaded from the SB and can be selected as SBD's whereas OC and OD are loaded from the BUS via microoperations. OA is shown in Figures 2.29 and 2.30; OB, OC, and OD, not shown, are identical.

Up to 16 devices can be connected to each of these output ports in their fully expanded configuration. The basic output port, as shown in Figure 2.30, consists only of a single data buffer, 64 bit wide and its associated condition register (the busy condition), and a Data Mark buffer (1 bit wide).



Device Selection on a fully expanded OA

Figure 2.29



Device Interfacing on Output Port

Figure 2.30

Notation	Microoperation
OAA	Activate Port, i.e., write OA
OAD := CM|EX|SB	Load OA Device Register from CM|EX|SB
OADC	Clear OA Device Register
OAD +1	Increment the OA Device Register
OAD −1	Decrement the OA Device Register
OCA	Activate Port, i.e., write OC
OCD := CM|EX|SB	Load OC Device Register from CM|EX|SB
OCDC	Clear OC Device Register
OCD +1	Increment the OC Device Register
OCD −1	Decrement the OC Device Register
OC := BUS	Load OC from BUS (63:0)
OAR	Deactivate OA, i.e., reset device condition flip-flop
OCR	Deactivate OC, i.e., reset device condition flip-flop

Table 2.24

Microoperations for control of the OA and OC

The particular device which is selected for output in an expanded configuration is pointed to by a Device register called OAD. It can be loaded from CM|EX|SB, its contents determine which device is currently selected as shown in Figure 2.29. There is a condition associated with a selected device: space available, OASA whose value is the complement of the busy condition. The microoperations associated with the control of OA and OC are shown in Table 2.24. The microoperations for OB are identical to those for OA and the microoperations for OD are identical to those for OC. The OA Activate microoperation, OAA, has only effect when the OASA condition is true, in which case the data buffer and the data mark buffer for the selected device is loaded, and the busy flag set true. The device is then assumed to consume the data and reset the busy condition when it is done.

As an example, suppose we wish to write out the output of the AL onto device 13 of output port C. We could then write,

```
AL      ;OC:=BUS,OCD:=13.
        ;                        ;if ¬OCSA then HERE.
        ;OCA.|||||
```

Recall that on the input ports it is possible to test a data condition which is set by a device. Analogous with this, it is possible on output to write out an extra bit in addition to the data, the Data Mark bit, which is a data bit from CM. The device can, for example, treat this extra bit as a data condition. The microoperations for output port activate are now given by

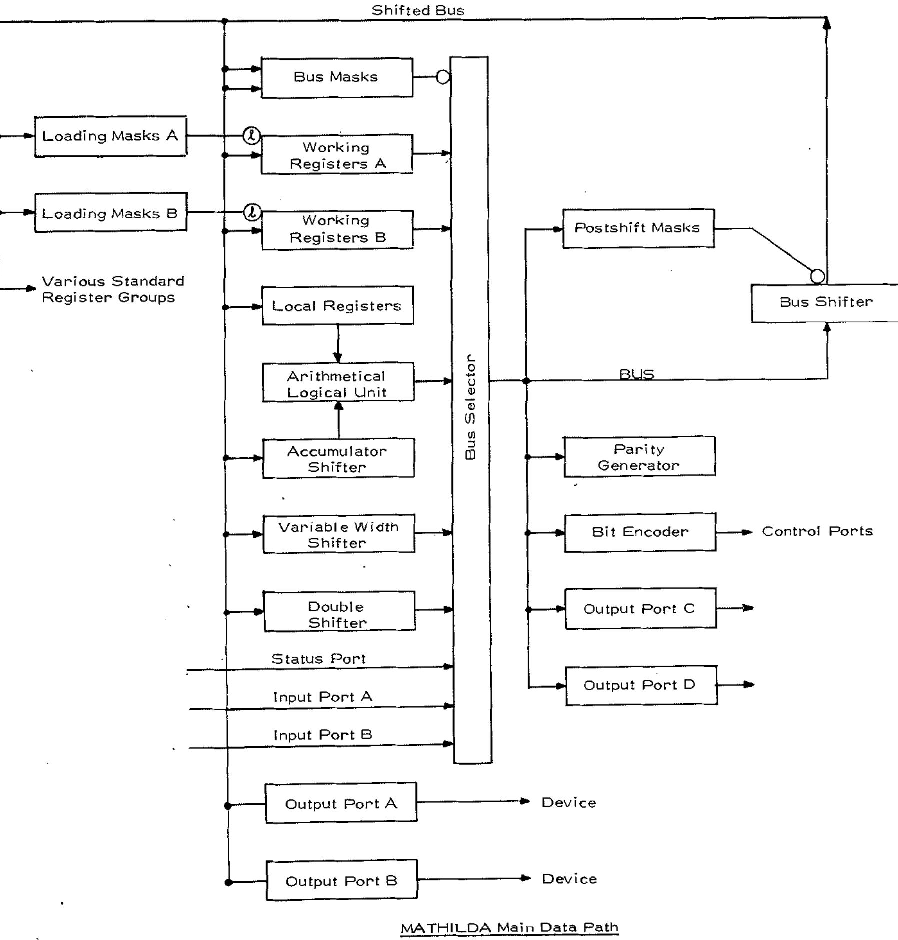
OAA1 activate with additional bit set to 1
OAA0 activate with additional bit set to 0
OAA activate with additional bit set to X (i.e., undefined)

The OAR (OA Reset) microoperation may be used upon deadstart to initialize the busy condition.

2.20 The MDP Structure

With the introduction of the output ports in the previous section we have a more complete MATHILDA MDP, the registers and functional units attached to it, and the control which can be exercised on these components. The MDP is now shown in Figure 2.31.

Let us summarize some of the information with respect to bus SOURCEs and



MATHILDA Main Data Path

Figure 2.31

DESTINATIONs. We have the following SOURCEs and DESTINATIONs for a MDP transport:

a) SOURCEs for BUS Transport

WA
WB
AL
VS
DS
SP
IA
IB

b) DESTINATIONs for 64-bit Load of SB with SBD Load

MA
MB
WA
WB
LR
OA
OB

c) Shifters which can load 64-bit SB via dedicated bits in every microinstruction

AS
VS
DS

Thus in the bus transport specification

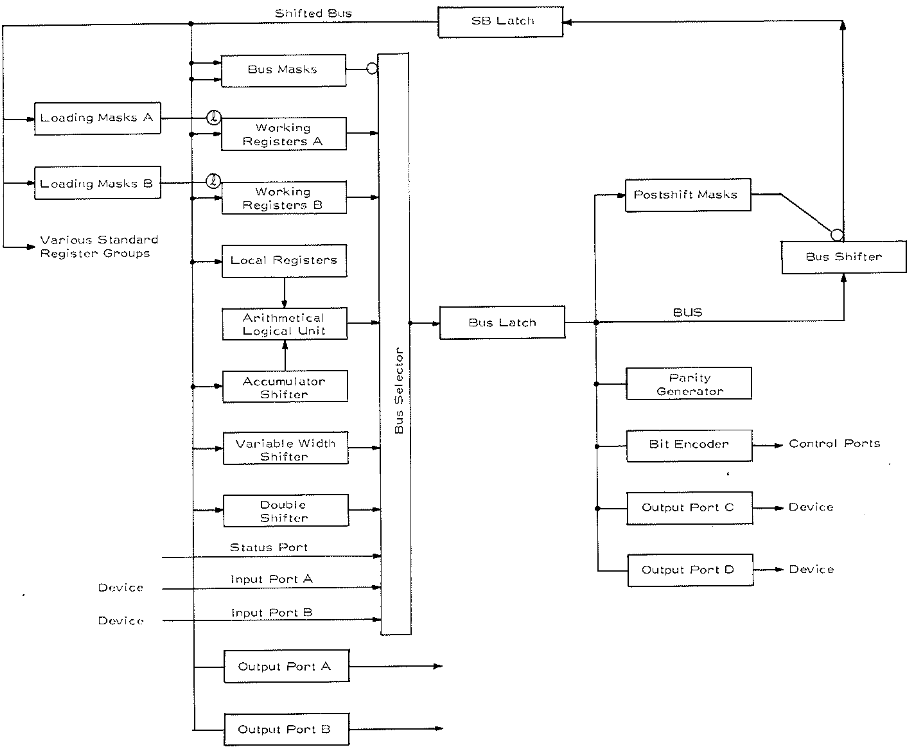
<LIST>:=<SOURCE>,

the LIST can consist of 1 destination from (b) above or any or all of the shifters, i.e.,

$SBD_b \{ , AS \} \{ , VS \} \{ , DS \} := SOURCE,$

where the $\{ \}$ indicates the option of inclusion in the LIST.

Recall that the SB can be loaded into LA and LB by execution of appropriate microoperations and the BUS can be loaded into PA, PB, OC, and OD by execution



MATHILDA Main Data Path
Figure 2.32

of appropriate microoperations. Also, a subfield of the SB can be loaded into various SG's and control ports throughout the system by executing the appropriate microoperation. Thus, many parallel loads of both the BUS and the SB may occur in any given microinstruction.

2.20.1 The Bus Latch and the Shifted Bus Latch

In order to realize the semantics of the microoperations which have been given, it is required that both the output of the bus selector and the output of the BS after postshift masking has occurred be "latched" (i.e., held temporarily in a buffer). Thus, a more correct representation of the MATHILDA MDP is shown in Figure 2.32.

BUS, then, is the name given to the output of the Bus Latch. There is a microoperation associated with the Bus Latch which sets all the bits of the BUS to 1. This action is completed before the BS and PM are used. This operation of setting BUS(63:0)=11...11 aborts the use of the selected SOURCE during a MDP transport and thus can be written in either of the equivalent forms,

<DEST>:=all 1s;

or

<DEST>:=SOURCE; BUS:=all 1s.

In a similar fashion, SB is the name given to the output of the SB Latch. There is a microoperation associated with the SB Latch which sets all the bits of the SB to 0. The operation of setting SB(63:0)=00...00 aborts the use of the BS and the PM during a MDP transport and can be written in the following equivalent ways:

<DEST>:=all 0s;

or

<DEST>:=<SOURCE> ; SB := all 0s.

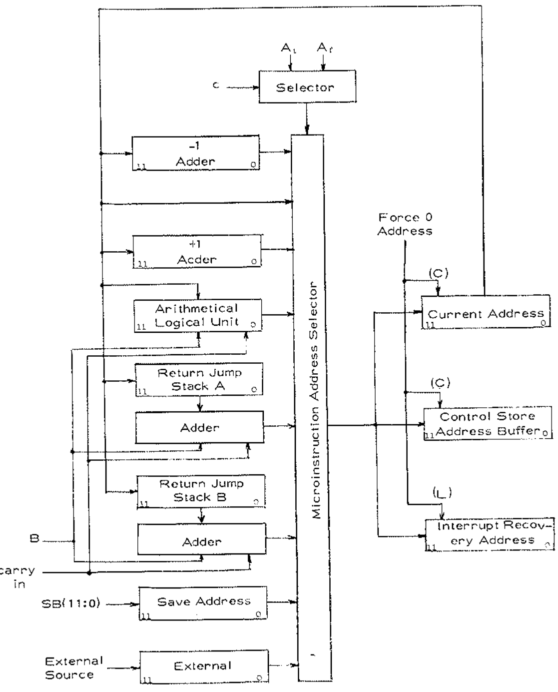
Note, however, the BUS is indeed available for use.

The semantics of the following examples should be obvious to the reader:

Notation	Interpretation
HERE -1	A - 1
HERE	A
HERE +1	A + 1
AL(A, B)	A function of A and B as computed by an arithmetical logical unit
RA + B	The contents of the top of a return jump stack, RA, added to B
RB + B	The contents of the top of a return jump stack, RB, added to B
SA	The contents of the Save Address register, SA
EX	The contents of the External register, EX

Table 2.25

Microinstruction Address Sources



Microinstruction Address Bus (Preliminary)

Figure 2.33

(a) WA: = all 0s.|||||

(b) DS ;SB: = all 0s,PA: = BUS.|||||

(c) WA: = all 0s;BUS: = all 1s,OC: = BUS.|||||

(d) WA ;OC: = BUS,SB: = all 0s,DS(1:0)S: = SB.|||||

2.21 The Control Unit

The control unit of the MATHILDA processor, shown in Figure 2.1, consists of (1) a control store and (2) a microinstruction sequencing capability. The random access control store consists of up to 4,096 words of 64-bit wide, 80 nanosecond monolithic storage. The microinstruction sequencing is described below.

2.21.1 Microinstruction Sequencing

The microinstruction sequencing hardware is a physical embodiment of the "if c then A_t else A_f " clause we have been using in the microprogramming examples. This is accomplished in the following way. The addresses A_t and A_f are selected from 8 possible address sources. Let A be the address of the current microinstruction and let B be data which is specified in the current microinstruction. The 8 possible address sources, which are explained in more detail shortly, are listed in Table 2.25. These address sources are realized by providing a microinstruction address bus which is shown in a limited form in Figure 2.33. One can see from this figure how the "if, then, else" clause is realized. There are 3-bits in each microinstruction which specify one of the 8 address sources of Table 2.25 to be used as the true branch address, denoted A_t. There are 3-bits in each microinstruction which specify one of the 8 address sources of Table 2.25 to be used as the false branch address, denoted A_f. There are 7 bits in each microinstruction used to specify 1 of 128 conditions which are testable in the system; the selected condition is denoted c. The state of the selected condition c determines which source, A_t or A_f, will be used to select the next microinstruction address source. If c = 1 then A_t will be used to select the address of the next microinstruction; if c = 0, then A_f will be used for this purpose. When a microinstruction address is selected, it is loaded into the Control Store Address Buffer so it can be used to fetch the microinstruction, and it is also loaded into the Current Address register so that it can be used in the next address computation, if required. The contents of the Current Address register has been used in previous examples under the symbolic name HERE. The "Force 0 Address" capability, the Interrupt Recovery Address register shown in Figure 2.33 will be discussed in later sections. Let us now discuss the address sources in detail.

The address sources A-1, A, and A+1 are straightforward and need not be dealt

SET SET
L A+B B
CM
CUAL Function
CUAL
A
B
Microinstruction Address Selector
carry-in (c or $\bar{c}$)
Data specified by the CM

Control Unit Arithmetical Logical Unit

Figure 2.34

with. It should be mentioned, however, that Control Store addresses are interpreted modulo the size of the Control Store.

2.21.2 The Control Unit Arithmetical Logical Unit

The Control Unit Arithmetical Logical Unit, CUAL, is functionally identical to the arithmetical logical unit which is connected to the MATHILDA bus structure except that it is 12-bits wide and not 64-bits wide. The CUAL functions are identical to those of the AL and are given in Table 2.9. The "A input" to these computations is the address of the CM and the "B input" is data specified by CM. The CUAL is shown as in Figure 2.34.

First, note that the CUAL Function register can only be loaded from the CM, i.e., CUALF:=CM. One can set the CUALF to add A and B, i.e., SET CUALF + and also to the logical function B, i.e., SET CUALF B. These are the only three microoperations associated with the CUAL. Only 5 bits are used to specify the function; the carry-in when required, is specified in another way. Let c denote the selected condition used to control the address selection and let $\bar{c}$ be its negation. There is a bit in each microinstruction, called the Carry-Input Selection Bit, CISB, which is used to determine if the carry-in is to be c or $\bar{c}$.

Example 1

Suppose the CUALF is set to A+B. Its output can be used to realize a relative jump when the CUAL is chosen as the selected address mode. If CISB='$\bar{c}$', the specification

if c then CUAL else HERE.

can be interpreted to mean:

if c then HERE + B else HERE.

Whereas, if CISB='c', the specification can be interpreted to mean:

if c then HERE + B + 1 else HERE.|||||

Example 2

Suppose the CUALF is set to B. Its output can then be used to realize an absolute jump when the CUAL is chosen as the selected address mode. This is a logical function and not affected by the carry-in.

if c then CUAL else CUAL.

can be interpreted to mean:

if c then B else B.|||||

The specification of the CISB will be given *implicitly*. If one chooses the CUAL output as microinstruction address source, we write

CUAL + Carry-in.

Choice of this specification as either an A_t or A_f will dictate the setting of the CISB.

For the first interpretation of Example 1 to be valid the specification would have to be written

if c then CUAL else HERE.

whereas if we meant the second interpretation we would have to write

if c then CUAL+1 else HERE.

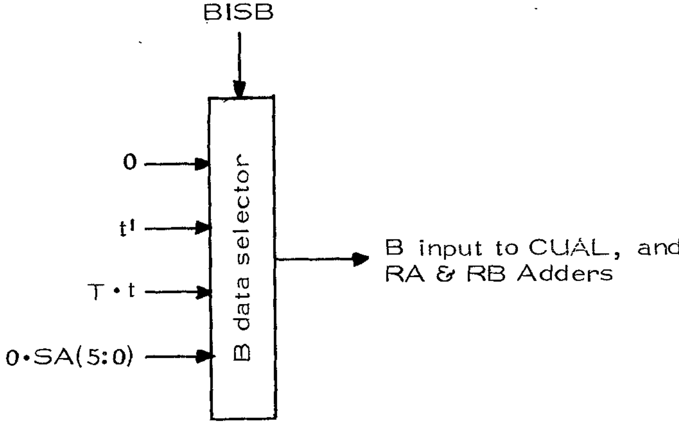
It should be obvious that the specification

if c then CUAL+1 else CUAL+1.

is an example of a microinstruction sequencing specification which is incompatible with the specification capability described above. Indeed if one wished to choose the address specification CUAL+1 irrespective of condition, one merely need write

CUAL+1.

in the microinstruction sequencing field of the microinstruction. This would have the same effect as writing, for example,



B data selection

Figure 2.35

Notation	B Data (12 bits)
0	A constant zero
t'	Sign-extended version of t
T • t	T concatenated with t
0 • SA(5:0)	The zero concatenated with the least significant part of SA

Table 2.26

B Data

if TRUE then CUAL+1.

where TRUE is a manifest system constant set to 1. There is also a manifest system constant, FALSE, which always has the value 0.

In order to complete the discussion of the CUAL we must discuss the specification of the data B. There are two 6-bit fields in the microinstruction which we shall call T and t. T and t are input into a selector along with 0 and SA(5:0) as shown in Figure 2.35, the output of which are shown in Table 2.26. There are 2 bits in every microinstruction, called the B-Input Selection Bits, BISB, which determine which of these computations will be used as the B data, if required, in the current address computation.

The notation t' means the 12 address bits are given by

$$t_5\, t_5\, t_5\, t_5\, t_5\, t_5\, t_5\, t_4\, t_3\, t_2\, t_1\, t_0,$$

i.e., in "sign extended" form. With the CUALF set to A+B and BISB='t" we then have a relative addressing capability of -31 to +32. The notations T • t and 0 • SA(5:0) denote concatenation.

The specification of the BISB will be given implicitly. One specifies the B value explicitly as a number, or as SA, in the address specification and this will dictate the setting of the BISB.

We can henceforth write the CUAL specifications as

CUAL (A,B) + Carry-in.

Both CU and A is redundant information since this is written in the microinstruction sequencing field of the microinstruction and we will use the shorter form

AL(B) + Carry-in

where B is a signed integer, $-2048 \leq B \leq 2047$, when combined in an arithmetic function with A, but may obviously lie in the interval $0 \leq B \leq 4095$ when used for absolute jumps.

Example 1
If the CUALF is set to A+B, then the specification

if c then AL(-18).

can be interpreted to mean

if c then HERE-18 else HERE+1.

where BISB is set to 't'' and CISB is set to $\bar{c}$.

Example 2
If the CUALF is set to A+B, then the specification

if c then AL(12)+1 else AL(12)

can be interpreted to mean

if c then HERE+13 else HERE+12

where BISB is set to 't'' and CISB is set to c, thus giving a conditional branch to one of two sequentially located microinstructions.

Example 3
If the CUAL is set to A+B, then the specification

if c then AL(SA) else SA

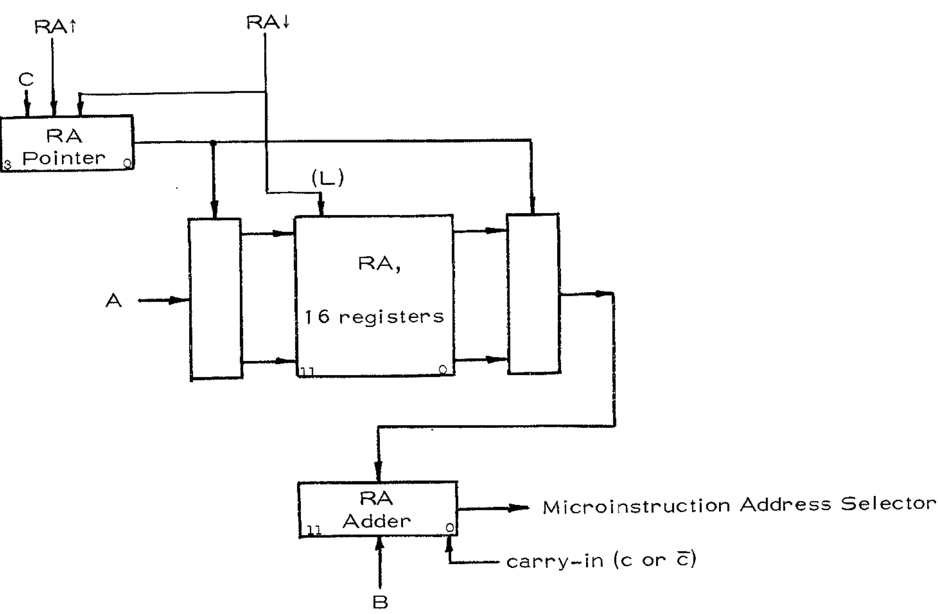
can be interpreted to mean

if c then HERE + SA(5:0) else SA(11:0)

where BISB is set to '0 • SA(5:0)' and CISB is set to $\bar{c}$.

Example 4
If the CUAL is set to B, then the specification

if c then AL(1975) else HERE-1



Return Jump Stack A, RA

Figure 2.36

Notation	Microoperation
RA ↓	Increment RAP and then Load RAS with the address of the current microinstruction (stack push)
RA ↑	Decrement RAP (stack pop)
RAPC	Clear the RAP

Table 2.27

Microoperations for control of RA

can be interpreted to mean

if c then 1975 else HERE-1

where the BISB is set to 'T • t' and the carry-in is not used since B is a logical CUAL function.

2.21.3 Return Jump Stack Facilities A and B

There are two return jump stack facilities associated with the microinstruction addressing facility. Each consists of

(1) a 12-bit wide, 16 element RG,
(2) a 4-bit wide RGP, and
(3) a 12-bit wide adder.

Figure 2.36 shows the return jump stack facility RA; RB, not shown, is identical. The microoperations associated with RA are shown in Table 2.27. The instructions for RB are identical.

Whenever the output of the RA adder is being used as the resulting address mode, the microoperation RA ↑ is executed. That is, the stack pointer is automatically maintained any time an address is added to the stack or whenever the RA adder is selected as the address mode. The use of RA is specified by writing

RA + B + carry-in.

This is seen immediately from Figure 2.36. The B data and the carry-in selection are exactly the same as those specified for the CUAL. The specification RA+1 or RB+1 will be interpreted to mean BISB='0' and the carry-in=1.

Example 1

Suppose we are in a routine at location n and wish to jump to a routine at location n+m. At location j of the second routine we wish to return to n+1. Assuming the CUALF:=B we could write

CS[n] ; RA ↓; AL(sub).

CS[sub]

CS[j] ; ; RA+1.

For the microinstruction at Location n, CISB='c̄' and BISB='t'' if $-32 \leq n \leq 31$ or BISB='T• t' otherwise. For the microinstruction at location j, CISB='c' and BISB='0'.

Example 2
It should be noted that the availability of 2 return jump stacks may facilitate the implementation of coroutines. For example, the microinstruction

CS[n] ; RA ↓; RB+1.

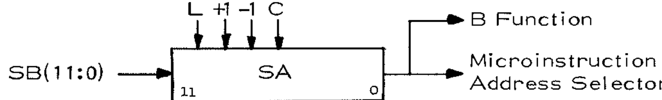
stores the current address, n, in one stack while simultaneously using RB as the selected address mode. RB is, of course, equal to RB+B+C=RB+0+1, i.e., BISB='0' and CISB='c'.

Example 3
A conditional return entry point can be obtained by using the specification

if c then RA+n else RA+n+1.

Here, CISB='c̄' and BISB='t'' if $-32 \leq n \leq 31$ or BISB='T • t' otherwise.

Example 4
Let us assume that a branch to a subroutine is required from a program segment and after completion a variable entry point return is required. In the example microcode following, the subroutine DECODE must place the displacement, d, in SA so that the computation RA+SA is in effect r+d.



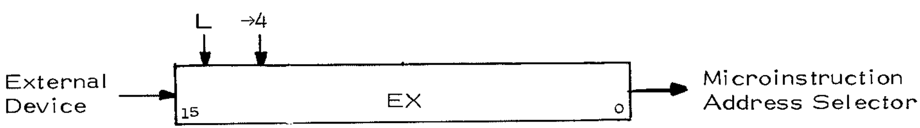
The Save Address Register, SA

Figure 2.37

SA:=SB
SA + 1
SA − 1
SAC

Table 2.28

Microoperations for control of SA



The External Register, EX

Figure 2.38

Notation	Microoperations
EX Load	Load the External register
EX → 4	Shift the External register 4 bits right cyclic

Table 2.29

Microoperations for control of EX

```
CS[n] PROG SEG:      .
                     .
                     .
CS[r]:               ;RA ↓    ;AL(sub)
                     .
                     .
                     .
CS[sub] DECODE:      .
                            .
                            .
                     ;      ; SA:=SB ; RA+SA
```

Here, CISB='c̄' and BISB='0•SA(5:0)'.

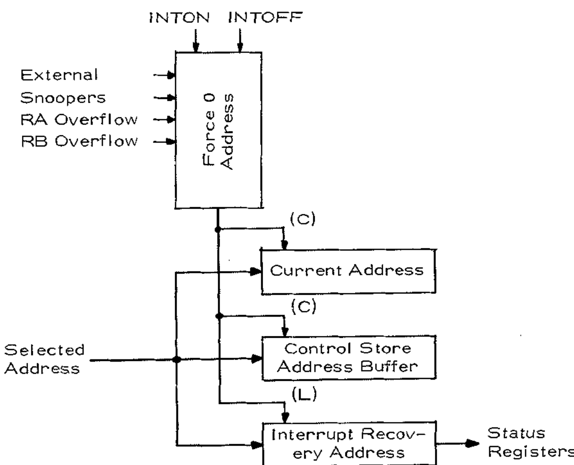
2.21.4 The Save Address Register

The Save Address register, SA, is shown in Figure 2.37. The microoperations associated with this register are shown in Table 2.28. SA provides a data path between the bus structure of MATHILDA and the control unit which controls the transactions on this structure. It can be used, for example, during the loading of control store, recovering from an interrupt, and in the B data computations.

2.21.5 The External Register

The External Register, EX, is a 16-bit wide right cyclic shifter which shifts 4 bit positions at a time. EX is loaded from an external device. If, for example, MATHILDA is to be connected as an input/output device to another processor, then the EX register provides one form of communications area for data sent to MATHILDA. EX is shown in Figure 2.38, the microoperations associated with EX are shown in Table 2.29.

EX can not only be used as a possible source for the address of the next instruction, but it can also be used as data for many of the control registers in the system, e.g., CA. When EX is to be used as the source of a microinstruction address, bits EX(11:0) are used. In fact, in all circumstances the data from the EX is always considered to be a contiguous string of bits of the required width starting with b_0.



The Force 0 Address Capability

Figure 2.39

Force 0 Address Conditions
External Signal
RA Overflow
RB Overflow
Snooper

Table 2.30

Force 0 Address Conditions

2.21.6 The Force 0 Address Capability

There are several conditions which if they occur during the execution of any microinstruction will disregard the address computation specified in the microinstruction sequencing portion of the microinstruction and fetch the next microinstruction from Control Store address 0. These conditions are listed in Table 2.30.

An external device may be connected to the External Signal condition to interrupt the operation of MATHILDA. If either RA or RB overflow, i.e., we have stacked more than 16 addresses, we will also force the address to 0. Finally the snoopers described in Section 2.23.2 can interrupt MATHILDA. This capability is shown in Figure 2.39.

Whenever a Force 0 Address Condition arises the following occurs: both the Control Store Address Buffer and the Current Address register are cleared, i.e., set to zero; the selected address is loaded into the Interrupt Recovery Address register, IRA; and the interrupt facility is turned off. The IRA contains the address of the microinstruction which would have been executed had the interrupt not occurred. The contents of the IRA can be gated onto the BUS through the Status Port explained in Section 2.17. . The IRA can then be used in conjunction with the SA facility previously described to restore the continuation address. The interrupt capability can be turned off and on by executing the microoperations INTOFF and INTON respectively.

2.21.7 The Microinstruction Address Bus

Having gained insight into the nature of the various address modes which can be used during microinstruction sequencing, we can now present a more detailed picture of the microinstruction address bus; it is shown as Figure 2.40. Because the number of control elements is small, they are also shown on this figure.

The microoperations associated with the control unit are brought together, for convenience, in Table 2.31. All but the last microoperations have been explained in previous sections. The CS Load operation is discussed next.

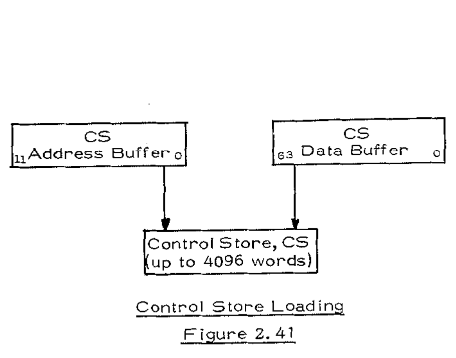
SA:=SB
SA + 1
SA - 1
SAC
CUALF:=CM
SET CUALF B
SET CUALF +
RA ↑
RA ↓
RAPC
RB ↑
RB ↓
RBPC
EX LOAD
EX → 4
INTON
INTOFF
CYL
CYS
CS LOAD

Table 2.31

Microoperations associated with the Control Unit

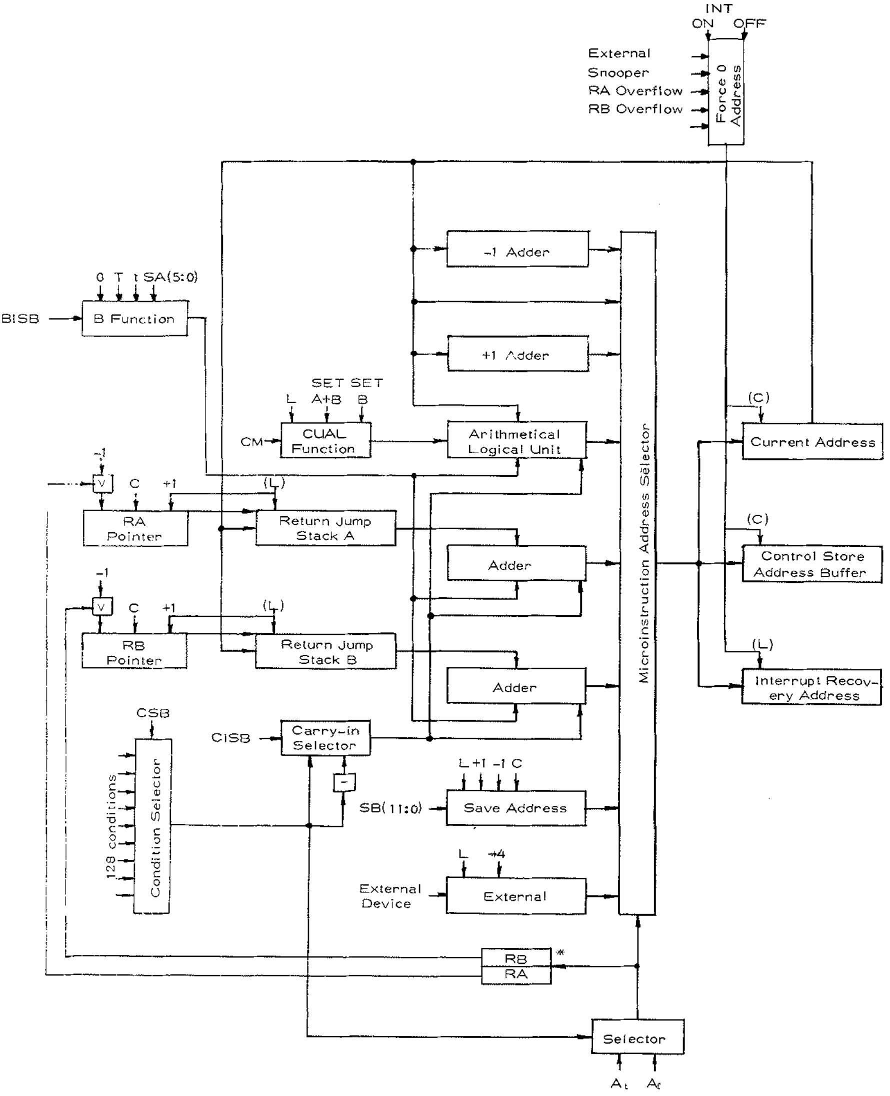
2.21.8 Control Store Loading

Control Store has, of course, both an address buffer and a data buffer, as shown in Figure 2.41. The CS Data Buffer is actually the OC register and as the CS Data Buffer is the only device no device selector is required. The CS Address Buffer is



Control Store Loading

Figure 2.41



Microinstruction Address Bus (Detailed)

Figure 2.40

* the address selector bits are decoded to determine if RA or RB are selected.

loaded from the output of the Microinstruction address selector as shown in Figure 2.40, which is, of course the address chosen as a result of the 'if c then A_t else A_f' evaluation. Let n be the address of the current microinstruction. The microoperation CS LOAD, if executed in the current microinstruction, can be interpreted as follows:

CS LOAD ::= Load the contents of the CS Data Buffer into the CS storage location pointed to by the CS Address Buffer *and then* choose n+1 as the address of the next microinstruction.

Example 1
The following microinstruction sequence might have been written to load CS[WA[0](11:0)] with WA[1]:

```
     ; WAPC, OCD:=8.
WA   ; SA:=SB, WAP+1.
WA   ; OC:=BUS                    ; if ¬OCSA then HERE.
     ; OCA.
     ; CS LOAD                    ;SA.|||||
```

However, the code which actually would be used to do this is as follows:

```
     ; WAPC.
WA   ; SA:=SB, WAP+1.
WA   ; OC:=BUS, CS LOAD           ; SA.|||||
```

This is because (1) the CS is the only device on OC so no device selection is necessary, (2) OC actually is the CS Data Buffer and space is always available thus eliminating the wait loop and (3) the CS writing is synchronized with the execution of the CS Load microoperation and thus no OCA is required to accomplish the normal synchronization, or load a device data buffer.

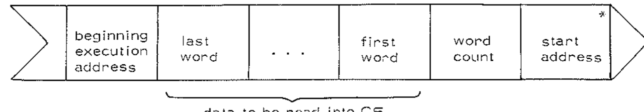
Example 2
The following example consists of microcode which will act as a loader, getting its information from a paper tape reader. The format of the tape is shown in the following diagram:

Unit	Symbolic Notation	Condition
AL	AL ALOV AL(0) AL(63) ONEOV TWOOV	AL(63:0) ≡ 1 ...... 1 AL carry-out and borrow-in bit bit 0 of AL input to bus selector bit 63 of AL input to bus selector 1's complement overflow 2's complement overflow
AS	AS(0) AS(V) AS(63)	bit 0 of the AS the variable bit of the AS bit 63 of the AS
BE	LSB1 MSB1 L1 L2 LD SGNLD LSBD SGNLSBD MSBD SGNMSBD BE(0) BEPGD	$LSB_1 \equiv 0$ $MSB_1 \equiv 63$ $L_1 \equiv 0$ (i.e., $MSB_1 = LSB_1$) $L_2 \equiv 0$ (i.e., $MSB_2 = LSB_2$) $L_1 = L_2$ sign $(L_2 - L_1)$ $LSB_1 = LSB_2$ sign $(LSB_2 - LSB_1)$ $MSB_1 = MSB_2$ sign $(MSB_2 - MSB_1)$ BE(0) ≡ 1 BEPG = '←'
BP	BP	BUS parity, BP=1 ⇒ odd parity
BUS	BUS	BUS(63:0) ≡ 0
CA	CA CA(0) CA(3) CA(4) CA(5) CA(6) CASPOV	CA zero bit 0 of CA bit 3 of CA bit 4 of CA bit 5 of CA bit 6 of CA CAP ≡ 1111 (CAP overflow)
CB	CB CB(0) CB(3) CB(4) CB(5) CB(6) CBSPOV	CB zero bit 0 of CB bit 3 of CB bit 4 of CB bit 5 of CB bit 6 of CB CBP ≡ 1111 (CBP overflow)
CR	CR	output of condition save registers

(continued)

Table 2.32

Partial Listing of System Conditions



* each rectangle is symbolic notation for eight 8-bit bytes.

Eight 8-bit bytes are read in and packed together to form a 64-bit word each time the subroutine labelled READ is called. The address where the first datum is to be written into CS is given as "start address". How many words are to be written into CS is given by word count, and the CS address at which execution is to begin after CS is loaded is given by "beginning execution address". The entry point for the loader is PTRLDR and it is assumed for the sake of this example that the reader is Device No. 4 on IB.

```
PRTLDR:                     ; IBD:=4, SET CUALF B   .Initialization
                            ; RA ↓                  ;AL(READ). Read start address
          AL                ; SA:=SB, RA ↓          ;AL(READ). Read Word Count
          AL                ; CB:=SB, RA ↓          ;AL(READ). Read 1st Word
LOOP:     AL                ; OC:=BUS, CS LOAD      ;SA.
                            ; SA+1, RA ↓            ;AL(READ)
                            ; CB-1                  ;if ¬ CB then AL(LOOP)
          AL                ; SA:=SB                ;SA. Read execution address and jump

READ:     AS,LR:=ALLOS      ;CA:=7                  .Initializations
FETCH:                      ;IBA                    .Request next 8-bit byte.
          LR:=IB            ;ALF:='AV B'            ;if ¬IBDA then HERE.
          AS:=AL,>8         ;CA-1,SETALFB           ;if ¬CA then AL(FETCH).|||||
                            ;                       ;RA+1
```

Asmall aside related to the notation in our previous examples: In Section 2.4.1 we had the following microinstruction sequences:

WAP:= 7.
if WA(63) then BITON.

Because at that time we did not wish to complicate the discussion with the details of the microinstruction sequencing unit we did not write:

(Continued)

Unit	Symbolic Notation	Condition
CU	EXDA	data available on EX
	RAPOV	RAP ≡ 1111 (RAP overflow)
	RAPUN	RAP ≡ 0000 (RAP underflow)
	RBPOV	RBP ≡ 1111 (RBP overflow)
	RBPUN	RBP ≡ 0000 (RBP underflow)
	INT	INT = 1 ⇒ INTON, INT = 0 ⇒ INTOFF
	CUALOV	CUAL overflow
	CYL	processor in "long Cycle" mode
DS	DS(i), i=0, . . . ,15	the indicated bit of the DS
	DS(j), j=V, V+1	the variable bits of the DS
I/O	IADA	data available on IA
	IADM	data condition on IA (Data Mark)
	IBDA	data available on IB
	IBDM	data condition on IB (Data Mark)
	OASA	space available on OA
	OBSA	space available on OB
	OCSA	space available on OC
	ODSA	space available on OD
KA	KA	state of console switches
KB	KB	
KC	KC	state of programmable switches
KD	KD	
LR	LR(0)	bit 0 of LR input to bus selector
	LR(63)	bit 63 of LR input to bus selector
SB	SB(0)	bit 0 of the shifted bus
	SB(1)	bit 1 of the shifted bus
	SB(62)	bit 62 of the shifted bus
	SB(63)	bit 63 of the shifted bus
System	TRUE	a binary one
	FALSE	a binary zero
VS	VS(0)	bit 0 of the VS
	VS(V)	the variable bit of the VS
	VS(63)	bit 63 of the VS
WA*	WA(0)	bit 0 of WA input to bus selector
	WA(63)	bit 63 of WA input to bus selector
	WAPOV	WAP ≡ 1111111 (WAP overflow)
WB*	WB(0)	bit 0 of WB input to bus selector
	WB(63)	bit 63 of WB input to bus selector
	WBPOV	WBP ≡ 1111111 (WBP overflow)

Table 2.32

Partial Listing of System Conditions

* See also Table 2.38.

; WAP:=7.
; if WA(63) then AL(BITON).

Wherever such labels were used, we had actually meant them to be shorthand for AL(label). It is now seen that we had assumed the microcoder had set the CUAL function to either A+B or B prior to the execution of the instruction containing the label in the <instruction sequencing> port. If the CUAL had been set to B, then the address of the CS location labelled BITON would have been used as the B-data resulting in an absolute jump. Whereas, if the CUAL had been set to A+B then the value of (BITON-HERE) would have been used as the B-data resulting in a relative jump. Actually, our examples have been given assuming that the CUAL has been set to B.

2.22 The Conditions, Condition Selector, and Condition Registers

There is the possibility of testing 128 conditions in the system. At this writing approximately 100 have been specified, leaving a reasonable amount of expandability in the system. The conditions and their symbolic notation are given in Table 2.32. The conditions in this table are grouped according to the functional unit with which they are associated. For convenience, the units are listed in alphabetical order.

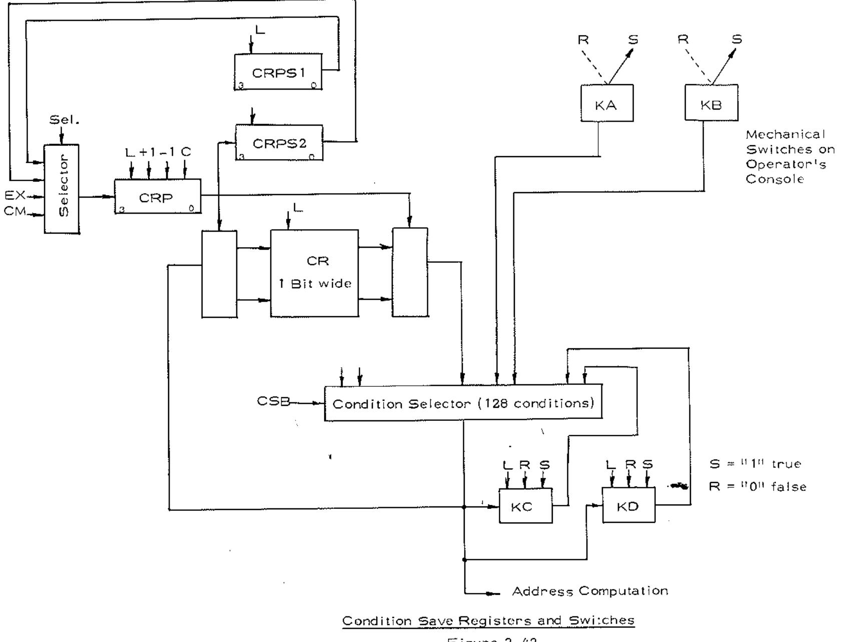
All 128 conditions are input into a condition selector. There are 7 bits in each microinstruction, called the Condition Selection Bits, CSB, which select a particular condition. The selected condition is input into

a) the A_t -A_f address selector (Section 2.21.1)
b) the carry-in selector (Section 2.21.2), and
c) a SG called the Condition Save Registers, CR.
d) two programmable switches, KC and KD.

This is shown in Figure 2.42. It can be seen from this figure that we can save the state of any condition as it arises and use it later when required. The microoperations associated with CR, KC and KD are given in Table 2.33.

Switches exist in two variants: KA and KB are console-switches, KC and KD are programmable switches that also can be loaded with the selected condition.

In the loading microoperation CR:=<SC>, Selected Condition, we can, instead of



Condition Save Registers and Switches
Figure 2.42

CR:=<SC>
CRP:=CM|EX|S1|S2
CRP + 1
CRP - 1
CRPC
CRPS1:=CM|EX|S1|S2
CRPS2:=CRP
KC:=<SC>
SET KC
CLEAR KC
KD:=<SC>
SET KD
CLEAR KD

Table 2.33
Microoperations for control of CR, KC and KD

Notation	Microoperation
CYL	Set cycle to long cycle
CYS	Set cycle to short cycle

Table 2.34
Microoperations to control the length of cycle

using the notation SC, use the symbolic notation given in Table 2.32. Thus, for example, if we wished to save the state of the ALOV condition in an instruction we would write:

CR:= ALOV

It should be obvious that since the SC goes to both the CR and the A_t -A_f selector one cannot specify a condition in the microinstruction sequencing field different from the SC in the CR:=<SC> microoperation within the same instruction. Thus

WA:=WB; WAP+1, CR:=BUS; if CA then RA+1.

is *not allowed*. It would have to be written as 2 microinstructions:

WA:=WB ; WAP+1, CR:=BUS.
; if CA then RA+1.

Microinstructions of the following type are obviously allowed:

WB:=DS; PG→3, AS←, CR:=BP; if BP then HERE-1.

2.22.1 Short and Long Cycle

It is obviously important to know when one can test a condition. The system can execute microinstructions in two different cycle times: a "short" cycle time and a "long" cycle time. The difference in these two cycles as it relates to the testing of conditions can be easily stated:

long cycle
When the machine is operating in long cycle mode *all* conditions which arise as a result of bus transport and microoperation execution are testable in the *same* microinstruction in which they arise,

short cycle
When the machine is operating in short cycle mode *all* conditions which arise as a result of bus transport and microoperation execution are testable in the *next* microinstruction to be executed.

Thus, if we are in long cycle and we write

WA:=WB; WAP+1; if BUS then RA+1.

we are testing whether or not if the current bus transport (WA:=WB) is such that BUS=0. Whereas, in short cycle, this microinstruction would mean we are testing the previous bus transport's condition. In order to test WA:=WB we would have to write 2 microinstructions,

```
WA:=WB ; WAP+1.
       ; if BUS then RA+1.
```

Thus, a microinstruction can be thought of being executed in the following sequential way:

Long cycle:

a) execute bus transport
b) execute microoperations
c) execute microinstruction sequencing based on the current conditions.

Short cycle:

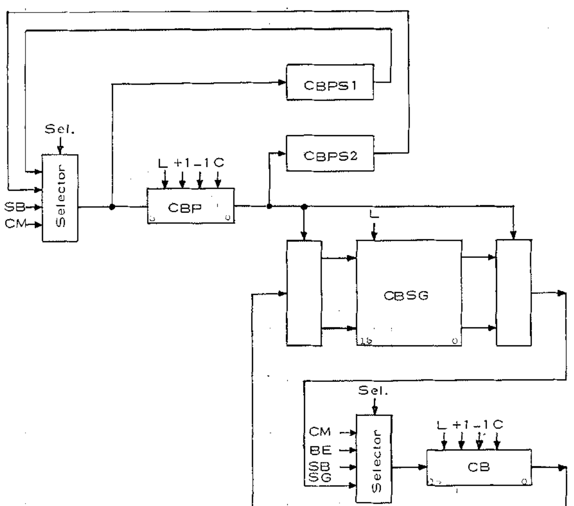
a) delay the conditions of the previous microinstruction
b) execute bus transport
c) execute microoperations
d) execute microinstruction sequencing based on the delayed conditions from the previous microinstruction.

It is obvious that all of the examples given previously have been executed in the "short cycle" mode (see the discussion in Section 2.4.1). This is, of course, the more difficult of two concepts; however, a reader who has started the document from the beginning should now be intuitively familiar with this concept.

There are two microoperations which allow the setting of the machine cycle as shownin Table 2.34. If either microoperation is executed in microinstruction M, it means that

a) the length of the cycle for instruction M *does not* change,
b) all subsequent microinstructions will be executed in the cycle length specified in microinstruction M until a change of cycle is initiated.

The condition CYL can be used to determine the length of the current cycle. If CYL=1, i.e., TRUE, this means the machine is currently operating in long cycle,



Counter B, CB
Figure 2.43

CB:=CM|BE|SB|SG
CB + 1
CB – 1
CBC
CBSG:=CB
CBP:=CM|SB|S1|S2
CBPS1:=CM|SB|S1|S2
CBPS2:=CBP
CBP + 1
CBP – 1
CBPC

Table 2. 35
Microoperations for control of CB, CBP, and CBSG

otherwise, it is operating in short cycle.

2.23 Auxiliary Control Facilities

The auxiliary control facilities associated with the MATHILDA processor as shown in Figure 2.1, i.e., the system counters and snooper facilities, will now be discussed.

2.23.1 Counter B

The system has 2 counters associated with it: Counter A, CA, has been introduced in Section 2.2, Counter B, CB, introduced here, is shown in Figure 2.43. A comparison of this figure with Figure 2.3 which shows CA shows that CB is identical with CA except that CA can be loaded from the EX register whereas CB can be loaded from the output of the BE, i.e., we have

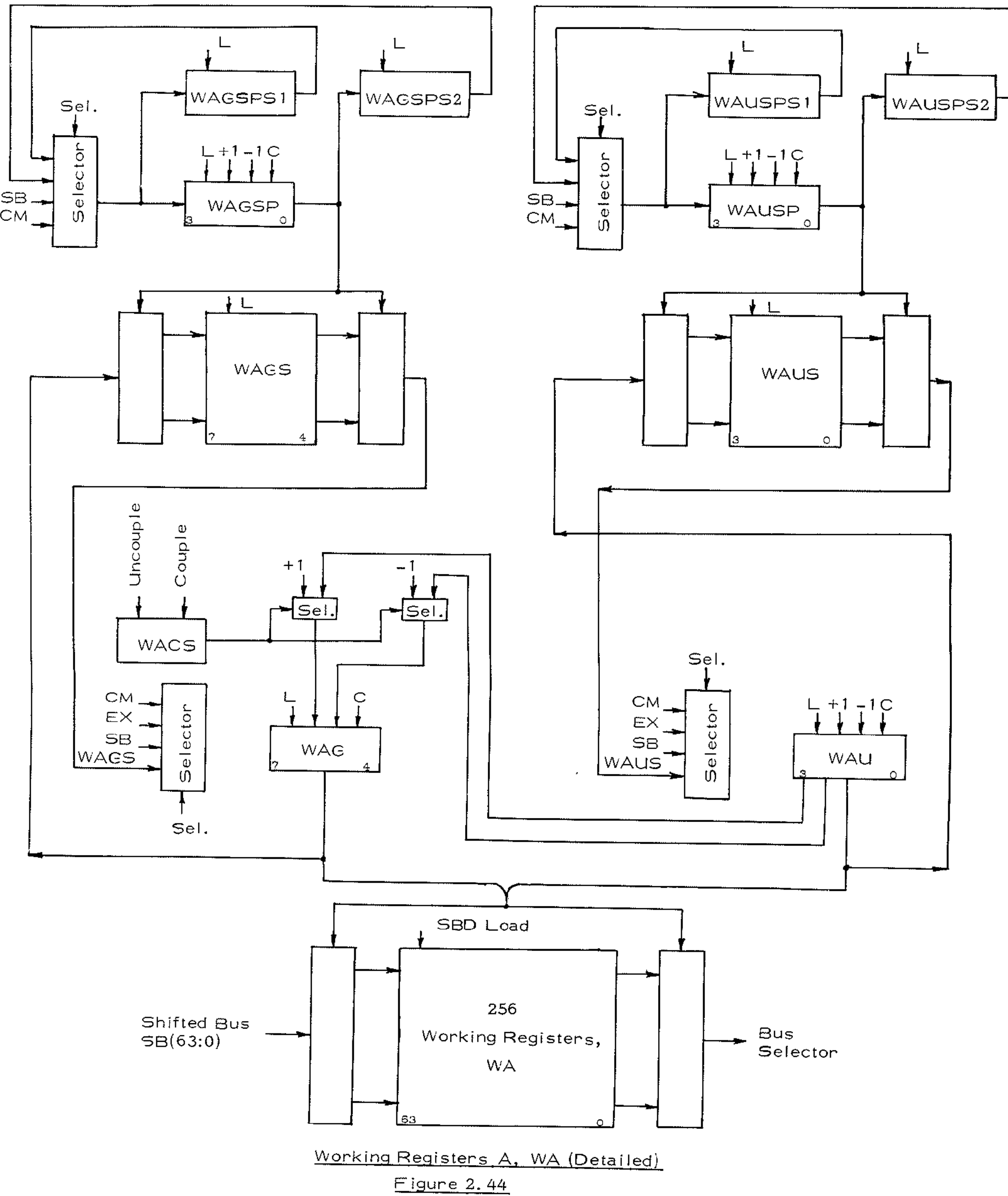
CA:=CM|EX|SB|SG, and CB:=CM|BE|SB|SG.

Note, the output of the BE is 6 bits, whereas CB is 16 bits wide. Whenever BE is selected as input CB the high order 10 bits of CB are set to 0. The microoperations associated with CB, CBSG, and CBP are given in Table 2.35. These are, of course, apart from the above difference, identical to those associated with CA and merely shown here for convenience. An example of the use of CB has been given as example 2 in Section 2.16. It should be quite obvious that CA and CB may be used independently of one another. One may count up in CA while counting down in CB, for example,

CA+1, CB-1. .

2.23.2 The Snooper Facility

The Snooper Facility consists of **a)** a Snooper Control Store and **b)** Snooper Resources (e.g. 2 groups of 16-registers, counter, and comparators). The Snooper unit works in the following way: when the address of the next microinstruction to be executed is sent to the MATHILDA Control Store address buffer, it is also gated into the Snooper Control Store address buffer. At the same time the microinstruction is fetched so that it can be executed, the contents of its associated Snooper Control Store location is fetched; in parallel with the microinstruction being executed, the contents of its associated Snooper Control Store just fetched is used to control the operation of the Snooper Resources. Snooper Control Store (80 nanosecond storage) is 16-bit wide and has the same number of words as the MATHILDA Control Store. A snooper word can specify, for example, any two registers which can be counted up (or down). The Snooper Facilities can be written through the



Working Registers A, WA (Detailed)
Figure 2.44

normal ports of the system, and be read through the Status Facility. Snooper Control Store is writable so that different data gathering routines can be associated with the same segment of microcode *without* changing the microcode. The user is allowed to establish the correspondence between any particular snooper resource and the routine upon which it is snooping. A more complete description of the Snoopers is given in [9].

2.24 An Alternative View of the Working Registers

The description of WA which was given in Section 2.4 introduced WA as a 256 element RG. In Figure 2.6 the address pointer, WAP, was shown to be 8-bits wide so that the WA registers could be addressed as 256 contiguous registers. In fact, the address pointer actually consists of two 4-bit pointers which had been "coupled" together to give the 8-bit wide pointer described in Section 2.4.. Figure 2.44 shows WA with its two 4-bit pointers called the Group and Unit pointer; WB, not shown, is identical.

When the microoperation WAP COUPLE is executed, the Group and Unit pointers are connected together to give the 8-bit wide pointer, WAP. After the microoperation WAP UNCOUPLE is executed, the Group and Unit pointers function as independent pointers. The low order 4-bits of the 8-bit address required to specify a particular register are given by the WA Unit pointer, WAU; the high order 4-bits of the address are given by the WA Group pointer, WAG. Thus, WA can be considered to be 16 RG's, each RG having 16 registers.

The microoperations associated with the WAU and WAG pointers are given in Table 2.36. (The similar microoperations for WB are not shown.)

If we wanted to point to the 9th unit of group 3 and then transfer its contents to the DS, we could write, assuming the pointers are uncoupled,

```
              ; WAG:=3, WAU:=9.
DS:=WA .|||||
```

The microoperations associated with WAP in Table 2.4 can now be given their appropriate meaning in terms of the microoperations in Table 2.36. Assuming WAU and WAG are coupled, we have

<table>
<tr><td>WAU:=CM|EX|SB|SG</td></tr>
<tr><td>WAU + 1</td></tr>
<tr><td>WAU − 1</td></tr>
<tr><td>WAUC</td></tr>
<tr><td>WAG:=CM|EX|SB|SG</td></tr>
<tr><td>WAG + 1</td></tr>
<tr><td>WAG − 1</td></tr>
<tr><td>WAGC</td></tr>
<tr><td>WAPCOUPLE</td></tr>
<tr><td>WAPUNCOUPLE</td></tr>
</table>

Table 2.36

Microoperations for control of the WAU and WAG pointers

<table>
<tr><td>WAUS:=WAU</td></tr>
<tr><td>WAUSP + 1</td></tr>
<tr><td>WAUSP − 1</td></tr>
<tr><td>WAUSPC</td></tr>
<tr><td>WAUSP:=CM|SB|S1|S2</td></tr>
<tr><td>WAUSPS1:=CM|SB|S1|S2</td></tr>
<tr><td>WAUSPS2:=WAUSP</td></tr>
<tr><td>WAGS:=WAG</td></tr>
<tr><td>WAGSP + 1</td></tr>
<tr><td>WAGSP − 1</td></tr>
<tr><td>WAGSPC</td></tr>
<tr><td>WAGSP:=CM|SB|S1|S2</td></tr>
<tr><td>WAGSPS1:=CM|SB|S1|S2</td></tr>
<tr><td>WAGSPS2:=WAGSP</td></tr>
</table>

Table 2.37

Microoperations for control of WAUS and WAGS

Unit	Symbolic notation	Condition
WA	WAUOV	WAU ≡ 1111 (WAU overflow)
	WAGOV	WAG ≡ 1111 (WAG overflow)
	WACS	WACS = 1 ⇒ WAU and WAG are coupled
WB	WBUOV	WBU ≡ 1111 (WBU overflow)
	WBGOV	WBG ≡ 1111 (WBG overflow)
	WBCS	WBCS = 1 ⇒ WBU and WBG are coupled

Table 2.38

Additional WA and WB Conditions

WAP+1 ::= WAU+1
WAP-1 ::= WAU-1
WAPC ::= WAUC and WAGC
WAP:=CM|EX|SB|SG ::= WAU:=CM|EX|SB|SG and WAG:=CM|EX|SB|SG

Let us now turn our attention to the pointer save capability shown in Figure 2.44. When WA is considered as 16 groups of 16 registers, the WAU and WAG pointers may be saved independently of one another. The microoperations associated with this facility are given in Table 2.37. As an example, suppose we are in group 3 and wish to work in group 8. Before working in group 8 we want to save the unit to which we are pointing in group 3. This is done by executing

WAUS:=WAU, WAG:=8. .

The microoperations associated with WAPS in Table 2.4 can now be given their appropriate meaning in terms of the microoperations in Table 2.39. Thus we have,

WAPS:=WAP ::= WAUS:=WAU and WAGS:=WAG
WAPSP+1 ::= WAUSP+1 and WAGSP+1
WAPSP-1 ::= WAUSP-1 and WAGSP-1
WAPSPC ::= WAUSPC and WAGSPC.

Let psource = CM|SB|S1|S2 then

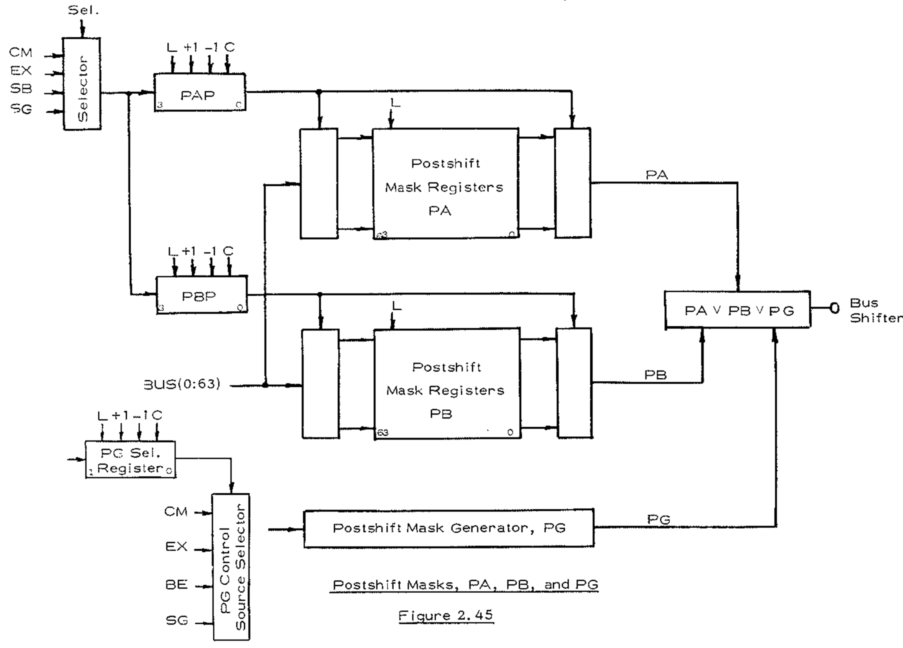
WAPSP:= psource ::=WAUSP:= psource and WAGSP:= psource
WAPSPS1:= psource ::=WAPSPS1:= psource and WAGSP1:= psource
WAPSPS2:=WAPSP ::=WAUSPS2:=WAUSP and WAGSPS2:=WAGSP

There are a few additional conditions which can now be added to Table 2.32, the partial listing of symbol conditions. These are given in Table 2.38.

Thus we can deal with WA or WB as either 256 contiguous registers or 16 groups of 16 registers. We can switch back and forth between either interpretation in a relatively straightforward way.

2.25 An Alternative View of the Postshift Masks

The description of the Postshift Masks which was given in Section 2.7 was structured to make the Postshift Masks look as much like the Bus Masks as possible, to enhance the understanding of this unit. In fact, the output of the BS is masked



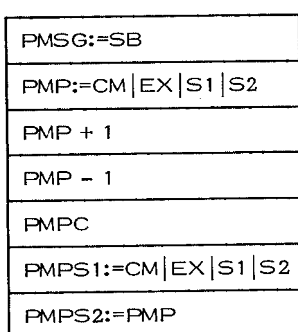
Postshift Masks, PA, PB, and PG

Figure 2.45

PB:=BUS
PBP:=CM|EX|SB|SG
PBP + 1
PBP - 1
PBPC

Table 2.39

Microoperations for control of PB



PMSG:=SB
PMP:=CM|EX|S1|S2
PMP + 1
PMP - 1
PMPC
PMPS1:=CM|EX|S1|S2
PMPS2:=PMP

Table 2.40

Microoperations for control of PMSG

during *every* bus transport by the mask which is specified to be

$$PM \vee PG$$

where

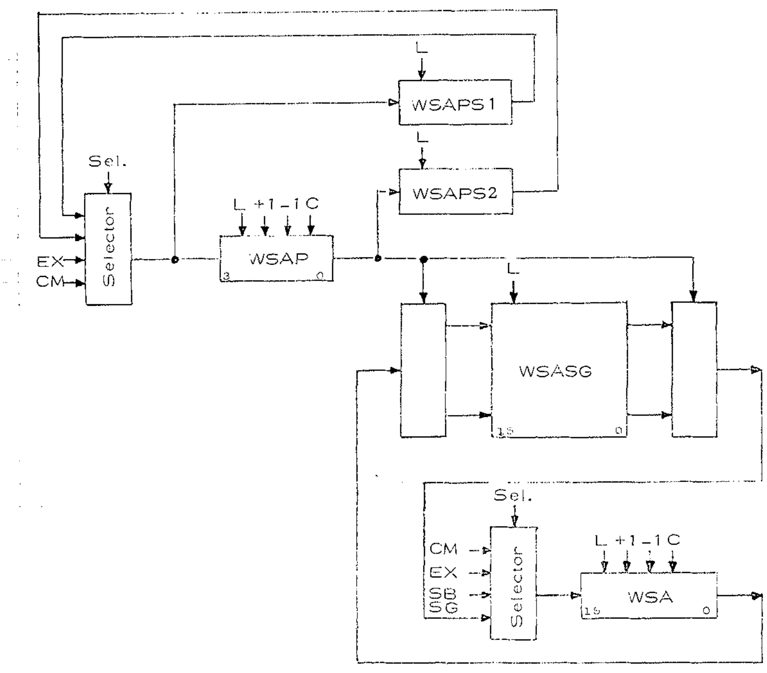
PM = PA ∨ PB
PA = an element of a 64-bit wide, 16 element RG called the Postshift Mask A registers
PB = an element of a 64-bit wide, 16 element RG called the Postshift Mask B registers
PG = the Postshift Mask Generator
∨ = logical "inclusive or".

In Section 2.7 we had introduced the mask to be PA∨ PG; here we had merely assumed all elements of PB to contain all 0's, implying that PM=PA. The actual situation is shown more clearly in Figure 2.45. The most important thing to note from this diagram is that the PA/PB structure is indeed the same as the MA/MB structure (see Figure 2.9). The microoperations associated with PB are shown in Table 2.39.

The name of the common SG associated with the PA pointer and the PB pointer is the Postshift Mask Standard Group, PMG. The microoperations associated with this SG are given in Table 2.40. We will assume that all elements of PB contain all 0's so that the effective mask is PA∨ PG and all of our previous standardizations for the use of this facility are still valid.

2.26 Wide Store Address

Mathilda is connected to a memory system called the Wide Store (WS). It is a 64 bit wide core memory (at present 32K words), which is also connected to other processors (e.g. RIKKE-1). Wide Store is operated as an i/o device, connected through the IA and OA data ports as the only device on these ports. Hence IAD and OAD have no interpretation, as no device selection is needed.



Wide Store Address, WSA

Figure 2.46

WSA
WSA := CM|EX|SB|SG
WSA +1
WSA -1
WSAC
WSASG
WSASG := WSA
WSAP := CM|EX|S1|S2
WSAP +1
WSAP -1
WSAPC
WSAPS1 := CM|EX|S1|S2
WSAPS2 := WSAP

Table 2.41

Microoperations for control of WSA

Symbolic notation	Condition
WSAB	Wide Store Address busy
WSAOR	WSA out of range
WSASPOV	WSAP = 1111 (WSAP overflow)

Table 2.42

WSA Conditions

IA thus acts as a memory buffer register on reading WS, and OA acts as the buffer register for writing WS. As memory address register there is a 16 bit register called WSA (Fig. 2.46) which looks very much like Counter A. WSA has an associated register group WSASG which can be used to save the contents of WSA for a possible later rewriting. WSA can also be loaded from CM, SB, and EX, and its contents can be incremented, decremented and cleared.

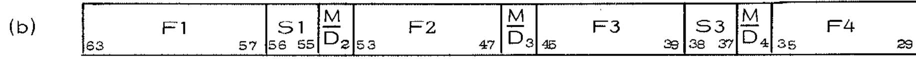
There is a condition (WSAOR) available to test whether the actual contents of WSA correspond to an address outside the physical address space. Also, since WSA interacts with an asynchronous device, there is a special condition WSAB available to test whether Wide Store has "used" the address in WSA.

WSAB acts as a "busy-flag" for WSA, and will be true from the moment a memory transfer has been requested and until WS has read the contents of WSA. Thus changing the contents of WSA while WSAB is true may result in strange effects.

Since it is not the purpose of this description to explain the details of the operation of Wide Store, we will refer the reader to other documentation on the memory system. Since WS is a resource which is shared with other processors, and erroneous operation may interfere with these, access to WS has to be done with extreme caution. It is advised to use available "system routines" for such purposes.

(a)

(b)



mop	Sel.	'M'	mop	'M'	mop	Sel.	'M'	mop
		'D'	data	'D'	$data_{BS}$		'D'	data

(c)

BISB	CISB	CSB	A_f	A_t
15 14	13	12 6	5 3	2 0

(d)

AS	VS	DS
28 27	26 25	24 23

Subfields of Microinstruction

Figure 3. 1

SOURCE	SBD
Symbolic Notation	Symbolic Notation
SP	no destination
AL	MA
VS	MB
DS	LR
WA	WA
WB	WB
IA	OA
IB	OB

Table 3.1

Symbolic Notation for SOURCE's and SBD's

3.0 Microinstruction Specification and Execution

We will in this section discuss the microinstruction format, the manner in which the instruction is executed, and then give a comprehensive table of all microoperations.

3.1 Microinstruction Format

Microinstructions are 64-bits wide. There are 4 major fields in a microinstruction. These fields specify

a) MDP transport (7 bits)
b) microoperations (mops) and data (35 bits)
c) microinstruction sequencing (16 bits)
d) control of AS, VS, and DS (6 bits)

These fields are shown in Figure 3.1 with their sub-fields named and their actual bit location in the microinstruction. Let us discuss each of them in more detail.

(A) The MDP Transport Field

Table 3.1 summarizes the symbolic notation for SOURCE's and SBD's. If the BS Enable bit='NBS', no bus shift occurs; if the BS Enable bit='BS' a bus shift occurs. As an example, the MDP transport specification WB:=DS will be shown symbolically as

BSE	SBD	SOURCE
'NBS'	'WB'	'DS'

(B) The microoperations and Data Field

The microoperations and data field can be considered to be made up of the following fields:

$$F_1, S_1, M/D_2, F_2, M/D_3, F_3, S_3, M/D_4, F_4$$

as shown in Figure 3.1.

The following comments should assist in understanding this diagram.

(B.1) Field F_1 always specifies a microoperation activation (1 of 128 mops)

if M/D_2 ='M' then F_2 specifies a microoperation activation (1 of 128 mops).
if M/D_3 ='M' then F_3 specifies a microoperation activation (1 of 128 mops).
if M/D_4 ='M' then F_4 specifies a microoperation activation (1 of 128 mops).

Therefore up to 4 microoperations may be specified in this field; for example,

; BSP+1, WBP+1, MBP+1, CA-1;

(B.2) We have seen that many microoperations concern the loading of a register from various sources, e.g.,

MAP:=CM|EX|SB|SG.

Such a mop must be placed either in field F_1 or F_3. If it is placed in F_1, then the 2 selection bits S_1 specify which source will be used. If the source specified is the CM then M/D_2 is set to 'D' and F_2 is used as data (similarly M/D_4 and F_4 are used with F_3). For example

MAP:=7

could be symbolically represented

F_1	S_1	$\frac{M}{D_2}$	F_2
'MAP:='	'CM'	'D'	'7'

Thus one sees that there can be at most 2 microoperations of this type in a microinstruction.

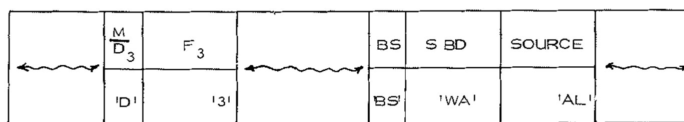
(B.3) Figure 3.1 also shows that if the BS control data is to be taken from the CM then F_3 is used as data. If the BS has been enabled, the control source is selected via field S_3. Thus the specification

A_f and A_t
Symbolic Notation
EX
AL
RB
RA
SA
A-1
A+1
A

Table 3.2
Symbolic Notations for A_t and A_f

WA:=AL, → 3

could be symbolically represented



(B.4) All of the possible microoperations are not available in each field F_1, F_2, F_3, and F_4. The microoperations which can be specified in each field are given in Section 3.3, the Comprehensive Tables of Microoperations for Individual Functional Units.

(C) The Microinstruction Sequencing Field
Table 3.2 summarizes the symbolic notation for A_t and A_f. Table 2.26 presents this information for the BISB (B-input selection bits).

Example 1) If BUS then HERE. could be represented

BISB	CISB	CSB	A_f	A_t
'0'		'BUS'	'A+1'	'A'

Example 2) If ALOV then RA+12. could be represented

BISB	CISB	CSB	A_f	A_t
't'	'$\bar{c}$'	'ALOV'	'A+1'	'RA'

However, this is incomplete and immediately raises the question where do T and t

Symbolic Notation	Shift/Load Control
\ /	Do Nothing
→	Shift Right
←	Shift Left
:=	Load

Table 3.3
Shift/Load Control Bits

come from? T is always the least significant 6 bits of F_3 and t is always the least significant 6 bits of F_4. BISB tells us, of course, how we will combine T, t, SA(5:0), and to yield the B-data. Thus the complete specification would be

↜↝	$\frac{M}{D_4}$	F_4	↜↝	BISB	CISB	CSB	A_f	A_t
	'D'	'12'		't'	'c'	'ALOV'	'A+1'	'RA'

(D) AS, VS, and DS Control Field.
The dedicated bits for shifter control are interpreted as shown in Table 3.3. Thus, the specification AS→, VS←, DS← could be represented symbolically as

AS	VS	DS
'→'	'←'	'←'

The specification AS, LR:=AL; DS← . would be given by

↜↝	AS	VS	DS	BS	SBD	SOURCE	BISB	CISB	CSB	A_f	A_t
	'L'		'←'	NBS	'LR'	'AL'	'0'		'TRUE'	'A+1'	'A+1'

3.2 Microinstruction Execution

As introduced in Section 2.4.1 and then explained in more detail in Section 2.22.1, the machine has both a long cycle and a short cycle. The result of that discussion, which is repeated here for convenience is that microinstructions can be thought of being executed in the following sequential way:

long cycle:

A) microinstruction fetch
B) MDP transport
C) execute microoperations
D) microinstruction sequencing based on current conditions

short cycle:

A) microinstruction fetch
B) MDP transport
C) execute microoperations
D) microinstruction sequencing based on delayed conditions.

Let us now examine each of the sequential steps in more detail.

A) Microinstruction fetch

1: Fetch the content of the CS [CS Address Buffer].
2: In short cycle save the value of all testable conditions.

B) MDP Transport

1: Selection of source
2: Masking by MA $\vee$ MB
3: Buffering in the BUS-latch
4: Shifting by BS if enabled
5: Masking by PA $\vee$ PB $\vee$ PG
6: Buffering in the SB-latch
7: Loading into a selected MDP destination

C) Microoperation Execution

1: Gate the data from S-fields and from F2, F3, F4 fields to their destinations irrespective of their expected or non-expected use.
2: Decode the F-fields (if enabled by M-fields).
3: Activate (clock) the specified clock 1-mops. (See Section 3.2.1.)
4: Activate the specified load/shift actions in AS, VS, and DS.
5: Activate (clock) the specified clock 2-mops. (See Section 3.2.1.)

D) Microinstruction Sequencing

1: Choosethe selected condition, c. [In short cycle c is the value of the selected condition prior to MDP transport, in long cycle use the new value of the condition.]
2: Select the carry-in and B-input into the return jump stack adders and the CUAL.
3: Compute the results of the return jump stacks' additions and the CUAL function.
4: Select the new address, using A_t if c=1 or A_f if c=0 as the microinstruction address bus selection, and load the address-buffers.
5: If RA and RB have been selected then pop the appropriate stack(s).

6: If a force-zero situation has occurred then load the IRA, and clear the address-buffers.

3.2.1 Clock Pulse 1 and Clock Pulse 2

Recall that the RG is a basic building element used in the system. A very common operation is to load an RG and then change its pointer (e.g. this was done quite frequently in our examples). Often, one also wished to save the address of the current element pointed to before the pointer is changed. It was decided that this capability should be allowed in one microinstruction and, furthermore, *every* RG in the system should be treated in the same uniform way.

The microinstruction

AS:=WA; WAPS:=WAP, WAP+1

means: take the contents of WA[WAP] and store it in the AS; then store the WAP in the WAPS registers and then increment WAP by 1. It means this because the BD load occurs in step B.7 and the microoperation WAPS:=WAP occurs at clock 1 and the microoperation WAP+1 occurs at clock 2 during step C. Thus, every RG in the system can be looked at in the following way:

a) it can be loaded or used as a source
b) its current pointer can be saved, if it has a save capability
c) its pointer can be changed after a) and b);

all with one microoperation.

An additional example is

WB:=AL, →; SET ALF+, WBU:=9, BSSC. |||||

which means, assuming the BSS:='BE': store the output of AL in WB[WBP] after shifting it the amount specified by the BE, change the ALF to A+B and change the WBU to 9, reset the BSS to 'CM' and then go to the next microinstruction.

3.3 Comprehensive Tables of Microoperations for Individual Functional Units

The following tables (represented in alphabetical order based on the abbreviations associated with the functional unit) show which microoperations can appear in which fields and at which clock pulse these microoperations are initiated. In these tables we use the following notation:

XX = EX|SB|SG
VV = BE|SB|SG
YY = SB|S1|S2
ZZ = EX|S1|S2
EE = EX|SB

Some particular points perhaps should be recalled and emphasized here:

a) use of these tables will show what space and time conflicts arise in the construction of a microinstruction. The reader is encouraged to review some of the examples of the earlier sections by constructing symbolic microinstructions similar to those presented in Section 3.1.

b) t comes from field F_4, so if t is being used, for example in relative addressing, a microoperation should not be specified in F_4.

c) T comes from field F_3, so if T is being used, for example in absolute addressing, a microoperation should not be specified in F_3.

d) data for the BS, if the CM is the control source, comes from F_3.

e) data for the PG, if the CM is the control source, comes from F_2.

Inside the table, ⊖ means, that the microoperation is not yet implemented in that field.

Outside the table, ⊖ means, that the microoperation is not yet implemented.

MICROOPERATIONS FOR Arithmetic Logical Unit, AL

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D2	F2	M/D	F3	S3	M/D	F4	MICROOPERATION	
2					M	ALP :=	ZZ CM	D	d d d d	Load the AL SG Pointer from CM|EX|S1|S2	
2					M	ALP +1				Increment AL SG Pointer	
2					M	ALP -1				Decrement AL SG Pointer	
2					M	ALPC				Clear AL SG Pointer	
2			M	ALPS1:=	M	ALPS1:=	ZZ CM	D	d d d d	Load the AL SG Save1 register from CM|EX|S1|S2	
1	ALPS2:=ALP									Load the AL SG Save2 register from the AL SG Pointer	
1			M	ALSG:=SB						Load the AL SG with SB(5:0)	
2	ALF:=	XX CM	D	d d d d d d						Load the AL Function register from CM|EX|SB|SG	
2			M	SET ALF +						Set AL Function to A+B (= LR+AS)	
2			M	SET ALF -						Set AL Function to A-B (= LR-AS)	
2			M	SET ALF A				M	SETALF A	Set AL Function to A (= LR)	
2			M	SETALF +1						Set AL Function to A+1 (= LR+1)	
2			M	SETALF B				M	SETALF B	Set AL Function to B (= AS)	
2			M	SETALF ALL0S				M	SETALF ALL0S	Set AL Function to generate 00. . . . 0	⊖
2			M	SETALF ALL1S				M	SETALF ALL1S	Set AL Function to generate 11. . . . 1	⊖

MICROOPERATIONS FOR Accumulator Shifter, AS

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D2	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2	AS(0)S:=	XX CM	D	d d d				M	AS(0):=	Load the AS(0) Source register from CM|EX|SB|SG
2	AS(63)S:=	XX CM	D	d d d				M	AS(63)S:=	Load the AS(63) Source register from CM|EX|SB|SG
2	AS(V)S:=	XX CM	D	d d d d d d				M	AS(V)S:=	Load the AS(V) Selection register from CM|EX|SB|SG
2								M	ASLL	Set the AS to a logical left shift
2								M	ASLR	Set the AS to a logical right shift
2					M	AS(V)SC				Clear the AS(V) Selection register
2					M	AS(V)S +1				Increment the AS(V) Selection register
2					M	AS(V)S -1				Decrement the AS(V) Selection register

MICROOPERATIONS FOR AVD (AS, VS, DS) Standard Group and parallel mops

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D_2	F2	M/D_1	F3	S3	M/D_3	F4	MICROOPERATION
2					M	AVDP:=	ZZ CM	D	d d d d	Load the AVD SG Pointer from CM|EX|S1|S2
2					M	AVDP +1				Increment the AVD SG Pointer
2					M	AVDP -1				Decrement the AVD SG Pointer
2					M	AVDPC				Clear the AVD SG Pointer
2			M	AVDPS1:=	M	AVDPS1:=	ZZ CM	D	d d d d	Load the AVDP Save1 register from CM|EX|S1|S2
1	AVDPS2:= AVDP				M	AVDPS2:= AVDP				Load the AVDP Save2 register from the CS Pointer
1			M	AVDSG:=SB						Load the AVD SG from SB(5:0)
2								M	AVDLL	Set AS, VS and DS to logical left shift
2								M	AVDLR	Set AS, VS, and DS to logical right shift
2					M	AVD(V)SC				Clear AS, VS, and DS Variable Bit Selection register
2	AVD(0)S:=	XX CM	D	d d d						Load AS(0), VS(0), and DS(1:0) Source register from CM|EX|SB|SG
2	AVD(63)S:=	XX CM	D	d d d						Load AS(63), VS(63), and DS(63:62) Source register from CM|EX|SB|SG
2	AVD(V)S:=	XX CM	D	d d d d d d						Load AS(V), VS(V), and DS(V) Selection register from CM|EX|SB|SG

MICROOPERATIONS FOR Bit Encoder, BE

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D_2	F2	M/D_1	F3	S3	M/D_3	F4	MICROOPERATION
2	BEM LOAD									Load results of MSB encoding into MSB_1
1			M	BEMI						MSB_1 and MSB_2 are interchanged
2								M	BEL LOAD	Load results of LSB encoding into LSB_1
1					M	BELI				LSB_1 and LSB_2 are interchanged
2	BELM LOAD				M	BELM LOAD				Load results of MSB encoding into MSB_1 AND load results of LSB encoding into LSB_1
1			M	BELMI				M	BELMI	MSB_1 and MSB_2 are interchanged AND LSB_1 and LSB_2 are interchanged
2	BEF:=	XX CM	D	d d d d						Load BE Function register from CM|EX|SB|SG
2			M	SET BEF LSB1						Set the BEF to LSB_1 (clear the BEF Function register)
1			M	BEPGL						Sets PG to generate from LSB if BE is control input
1			M	BEPGM						Sets PG to generate from MSB if BE is control input
2					M	BEP:=	ZZ CM	D	d d d d	Load BESG Pointer from CM|EX|S1|S2
2					M	BEP +1				Increment BESG Pointer
2					M	BEP -1				Decrement BESG Pointer
2					M	BEPC				Clear BESG Pointer
2			M	BEPS1:=	M	BEPS1:=	ZZ CM	D	d d d d	Load BEP Save1 register from CM|EX|S1|S2
1	BEPS2:=BEP									Load BEP Save2 register from BE Pointer
1			M	BESG:=SB						Load BE SG from SB(3:0)

MICROOPERATIONS FOR Bus Shifter, BS

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2	BSS:=		M	BSS:=	M	BSS:=	d d			Load the BS Selection register with dd, dd='CM' | 'EX' | 'BE' | 'SG'
2								M	BSS +1	Increment the BS Selection register Θ
2								M	BSS -1	Decrement the BS Selection register Θ
2								M	BSSC	Clear the BS Selection register
					D	d d d d d d				(THIS DATA IS REQUIRED WHENEVER THE BUS SHIFTER CONTROL IS USING CM AS DATA)
2	BSP:=	ZZ CM	D	d d d d						Load BS SG pointer from CM | EX | S1 | S2
2	BSP +1									Increment BS SG Pointer
2	BSP -1									Decrement BS SG Pointer
2	BSPC									Clear BS SG Pointer
2	BSPS1:=	ZZ CM	D	d d d d						Load BSP Save1 register from CM | EX | S1 | S2
1					M	BSPS2:=BSP				Load BSP Save2 register from BS SG Pointer
1								M	BSSG:=SB	Load BS SG from SB(5:0)

MICROOPERATIONS FOR Counter A, CA

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2	CA:=	XX CM	D	d d d d d d d	D	d d d d d d d	dd	M	CA:=	Load CA from CM (16 bits), SB (16 bits), EX (16 bits), or CASG (16 bits)
2	CA +1				M	CA +1		M	CA +1	Increment CA
2	CA -1				M	CA -1		M	CA -1	Decrement CA
2	CAC				M	CAC		M	CAC	Clear CA
2	CAP:=	YY CM	D	d d d d						Load the CA SG Pointer from CM | SB | S1 | S2 Θ
2	CAP +1 Θ		M	CAP +1						Increment CA SG Pointer
2	CAP -1 Θ		M	CAP -1						Decrement CA SG Pointer
2	CAPC Θ		M	CAPC						Clear CA SG Pointer
1			M	CASG:=CA				M	CASG:=CA Θ	Load CA SG from CA
2	CAPS1:=	YY CM	D	d d d d						Load CAP Save1 register from CM | SB | S1 | S2 Θ
1					M	CAPS2:=CAP				Load CAP Save2 register from CA SG Pointer Θ

MICROOPERATIONS FOR Counter B, CB

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2	CB:=	VV CM	D	d d d d d d d	D	d d d d d d d	dd	M	CB:=	Load CB from CM (16 bits), SB (16 bits), BE+ (6 bits), or CBSG (16 bits)
2	CB +1		M	CB +1				M	CB +1	Increment CB
2	CB -1		M	CB -1				M	CB -1	Decrement CB
2	CBC		M	CBC				M	CBC	Clear CB
2					M	CBP:=	YY CM	D	d d d d	Load the CB SG Pointer from CM | SB | S1 | S2 Θ
2					M	CBP +1				Increment CB SG Pointer
2					M	CBP -1				Decrement CB SG Pointer
2					M	CBPC				Clear CB SG Pointer
1	CBSG:=CB Θ				M	CBSG:=CB				Load CB SG from CB
2					M	CBPS1:=	YY CM	D	d d d d	Load CBP Save1 register from CM | SB | S1 | S2 Θ
1			M	CBPS2:=CBP						Load CBP Save 2 register from CB SG Pointer Θ
										+) when BE is selected as the source, the high order 10 bits of CB are set to 0

MICROOPERATIONS FOR Condition Save Register, CR and switches KC, KD

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION	
2					M	CRP:=	ZZ CM	D	d d d d	Load CR RG Pointer from CM|EX|S1|S2	⊖
2					M	CRP +1				Increment CR RG Pointer	⊖
2					M	CRP −1				Decrement CR RG Pointer	⊖
2					M	CRPC				Clear CR RG Pointer	⊖
2			M	CRPS1:=	M	CRPS1:=	ZZ CM	D	d d d d	Load CR RG Save1 buffer from CM|EX|S1|S2	⊖
1	CRPS2:=CRP									Load CR RG Save2 buffer from CR RG Pointer	⊖
S*	CR:=<SC>		M	CR:=<SC>				M	CR:=<SC>	Load CR RG with the current Selected Condition	⊖
1 1			M M	SETKC KCC						KC:= true KC:= false	
S*	KC:=<SC>		M	KC:=<SC>						Load KC with the current Selected Condition	
1 1								M M	SETKD KDC	KD:= true KD:= false	
S*	KD:=<SC>							M	KD:=<SC>	Load KD with the current Selected Condition	
	S* = special depending on short or long cycle to give the value of the condition used in sequencing.										

MICROOPERATIONS FOR Control Unit, CU

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION	
1			M	SA:=SB						Load Save Address register from SB(11:0)	
1								M	SA +1	Increment Save Address	
1								M	SA −1	Decrement Save Address	
1								M	SAC	Clear Save Address	
1					M	CUALF:=		D	d d d d d	Load CU AL Function register with d d d d d	
1					M	SET CU ALF +				Set CU AL Function register to A+B	
1	SETCUALF B									Set CU AL Function register to B	
1			M	RA ↑						Decrement RA Pointer	
1	RA ↓		M	RA ↓	M	RA ↓				Increment RA Pointer and then Load RA	
1			M	RAPC						Clear RA Pointer	
1	RB ↑									Decrement RB Pointer	
1	RB ↓		M	RB ↓	M	RB ↓				Increment RB Pointer and then Load RB	
1	RBPC									Clear RB Pointer	
1			M	EX LOAD						Load the External register	⊖
1			M	EX →						Shift the External register 4 bits right cyclic	⊖
1			M	CS LOAD						Load control store and then choose A+1 as the address of the next microinstruction	
1			M	INTON	M	INTON				Enable interrupt conditions to force 0 address	⊖
1			M	INTOFF	M	INTOFF				Disable interrupt conditions to force 0 address	⊖
1			M	STOP A				M	STOP A	} if the corresponding console switch A or B is turned on then stop execution else do nothing	
1								M	STOP B	}	
1								M	CYL	} Sets the mode of the processor to be in Long resp. Short cycle, starting with the execution of the next instruction.	⊖
1								M	CYS	}	⊖
1	NOOP1		M	NOOP2	M	NOOP3		M	NOOP4	No Operation (dummy)	

MICROOPERATIONS FOR Double Shifter, DS

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D2	F2	M/D3	F3	S3	M/D4	F4	MICROOPERATION
2	DS(1:0)S :=	XX CM	D	d d d						Load DS(1:0) Source register from CM|EX|SB|SG
2	DS(63:62)S :=	XX CM	D	d d d						Load DS(63:62) Source register from CM|EX|SB|SG
2	DS(V)S :=	XX CM	D	d d d d d d						Load DS(V) Selection register from CM|EX|SB|SG
2								M	DSLL	Set the DS to logical left shift
2								M	DSLR	Set the DS to logical right shift
2					M	DS(V)SC				Clear DS(V) Selection register
2					M	DS(V)S +1				Increment DS(V) Selection register
2					M	DS(V)S -1				Decrement DS(V) Selection register

MICROOPERATIONS FOR Input Port A, and Input Port B, IA and IB

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D2	F2	M/D3	F3	S3	M/D4	F4	MICROOPERATION
1	IAD:=	EE CM	D	d d d d						Load IA Device register from CM|EX|SB
2	IAA		M	IAA				M	IAA	Activate device, i.e. initiate read
1			M	IADC						Clear IA Device register
1			M	IAD +1						Increment IA Device register
1			M	IAD -1						Decrement IA Device register
1	IBD:=	EE CM	D	d d d d						Load IB Device register from CM|EX|SB
2	IBA		M	IBA				M	IBA	Activate device, i.e. initiate read
1			M	IBDC						Clear IB Device register
1			M	IBD +1						Increment IB Device register
1			M	IBD -1						Decrement IB Device register

MICROOPERATIONS FOR Loading Mask Registers A, LA

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D2	F2	M/D3	F3	S3	M/D4	F4	MICROOPERATION
2	LAP :=	ZZ CM	D	d d d d						Load LASG Pointer from CM|EX|S1|S2
2	LAP +1		M	LAP +1				M	LAP +1	Increment LASG Pointer
2	LAP -1		M	LAP -1				M	LAP -1	Decrement LASG Pointer
2	LAPC							M	LAPC	Clear LASG Pointer
2	LAPS1 :=	ZZ CM	D	d d d d				M	LAPS1:=	Load LAP Save1 register from CM|EX|S1|S2
1					M	LAPS2:=LAP				Load LAP Save2 register from LA Pointer
1								M	LA := SB	Load LA from SB(63:0)

MICROOPERATIONS FOR Loading Mask Registers B, LB

Field widths: 7 | 2 | 1 | 7 | 1 | 7 | 2 | 1 | 7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2					M	LBP :=	ZZ CM	D	d d d d	Load LBSG Pointer from CM | EX | S1 | S2
2			M	LBP +1	M	LBP +1		M	LBP +1	Increment LBSG Pointer
2			M	LBP -1	M	LBP -1		M	LBP -1	Decrement LBSG Pointer
2					M	LBPC		M	LBPC	Clear LBSG Pointer
2			M	LBPS1:=	M	LBPS1:=	ZZ CM	D	d d d d	Load LBP Save1 register from CM | EX | S1 | S2
1	LBPS2:=LBP									Load LBP Save 2 register from LB Pointer
1			M	LB := SB						Load LB from SB(63:0)
2			M	LPC						Clear LASG and LBSG pointer

MICROOPERATIONS FOR Local Registers, LR

Field widths: 7 | 2 | 1 | 7 | 1 | 7 | 2 | 1 | 7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2	LRIP := DS									Load LR Input Pointer with DS(V+1:V)
2	LRIP +1									Increment LR Input Pointer
2	LRIP -1									Decrement LR Input Pointer
2	LRIPC									Clear LR Input Pointer
2								M	LROP := DS	Load LR Output Pointer with DS(V+1:V)
2								M	LROP +1	Increment LR Output Pointer
2								M	LROP -1	Decrement LR Output Pointer
2								M	LROPC	Clear LR Output Pointer
2			M	LRP := DS	M	LRP := DS				Load both LRIP and LROP with DS(V+1:V)
2			M	LRPC	M	LRPC				Clear both LRIP and LROP
2			M	LRP +1	M	LRP +1				Increment both LRIP and LROP
2			M	LRP -1	M	LRP -1				Decrement both LRIP and LROP

MICROOPERATIONS FOR Bus Mask Registers, MA and MB

Field widths: 7 | 2 | 1 | 7 | 1 | 7 | 2 | 1 | 7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2	MAP :=	XX CM	D	d d d d				M	MAP :=	Load MA Pointer from CM | EX | SB | SG
2	MAP +1		M	MAP +1				M	MAP +1	Increment MA Pointer
2	MAP -1		M	MAP -1				M	MAP -1	Decrement MA Pointer
2	MAPC		M	MAPC				M	MAPC	Clear MA Pointer
2	MBP :=	XX CM	D	d d d d				M	MBP :=	Load MB Pointer from CM | EX | SB | SG
2	MBP +1				M	MBP +1		M	MBP +1	Increment MB Pointer
2	MBP -1				M	MBP -1		M	MBP -1	Decrement MB Pointer
2	MBPC				M	MBPC		M	MBPC	Clear MB Pointer
2					M	BMP :=	ZZ CM	D	d d d d	Load BM SG Pointer from CM | EX | S1 | S2
2					M	BMP +1				Increment BM SG Pointer
2					M	BMP -1				Decrement BM SG Pointer
2					M	BMPC				Clear BM SG Pointer
2			M	BMPS1 :=	M	BMPS1 :=	ZZ CM	D	d d d d	Load BMP Save1 register from CM | EX | S1 | S2
1	BMPS2 := BMP									Load BMP Save2 register from the BMPP
1			M	BMSG:=SB						Load BM SG with SB(3:0)

MICROOPERATIONS FOR MDP-transport

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D_2	F2	M/D_3	F3	S3	M/D_4	F4	MICROOPERATION
			D	d d d d d d d	D	d d d d d d d	dd			THIS DATA IS REQUIRED WHENEVER SP IS CHOSENAS SOURCE AND SPP='CM'(=0)
					D	d d d d d d				THIS DATA IS REQUIRED WHENEVER BUS SHIFTER IS ENABLED AND BSS='CM'(=0)
			D	d d d d d d d						THIS DATA IS REQUIRED WHENEVER MASK GENERATOR IS ENABLED AND PGS='CM'(=0)
C*	BUS:=ALL 1S									Forces BUS=11....1 (used by assembler whenever <SOURCE>=ALL 1S or <SOURCE>=ALL 0S)
C**					M	SB:=ALL0S				Forces SB=00....0
	C* and C** are special clocks.									

MICROOPERATIONS FOR Output Ports A, B, C, and D, OA, OB, OC, and OD

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D_2	F2	M/D_3	F3	S3	M/D_4	F4	MICROOPERATION
1					M	OAD:=	EE CM	D	d d d d	Load OA Device register from CM|EX|SB
2	OAA	d			M	OAA		M	OAA	Activate device i. e. initiate write with DM_1 := d
2			M	OAR						Deactivate device (Reset)
1					M	OAD +1				Increment OA Device register
1					M	OAD -1				Decrement OA Device register
1					M	OADC				Clear OA Device register
1					M	OBD:=	EE CM	D	d d d d	Load OB Device register from CM|EX|SB
2	OBA	d			M	OBA		M	OBA	Activate device i. e. initiate write with DM_1 := d
2			M	OBR						Deactivate device (Reset)
1			M	OBD +1						Increment OB Device register
1			M	OBD -1						Decrement OB Device register
1			M	OBDC						Clear OB Device register
1					M	OCD:=	EE CM	D	d d d d	Load OC Device register from CM|EX|SB
2	OCA	d			M	OCA		M	OCA	Activate device i. e. initiate write with DM_1 := d
2			M	OCR						Deactivate device (Reset)
1					M	OCD +1				Increment OC Device register
1					M	OCD -1				Decrement OC Device register
1					M	OCDC				Clear OC Device register
1	OC:=BUS		M	OC:=BUS						Load OC from BUS(63:0)
1					M	ODD:=	EE CM	D	d d d d	Load OC Device register from CM|EX|SB
2	ODA	d	M	ODA				M	ODA	Activate device i. e. initiate write with DM_1 := d
2			M	ODR						Deactivate device (Reset)
1			M	ODD +1						Increment OD Device register
1			M	ODD -1						Decrement OD Device register
1			M	ODDC						Clear OD Device register
1			M	OD:=BUS						Load OD from BUS(63:0)

MICROOPERATIONS FOR Postshift Masks (PA, PB) and PMSG

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2	PAP:=	XX CM	D	d d d d						Load PA Pointer from CM|EX|SB|SG
2	PAP +1				M	PAP+1		M	PAP +1	Increment PA Pointer
2	PAP -1				M	PAP-1		M	PAP -1	Decrement PA Pointer
2	PAPC							M	PAPC	Clear PA Pointer
1					M	PA:=BUS				Load PA RG from BUS(63:0)
2	PBP:=	XX CM	D	d d d d						Load PB Pointer from CM|EX|SB|SG
2	PBP +1							M	PBP +1	Increment PB Pointer
2	PBP -1							M	PBP -1	Decrement PB Pointer
2	PBPC							M	PBPC	Clear PB Pointer
1					M	PB:=BUS				Load PB RG from BUS(63:0)
2					M	PMP:=	ZZ CM	D	d d d d	Load PM SG Pointer from CM|EX|S1|S2
2					M	PMP +1				Increment PM SG Pointer
2					M	PMP -1				Decrement PM SG Pointer
2					M	PMPC				Clear PM SG Pointer
2			M	PMPS1:=	M	PMPS1:=	ZZ CM	D	d d d d	Load PMP Save1 register from CM|EX|S1|S2
1	PMPS2:=PMP									Load PMP Save2 register from PM SG Pointer
1			M	PMSG:=SB						Load PMSG from SB(3:0)
2					M	PABC				Clear PA and PB Pointer

MICROOPERATIONS FOR Postshift Mask Generator, PG

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2					M	PGS:=	dd			Mask Generator Control Source Selection register is set to dd, dd='CM'|'EX'|'BE'|'SG'
2			M	PGS +1				M	PGS +1	Increment PG Selection register
2			M	PGS -1				M	PGS -1	Decrement PG Selection register
2			M	PGSC				M	PGSC	Clear PG Selection register
0			D	d d d d d d d						THIS DATA IS REQUIRED WHENEVER THE MASK GENERATOR CONTROL IS USING CM AS DATA
2					M	PGP:=	ZZ CM	D	d d d d	Load PG SG Pointer from CM|EX|S1|S2
2					M	PGP +1				Increment PG SG Pointer
2					M	PGP -1				Decrement PG SG Pointer
2					M	PGPC				Clear PG SG Pointer
2			M	PGPS1:=	M	PGPS1:=	ZZ CM	D	d d d d	Load PG Save1 register from CM|EX|S1|S2
1	PGPS2:=PGP									Load PG Save2 register from PGP
1			M	PGSG:=SB						Load PG SG from SB(6:0)

MICROOPERATIONS FOR Status Port, SP

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D2	F2	M/D3	F3	S3	M/D4	F4	MICROOPERATION
			D	d d d d d d d	D	d d d d d d d	dd			THIS DATA IS REQUIRED WHENEVER THE STATUS PORT IS BEING USED AS SOURCE AND SPP= 0 (= 'CM')
1	SPP:=	EE CM	D	d d d d d d				M	SPP :=	Load Status Port Pointer from CM|EX|SB
1								M	SPP +1	Increment the Status Port Pointer
1								M	SPP -1	Decrement the Status Port Pointer
1								M	SPPC	Clear the Status Port Pointer

MICROOPERATIONS FOR Variable Width Shifter, VS

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D2	F2	M/D3	F3	S3	M/D4	F4	MICROOPERATION
2	VS(0)S :=	XX CM	D	d d d				M	VS(0)S :=	Load the VS(0) Source register from CM|EX|SB|SG
2	VS(63)S :=	XX CM	D	d d d				M	VS(63)S :=	Load the VS(63) Source register from CM|EX|SB|SG
2	VS(V)S :=	XX CM	D	d d d d d d				M	VS(V)S :=	Load the VS(V) Selection register from CM|EX|SB|SG
2			M	VSLL						Set the VS to a logical left shift
2			M	VSLR						Set the VS to a logical right shift
2	VS(V)SC									Clear the VS(V) Selection register
2	VS(V)S +1									Increment the VS(V) Selection register
2	VS(V)S -1									Decrement the VS(V) Selection register

MICROOPERATIONS FOR Working Registers, WA

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2	WAU :=	XX CM	D	d d d d						Load WA Unit pointer from CM|EX|SB|SG
2	WAU +1							M	WAU +1	Increment WA Unit pointer
2	WAU -1							M	WAU -1	Decrement WA Unit pointer
2	WAUC							M	WAUC	Clear WA Unit pointer
2					M	WAG :=	XX CM	D	d d d d	Load WA Group pointer from CM|EX|SB|SG
2					M	WAG +1				Increment WA Group pointer
2					M	WAG -1				Decrement WA Group pointer
2					M	WAGC				Clear WA Group pointer
2	WAP :=	XX CM	D	d d d d			XX CM	D	d d d d	Load WA Unit pointer from CM|EX|SB|SG AND load WA Group pointer from CM|EX|SB|SG
2	WAPC									Clear WA Unit pointer and WA Group pointer *)
1								M	WAPCOUPLE	Couple WA Unit and Group pointers to form an 8 bit counter
1								M	WAP-UNCOUPLE	Uncouple WA Unit and Group pointers to form two independent 4 bit counters

*) WAP +1 is equivalent to WAU +1, and WAP -1 to WAU -1, assuming that the unit and group pointers are coupled.

MICROOPERATIONS FOR WA Unit and Group Standard Groups, WAUS and WAGS

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION	
1	WAUS:=WAU ⊖							M	WAUS:=WAU	Load WA Unit SG with WAU	
2					M	WAUSP:=	YY CM	D	d d d d	Load WAUS Pointer from CM|SB|S1|S2	⊖
2								M	WAUSP +1	Increment WA Unit SG Pointer	
2								M	WAUSP -1	Decrement WA Unit SG Pointer	
2								M	WAUSPC	Clear WA Unit SG Pointer	
2					M	WAUSPS1:=	YY CM	D	d d d d	Load WAUSP Save 1 register from CM|SB|S1|S2	⊖
1			M	WAUSPS2:= WAUSP						Load WAUSP Save2 register from WAUSP	⊖
1	WAGS:=WAG							M	WAGS:=WAG ⊖	Load WA Group SG with WAG	
2	WAGSP:=	YY CM	D	d d d d						Load WAGSP from CM|SB|S1|S2	⊖
2	WAGSP +1									Increment WA Group SG Pointer	
2	WAGSP -1									Decrement WA Group SG Pointer	
2	WAGSPC									Clear WA Group SG Pointer	
2	WAGSPS1:=	YY CM	D	d d d d						Load WAGSP Save1 register from CM|SB|S1|S2	⊖
1					M	WAGSPS2:= WAGSP				Load WAGSP Save 2 register from WAGSP	⊖
1					M	WAPS:=WAP		M	WAPS:=WAP	Load WA Unit and WA Group SG with WAU and WAG respectively	
2	WAPSP:=	YY CM	D	d d d d			YY CM	D	d d d d	Load WAUPSP and WAGSP from CM|SB|S1|S2	⊖
2								M	WAPSP +1	Increment WA Unit and WA Group SG Pointers	
2								M	WAPSP -1	Decrement WA Unit and WA Group SG Pointers	
2								M	WAPSPC	Clear WA Unit and WA Group SG Pointers	
2	WAPSPS1:=	YY CM	D	d d d d			YY CM	D	d d d d	Load WAUSP and WAGSP Save1 registers from CM|SB|S1|S2	⊖
1								M	WAPSPS2:= WAPSP	Load WAUSP and WAGSP Save2 registers from WAUSP and WAGSP respectively	⊖

MICROOPERATIONS FOR Working Registers, B, WB

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D_2	F2	M/D_3	F3	S3	M/D_4	F4	MICROOPERATION
2					M	WBU :=	XX CM	D	d d d d	Load WB Unit pointer from CM|EX|SB|SG
2			M	WBU +1				M	WBU +1	Increment WB Unit pointer
2			M	WBU -1				M	WBU -1	Decrement WB Unit pointer
2			M	WBUC				M	WBUC	Clear WB Unit pointer
2	WBG :=	XX CM	D	d d d d						Load WB Group pointer from CM|EX|SB|SG
2			M	WBG +1						Increment WB Group pointer
2			M	WBG -1						Decrement WB Group pointer
2			M	WBGC						Clear WB Group pointer
2	WBP :=	XX CM	D	d d d d			XX CM	D	d d d d	Load WB Unit pointer from CM|EX|SB|SG AND load WB Group pointer from CM|EX|SB|SG
2			M	WBPC						Clear WB Unit pointer and WB Group pointer *)
1					M	WBPCOUPLE				Couple WB Unit pointer and Group pointers to form an 8 bit counter
1					M	WBP-UNCOUPLE				Uncouple WB Unit pointer and Group pointer to form two independent 4 bit counters

*) WBP +1 is equivalent to WBU +1, and WBP -1 to WBU -1, assuming that the unit and group pointers are coupled.

MICROOPERATIONS FOR WB Unit and Group Standard Groups, WBUS and WBGS

7	2	1	7	1	7	2	1	7

C_p	F1	S1	M/D_2	F2	M/D_3	F3	S3	M/D_4	F4	MICROOPERATION	
1			M	WBUS:=WBU	M	WBUS:=WBU Θ				Load WB Unit SG from WBU	
2	WBUSP:=	YY CM	D	d d d d						Load WBUS Pointer from CM|SB|S1|S2	Θ
1			M	WBUSP +1						Increment WB Unit SG Pointer	
2			M	WBUSP -1						Decrement WB Unit SG Pointer	
2			M	WBUSPC						Clear WB Unit SG Pointer	
2	WBUSPS1:=	YY CM	D	d d d d						Load WBUSP Save1 register from CM|SB|S1|S2	Θ
1								M	WBUSPS2:= WBUSP	Load WBUSP Save2 register from WBUSP	Θ
1			M	WBGS:=WBG Θ	M	WBGS:=WBG				Load WB Group SG from WBG	
2					M	WBGSP:=	YY CM	D	d d d d	Load WBGS Pointer from CM|SB|S1|S2	Θ
2					M	WBGSP +1				Increment WB Group SG Pointer	
2					M	WBGSP -1				Decrement WB Group SG Pointer	
2					M	WBGSPC				Clear WB Group SG Pointer	
2					M	WBGSPS1:=	YY CM	D	d d d d	Load WBGSP Save1 register from CM|SB|S1|S2	Θ
1	WBGSPS2:= WBGSP									Load WBGSP Save2 register from WBGSP	Θ
1			M	WBPS:=WBP	M	WBPS:=WBP				Load WB Unit and WB Group SG with WBU and WBG respectively	
2	WBPSP:=	YY CM	D	d d d d			YY CM	D	d d d d	Load WBUS and WBGS Pointers from CM|SB|S1|S2	Θ
2					M	WBPSP +1				Increment WB Unit and WB Group SG Pointers	
2					M	WBPSP -1				Decrement WB Unit and WB Group SG Pointers	
2					M	WBPSPC				Clear WB Unit and WB Group SG Pointers	
2	WBPSPS1:=	YY CM	D	d d d d			YY CM	D	d d d d	Load WBUSP and WBGSP Save1 registers from CM|SB|S1|S2	Θ
1								M	WBPSPS2:= WBPSP	Load WBUSP and WBGSP Save2 registers from WBUSP and WBGSP respectively	Θ

MICROOPERATIONS FOR Wide Store Address, WSA

7	2	1	7	1	7	2	1	7

C_p	F1	S1	MD_2	F2	MD_3	F3		MD_4	F4	MICROOPERATION
2	WSA :=	XX / CM	D	d d d d d d d	D	d d d d d d d	d d			Load WSA from CM|EX|SB|SG
2	WSA + 1		M	WSA + 1				M	WSA + 1	Increment WSA
2	WSA - 1		M	WSA - 1				M	WSA - 1	Decrement WSA
2	WSAC		M	WSAC				M	WSAC	Clear WSA
2	WSAP :=	ZZ / CM	D	d d d d						Load WSA SG Pointer from CM|EX|S1|S2
2	WSAP + 1									Increment WSA SG Pointer
2	WSAP - 1									Decrement WSA SG Pointer
2	WSAPC									Clear WSA SG Pointer
1	WSASG:=WSA		M	WSASG:=WSA				M	WSASG:=WSA	Load WSA SG from WSA
2	WSAPS1 :=	ZZ / CM	D	d d d d				M	WSAPS1 :=	Load WSAP Save 1 register from CM|EX|S1|S2
1					M	WSAPS2 := WSAP				Load WSAP Save 2 register from WSA SG Pointer

3.4 Assembler Notation

The notation used in this book forms the basis of the assembly language for Mathilda described in [3]. However the character representation differs somewhat, due to the DEC-10 character set.

Below we give the equivalent ASCII-characters, accepted by [3] for the symbols used here.

Symbol	ASCII-representation
→	>
←	<
↓	!
↑	↑
∧	&
∨	@
¬		

For convenience, we also give a complete list of recognizable conditions, AL- and BE-functions.

Conditions			
AL(0)	CR	KD	TRUE
AL(63)	DS(0)	L1	VS(0)
AL	DS(1)	L2	VS(63)
ALOV	DS(2)	LD	VS(V)
AS(0)	DS(3)	LR(0)	WA(0)
AS(63)	DS(4)	LR(63)	WA(63)
AS(V)	DS(5)	LSB1	WACS
BEPGD	DS(6)	LSBD	WAGOV
BE(0)	DS(7)	MSB1	WAGSPOV
BP	DS(8)	MSBD	WAPOV
BUS	DS(9)	OASA	WAPSPOV
BUSPAR	DS(10)	OBSA	WAUOV
CA(0)	DS(11)	OCSA	WAUSPOV
CA(3)	DS(12)	ODSA	WB(0)
CA(4)	DS(13)	ONE	WB(63)
CA(5)	DS(14)	RAPOV	WBCS
CA(6)	DS(15)	RAPUN	WBGOV
CA	DS(V)	RBPOV	WBGSPOV
CASPOV	DS(V+1)	RBPUN	WBPOV
CB(0)	FALSE	SB(0)	WBPSPOV
CB(3)	IADA	SB(1)	WBUOV
CB(4)	IADM	SB(62)	WBUSPOV
CB(5)	IBDA	SB(63)	WSAB
CB(6)	IBDM	SGNLD	WSAOR
CB	KA	SGNLSBD	WSASPOV
CBSPOV	KB	SGNMSBD	ZERO
CYL	KC		

<table>
<tr><th colspan="3">AL-functions</th></tr>
<tr><th colspan="2">ARITHMETIC</th><th>LOGICAL</th></tr>
<tr><th>carry-in = 0</th><th>carry-in = 1</th><th></th></tr>
<tr><td>A</td><td>A+1</td><td>|A</td></tr>
<tr><td>A@B</td><td>A@B+1</td><td>|A&|B</td></tr>
<tr><td>A@|B</td><td>A@|B+1</td><td>|A&B</td></tr>
<tr><td>MINUS1</td><td>MINUS1+1</td><td>ALLOS</td></tr>
<tr><td>A+(A&|B)</td><td>A+(A&|B)+1</td><td>|A@|B</td></tr>
<tr><td>(A@B)+(A&|B)</td><td>(A@B)+(A&|B)+1</td><td>|B</td></tr>
<tr><td>A-B-1</td><td>A-B</td><td>AEQVB</td></tr>
<tr><td>(A&|B)-1</td><td>(A&|B)</td><td>A&|B</td></tr>
<tr><td>A+(A&B)</td><td>A+(A&B)+1</td><td>|A@B</td></tr>
<tr><td>A+B</td><td>A+B+1</td><td>ANEQVB</td></tr>
<tr><td>A@|B+(A&B)</td><td>(A@|B)+(A&B)+1</td><td>B</td></tr>
<tr><td>(A&B)-1</td><td>A&B</td><td>A&B</td></tr>
<tr><td>A+A</td><td>A+A+1</td><td>ALL1S</td></tr>
<tr><td>(A@B)+A</td><td>A@B+A+1</td><td>A@|B</td></tr>
<tr><td>(A@|B)+A</td><td>(A@|B)+A+1</td><td>A@B</td></tr>
<tr><td>A-1</td><td>A</td><td>A</td></tr>
</table>

BE-functions
LSB1
LSB1-1
MSB1
MSB1+1
L1
L2-L1
LSB2-LSB1
MSB2-MSB1
[LSB1/2]+1
[LSB1-1/2]+1
[MSB1/2]+1
[MSB1+1/2]+1
[L1/2]+1
[L2-L1/2]+1
[LSB2-LSB1/2]+1
[MSB2-MSB1/2]+1

4.0 INDEX TO FIGURES, TABLES AND SYMBOLS

4.1 List of Figures.

Figure No.	*Title*	*Page*
2.1	The MATHILDA Processor	6
2.2	Typical Register Group	6
2.3	Typical Standard Group	7
2.4	Counter A, CA	8
2.5	Sub-system of the MDP	10
2.6	Working Registers A, WA	11
2.7	Bus Shifter, BS	16
2.8	MDP Sub-system of Fig. 2.4 expanded with BM	19
2.9	Bus Masks, MA and MB	19
2.10	MDP Sub-system of Fig. 2.8 expanded with PM	21
2.11	Postshift Masks, PA and PG	22
2.12	Arithmetical Logical Unit, AL	24
2.13	Local Registers, LR	26
2.14	Accumulator Shifter, AS	27
2.15	MDP Sub-system of Fig. 2.10 expanded with LR, AL, and AS	29
2.16	Variable Width Shifter, VS	31
2.17	Double Shifter, DS	32
2.18	MDP Sub-system of Fig. 2.15 expanded with VS and DS	33
2.19	Counting Loop for Counting Number of Bits set to 1 in a Word	33
2.20	AS, VS, and DS Control	34
2.21	MDP Sub-system of Fig. 2.18 expanded with LA and LB	35
2.22	Loading Mask Registers A, LA	35
2.23	MDP Sub-system of Fig. 2.21 expanded with BPG	36
2.24	Bit Encoder, BE	37
2.25	MDP Sub-system of Fig. 2.23 extended with BE	40
2.26	The Status Port, SP	40
2.27	Device Interfacing on Input Port	42
2.28	Device Selection on a fully expanded IA	42

2.29	Device Selection on a fully expanded OA	43
2.30	Device Interfacing on Output Port	43
2.31	MATHILDA Main Data Path	44
2.32	MATHILDA Main Data Path	45
2.33	Microinstruction Address Bus (Preliminary)	46
2.34	Control Unit Arithmetical Logical Unit	47
2.35	B data selection	49
2.36	Return Jump Stack A, RA	51
2.37	The Save Address Register, SA	53
2.38	The External Register, EX	53
2.39	The Force 0 Address Capability	54
2.40	Microinstruction Address Bus (Detailed)	55
2.41	Control Store Loading	54
2.42	Condition Save Registers and Switches	58
2.43	Counter B, CB	60
2.44	Working Registers A, WA (Detailed)	61
2.45	Postshift Masks PA, PB and PG	63
2.46	Wide Store Address, WSA	63a
3.1	Subfields of Microinstructions	64

4.2 List of Tables

2.1	Microoperations for the control of an RG	7
2.2	Microoperations for control of CA	8
2.3A	Microoperations for control of CASG and CASGP	9
2.3B	Microoperations for control of CASG and CAP	9
2.4	Microoperations for control of WA and WB	12
2.5	Microoperations for control of the BS	16
2.6	Microoperations for control of the BM	20
2.7	PG Output	22
2.8	Microoperations for control of the PM	23
2.9	AL Functions	25
2.10	Microoperations for control of the AL	25
2.11	Microoperations for control of the LR	26
2.12	Microoperations for control of the AS	28
2.13	Microoperations for control of the VS	31
2.14	Microoperations for control of the DS	32
2.15	Microoperations for control of the AVD SG	34
2.16	Parallel AVD Microoperations	34
2.17	Microoperations for control of LA and LB	36
2.18	Bit Encoder Functions	37
2.19	Microoperations for control of BE	38
2.20A	Bit Encoder Conditions	39
2.20B	Conditions Related to BE Usage	39
2.21	Status Information	40
2.22	Microoperations for control of SPP	41
2.23	Microoperations for control of IA and IB	42
2.24	Microoperations for control of OA and OC	43
2.25	Microinstruction Address Sources	46

2.26	B Data	49
2.27	Microoperations for control of RA	51
2.28	Microoperations for control of SA	53
2.29	Microoperations for control of EX	53
2.30	Force 0 Address Conditions	54
2.31	Microoperations associated with the Control Unit	54
2.32	Partial Listing of System Conditions	56-57
2.33	Microoperations for control of CR, KC and KD	58
2.34	Microoperations to control the length of cycle	58
2.35	Microoperations for control of CB, CBP and CBSG	60
2.36	Microoperations for control of the WAU and WAG pointers	62
2.37	Microoperations for control of WAUS and WAGS	62
2.38	Additional WA and WB Conditions	62
2.39	Microoperations for control of PB	63
2.40	Microoperations for control of PMSG	63
2.41	Microoperations for control of WSA	63a
2.42	WSA Conditions	63a
3.1	Symbolic Notation for SOURCE's and SBD's	64
3.2	Symbolic Notations for A_t and A_f	66
3.3	Shift/Load Control Bits	67

4.3 Table of First Occurrence or Definitions of Selected Abbreviations and Symbols

Symbol	*Interpretation*	*Page*
A_t , A_f	Address Specifications	12
AL	Arithmetical Logical Unit	24
AS	Accumulator Shifter	27
AVD	AS/VS/DS Resources	34
BE	Bit Encoder	36
BISB	B-input Selection Bits	49
BM	Bus Masks	19
BPG	Bus Parity Generator	36
BS	Bus Shifter	16
BSE	Bus Shifter Enable bit	64
BUS	the BUS	10
C	Clear (Set to 0...0)	7
C_p	Clock Pulse	69
CA	Counter A	8
CB	Counter B	60
CISB	Carry-in Selectin Bit	47
CM	Current Microinstruction	8
CR	Condition Save Register	57
CS	Control Store	54
CSB	Condition Selection Bits	57
CU	Control Unit	46
CUAL	Control Unit Arithmetical Logical Unit	47
DESTINATION	Destination, SBD or D	10
DS	Double Shifter	32
EX	External Register	53
IA	Input Port A	41

IB	Input Port B	41
IRA	Interrupt Recovery Address	54
KA	Console Switch A	57
KB	Console Switch B	57
KC	Programmable Switch C	57
KD	Programmable Switch D	57
L	Microoperation specified load	7
LA	Loading Masks A	34
LB	Loading Masks B	34
LR	Local Registers	26
LSB	A Bit Pointer (available through BE)	36
MA	Mask A Register	19
MB	Mask B Register	19
MDP	Main Data Path	10
MSB	A Bit Pointer (available through BE)	36
OA	Output Port A	42
OB	Output Port B	42
OC	Output Port C	42
OD	Output Port D	42
PA	Postshift Mask A Registers	22
PB	Postshift Mask B Registers	63
PG	Postshift Mask Generator	22
PM	Postshift Masks (PAVPB)	63
RA	Return Jump Stack A	51
RB	Return Jump Stack B	51
RG	Register Group	6
SA	Save Address Register	53
SB	Shifted Bus	10
SBD	Shifted Bus Destination	10
SBD Load	SB Destination Load	12

SC Selected Condition 57
Sel Selection by dedicated bits in the microinstruction 7
SG Standard Group 7
SOURCE the input to the BUS 10
SP Status Port 40
V Variable Bit 27
VS Variable Shifter 31
WA Working Registers A 11,61
WB Working Registers B 11,61

WSA Wide Store Address 63

$\neg$ c, $\bar{c}$ Logical "negation" of condition c 14,47
$\equiv$ Logical "equivalence" 25
$\not\equiv$ Logical "nonequivalence" 25
$\rightarrow$ Right Shift 18
$\rightarrow$ Postshift Mask Generation Direction 23
$\leftarrow$ Left Shift 18
$\leftarrow$ Postshift Mask Generation Direction 23
{ } "Meta" parenthesis (used for grouping) 11
| Possible Alternate Sources 7
——► Input 6
——○ Mask Application 19
——ⓛ Loading Mask Application 35
< > "Meta" brackets enclosing meta symbols 9
↑ Push 51
↓ Pop 51
'...' Resource Name Encoding 16
[i] Indexing operator 7
(i:j) Contiguous bit string specification 8
+1 Increment of counter or pointer 7

-1	Decrement of counter or pointer	7
|||||	Indication of the end of an example	15
:=	Equivalent to L or SBD Load	7
::=	is defined to be	62
o	Concatenation	49

References

[1] Robert Dorin: "A Viable Host Machine for Research in Emulation", Department of Computer Science Report 39-72-mu, State University of New York at Buffalo, Amherst, New York, 1972.

[2] Peter Kornerup, B.D. Shriver: "An Overview of the MATHILDA System", DAIMI PB-34, Department of Computer Science, University of Aarhus, Denmark; also in SIGMICRO Newsletter, Jan. 1975.

[3] I.H. Sørensen, E. Kressel: "RIKKE-MATHILDA microassemblers and simulators on the DECsystem-10". DAIMI MD-28, Department of Computer Science, University of Aarhus, Denmark, December 1977.

[4] Ole Brun Madsen: "BPL - a hardware and software description language", RECAU, University of Aarhus, Denmark, 1972.

[5] Ole Brun Madsen: "KAROLINE - a network computer project", RECAU, University of Aarhus, Denmark, 1972.

[6] Robert F. Rosin: "The Significance of Microprogramming", proceedings from the International Computing Symposium 1973, Davos, Switzerland.

[7] Bruce D. Shriver: "Microprogramming and Numerical Analysis", IEEE Transactions on Electronic Computers, Special Issue on Microprogramming, July 1971.

[8] Bruce D. Shriver: "A Small Group of Research Projects in Machine Design for Scientific Computation", DAIMI PB-14, Department of Computer Science, University of Aarhus, Denmark, April 1973.

[9] Bruce D. Shriver: The Preliminary Description of the MATHILDA Snooper Facility", unpublished paper, Department of computer Science, University of Southwestern Louisiana, Lafayette, August 1974.

[10] J. Staunstrup: A Description of the RIKKE1 System", DAIMI PB-29, Department of Computer Science, University of Aarhus, Denmark, May 1974.