A Description of the MATHILDA System

by

Bruce D. Shriver

DAIMI PB-13
April 1973

Matematisk Institut   Aarhus Universitet
DATALOGISK AFDELING
Ny Munkegade - 8000 Aarhus C - Danmark
Tlf. 06-12 83 55

Department of Computer Science

A Description of the MATHILDA System*

by

Bruce D. Shriver

* This work is supported by the Danish Research Council Grant 511-1546.

April 1973

Abstract

A dynamically microprogrammable processor called MATHILDA is described. MATHILDA has been designed to be used as a tool in emulator 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 bus structure, a horizontally 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.

Table of Contents

Foreword

It is the purpose of this document to give an introductory (yet reasonably detailed) description of the MATHILDA System. The bus structure, the registers and functional units attached to it, and the control which can be exercised on these components 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 bus structure. The examples are written in an imaginary (syntatically sugared) microassembly language. 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 components 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 underscores the overall simplicity and homogeneity of the structure and its components.

A Description of the MATHILDA System*

by

B. D. Shriver

1.0 Introduction

MATHILDA is a dynamically microprogrammable processor which has been designed to be used as a tool in emulation-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 mini-computer 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 [1]. To support this group and to make the use of such a mini-computer 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 [2], 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 [3, 4] 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 micro-programmed 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 (32K, 64-bit

* This work is supported by the Danish Research Council, Grant 511-1546

wide, 1.4 μs access 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 design was started in May 1972 and completed during the summer of 1972. The construction of the resulting machine, MATHILDA, is due to be completed in June 1973.

The motivation for building the MATHILDA instead of purchasing a commercially available machine can be summarized as follows. First, there were (to the author's knowledge) no commercially available dynamically microprogrammable processors at the time we started our 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 criteria 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 emulator 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 [5]. We, therefore, tried to have various application-oriented and software ideas be 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 [6]). 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.

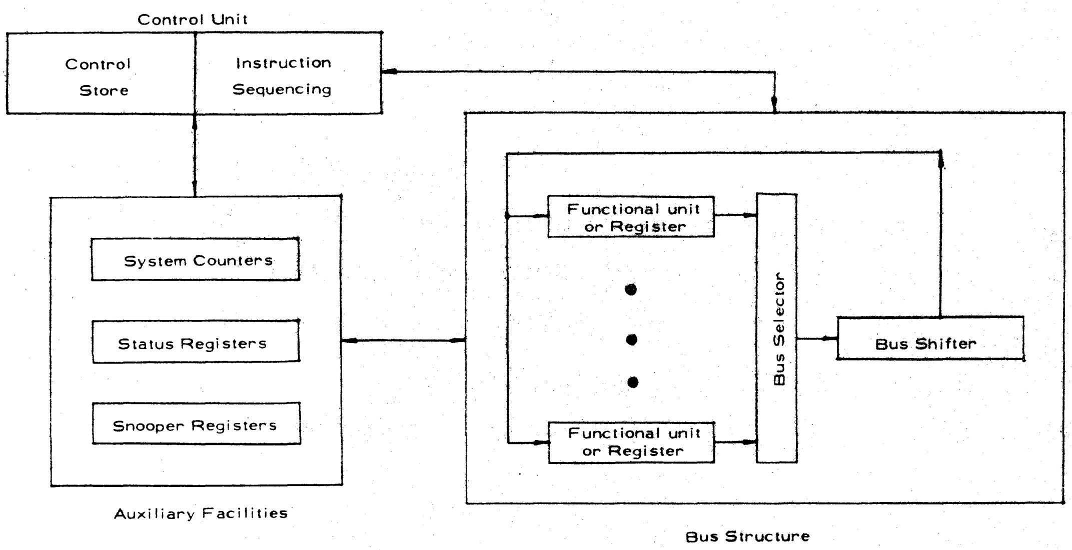
Another criterion was to have a clean and consistent way of dealing with timing problems. We decided not to force the speed;

rather we would have a slower machine than obtainable with the componetry at hand, and thus one, hopefully, with a reduced set of timing idiosyncrasies. 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" whereever such 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 1 year, it was decided that the additional RIKKE's (now called RIKKE-1's) 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-layout and construction to be achieved. As an example, the bus structure is laid out on one print board, 8-bits wide. Two of these boards, interconnected, comprise one RIKKE-1 bus structure with all registers, shifters, etc. Four of these RIKKE-1 boards, interconnected, give the MATHILDA bus structure.

2.0 The MATHILDA System

MATHILDA, as has been stated earlier, is a microprogrammed controlled bus structure. The major elements of the system are shown in Figure 2.1 and are the: 1) bus structure, 2) control unit, and 3) auxiliary facilities. In the following sections we will describe each of these systems independently and give examples of their utilization.



MATHILDA System

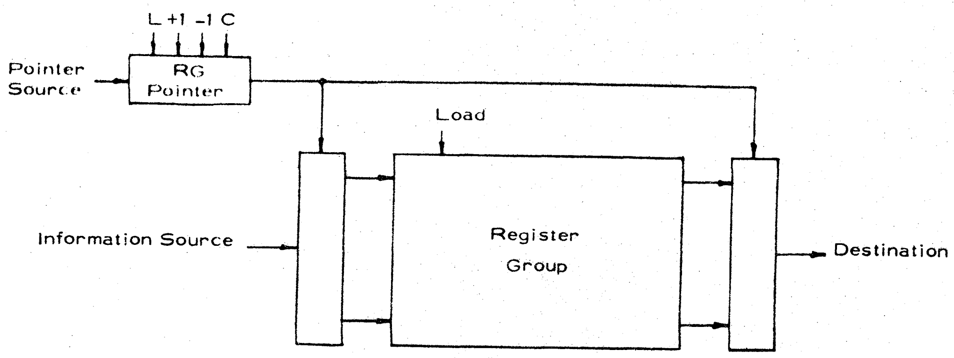
Figure 2.1

2.1 The Register 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 Figoure 2.2. 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 information, is pointed to by the RG address register. This register is called the Register Group Pointer, RGP, 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 Occurrance of Abbreviations and Symbols", beginning on page 115, for the page of first occurrance.



Typical Register Group

Figure 2.2

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 below in Table 2.1.

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

Table 2.1

Microoperations for the control of an RG

The symbolic notation RGP+1, RGP-1, etc. is the notation which is used with our microassembler, and all of our examples will be shown using this notation. The abbreviation 'RG' will often be replaced by the abbreviation of the name of the functional unit with which that particular RG is associated. Not all of the RGP's will have the microoperation

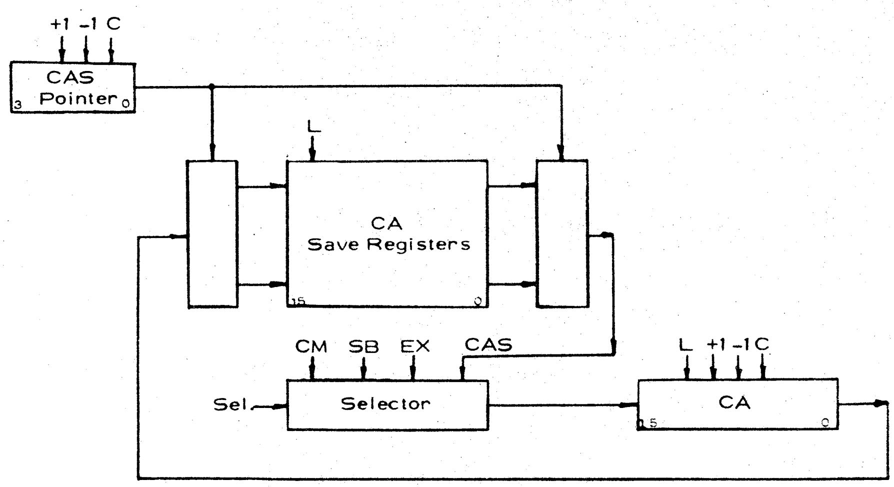
RGP := Pointer Source

associated with them. For those RGP's which do have this microoperation it will be seen that 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.

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 2.3.



Counter A, CA

Figure 2.3

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.

Symbolic Notation		Microoperation
L	CA:=CM|EX|SB|CAS	Load CA from either CM, EX, SB, or CAS. Note the use of "|" to mean "or" in the symbolic notation for this microoperation.
+1	CA + 1	Increment C A 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

Both the box labelled "Selector" in Figure 2.3 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, SB (discussed in Section 2,5),

and 4) from an element of a 16-bit wide, 16 element RG called the Counter A Save Registers, CAS.

Thus the microoperation

CA := 37

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

CA := CAS

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

Symbolic Notation		Microoperation
L	CAS := CA	Load the element of CAS pointed to by CASP with CA
+1	CASP + 1	Increment the CASP by 1
-1	CASP - 1	Decrement the CASP by 1
C	CASPC	Clear (i.e., set to zero) CASP

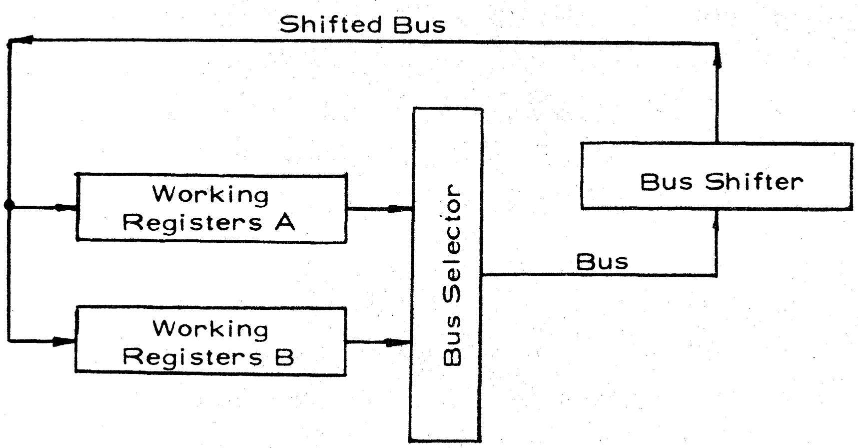
Table 2.3
Microoperations for control of CAS and CASP

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 2.3 in subsequent examples.

2.3 Bus Transport

Having introduced some elementary notions we will now examine in some detail the bus structure, the registers and functional units attached to it, and the control which can be exercised on these components. We will construct the bus structure 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 bus transport by considering a sub-system of the bus structure consisting of the Working Registers A, WA , Working Registers B, WB, and the Bus Shifter, BS, as shown in Figure 2.4. The exact nature of WA, WB, and BS is not important to us here.



Sub-system of the Bus Structure

Figure 2.4

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 bus transport may be shifted a specified amount in the BS. The output of the BS, called the Shifted Bus, SB, can then be stored in at least one of seven possible 64-bit destinations (called Bus Destinations, BD, or DESTINATION). Two such BD's are shown in Figure 2.4, 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 bus 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 BD's, the bus transport information is written

SOURCE, BS Specification

or

SOURCE

as is appropriate. The SOURCE may be stored in destinations other than BD's during a bus 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 bus transport specification is written as a list of destinations separated by commas, i.e.,

LIST := SOURCE, BS Specification

or

LIST := SOURCE

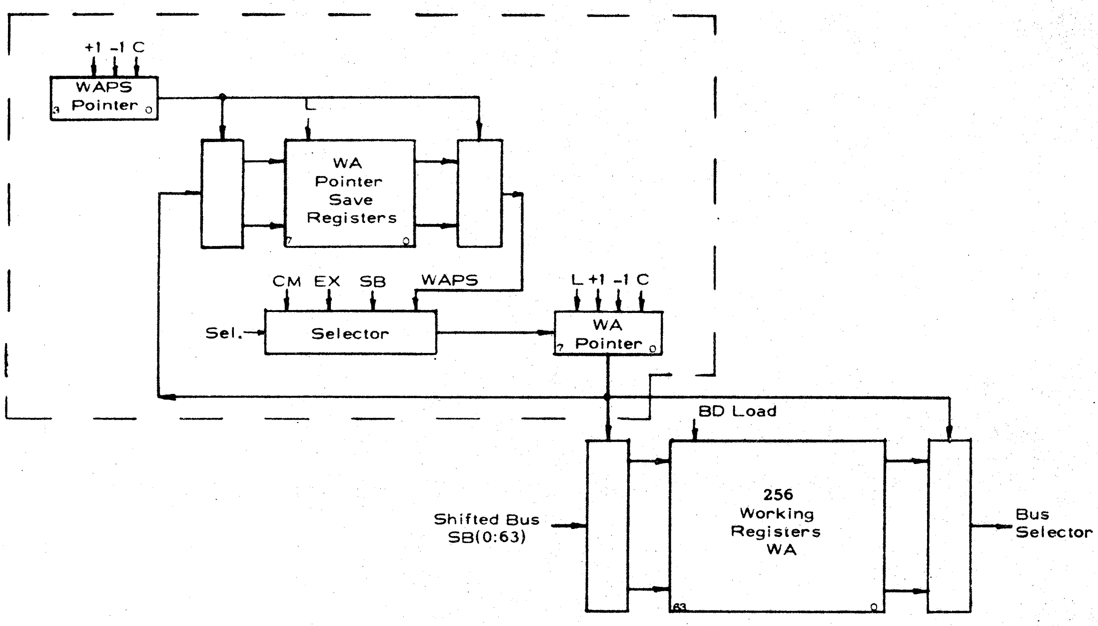
where

LIST::= $d_1, \ldots, d_n$. The value of n and the units which can serve as destinations, d_i, 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 2.5 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 the dashed-line box in Figure 2.5.) Therefore, WAP not only points to which element of WA can be used as a SOURCE for bus transport (or used as a BD), but also can be stored in an RG



Working Registers, A, WA
Figure 2.5

called the WAP Save registers, WAPS. This is identical to CA being saved. Also, as indicated in the box labelled "Selector" in Figure 2.5 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.

The WA (and WB) registers are not loaded by a microoperation but rather as a result of being chosen as a BD in a bus transport specification; thus the loading of these registers is shown by the function "BD Load" on Figure 2.5. This notation will be used in all subsequent drawings. There are 8 microoperations shown in Figure 2.5 associated with the use of WA. These are listed along with the corresponding microoperations for WB in symbolic form in Table 2.4. The actual microoperation descriptions can be extracted form the previous tables and are not repeated here.

WAP := CM| EX| SB |WAPS	WBP := CM| EX| SB| WBPS
WAP + 1	WBP + 1
WAP -1	WBP - 1
WAPC	WBPC
WAPS := WAP	WBPS := WBP
WAPSP + 1	WBPSP + 1
WAPSP -1	WBPSP - 1
WAPSPC	WBPSPC

Table 2.4

Microoperations for control of WA and WB

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 our imaginary microassembler. The format of a microinstruction is:

"A: bus transport; microoperations and data; microinstruction sequencing." ,

where

a) "A" is a symbolic name for the address of the microinstruction,

b) "bus transport" is a field giving the bus 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) "microinstruction 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 selected condition is true then choose address A_t as the address of the next microinstruction else choose A_f.

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 2.20.1. However, conditions and address functions will be introduced as needed for examples. If no condition is to be considered, i.e., if $A_t = A_f$, the sequencing information will merely be written A_t (and not "if c then A_t else A_t" where c is an arbitrary condition).

Thus, the microinstruction labelled n,

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 n+1. The action associated with <u>every</u> microoperation specified in a microinstruction is completed <u>before</u> 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 WB9 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 WB10 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): <u>all</u> conditions which arise as a result of bus transport and microoperation execution specified by a particular microinstruction, M, are testable in the <u>next</u> 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:

(a) save the conditions of the previous microinstruction

(b) execute bus transport

(c) execute microoperations

(d) execute microinstruction sequencing based on saved conditions.

Let us introduce the notion that bit 63 of the WA input to the bus selector is testable, that is, bit 63 of the element of WA which is pointed to by WAP. If we wish, for example, to test bit 63 of WA7, and if it is set to 1, jump to the microinstruction labelled BITON, else continue with the next microinstruction, we could write,

```
n-1 :   ; WAP := 7
n   :   ; if WA(63) = 1 then BITON else n+1.
n+1 :
```

We could not write

```
n   :   ; WAP :=7; if WA(63) = 1 then BITON else n+1.
```

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

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

We have introduced some standard defaults in this example:

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 microinstruction sequencing field 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 "; microinstruction sequencing." is replaced by " . ".

d) The instruction sequence shown is assumed to be located sequentially in control store and the symbolic address name is used only when n needed in the microinstruction sequencing field.

e) The symbol ■ will be used to indicate the end of the group of microinstructions in the example.

The symbolic names 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) = 1 then HERE+1 else 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 ; HERE+2.
        ; WAP+1, CA-1 ; if CA = 0 then NONE else HERE+1.
        ; if WA(63) = 1 then HERE+1 else 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 register 0 of WB and furthermore that you wish to save it in register 0 of CAS because it may be written over if a transfer is made to WB. A possible code segment is,

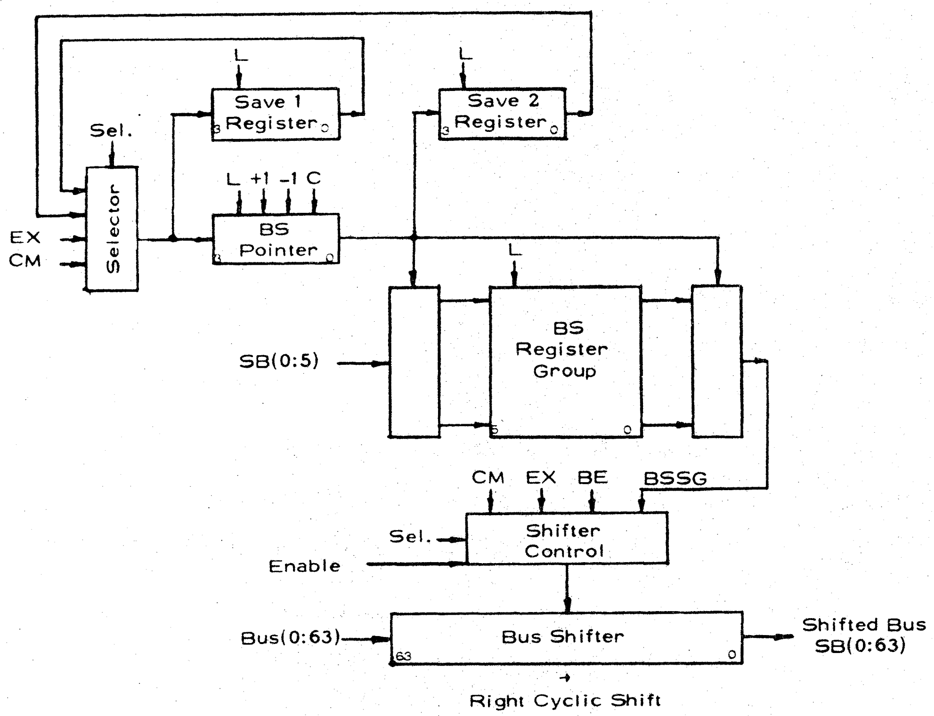
```
        ; WAPC, WAPSPC, WBPC.
WB      ; CASPC, CA := SB.
        ; CAS := CA ; HERE+2.
        ; WAP+1 ; if CA = 0 then NONE else HERE+1.
        ; CA-1 ; if WA(63) = 1 then HERE+1 else HERE-1.
WB:=WA  ; WAPS := WAP.  ■
```

If the A_f address is HERE+1 we will only write, from now on, if c then A_t. Thus, the fourth instruction of the above example would be written

; WAP+1; if CA=0 then NONE. ■

2.5 The Bus Shifter

The Bus Shifter, BS, introduced in Figure 2.4 and shown in more detail in Figure 2.6 is a 64-bit wide right cyclic shifter which can be set to shift n bits, $0 \leq n \leq 63$. There exists a dedicated bit in each microinstruction to control the BS which indicates whether or not the BS should be used (enabled) during the current bus transport. If the BS is not enabled, no shift will occur.



Bus Shifter, BS

Figure 2.6

If we wish to use the BS, the amount of shift can be selected from one of four possible sources as shown in the box labelled "Shift Control" in Figure 2.6, 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. The bus transport specification

WA:=WB

means: take the element of WB pointed to by the WBP and store it in the element of WA pointed to by the WAP without shifting it. While the bus transport specification

WA:=WB, → 3

means: take the element of WB pointed to by the WBP, shift it 3 bits right cyclic and then store it in the element of WA pointed to by 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

$$WB := WA, \leftarrow 24$$

to mean the same thing as

$$WB := WA, \rightarrow 40$$

thus using ←(left shift) or →(right shift) whichever makes the understanding of the processing clearer. The microassembler will make the above computation and insert the correct amount for left shifting.

The BS specification in the bus transport field of the microinstruction is given by

$$\left\{ \begin{matrix} \rightarrow \\ \leftarrow \end{matrix} \right\} \text{CM} \mid \text{EX} \mid \text{BE} \mid \text{BSSG}$$

where the microassembler makes the above computation only if the first alternative is selected as the source of BS control. The use of ← 11 → are dummy when used with the three other alternatives.

Having seen how the BS is controlled and how we specify this control, let us turn our attention to the BS register group Pointer, BSP. We see in Figure 2.6 that the data which can be loaded into the BSP can also be loaded into an additional register called the BS Save1 register, BSS1. If, for example, we know in advance the address of a particular register fo the BSSG, which we will want to use as shift data (e.g., some highly used shift constant), we can store this pointer in BSS1 by loading BSS1 from the CM,

$$BSS1 := CM.$$

Whenever we wish to use this stored pointer we can load it into the BSP by executing

BSP:=BSS1.

Now notice in Figure 2.6 that the BSP not only points to the element of the BSRG which can be chosen as data for the shift control unit, but also can be stored in a register called the BS Save 2 register, BSS2. Suppose we are pointing to a particular element of the BSSG for the current shift control data and in the next microinstruction we wish to have register 9 of the BSSG to be used as shift data, <u>but</u> we do not wish to loose the pointer to our current control data. The following microinstruction achieves this,

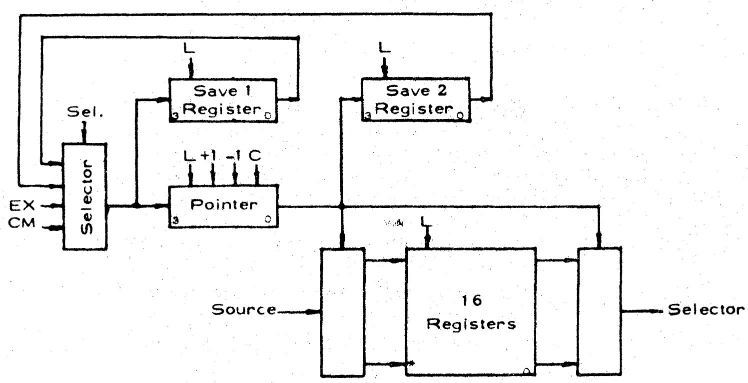
; BSS2:=BSP, BSP:=9. ■

Thus at some later time if we execute

BSP:=BSS2

the pointer information which had been saved in BSS2 would be restored.

A 16 element RG with the two Save registers and Pointer as shown in Figure 2.7 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, i.e., it will not be explicitly be drawn out each time as it was in Figure 2.6. Each SG will, however, be given a name closely associated with the particular functional unit to which it is connected as, for example, in the current discussion the SG associated with the BS is called the BSSG.



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

Typical Standard Group

Figure 2.7

Table 2. 5, below, lists the seven 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(0:5).

BSP:=CM|EX|BSS1|BSS2
BSP+1
BSP-1
BSPC
BSS1:=CM|EX|BSS1|BSS2
BSS2:=BSP
BSSG:=SB

Table 2. 5
Microoperations for control of the BS

Example:

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

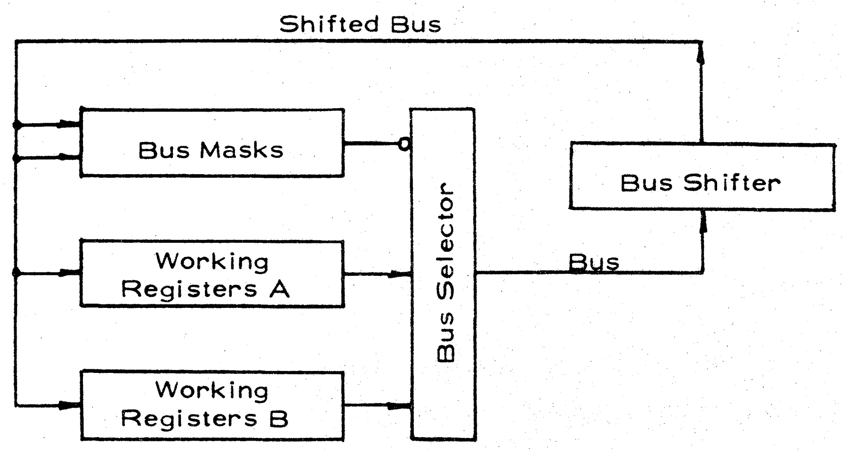
0⟷0 (63–22)	WA Adr (21–14)	WB Adr (13–6)	Lshft Data (5–0)

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

```
WB,→14          ; WAP:=SB.
WB              ; BSSG:=SB, WBPS:=WBP.
WB,→6           ; WBP:=SB.
WA:=WB,←BSSG    ; WBP:=WBPS.
```
■

2.6 Bus Masks

Let us now expand the initial bus structure given in Figure 2.4 by adding the Bus Masks (BM) as shown in Figure 2.8.



Expanded Bus Structure

Figure 2.8

The BM allow 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 ($0 \leq i \leq 63$) 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 are not an input to the bus selector but affect 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 $\longrightarrow$ (which means "input").

The SOURCE is masked during <u>every</u> bus transport by the mask which is specified to be

$$MA \vee MB$$

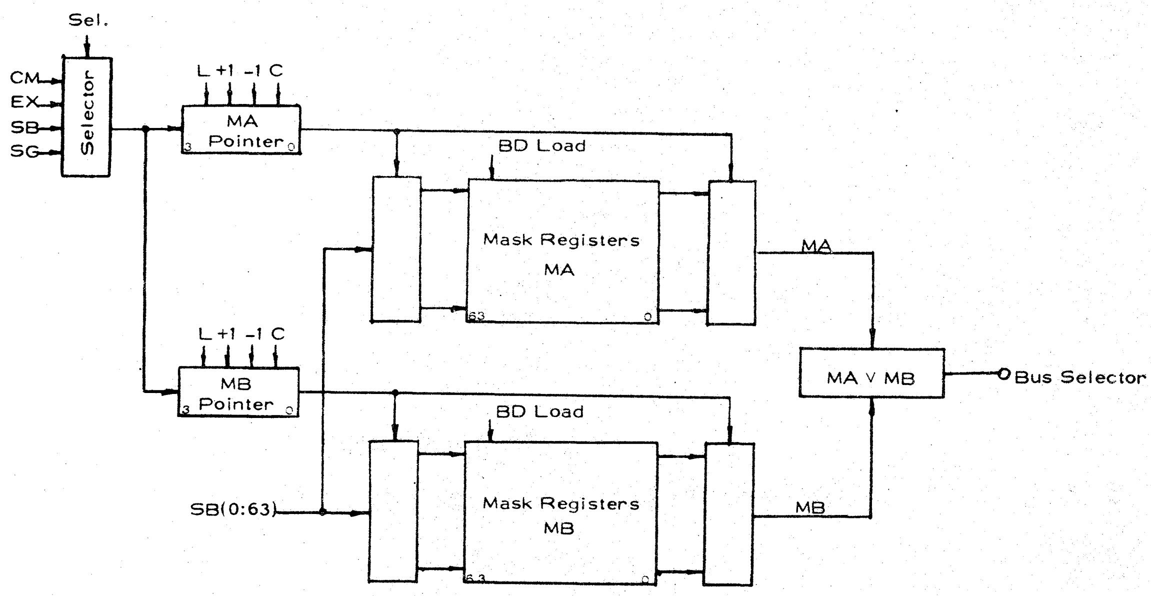
where,

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. Upon dead start, the system is

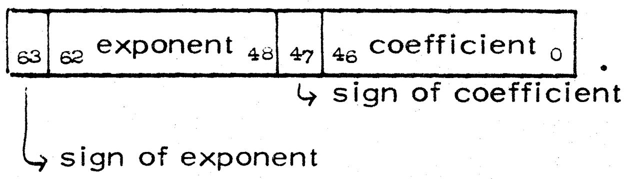


Bus Masks, MA and MB

Figure 2.9

such that the "no mask", i.e., 64 1's, is in register 0 of MA and the "bus clear mask", i.e., 64 0's, is in register 1 of MA. We will assume this to be the case throughout normal operation of the system. One can then look upon the pointer MAP as a switch for the use of the bus masks: 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, of course, not the only interpretation of the use of the BM but it is a convenient one and one which we will normally employ unless otherwise stated.

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
MB0	1	0 ←		→ 0
MB1	0	1 ←→ 1	0	←→ 0
MB2	0	←→ 0	1	0 ←→ 0
MB3	0	←→	0	1 ←→ 1

The following code will decompose a floating point number found in the register of WA pointed to by WAP and store the information as follows,

a) sign of the exponent in bit 63 of WB0
b) magnitude of the exponent shifted 1 in WB1
c) sign of coefficient in bit 63 of WB2
d) magnitude of the coefficient shifted 16 in WB3.

```
                ; MAPC.
                ; MAP+1, MBPC, WBPC.
WB:=WA          ; MBP+1, WBP+1.
WB:=WA, ← 1     ; MBP+1, WBP+1.
WB:=WA, ← 15    ; MBP+1, WBP+1.
WB:=WA, ← 16    ;              ■
```

It is suggested by this example that when one is decomposing formatted information (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, 49, and 48 to be stored in the first 4 registers of the BSSG. The above decomposition and storage could be written as the following 3 microoperations

```
          ; CA:=3, MAPC.
          ; BSPC, WBPC, MBPC, MAP+1.
WB:=WA, ←BSSG; BSP+1, WBP+1, MBP+1, CA-1; if CA ≠0 then HERE. ■
```

The MA Pointer (MAP) and the MB Pointer (MBP) both of which were used in the above examples are loadable either separately or together; thus we can execute the microoperations

MAP := CM| EX| SB| SG,
MBP := CM| EX| SB| SG, or
MAP, MBP := CM| EX| SB| SG.

The name of the SG associated with the BM is the Bus Mask Pointer (BMP) Standard Group. The following table lists the microoperations associated with MA, MB, and BMP.

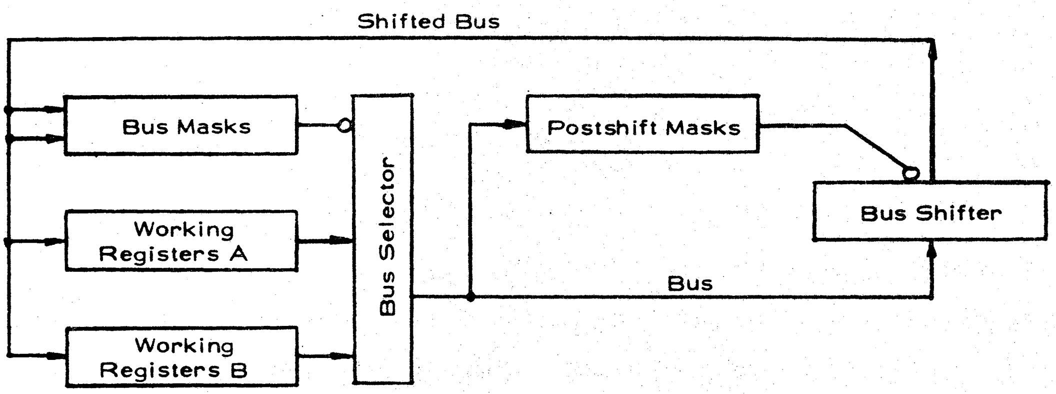
<table>
<tr><td>MAP+1
MAP-1
MAPC
MAP:=CM| EX| SB| SG</td><td>MBP+1
MBP-1
MBPC
MBP:=CM| EX| SB| SG</td></tr>
<tr><td colspan="2">MAP, MBP:=CM| EX| SB| SG</td></tr>
<tr><td colspan="2">BMP:=SB
BMPP:=CM| EX |BMPS1 | BMPS2
BMPP+1
BMPP-1
BMPPC
BMPS1 :=CM| EX| BMPS1 | BMPS2
BMPS2:=BMPP</td></tr>
</table>

Table 2.6

Microoperations for control of the BM

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 (PM) as shown in Figure 2.10.



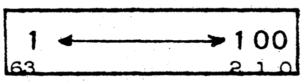
Expanded Bus Structure

Figure 2.10

One of the purposes of the PM 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 simulating a logical shift. As an example, if the bus 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 bus transport

WB:=WA, → 6

with the mask

0000001 ⟵⟶ 1
63 58 57 0

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 <u>every</u> bus transport by the mask which is specified to be

PA ∨ PG

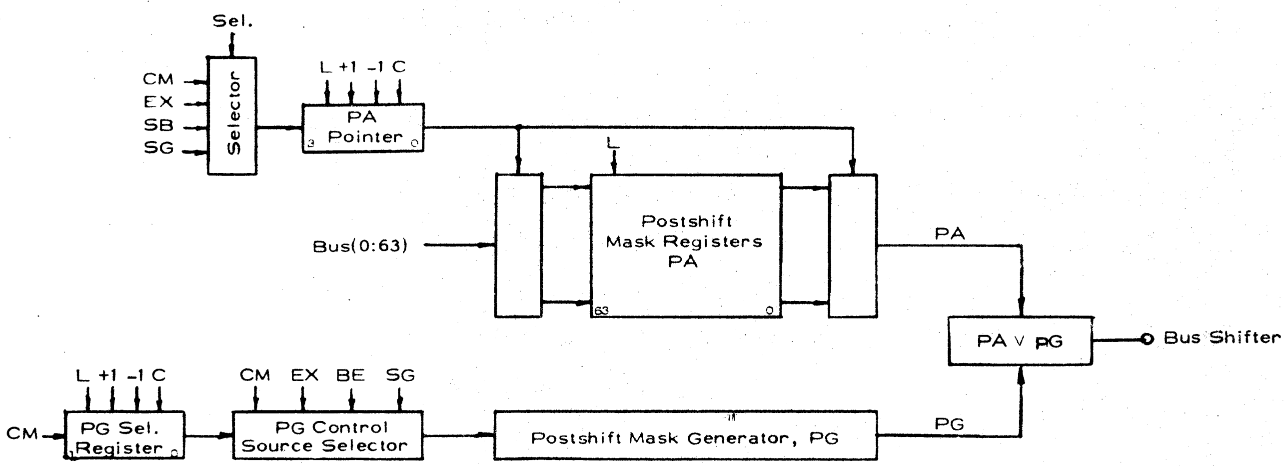
where,

PA = an element of a 64-bit wide, 16 element RG called the Postshift Mask A registers,

PG = a functional unit called the Postshift mask Generator,

∨ = 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.



Postshift Masks, PA and PG

Figure 2.11

The PG is a functional unit which can generate a string of j 0's ($0 \leq j \leq 64$) 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 be used as data for the mask fgeneration is determined by the contents of a 2-bit Postshift mask Generator Selection register (PGS) as shown in this figure and in Table 2.7 below.

Contents of PGS	Source of DATA
00	CM
01	EX
10	BE
11	SG

Table 2.7

Source of Data for Postshift Mask Generation

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

PG "arrow" n

where,

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

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

WB:=WA, $\leftarrow$ 2; PG$\leftarrow$2

and WB:=WA, $\rightarrow$ 6; PG$\rightarrow$6

Upon dead start, the system is such that the mask of all 1's is in register 0 of PA, and the mask of all 0's is in register 1 of PA. This is identical to the situation in MA. We will assume this to be the case throughout normal operation of the system. 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. This is, of course, not the only interpretation of the use of the postshift masks, but it is a convenient one and one which we shall normally employ unless otherwise stated.

Table 2.8 is a list of the microoperations associated with the PM. The first half of this table deals with PA. The second half of this table 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 AB Pointer (PABP). It is not discussed here but in Section 2.25.

Operations associated with PA
PA := BUS
PAP := CM|EX|SB|SG
PAP +1
PAP -1
PAPC
Operations associated with PG and PGSG
PGS := CM
PGS +1
PGS -1
PGSG := SB
PGP := CM|EX|PGS1|PGS2
PGP +1
PGP -1
PGPC
PGS1 := CM|EX|PGS1|PGS2
PGS2 := PGP

Table 2.8

Microoperations for the control of the PM

Let us extend the example of Section 2.6 in which we emulated a virtual machine instruction which performed a register to register transfer combined with left/right cyclic shifting. As shown below, if we use the PG we can execute an instruction which will take a given WB register (WB Adr), shift it left/right logical or cyclic (Shift & Mask Data), and then store it in a WA register (WA Adr). If the data for the instruction is in the current WB register pointed at by WBP in the form

0 ⟷ 0	WA Adr	WB Adr	Mask Data	Shift Data
63 29	28 21	20 13	12 6	5 0

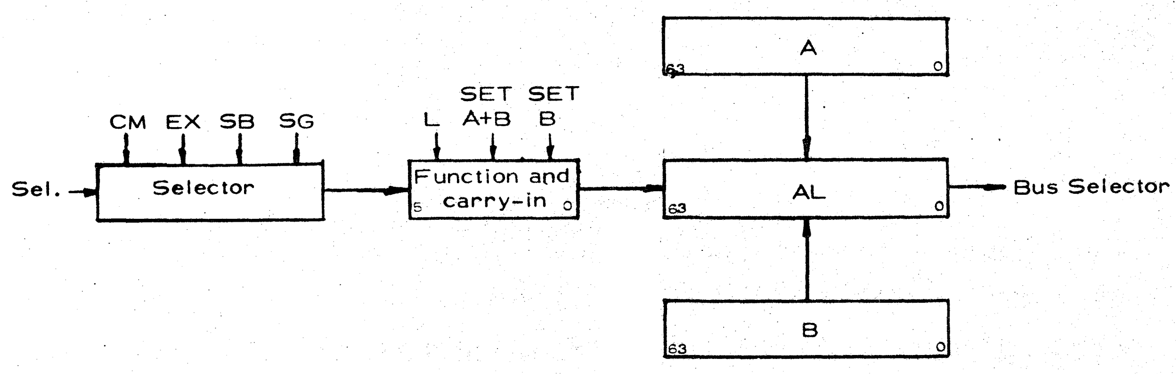
a possible code sequence would be,

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

Note well, there are two important assumptions in this example. The first is that MAP = 0 upon entry to this code, i.e., a bus mask is not applied to the source, 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: bus transport normally occurs in an unmasked fashion; if a particular code segment requires the use of a masking facility it is responsible for leaving the system in this normal state after such masking occurs.

2.8 The Arithmetical and Logical Unit

We will now add additional computational capability to the bus structure 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.



Arithmetical Logical Unit, AL

Figure 2.12

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.

ARITHMETIC	LOGICAL
A	$\overline{A}$
$A \vee B$	$\overline{A} \wedge \overline{B}$
$A \vee \overline{B}$	$\overline{A} \wedge B$
minus 1*	all 0's
$A + (A \wedge \overline{B})$	$\overline{A} \vee \overline{B}$
$(A \vee B) + (A \wedge \overline{B})$	$\overline{B}$
$A - B - 1$	$A \equiv B$
$(A \wedge \overline{B}) - 1$	$A \wedge \overline{B}$
$A + (A \wedge B)$	$\overline{A} \vee B$
$A + B$	$A \not\equiv B$
$A \vee \overline{B} + (A \wedge B)$	B
$(A \wedge B) - 1$	$A \wedge B$
$A + A$	all 1's
$(A \vee B) + A$	$A \vee \overline{B}$
$(A \vee \overline{B}) + A$	$A \vee B$
$A - 1$	A

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

Table 2.9

AL Functions

The 6 control bits which specify the current operation for the AL are the contents of the AL Function and Carry-in register (ALF) which can be loaded, ALF := CM|EX|SB|SG, set to the arithmetic addition operation A+B and set to the logical function B. 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.

ALF := CM EX SB SG
SET ALF +
SET ALF B
ALSG := SB
ALP := CM|EX|ALS1|ALS2
ALP +1
ALP -1
ALPC
ALS1 := CM|EX|ALS1|ALS2
ALS2 := ALP

Table 2.10

Microoperations for control of the AL

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 $\wedge$ B

etc.

The AL is always running. If the ALF is changed in 1 microinstruction, then the result of the newly computed function is available for bus transport in the very next microoperation. Thus the microinstructions

; ALF := all 1's, PAP +1, PGS := 'CM'.

WA := AL ; PG $\rightarrow$ 48, PAP -1. ■

will put a string of 16 1's in the WA register pointed to by WAP. The 1's will be least significant bit, b_0, justified.

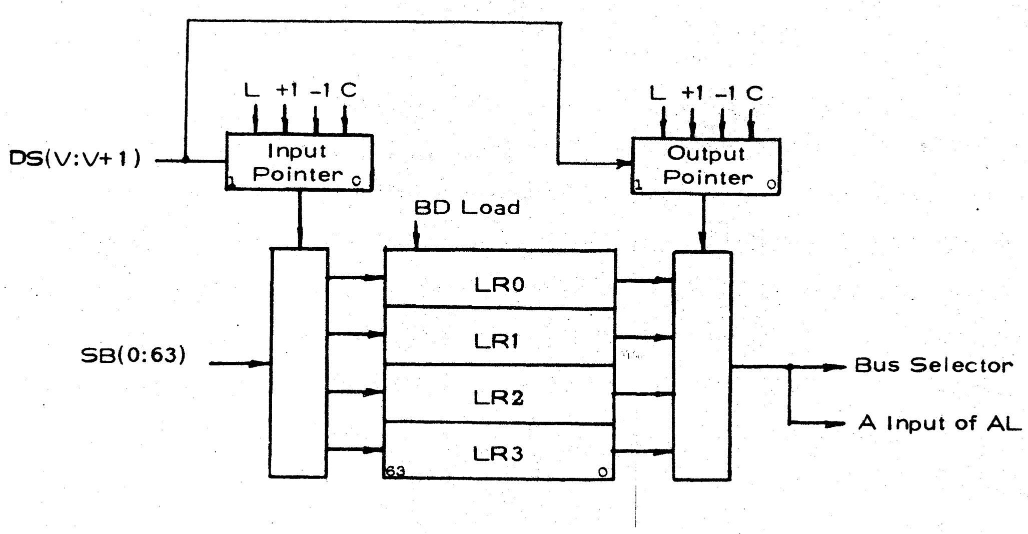
There are many testable conditions concerning the operation of the AL. A few of these are

Symbolic Notation	Condition
AL	result of AL operation all 0's
AL(0)	bit 0 of the result of the AL operation
AL(63)	bit 63 of the result of the AL operation
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, are 4 64-bit wide registers which have independent input and output pointers. The input pointer, LRIP, points to a LR which can be used as a BD for the current bus transport. The output pointer, LROP, points to a LR which can be used as either the A input to the AL or as the SOURCE for the current bus transport.



Local Registers, LR

Figure 2.13

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:V+1), 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 micro-operations associated with the control of the LR.

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

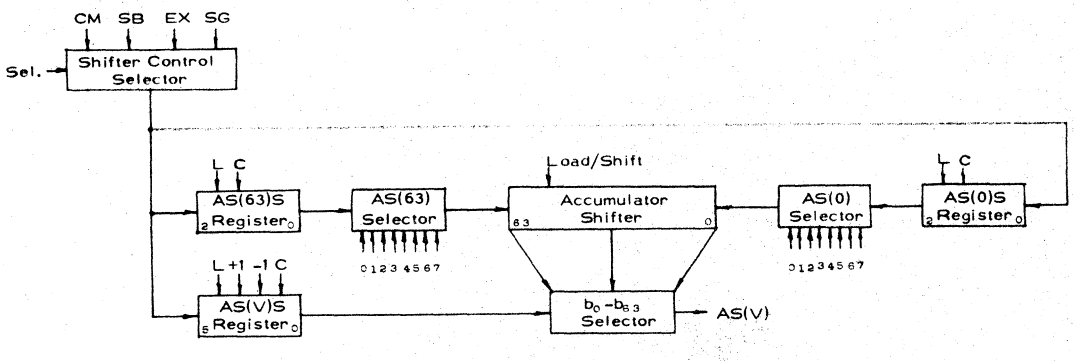
Table 2.11

Microoperations for 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 AS can serve as a bus DESTINATION, but to be read, its contents must be gated through the AL with the ALF set to AS. The AS, shown in Figure 2.14, is a 1-bit shifter which can shift left, shift right, 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 2 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 input to a selector (labeled the b_0-b_{63} 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 bus 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 the above figure 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) 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 AS.
WA := AL; PG→16 . ■

b) In Figure 2.14 it is seen that bits AS(0) and AS(63) can be filled by 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. 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. These latter inputs are of an experimental nature and uses will be demonstrated in later examples.

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. 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 is obvious. Table 2.12 lists the microoperations associated with the control of the AS. Note the AS can be set to a logical left, ASLL, or logical right, ASLR, shift.

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

Table 2.12
Microoperations for control of the AS

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(0:63), 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, the operation is specified in the "bus transport" field of the microinstruction. As an example, the microinstruction

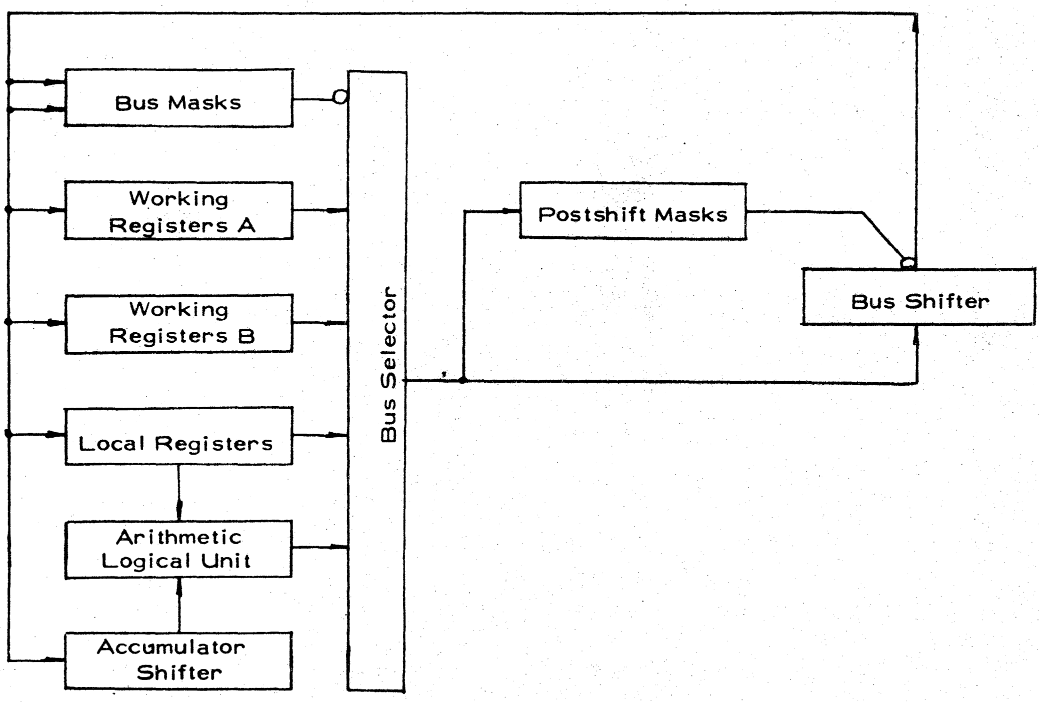
WA := AL; AS←.

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 a LR and then increments the WB pointer. If the AS is not employed during a given microinstruction, it does not appear in the specification of that microinstruction.

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

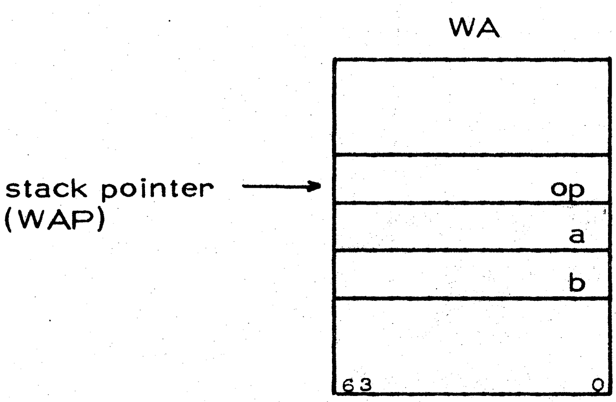


Expanded Bus Structure

Figure 2.15

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

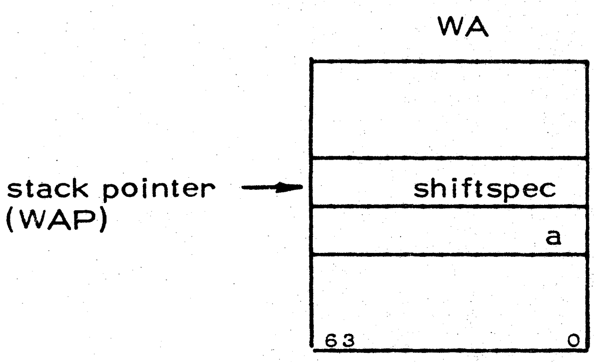
Example 1) Let us consider WA as a stack as shown below.



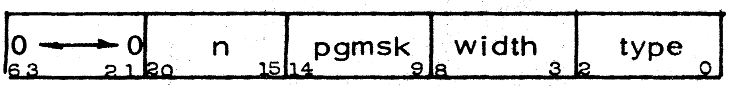
We wish to take two operands, a and b, and an arithmetical or logical operator, op, from the stack and place a op b on the new top of stack. The following microinstruction sequence does this.

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:



where type = encoding found in the table of Figure 2.14 for logical, cyclic, etc. shift,

width = width of shifter -1 , $1 \leq$ width of shifter ≤ 64

pgmsk = PG mask specification,

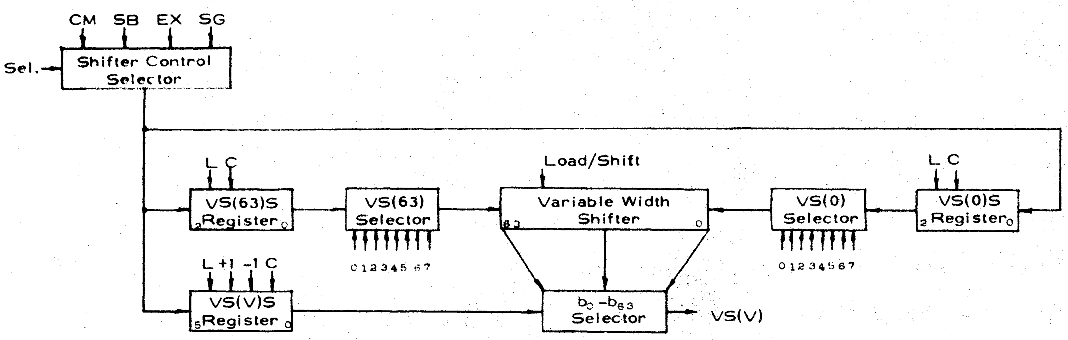
n = number of shifts -1 , $1 \leq$ number of shifts ≤ 64

The following microinstructions execute the desired operation.

```
WA                 ; AS(0)S := SB.
WA, → 3            ; AS(V)S := SB.
WA, → 9            ; PGSG := SB.
WA, → 15           ; CA := SB, WAP +1.
AS := WA           ; PGS := SG, PAP +1, SET ALF AS.
                   ; CA -1, AS ←; if CA ≠ 0 then HERE.
WA := AL           ; PAP -1. ■
```

2.11 The Variable Width Shifter

The Variable Width Shifter, VS, is a shifter functionally identical to the AS. 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 VS can be a SOURCE or DESTINATION for a bus transport. It is shown in Figure 2.16.



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

The microoperations associated with the VS are identical to those associated with the AS and are listed below in Table 2.13.

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

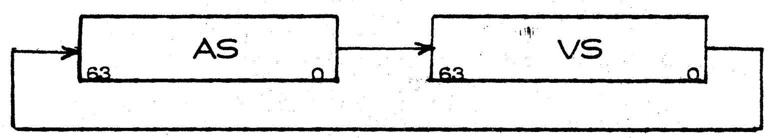
Table 2.13

Microoperations for control of the VS

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(0:63), or remain idle.

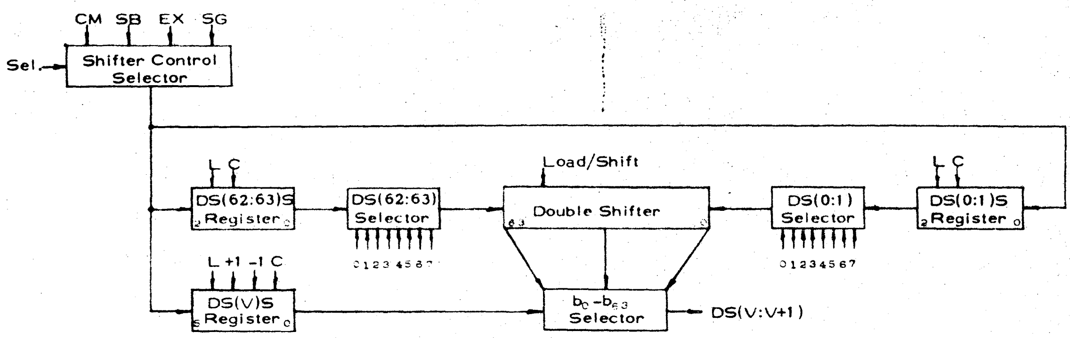
Assuming the previous AS/VS connection has been made, subsequent execution of the microoperations

$$AS \rightarrow, \; VS \rightarrow$$

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 bits 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 a SOURCE for or a DESTINATION of a bus transport.



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	$\overline{CR}$	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)	spare	spare

Double Shifter, DS

Figure 2.17

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.

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

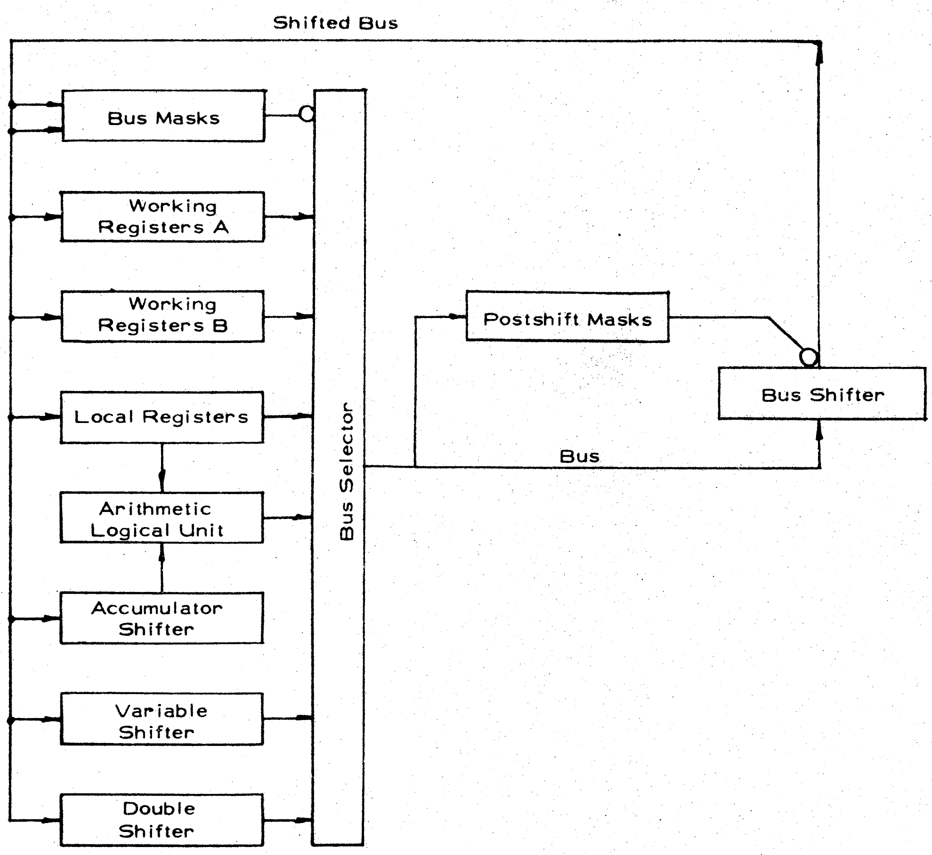
Table 2.14

Microoperations for control of the DS

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(0:63), 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 bus structure is shown in Figure 2.18.



Expanded Bus Structure

Figure 2.18

Example 1)

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

a) Load the LR with the following constants

LR0 := 0
LR1 := 1
LR2 := 1
LR3 := 2

b) Clear the AS (considered here as an accumulator)

c) Set the AL to addition

d) Transfer the data to the DS

e) Do the following 32 times and then do (f)

i) if DS(0:1) ≡ 00 then accumulate LR0 + AS
if DS(0:1) ≡ 01 then accumulate LR1 + AS
if DS(0:1) ≡ 10 then accumulate LR2 + AS
if DS(0:1) ≡ 11 then accumulate LR3 + AS

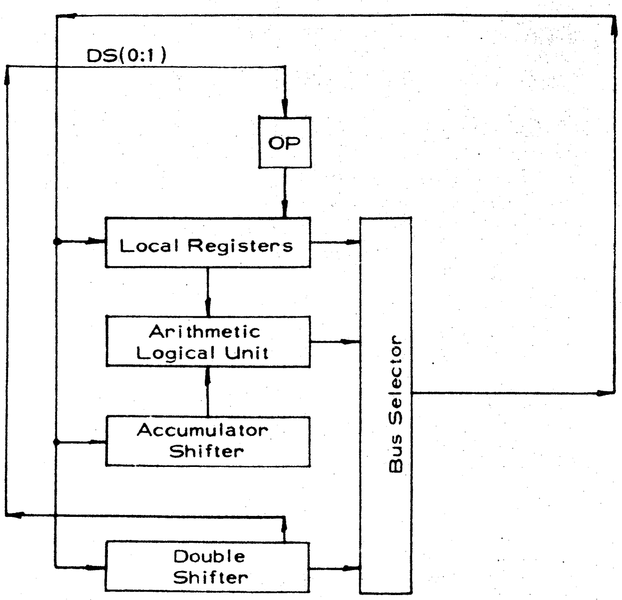
ii) shift DS →

f) Store the accumulated result which is in AS

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;  ALF := all 1's, LRP +1 , PAP +1 .
LR := AL        ; PG →63 , LRIP +1 , DS(V)SC, PAP -1 .
LR := LR        ; ALF := LR + AS, LRIP +1 .
LR := LR, ← 1 ; CA := 31 , LROP := DS.
AS := AL        ; CA -1 , DS → 1 , LROP := DS; if CA ≠ 0 then
                  HERE.
WA := AL        . ■
```

The subset of the bus 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.



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

Figure 2.19

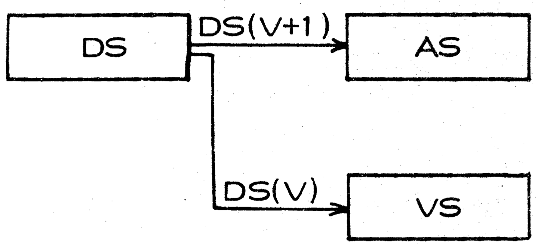
Example 2)

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 left 32 bits of the current WB register and then odd numbered bits (b_1, b_3, etc.) in the right 32 bits of this register so that the result appears as

b_{63} ..... b_5 b_3 b_1	b_{62} ..... b_4 b_2 b_0

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

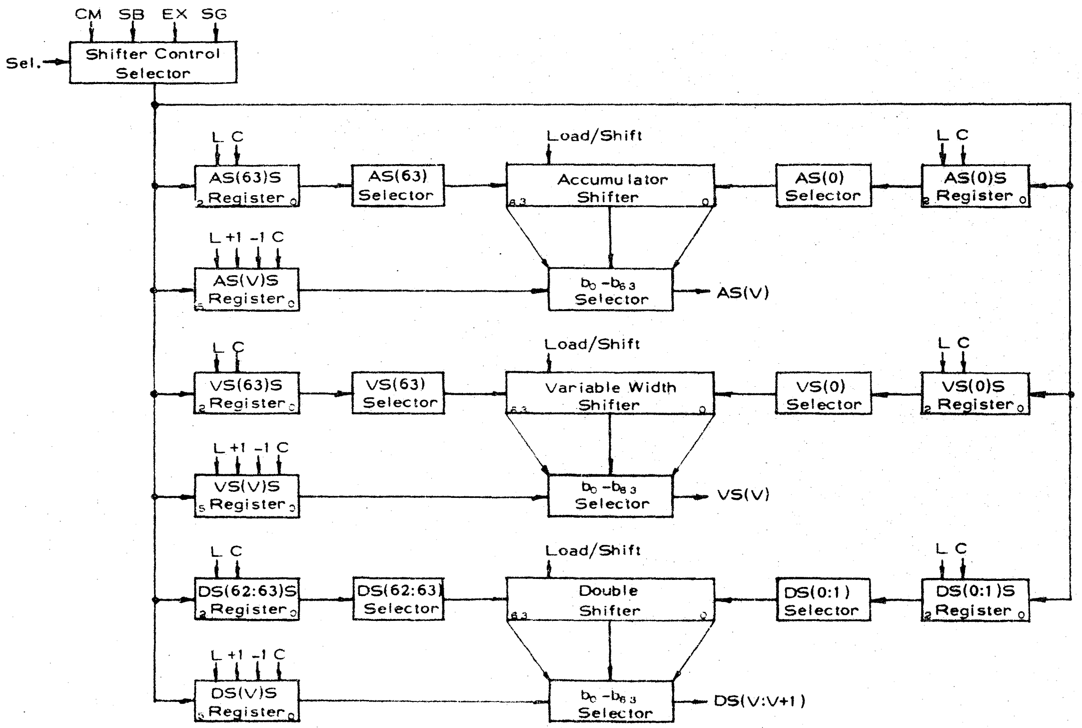


one can accomplish the following way:

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

2.13 The Common 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.



AS, VS, and DS Control

Figure 2.20

The SG which is associated with this selector is called the Common Shifter SG. Various shifter control data can be stored in this SG for various shifter interconnections and then used in environment prologues. The microoperations associated with the CS SG are shown in Table 2.15.

CSP := CM|EX|S1|S2
CSP +1
CSP -1
CSPC
CSS1 := CM|EX|S1|S2
CSS2 := CSP
CSSG := SB

Table 2.15

Microoperations for control of the CS SG

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.

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

Table 2.16

Parallel CS Microoperations

2.14 Loading Masks

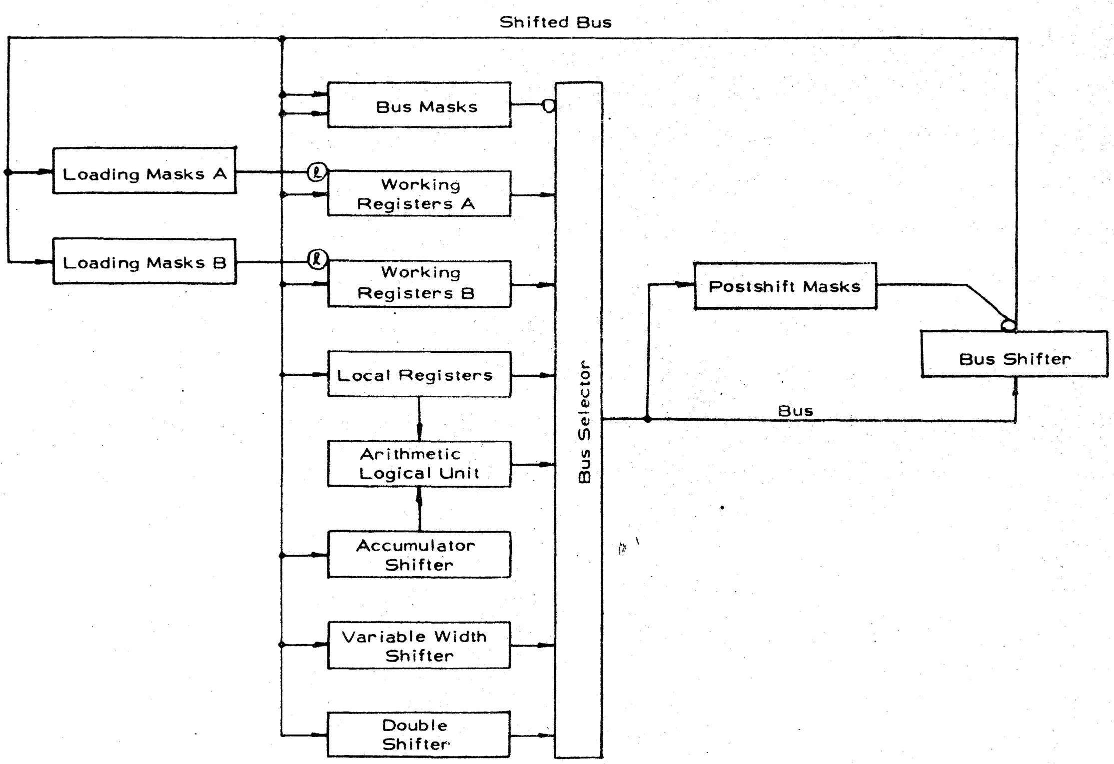
Associated with WA there is a SG of loading masks called Loading Masks A, LA. Associated with WB there is a SG of loading masks called Loading Masks B, LB. In what follows we will describe only LA; LB is identical in function. The purpose of the loading masks, LA and LB, is to be able to specify which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

00 . . . . . . 00 111111
63   6 5   0

were pointed at by the LA pointer, LAP, then, when the bus transport

WA := AL

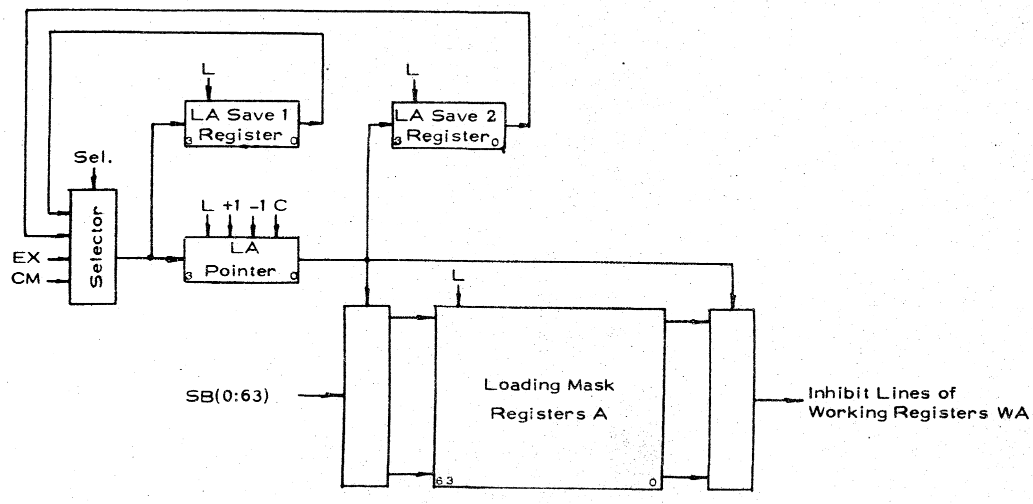
is executed, bits SB(0:5) would be gated into the WA register pointed to by WAP in bit positions b_0 through b_5 respectively while bits b_6 through b_{63} would not change their value. When WA is selected as a SOURCE for bus transport the mask LA acts in the following fashion: if bit i $(0 \leq i \leq 63)$ of the mask is a 1, then bit i of WA is transmitted. If bit i of the mask is a 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 bus structure with the loading masks added.



Expanded Bus Structure

Figure 2.21

Figure 2.22 shows a more detailed sketch of LA; LB, not shown, is identical.



Loading Mask Registers A, LA

Figure 2.22

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.

<table>
<tr><td>LA := SB(0:63)</td><td>LB := SB(0:63)</td></tr>
<tr><td>LAP := CM| EX| S1 | S2</td><td>LBP := CM| EX| S1 | S2</td></tr>
<tr><td>LAP +1</td><td>LBP +1</td></tr>
<tr><td>LAP -1</td><td>LBP -1</td></tr>
<tr><td>LAPC</td><td>LBPC</td></tr>
<tr><td>LAS1 := CM| EX| S1 | S2</td><td>LBS1 := CM| EX| S1 | S2</td></tr>
<tr><td>LAS2 := LAP</td><td>LBS2 := LBP</td></tr>
</table>

Table 2.17

Microoperations for control of LA and LB

Upon the dead start, the system is such that the "full load" and "full read out" mask, i.e., 64 1's is in register 0 of LA and register 0 of LB. We will assume this to be the case throughout normal operation of the system. One can then look upon the pointers LAP and LBP as selection switch for the use of the loading masks. If LAP = 0 then no loading mask is applied to WA, if LAP $\neq$ 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, but it is a convenient one and one which we will normally employ unless otherwise stated.

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 WB0 leaving the high order 16 bits the same. If the mask

0 ⟷	0	1	⟷ 1
63	48	47	0

is in LB9, the following microinstruction sequence accomplishes this:

; LBP := 9, WBPC.
WB := DS, $\rightarrow$ 16 ; LBPC. ■

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

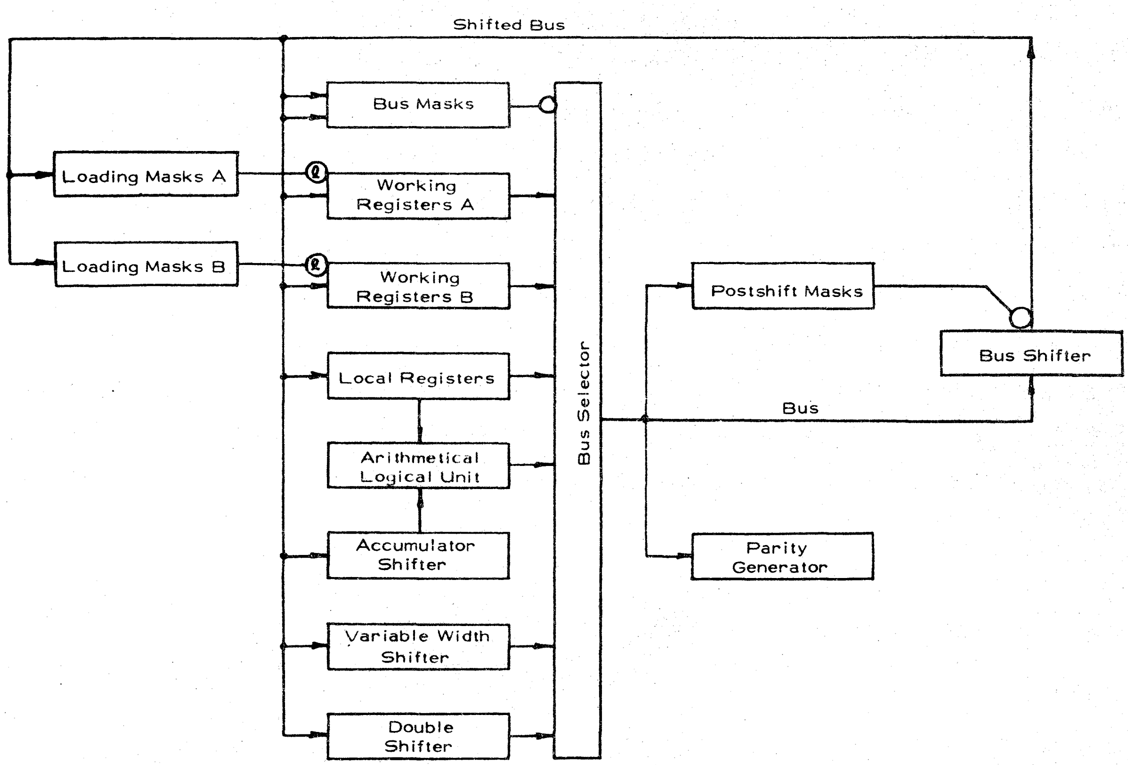
; ALF := all 1's, LBP := 9.
; PGS := CM, PAP +1 .
AL ; PG $\rightarrow$ 16, LB := SB, PAP -1. ■

generates the mask and stores it in LB9. 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 Parity Generator

The parity generator 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 processing wherein 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.



Expanded Bus Structure

Figure 2.23

2.16 The Bit Encoder

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

b_{63} b_{62} . . . b_1 b_0

.

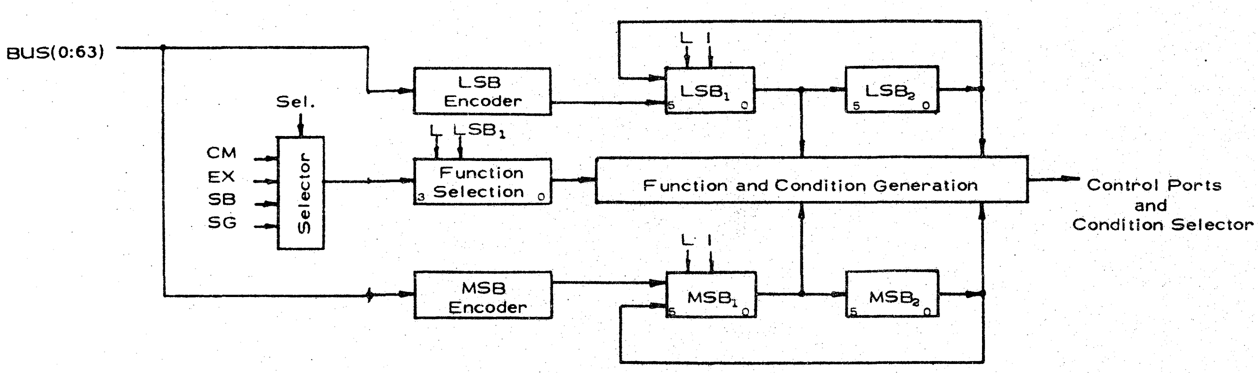
Let us scan this string of bits from the right to the left, i.e., starting with bit b_0 and finishing with bit b_{63}. LSB will denote the value of the subscript of the first, nonzero bit encountered while MSB will denote the value of the subscript of the last nonzero bit encountered in this string.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

$0 \longleftrightarrow 0 \; 1 \leftrightsquigarrow 1 \; 0 \longleftrightarrow 0$
b_{63} . . . b_k . . . b_j . . . b_0

$\uparrow$ MSB=k $\quad$ $\uparrow$ LSB=j

where $k \geq j$. If $k = j$ there are, of course, no bits between b_k and $_j b_j$; if $k > j$, the $k-j-1$ bits between b_k and b_j may be any arbitrary string of $(k-j-1)$ 0's and 1's. If the BUS $\equiv 0$, then a condition is set true and LSB and MSB are set to 0.

There is, on the MATHILDA System, a functional unit called the Bit Encoder, BE, which, during every bus transport, encodes the MSB and LSB associated with the BUS. The BE, shown in Figure 2.24, can also manipulate these quantities.



Bit Encoder, BE

Figure 2.24

During each bus transport an "LSB encoder" and an "MSB encoder" determines the LSB and MSB associated with the current 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 16 different functions with the variables LSB_1, LSB_2, MSB_1, and MSB_2. These functions are given in Table 2.18 where L_i = MSB_i-LSB_i, i = 1, 2.

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

Table 2.18

Bit Encoder Functions

Which particular function is to be the output of the BE is determined by the contents of the BE Function Selection register

$$BEF := CM|EX|SB|SG.$$

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.,

$$BEF := LSB_1 .$$

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 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 BE SG.

Notation	Microoperation
BEL Load	LSB_2 :=LSB_1 and then LSB_1 :=LSB encoding
BEM Load	MSB_2 :=MSB_1 and then MSB_1 :=MSB encoding
BELM Load	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 register from CM|EX|SB|SG
SET BEF LSB_1	Set BEF to LSB_1
BEPGL	Set PG to generate from b_0 if BE is control input
BEPGM	Set PG to generate from b_{63} if BE is control input
BE SG := SB	
BEP := CM|EX|S1|S2	
BEP +1	
BEP -1	
BEPC	
BES1 := CM|EX|S1|S2	
BES2 := BEP	

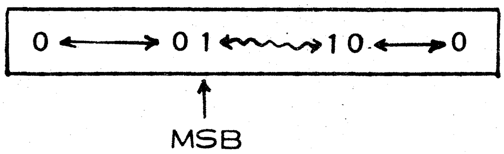
Table 2.19

Microoperations for control of BE

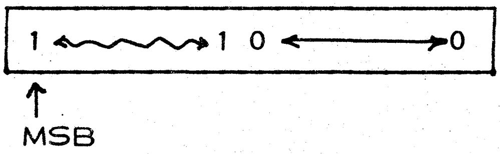
Example 1

We wish to take the contents of the WA register pointed to by 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.

WA before shift



WA after shift



The following microinstructions accomplish this.

DS := WA; BEM Load, BEF := MSB_1 +1.
WA := DS, ← BE. ■

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

$$\left[\frac{(MSB_1 - LSB_1)}{2}\right] + 1.$$

times, provided we shift the data LSB_1 places to the right before counting. The following microoperations accomplish this,

DS := WA	; BELM Load, BEF := LSB_1 .
DS := DS, →BE	; BEF := [$(MSB_1 - LSB_1)/2$] +1.
	; CB := BE .
	; ALF := all 0's, LRPC.
AS, LR := AL	; ALF := all 1's, LRP +1 , PAP +1 .
LR := AL	; PG→ 63, LRIP +1 , DS(V)SC, PAP -1 .
LR := LR	; ALF := LR + AS, LRIP +1 .
LR := LR, ←1	; CB -1 , LROP := DS.
AS := AL	; CB =1 , DS → 1 , LROP := DS; if CB ≠ 0 then HERE
WA := AL	. ■

Note that this code is only 2 instructions longer than the code on page 43. Counter B, CB, used in this example can be loaded from the BE (see Section 2.23.1).

2.16.1 Bit Encoder Conditions

There are conditions associated with each of the BE functions. These are listed below along side the entries of Table 2.18 as a matter of convenience.

	Function	Conditions
F	LSB_1	LSB_1 = all 0's
F	LSB_1-1	LSB_1-1 = all 0's
F	MSB_1	MSB_1 = all 1's
F	MSB_1+1	MSB_1+1 = all 1's
F	L_1	$MSB_1=LSB_1$ (i.e., $L_1=0$)
F	$\Delta L=L_2-L_1$	$L_2 = L_1$, sign (L_2-L_1), $L_2 = 0$
F	LSB_2-LSB_1	$LSB_2 = LSB_1$, sign (LSB_2-LSB_1)
F	MSB_2-MSB_1	$MSB_2 = MSB_1$, sign (MSB_2-MSB_1)
	$\left[\frac{F}{2}\right]+1$ [ ] ::= integer part of	same as above

Table 2.20

Bit Encoder Functions and Conditions

The important thing to understand about the conditions is that <u>all</u> of them are avialable for testing irrespective of which particular BE function is specified. The LSB and MSB encoding process yields a testable condition which indicates whether bits b_0 through b_{63} are all zero; this condition is noted 'BUS $\equiv$ 0'. Thus we can write, for example,

$$\text{if BUS} \equiv 0 \text{ then } A_t \text{ else } A_f.$$

And, as a last condition on BE, we can test BE(0), i.e., bit 0 of the BE output.

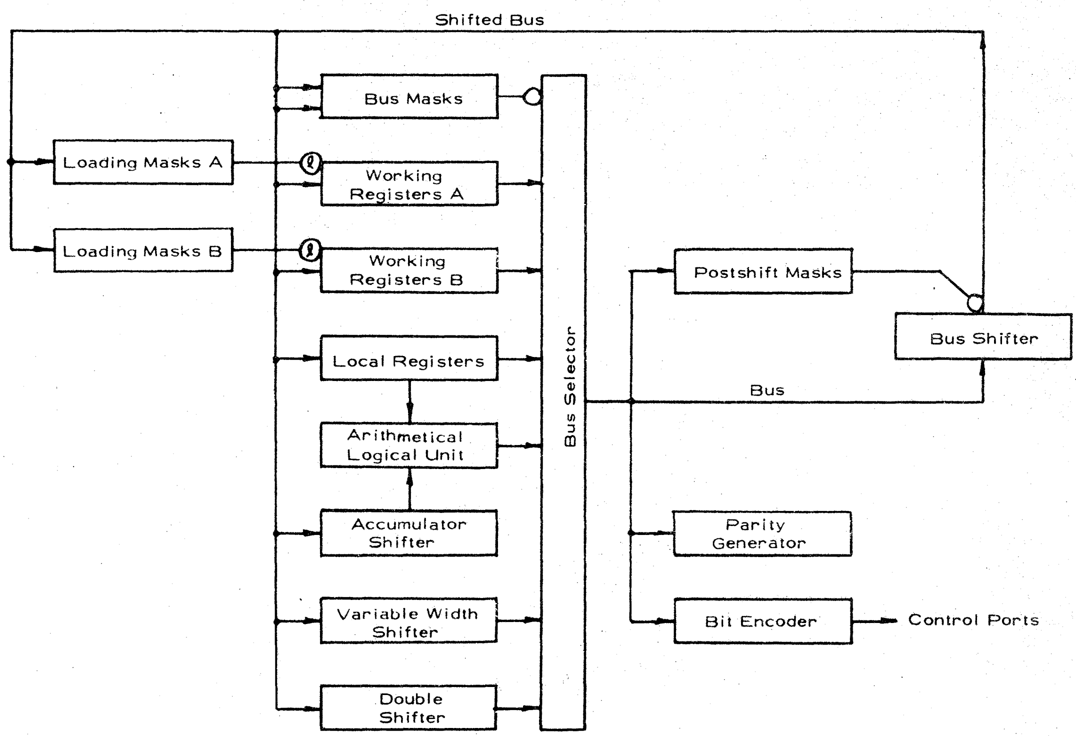
Example

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

VS ; BELM Load.
; if L_1 =0 then ONEBIT.

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

Since the BE has as its inputs 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.

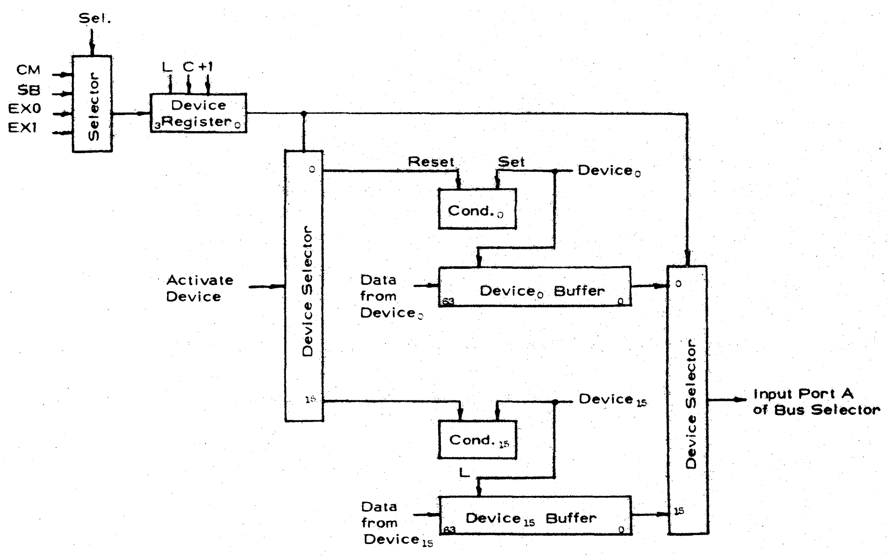


Expanded Bus Structure

Figure 2.25

2.17 Input Ports

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. IA is shown in Figure 2.26; IB, not shown, is identical.



Input Port A, IA

Figure 2.26

The particular device which is selected to be read is pointed to by a Device Register. There are two conditions associated with a selected device: a) data available, IADA, and b) data condition, IADC. All devices must be able to set the first condition. The second condition can be set by devices which can transmit two different sorts of information, for example control information and data. When a device is read, both the IADA and IADC conditions are reset. The microoperations associated with the control of IA and IB are given in Table 2.21.

Notation	Microoperation
IAA	Activate Port, i.e., read IA
IAD:=CM|EX0|SB|EX1	Load IA Device Register from CM|EX0|SB|EX1 *
IADC	Clear IA Device Register
IAD +1	Increment IA Device Register
IBA	Activate Port, i.e., read IB
IBD:=CM|EX0|SB|EX1	Load IB Device Register from CM|EX0|SB|EX1 *
IBDC	Clear IB Device Register
IBD +1	Increment IB Device Register

Table 2.21

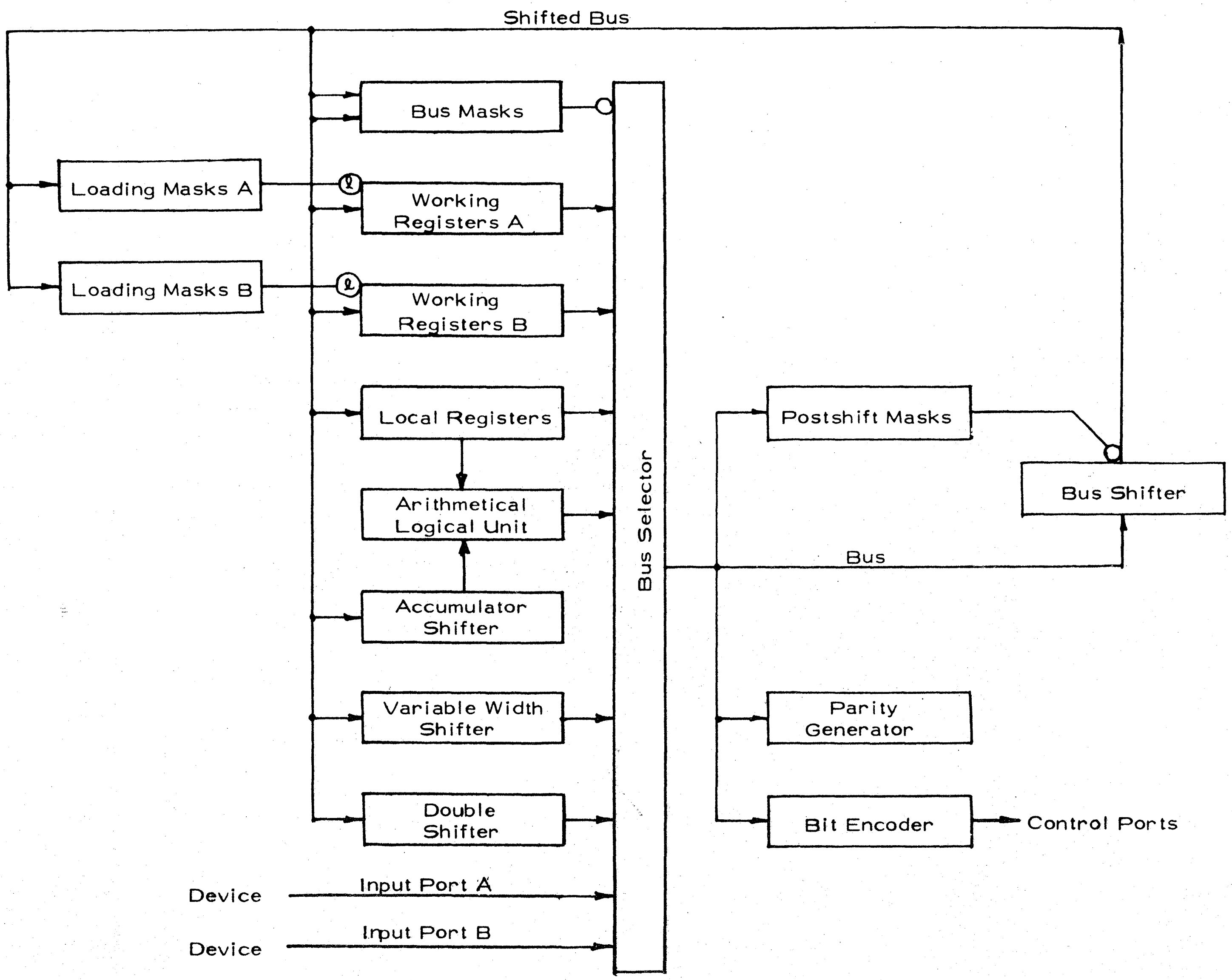
Microoperations for control of IA and IB

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

```
                ; IAD := 9.
                ; IAA; if IADA then HERE + 1 else HERE.
      AS := IA. ■
```

The expanded bus structure can now be shown as Figure 2.27.

* See Section 2.20.5 for a description of EX0 and EX1.

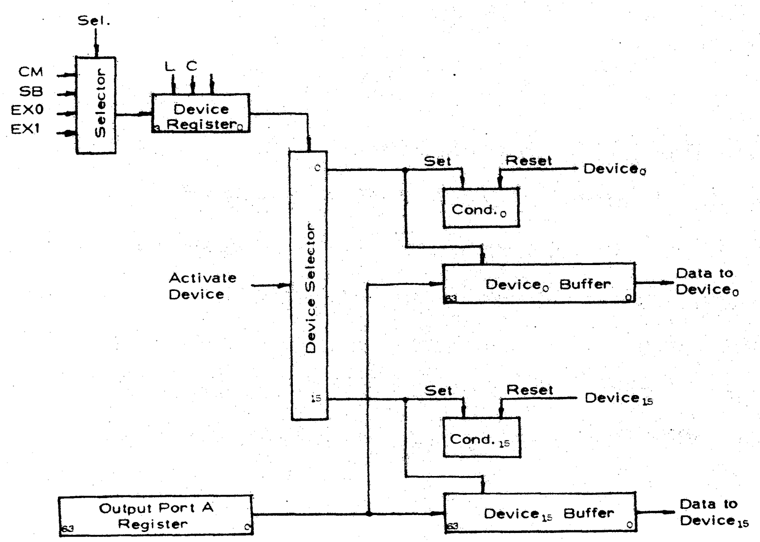


Expanded Bus Structure

Figure 2.27

2.18 Output Ports

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 bus DESTINATIONS whereas OC and OD are loaded from the BUS and cannot be selected as bus DESTINATIONS, but must be loaded by a microoperation. OA is shown in Figure 2.28; OB, OC, and OD, not shown, are identical.



Output Port A, OA

Figure 2.28

The particular device which is selected for output is pointed to by a Device register. There is a condition associated with a selected device: space available, OASA. The microoperations associated with the control of OA and OC are shown in Table 2.22. The microoperations for OB are identical to those for OA and the microoperations for OD are identical to those for OC.

Notation	Microoperation
OAA	Activate Port, I.e., write OA
OAD:=CM|EX0|SB|EX1	Load OA Device Register from CM|EX0|SB|EX1
OADC	Clear OA Device Register
OCA	Activate Port, i.e., write OC
OCD:=CM|EX0|SB|EX1	Load OC Device Register from CM|EX0|SB|EX1
OCDC	Clear OC Device Register
OC:=BUS	Load OC from BUS(0:63)

Table 2.22

Microoperations for control of OA and OC

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,

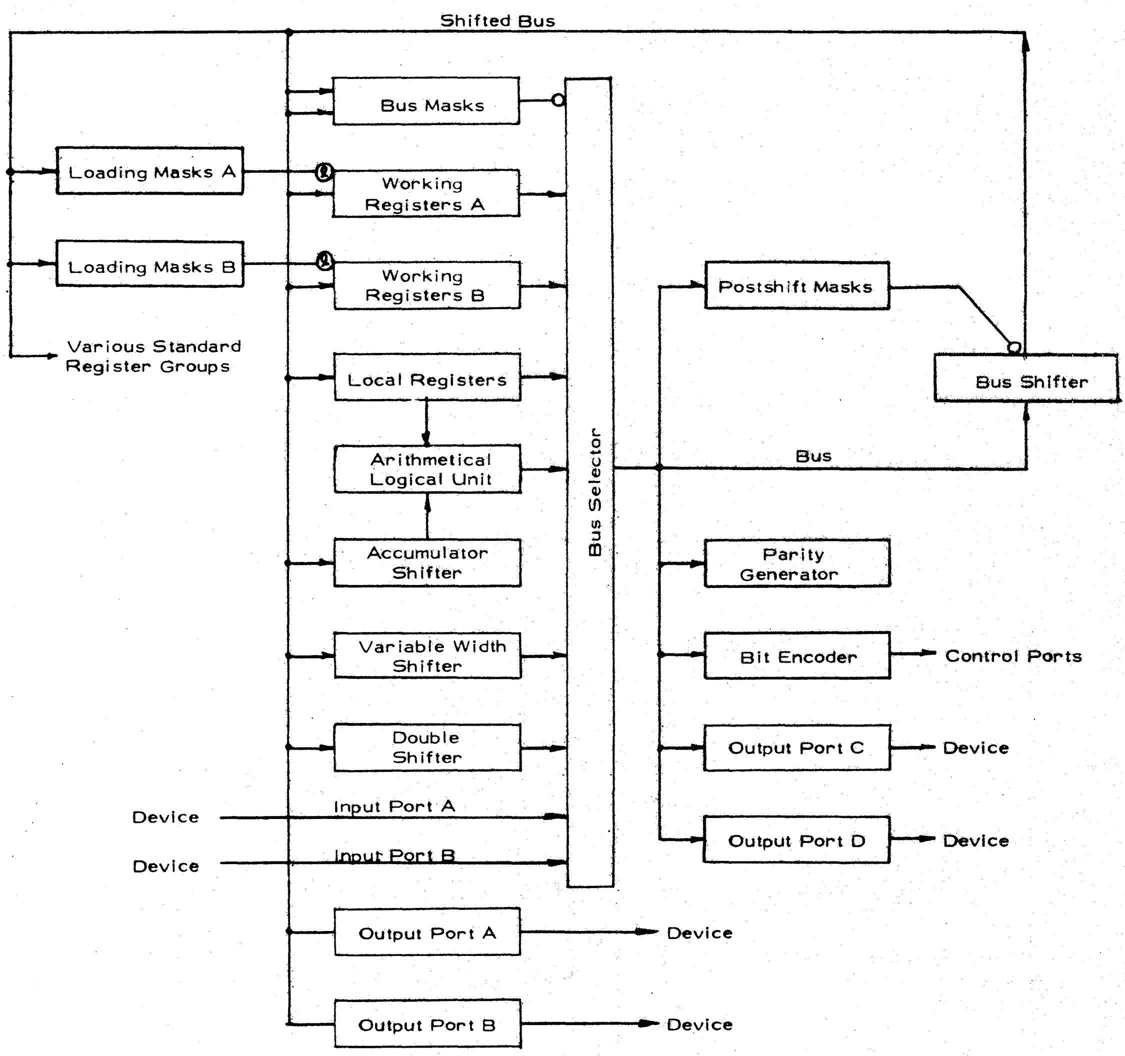
```
AL  ; OC := BUS, OCD := 13.
    ; if OCSA then HERE+1 else HERE.
    ; OCA. ■
```

There is one additional feature associated with the "activate" micro-operation. 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 device can, for example, treat this extra bit as a data condition. The micro-operations 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 undefined.

2.19 The Bus Structure

With the introduction of the output ports in the previous section we have completed a description of (with only very minor modifications) the MATHILDA Bus Structure, the registers and functional units attached to it, and the control which can be exercised on these components. The Bus Structure is now shown in Figure 2.29.



MATHILDA Bus Structure

Figure 2.29

Let us summarize some of the information with respect to bus SOURCEs and DESTINATIONs. We have the following SOURCEs and DESTINATIONS for a bus transport:

a) SOURCEs for Bus Transport

WA
WB
LR
AL
VS
DS
IA
IB

b) DESTINATIONS for 64-bit Load of SB with BD 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 and any or all of the shifters, i.e.,

$BD_b [, AS] [, VS] [, DS]$:= 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 of appropriate microoperations. Also, a subfield of the SB (always a contiguous string starting with bit b_0) 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.

There are three important restrictions on the above bus transport specifications:

a) the specifications WA := WA or WB := WB are not allowed,

b) the specification LR := LR is only meaningful when LRIP $\neq$ LROP,

c) one cannot use a mask (MA, MB, PA, LA, LB) and load the register containing that mask in the same microinstruction.

2.20 The Control Unit

The control unit of the MATHILDA system, shown in Figure 2.1 on page 4, 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.

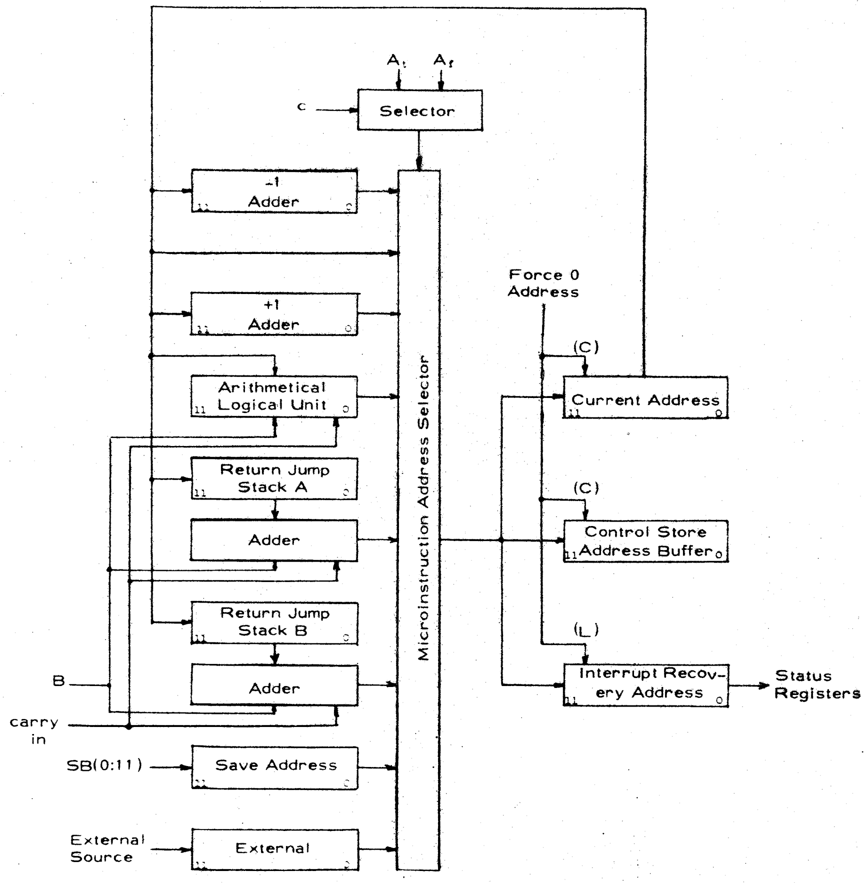
2.20.1 Microinstruction Sequencing

The microinstruction sequencing hardware is a physical embodiment of the "if c then A_t else A_f" clause we have been using in our 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 2.23.

Notation	Interpretation
A-1	Current address - 1
A	Current address
A+1	Current address + 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 2.23
Microinstruction Address Sources

These address sources are realized by providing a microinstruction address bus which is shown in a limited form in Figure 2.30.



Microinstruction Address Bus (Preliminary)

Figure 2.30

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 2.23 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 2.23 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, and the Status Registers shown in Figure 2.30 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 straight forward and need not be dealt with. It should be mentioned, however, that Control Store addresses are interpreted modulo the size of the Control Store.

2.20.2 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 the address of the current microinstruction and the "B input" is data specified in the current microinstruction. The CUAL is shown as in Figure 2.31.

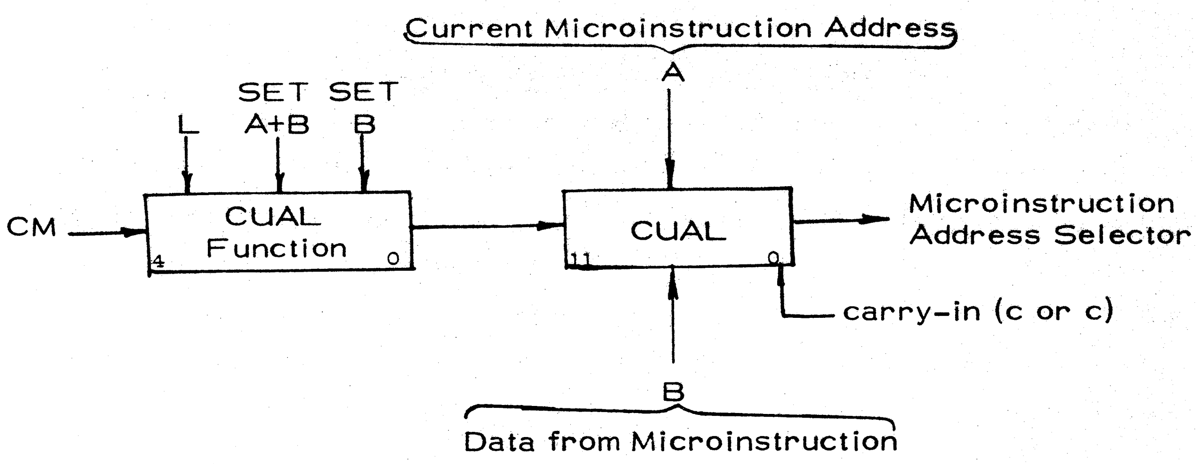


Figure 2.31

Control Unit Arithmetical Logical Unit

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 the carry-in as shown in Table 2.24.

CISB	Carry-in
0	$\bar{c}$
1	c

Table 2.24
Carry-in Selection

Example 1) Suppose the CUALF is set to A+B; this is a relative jump. If CISB = 0, the specification

if c then CUAL else HERE

can be interpreted to mean:

if c then HERE + B else HERE.

Whereas, if CISB = 1, the specification can be interpreted to mean:

if c then HERE + B + 1 else HERE.

Example 2) Suppose the CUALF is set to B; this is an absolute jump. 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.

In our microassembler, 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 imcompatible 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,

if TRUE then CUAL+1 else CUAL ; ,

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 2 6-bit fields in the micro-instruction which we shall call T and t. T and t are input into a function box which makes the computations shown in Table 2.25. 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.

BISB	B data
00	0
01	Tt
10	$^{t}\text{sign}^{t}$
11	T0

Table 2.25
B data Selection

The notation $t_{sign}t$ 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=10 we then have a relative addressing capability of ±32. The notations Tt and T0 denote concatenation.

In our microassembler, the specification of the BISB will be given implicitly. One specifies the B value explicitly as a decimal number in the address specification and this will dictate the setting of the BISB.

We will hence forth write the CUAL specifications as

CUAL (A, B) + Carry-in.

Both CU and A are 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

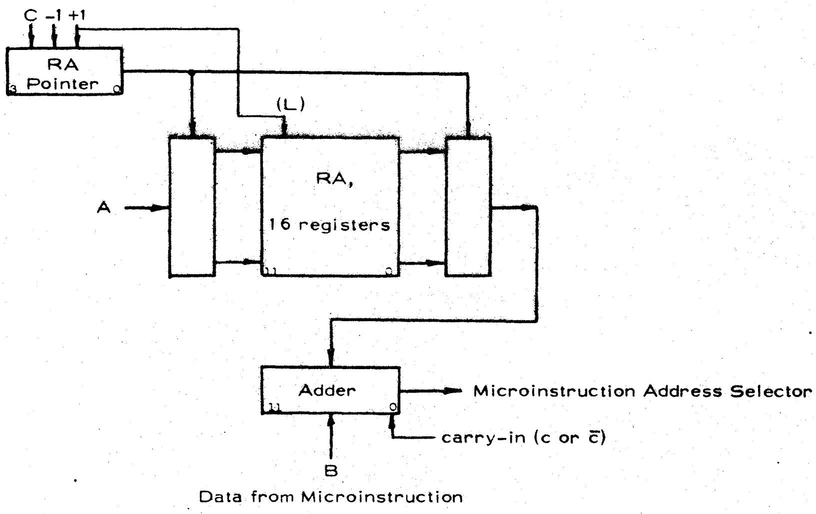
where B is a signed integer, $-2048 \leq B \leq 2048$, when combined in an arithmetic function with A, but may obviously lie in the interval $0 \leq B \leq 4095$ when used for absolute jumps.

Example 1) If the CUALF is set to A+B and BISB=10, then the specification
if c then AL(-18).
can be interpreted to mean
if c then HERE-18 else HERE+1.

Example 2) If the CUALF is set to A+B and BISB=10, then the specification
if c then AL(12) else AL(12)+1
can be interpreted to mean
if c then HERE+12 else HERE+13
thus giving a conditional branch to one of two sequentially located microinstructions.

2.20.3 Return Jump Stacks A and B

There are two return jump stacks associated with the microinstruction addressing facility. They are called RA and RB. Each is a 12-bit wide, 16 element RG. RA is shown in Figure 2.32; RB, not shown, is identical.



Return Jump Stack A, RA

Figure 2.32

The microoperations associated with RA are shown in Table 2.26. The instructions for RB are identical.

Notation		Microoperation
+1 $\wedge$ (L)	RA ↓	Increment RAP and then Load RA with the address at the current microinstruction
-1	RA ↑	Decrement RAP
c	RAPC	Clear the RAP

Table 2.26

Microoperations for control of RA

Whenever the top of the RA stack is used in the computation of the address A the next microoperation, the microoperation RA ↑ is executed, i.e., the stack pointer is automatically maintained any time something is added to the stack or whenever the stack is used in an address computation. The use of RA is specified by writing

RA + B + carry-in.

This is seen immediately from Figure 2.32. 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 B=0 and the carry-in=1.

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

n : ;RA ↓ ;AL(m).

m:

⋮

j: ; ;RA+1.

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

n: ;RA ↓ ;RB+1.

stores the current address in one stack while simultaneously using the other stack as a source in the computation of the address of the next microinstruction.

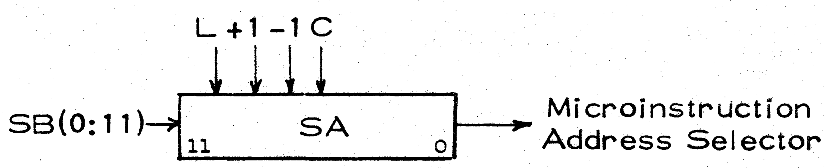
Example 3) A conditional return entry point can be obtained by using the specification

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

An important point must be raised here. It was stated on page 12: "The action associated with <u>every</u> microoperation specified in a microinstruction is completed <u>before</u> the next microinstruction is executed." There is <u>only one</u> exception to this rule and it is the action associated with the microoperation RA↓ (and RB↓ obviously). It was not important at the time the rule was introduced, but it is important now. The action associated with RA↓ and RB↓ require 2 microinstruction cycles to be completed and not 1 microinstruction cycle. Thus, if one loads RA in a given microinstruction, RA cannot be used as an address source in the very next microinstruction executed. The same is, of course, true for RB. (This is discussed further in Section 3.2.1.)

2.20.4 The Save Address Register

The Save Address register, SA, is shown in Figure 2.33.



The Save Address Register, SA

Figure 2.33

The microoperations associated with this register are shown in Table 2.27.

SA := SB
SA +1
SA -1
SAC

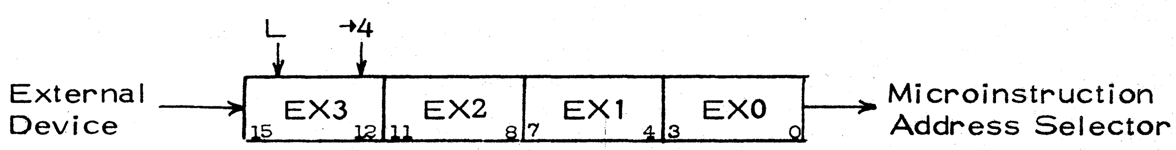
Table 2.27

Microoperations for control of SA

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 and recovering from an interrupt (see Sections 2.20.8 and 2.20.6 respectively).

2.20.5 The External Register

The External Register, EX, is a 16-bit wide right cyclic shifter which shifts 4 bits 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. The 16- bits of the EX register can be thought as consisting of four 4-bit bytes as shown in Figure 2.34.



The External Register, EX

Figure 2.34

The microoperations associated with EX are shown in Table 2.28.

Notation	Microoperations
EX Load	Load the External register
EX $\rightarrow$ 4	Shift the External register 4 bits right cyclic

Table 2.28
Microoperations for control of EX

EX can not only be used as a possible source for the address of the next microinstruction, 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, the right most 12-bits are used, i.e., bytes EX2, EX1, and EX0. In fact, in all circumstances (except in conjunction with the Device Registers of the input/output ports) the datum from the EX is always considered to be a contiguous string of bits of the required width starting with b_0. For example if EX is designated as the control source for the BS, the bits EX(0:5) are used to specify the shift amount. When EX is used as a data source for the loading of input/output port Device Registers (IAD, IBD, OAD, OBD, OCD, and ODD) both bytes EX1 and EX0 are considered data; not contiguous data, but 2 separate 4-bit data items.

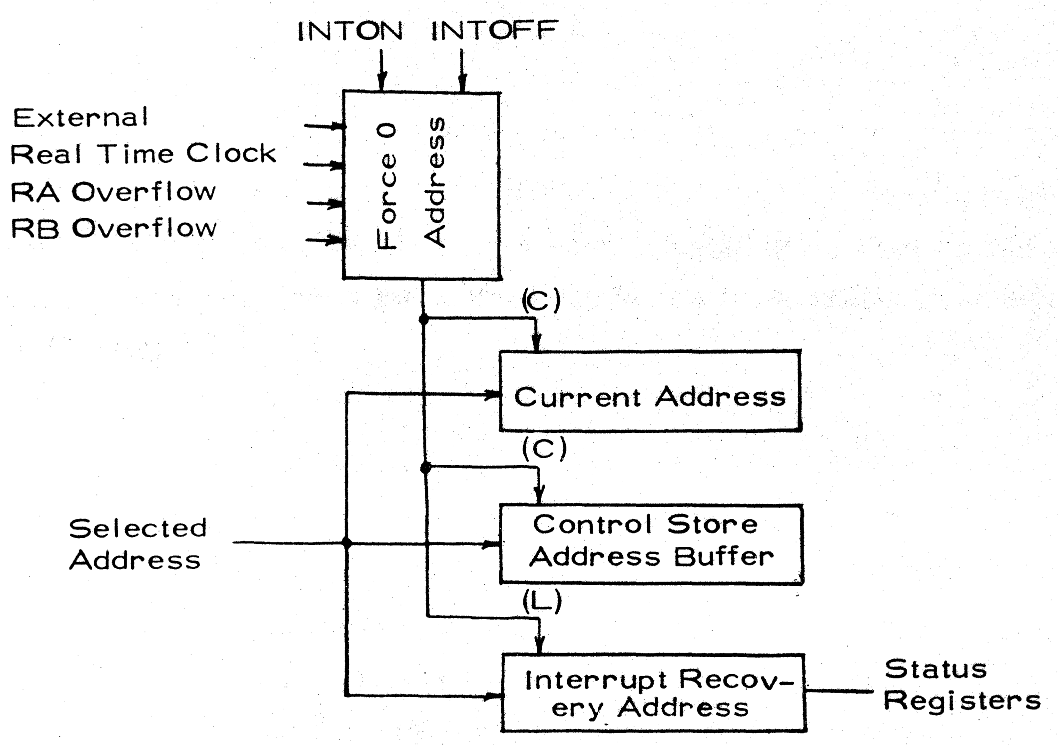
2.20.6 The Force 0 Address Capability

There are 4 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 2.29.

Force 0 Address Conditions
External Signal
Real Time Clock Overflow
RA Overflow
RB Overflow

Table 2.29
Force 0 Address Conditions

An external device may be connected to the External Signal condition to interrupt the operation of MATHILDA. A Real Time Clock, RTC, (Section 2.22), is available in the system which can count up to 60 sec. The overflow of the RTC causes the next microinstruction address to 0. If either RA or RB overflow, i.e., we have stacked more than 16 addresses, we will also force the address to 0. This capability is shown in the following way:



The Force 0 Address Capability

Figure 2.35

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 Registers explained in Section 2.23.3. 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.

2.20.7 The Microinstruction Address Bus

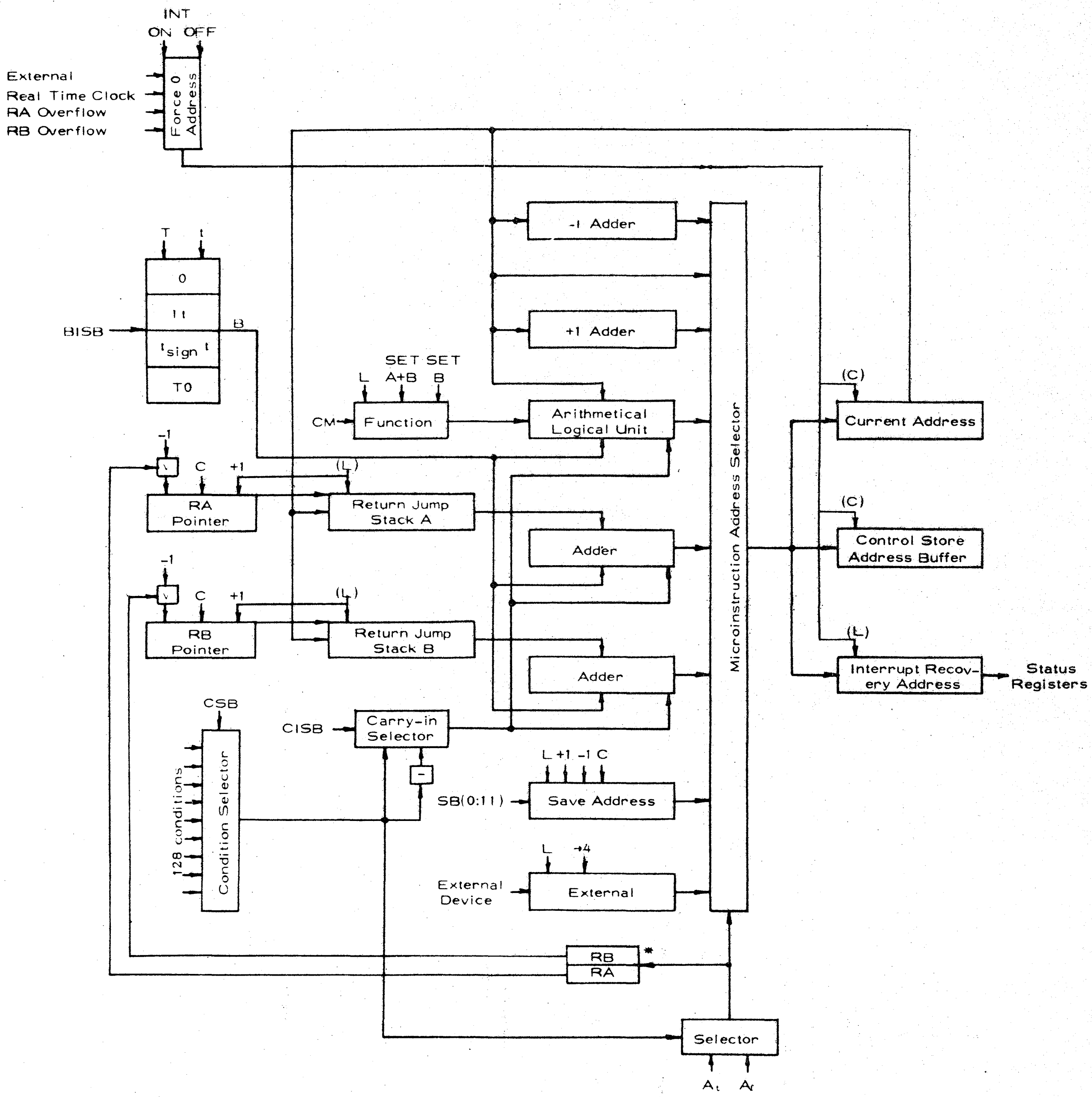
Having gained insight into the nature of the various address sources which can be used during microinstruction sequencing, we can now present a more detailed picture of the microinstruction address bus and it is shown as Figure 2.36. Because the number of control elements is small, they are also shown on this figure.

The microoperations associated with the control unit are brought together, for convenience, in Table 2.30. 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
CS Load

Table 2.30.

Microoperations associated with the Control Unit



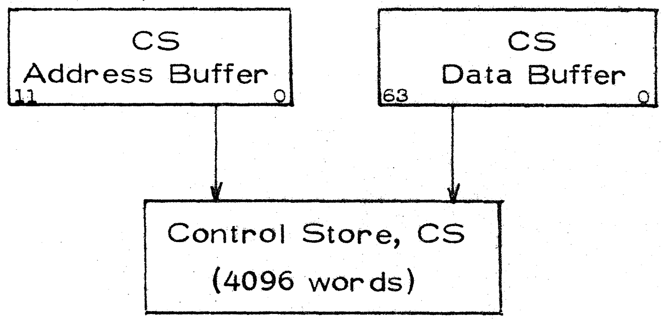
Microinstruction Address Bus (Detailed)

Figure 2.36

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

2.20.8 Control Store Loading

Control Store has, of course, both an address buffer and a data buffer, as shown below.



The CS Address Buffer is loaded from the Microinstruction Address Selector as shown in Figure 2.30. The CS Data Buffer is actually Device number 15 associated with Output Port A, OA. Let A 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 A+1 as the address of the next microinstruction.

Example

Load the contents of WA1 into the CS storage location specified by the rightmost 12 bits of WA0.

	; WAPC, OAD := 15.
WA	; SA := SB, WAP +1.
OA := WA	; if OASA then HERE +1 else HERE.
	; OAA.
	; CS Load; SA.
	; continue ■

2.21 The Conditions, Condition Selector, and Condition Registers

There is the possibility of testing 128 conditions in the system. At this writing there have been 100 specified, leaving a reasonable amount of expandability in the system. The conditions and their symbolic notation are given in Table 2.31.

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.

Unit	Symbolic Notation	Condition
AL	AL	are bits AL(0:63) $\equiv 0$
	ALOV	AL carry-out and borrow-in bit
	AL(0)	bit 0 of AL input to bus selector
	AL(63)	bit 63 of AL input to bus selector
	ONEOV	1's complement overflow
	TWOOV	2's complement overflow
AS	AS(0)	bit 0 of the AS
	AS(V)	the variable bit of the AS
	AS(63)	bit 63 of the AS
BE	LSB1	is $LSB_1 \equiv 000000$
	MSB1	is $MSB_1 \equiv 111111$
	L1	is $L_1 = 0$ (i.e., $MSB_1 = LSB_1$)
	L2	is $L_2 = 0$ (i.e., $MSB_2 = LSB_2$)
	LSB1 -1	is $LSB_1 - 1 = 000000$
	MSB 1+1	is $MSB_1 + 1 = 111111$
	LSBD	is $(LSB_1 - LSB_2) = 0$
	SGNLSBD	sign of LSBD (SGNLSBD=0$\Rightarrow$LSBD$\geq$0)
	MSBD	is $(MSB_1 - MSB_2) = 0$
	SGNMSBD	sign of MSBD (SGNMSBD=0$\Rightarrow$MSBD$\geq$0)
	LD	is $L_1 - L_2 = 0$
	SGNLD	sign of LD (SGNLD=0$\Rightarrow L_1 \geq L_2$)
	BEPGD	BE postshift mask generator director BEPGD=0$\Rightarrow$L, BEPGD=1$\Rightarrow$M
	BE(0)	bit 0 of the output of the BE
BP	BP	BUS parity, BP=1 $\Rightarrow$ odd parity
BUS	BUS	BUS(0:63) $\equiv 0$
CA	CA	is CA zero
	CA(3)	bit 3 of CA
	CA(4)	bit 4 of CA
	CA(5)	bit 5 of CA
	CA(6)	bit 6 of CA
	CASPOV	CASP $\equiv$ 1111 (CASP overflow)
CB	CB	is CB zero
	CB(3)	bit 3 of CB
	CB(4)	bit 4 of CB
	CB(5)	bit 5 of CB
	CB(6)	bit 6 of CB
	CBSPOV	CBSP $\equiv$ 1111 (CBSP overflow)

(cont.)

Unit	Symbolic Notation	Condition
CR	CR	output of condition save registers
CU	EXDA RAPOV RAPUN RBPOV RBPUN INT CUALOV	data available on EX RAP ≡ 1111 (RAP overflow) RAP ≡ 0000 (RAP underflow) RBP ≡ 1111 (RBP overflow) RBP ≡ 0000 (RAP underflow) INT=1⇒INTON, INT=0⇒INTOFF CUAL overflow
DS	DS(i),i=0,...,15 DS(j),j=V,V+1	the indicated bit of the DS the variable bits of the DS
I/O	IADA IADC IBDA IBDC OASA OBSA OCSA ODSA	data available on IA data condition on IA data available on IB data condition on IB space available on OA space available on OB space available on OC space available on OD
LR	LR(0) LR(63)	bit 0 of LR input to bus selector bit 63 of LR input to bus selector
RTC	RTCOV	Real Time Clock overflow toggle
SB	SB(0) SB(1) SB(62) SB(63)	bit 0 of the shifted bus bit 1 of the shifted bus bit 62 of the shifted bus bit 63 of the shifted bus
System	TRUE FALSE	a binary one a binary zero
VS	VS(0) VS(V) VS(63)	bit 0 of the VS the variable bit of the VS bit 63 of the VS
WA	WA(0) WA(15) WA(63) WAPOV WAPSPOV	bit 0 of WA input to bus selector bit 15 of WA input to bus selector bit 63 of WA input to bus selector WAP ≡ 11111111 (WAP overflow) WAPSP ≡ 11111111 (WAPSP overflow)
WB	WB(0) WB(15) WB(63) WBPOV WBPSPOV	bit 0 of WB input to bus selector bit 15 of WB input to bus selector bit 63 of WB input to bus selector WBP ≡ 11111111 (WBP overflow) WBPSP ≡ 11111111 (WBPSP overflow)

Table 2.31

Partial Listing of System Conditions*

* See also Table 2.38

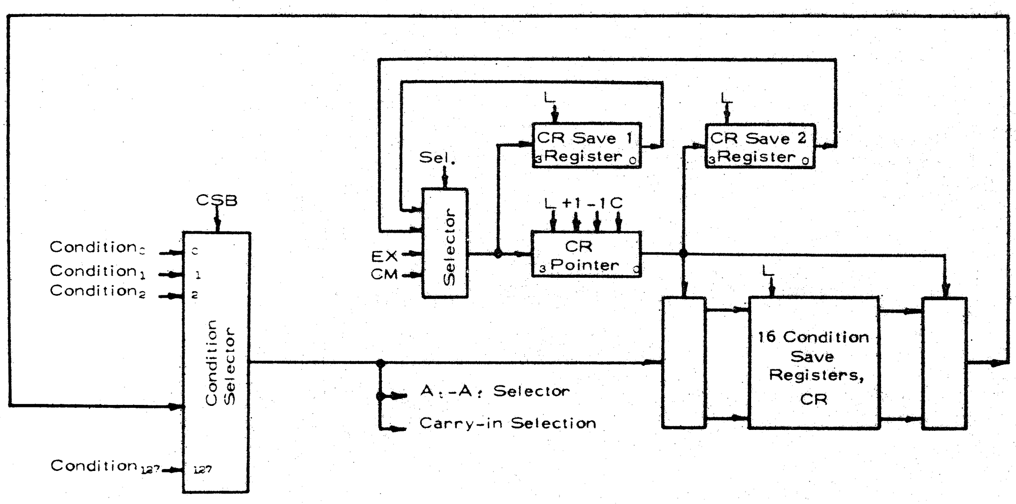
All 128 conditions are input into a condition selector. There are 7 bits in each microinstruction, called the Condition Selection Bits, CSB, which select a particular condition. The selected condition is input into

a) the A_t -A_f address selector (Section 2.20.1),

b) the carry-in selector (Section 2.20.2), and

c) a SG called the Condition Save Registers, CR.

This is shown in Figure 2.37.



Condition Selector and Condition Registers

Figure 2.37

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 are given below in Table 2.32.

CR := SC
CRP := CM | EX | S1 | S2
CRP +1
CRP -1
CRPC
CRS1 := CM | EX | S1 | S2
CRS2 := CRP

Table 2.32

Microoperations for control of CR

In the loading microoperation CR := SC (Selected Condition), we can, instead of using the notation SC, use the symbolic notation given in Table 2.31. Thus, for example, if we wish 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 that one cannot specify a condition in the microinstruction sequencing field different from the SC in the CR := SC microoperation within the same microinstruction. Thus

WA := WB; WAP +1, CR := BUS; if CA=0 then RA +1.

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

WA := WB ; WAP +1, CR := BUS.
; if CA = 0 then RA +1.

Statements of the following type are obviously allowed:

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

2.21.1 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 ≡ 0 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 ≡ 0 then RA +1.
```

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

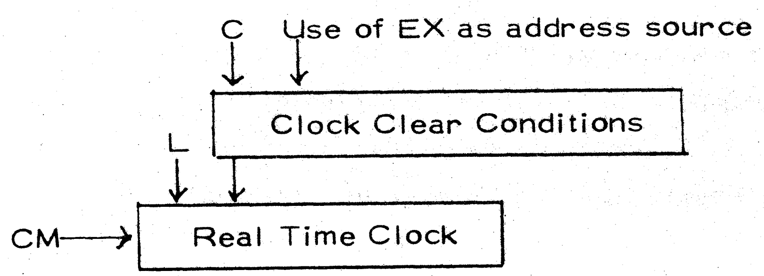
Long cycle:
a) execute bus transport
b) execute microoperations
c) execute microinstruction sequencing based on the current conditions

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.

2.22 The Real Time Clock

The Real Time Clock, RTC of the MATHILDA system is shown in Figure 2.38.



Real Time Clock
Figure 2.38

The clock can count up to 60 seconds. Whenever 60 seconds is reached two things occur, provided the INTON microoperation has been executed:

1) a Real Time Clock overflow Toggle, RTCT, is turned on and the clock is reset to 0,
2) the next microinstruction to be executed is obtained from control store location 0.

The clock is cleared whenever the microoperation RTCC is executed or whenever the EX input is selected as the address source for the address of the next microinstruction capability (see Section 2.20.6). One does not need to have the RTC count up from 0 before it overflows. A base value can be loaded by execution of the instruction RTC := CM.

In the microassembler the data will be specified in seconds. Thus, 4 seconds will elapse between the execution of the microoperation

RTC := 56

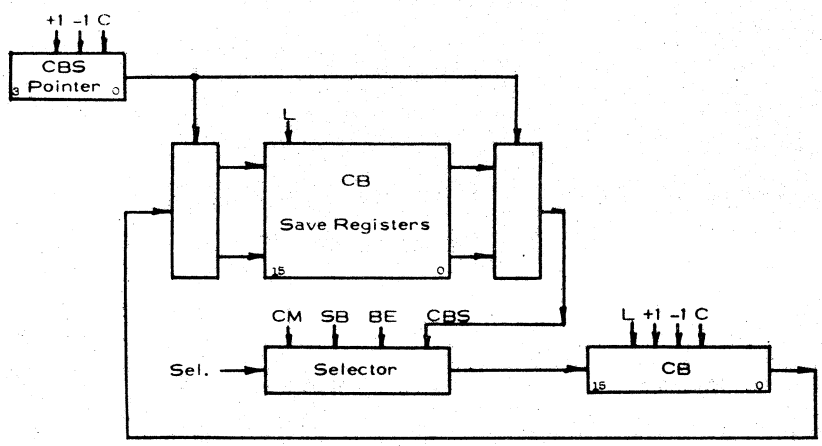
and the turning on of the RTC overflow toggle. The RTC overflow toggle can be turned off by executing the microoperation RTCT OFF.

2.23 Auxiliary Facilities

The auxiliary facilities associated with the MATHILDA system as shown in Figure 2.1, i.e., the system counters, status registers, and snooper registers, 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 2.39.



Counter B, CB

Figure 2.39

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| SB| EX| CAS

and CB := CM| SB| BE| CBS .

Note, the output of the BE is 6 bits, whereas CB is 16 bits wide. Whenever BE is selected as input to CB the high order 10 bits of CB are set to 0. The microoperations associated with CB, CBS, and CBSP are given in Table 2. 33. These are, of course, apart from the above difference,identical to those associated with CA and merely shown here for convenience.

CB := CM |SB |BE |CBS
CB + 1
CB - 1
CBC
CBS := CB
CBSP + 1
CBSP - 1
CBSPC

Table 2. 33

Microoperations for control of CB, CBS, and CBSP

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 are not connected in any way whatsoever and may be used independent of one another. One may count up in CA while counting down in CB, for example,

;CA + 1, CB - 1. .

2.32.2 The Snooper Store and Snooper Registers

The Snooper unit provides a facility for the gathering of data concerning the operation of the system. The facility consists of (a) a Snooper Store and (b) 16 Snooper Registers. The Snooper Store consists of up to 4,096 words of 4-bit wide, 80 nanosecond monolithic storage. It has the same number of words as the Control Store and is addressed in a cyclic fashion consistent with its size. The Snooper Registers are 32-bit wide registers which can be cleared and counted up. The Snooper unit works in the following way: when the address of the next microinstruction to be executed is sent to the Control Store Address Buffer, it is also sent to the Snooper Store Address Buffer; at the same time the microinstruction is fetched so that it can be executed, the contents of its associated Snooper Store location is fetched; the contents of the associated Snooper Store location identifies which of the 16 Snooper Registers is to be incremented during the execution of that particular microinstruction. Thus, during the execution of every microinstruction, a specified Snooper Register is incremented.

The Snooper Store can be written and the Snooper Registers read through the normal input/output facilities of the system. Snooper Store is writeable so that different data gathering routines can be associated with the same segment of microcode without changing the microcode. Snooper Store is loaded via OB, Device 1. If we load OB with the following information

OB	$0 \longleftrightarrow 0$ (63 … 16)	Contents (15 … 12)	Address (11 … 0)

,

then the execution of OBA when OBD is set to 1 will store OB(12:15) into the Snooper Store location specified by OB(0:11).

The contents of any particular Snooper Register, SRi, i=0,...,15, can be read through IB. Devices 1 through 8 of IB are associated with the Snooper Registers as shown in Table 2.34.

Device	IB(32:63)	IB(0:31)
1	SR 0	SR 1
2	SR 2	SR 3
3	SR 4	SR 5
4	SR 6	SR 7
5	SR 8	SR 9
6	SR 10	SR 11
7	SR 12	SR 13
8	SR 14	SR 15

Table 2. 34
IB Devices and the Snooper Registers

Thus, for example, if we wish to place the contents of SRB in bits 0 through 31 of LR0, we could write

; IBD := 5, LRIPC, PAP+1.
LR := IB, BS $\rightarrow$ 32 ; PG $\rightarrow$ 32, PAP-1. ■

A few points should be stated about this example. The IBA microoperation was not used, nor were either of the conditions IBDA or IBDC tested before input was made. This is explained as follows. The Snooper Registers are "dedicated" input devices, always available to be read. The IBA microoperation when used with Devices 1-8 is used to clear both of the Snooper Registers associated with the particular Device number.

There is also a tally of the total number of microinstructions which have been executed in the system. Device 9 on IB is a 64-bit wide Micro Instruction count register, MI, which is incremented everytime a microinstruction is executed. It can be cleared by executing IBA when IBD is set to 9. Thus the MI appears functionally identical to a Snooper and is included in this section.

2.23.3 The Status Registers

The Status facility establishes a data path between various control registers, address registers, and counters of the system and the BUS. Just as with the Snooper facility, this is done through the normal input facility of the system and, again, IB is used. Let us consider IB to be made of eight 8-bit bytes labelled IBj where IBj = IB(0+j8:7+j8), j = 0,. . .,7. For example, IB Byte 2, IB2 = IB(16:23). Table 2.35 shows which system elements are associated with Devices 10 and 11 on IB.

<table>
<tr><td>Device</td><td>IB7</td><td>IB6</td><td>IB5</td><td>IB4</td><td>IB3</td><td>IB2</td><td>IB1</td><td>IB0</td></tr>
<tr><td>10</td><td>CUF</td><td>BEF</td><td>WBP</td><td>WAP</td><td colspan="2">CB</td><td colspan="2">CA</td></tr>
<tr><td>11</td><td>CUALF</td><td>BE</td><td colspan="2">EX</td><td colspan="2">IRA</td><td colspan="2">SA</td></tr>
<tr><td>12</td><td colspan="8">Spare</td></tr>
</table>

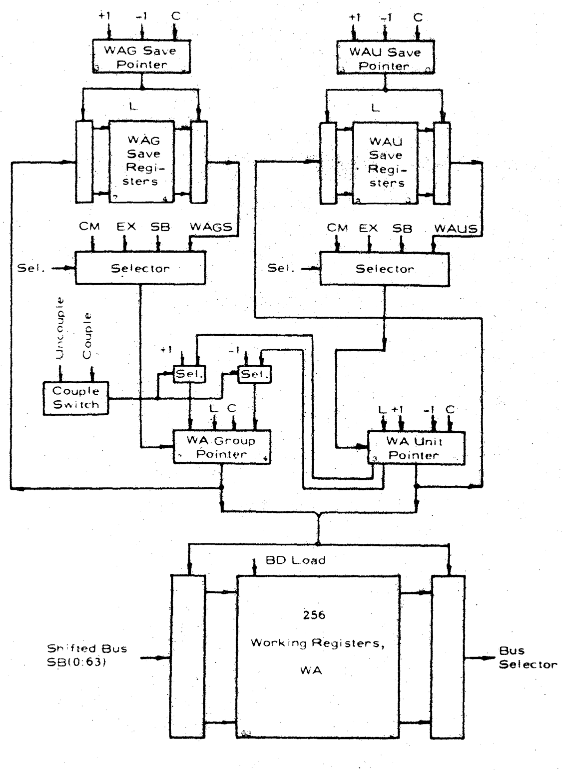
Table 2.35
Status Information

Devices 10, 11, and 12 on IB are the "Status Registers" of the system. Just as with the Snooper Registers, they are "dedicated" input devices. The IBA microoperation and the IBDA and IBDC conditions have no meaning when used with these devices. Suppose, for example we wish to store the output of the BE in the AS - recall the output of the BE had previously only been input to various control ports in the system. The following instructions connect it to the BUS and store it in the AS

;IBD := 11, PAP+1.
AS := IB, BS→48 ;PG→56, PAP-1. ■

2.24 An Alternate 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 2.5 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 2.40 shows WA with its two 4-bit pointers called the Group and Unit pointer; WB, not shown, is identical.



Working Registers A, WA (Detailed)

Figure 2.40

When the microoperation COUPLE A is executed, the Group and Unit pointers are connected together to give the 8-bit wide pointer, WAP. After the microoperation UNCOUPLE A 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.)

WAU := CM | EX | SB | WAUS
WAU + 1
WAU - 1
WAUC
WAG := CM | EX | SB | WAGS
WAG + 1
WAG - 1
WAGC

Table 2.36
Microoperations for control of the WAU and WAG pointers

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

WAP + 1 ::= WAU + 1
WAP - 1 ::= WAU - 1
WAPC ::= WAUC and WAGC
WAP := CM | EX | SB | WAPS ::= WAU := CM | EX | SB | WAUS
and WAG := CM | EX | SB | WAGS .

Let us now turn our attention to the pointer save capability shown in Figure 2.40. When WA is considered as 16 groups of 16 registers, the WAU and WAG pointers may be saved independent of one another. The microoperations associated with this facility are given in Table 2.37.

WAUS := WAU
WAUSP + 1
WAUSP - 1
WAUSPC
WAGS := WAG
WAGSP + 1
WAGSP - 1
WAGSPC

Table 2. 37

Microoperations for control of WAUS and WAGS

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 which we are pointing to 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. 37. 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.

There are a few additional conditions which can now be added to Table 2. 31, the partial listing of system conditions. These are given below in Table 2. 38.

Unit	Symbolic notation	Condition
WA	WAUOV	WAU $\equiv$ 1111 (WAU overflow)
	WAGOV	WAG $\equiv$ 1111 (WAG overflow)
	WAUSPOV	WAUSP $\equiv$ 1111 (WAUSP overflow)
	WAGSPOV	WAGSP $\equiv$ 1111 (WAGSP overflow)
	WACS	WACS = 1 $\Rightarrow$ WAU and WAG are coupled
WB	WBUOV	WBU $\equiv$ 1111 (WBU overflow)
	WBGOV	WBG $\equiv$ 1111 (WBG overflow)
	WBUSPOV	WBUSP $\equiv$ 1111 (WBUSP overflow)
	WBGSPOV	WBGSP $\equiv$ 1111 (WBGSP overflow)
	WBCS	WBCS = 1 $\Rightarrow$ WBU and WBG are coupled

Table 2.38
Additional WA and WB Conditions

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

$$PA \vee PB \vee PG$$

where

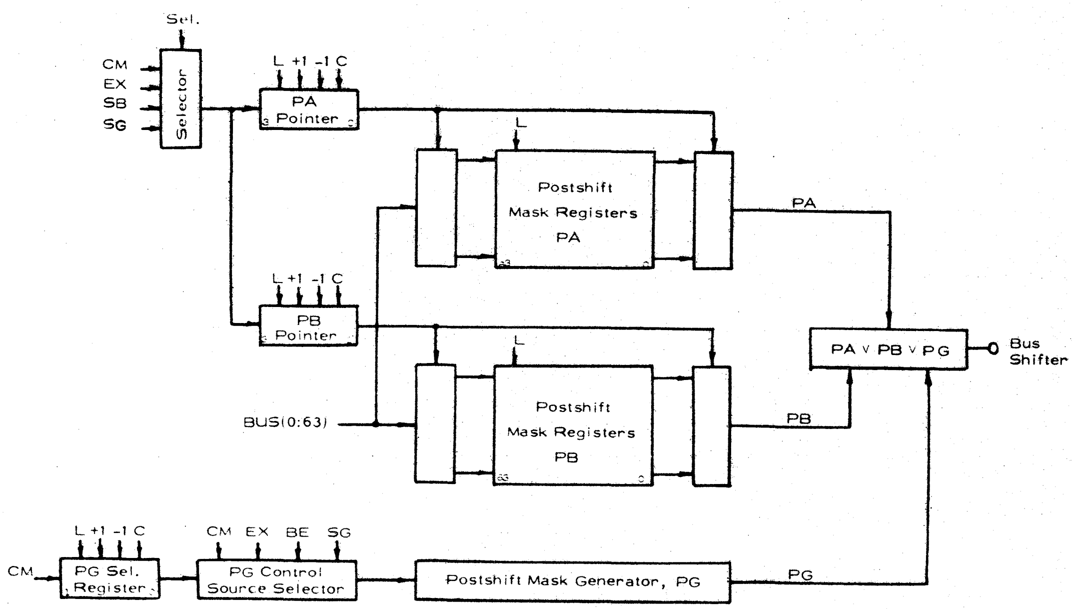
PA = an element of a 64-bit wide, 16 element RG called the Postshift Mask A registers

PB = an element of a 64-bit wide, 16 element RG called the Postshift Mask B registers

PG = the Postshift Mask Generator

V = logical "inclusive or".

In Section 2.7 we had introduced the mask to be PAVPG; here we had merely assumed all elements of PB to contain all 0's. The actual situation is shown more clearly in Figure 2.41.



Postshift Masks, PA, PB, and PG

Figure 2.41

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 then

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

Table 2.39

Microoperations for control of PB

The name of the SG associated with the PA pointer and the PB pointer is the Postshift AB Pointer, PABP. The microoperations associated with this SG are given in Table 2.40.

PABP := SB
PABPP := CM|EX|S1|S2
PABPP + 1
PABPP - 1
PABPPC
PABPS1 := CM|EX|S1|S2
PABPS2 := PABPP

Table 2.40

Microoperations for control of PABP

We will assume that all elements of PB contain all 0's so that the effective mask is PA^{V}PG and all of our previous standardizations for the use of this facility are still valid.

3.0 Microinstruction Specification and Execution

We will in this section discuss the microinstruction format, the manner in which the instruction is executed, an 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) bus transport
(b) microoperations and data
(c) microinstruction sequencing
(d) control of AS, VS, and DS

These fields are shown below with their sub-fields named and their actual bit location in the microinstruction.

(a) bus transport (7 bits)

BS	BD	SOURCE
22	21 19	18 16
1	3	3

Bus Shifter Enable Bit

(b) microoperations and data (35 bits)

mops	mops/data	mops/data	mops/data
63 57	56 47	46 39	38 29
7	10	8	10

mops = microoperations

(c) microinstruction sequencing (16 bits)

BISB	CISB	Condition Selection	A_f	A_t
15 14	13	12 6	5 3	2 0
2	1	7	3	3

Carry-in selection bits

B-input selection bits

(d) AS, VS, and DS control (6 bits)

AS	VS	DS
28 27	26 25	24 23
2	2	2

Shift/Load Control for the Shifters

Let us discuss each of these in more detail.

(A) The Bus Transport Field

Table 3.1 shows the correspondence between the symbolic notation for SOURCE's and BD's and their binary representations.

SOURCE		BD	
Symbolic Notation	Binary Notation	Symbolic Notation	Binary Notation
LR	000	no destination	000
AL	001	MA	001
VS	010	MB	010
DS	011	LR	011
WA	100	WA	100
WB	101	WB	101
IA	110	OA	110
IB	111	OB	111

Table 3.1

Symbolic and Binary Notation for SOURCE's and BD's

If the BS Enable bit = 0, no BS occurs ; if the BS Enable bit = 1 a BS Shift occurs. The control source for BS control is given in the microoperations and data field as is seen in (B) below. Thus the specification

BS	BD	SOURCE
0	101	011

is the binary representation of our bus transport specification

WB := DS . We will show this symbolically as

BS	BD	SOURCE
	WB	DS

as we have no need of binary representations in this report.

(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, $\frac{M}{D_2}$, F_2, $\frac{M}{D_3}$, F_3, S_3, $\frac{M}{D_4}$, F_4 as shown in Figure 3.1 .

7	2	1	7	1	7	2	1	7
F1	S1	$\frac{M}{D_2}$	F2	$\frac{M}{D_3}$	F3	S3	$\frac{M}{D_4}$	F4
63 57	56 55	54	53 47	46	45 39	38 37	36	35 29
mop	Sel.	M	mop	M	mop	Sel.	M	mop
		D	data	D	data_{BS}		D	data

Figure 3.1

Microoperation and Data Field

The following comments should assist in understanding this diagram.

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

if $\frac{M}{D_2} = 1$ then F_2 specifies a microoperation (1 of 128 mops).

if $\frac{M}{D_3} = 1$ then F_3 specifies a microoperation (1 of 128 mops).

if $\frac{M}{D_4} = 1$ then F_4 specifies a microoperation (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 microoperation 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 $\frac{M}{D_2}$ is set to D and F_2 is used as data (similarly $\frac{M}{D_4}$ 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

WA := AL, BS → 3

could be symbolically represented

	$\frac{M}{D_3}$	F_3	S_3		BS	BD	SOURCE	
	D	3	CM		BS	WA	AL	

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) <u>The Microinstruction Sequencing Field</u>

Table 3.2 shows the correspondence between the symbolic notation for A_t and A_f and their binary representations.

A_t and A_f	
Symbolic Notation	Binary Notation
EX	000
AL	001
RB	010
RA	011
SA	100
A-1	101
A+1	110
A	111

Table 3.2

Symbolic and Binary Notations for A_t and A_f

A similar table can be given for the symbolic and binary notations for the conditions but is not given here because of its length. Tables 2.24 and 2.25 present this information for the CISB (Carry-in selection bit) and BISB (B-input selection bits) respectively. We will give all of our examples symbolically.

Example 1) If BUS ≡ 0 then HERE. could be represented

BISB	CISB	Condition Selection	A_f	A_t
0		BUS	A+1	A

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

BISB	CISB	Condition Selection	A_f	A_t
$^{t}\text{sign}^{t}$		ALOV	A+1	RA+B

However, this is incomplete and immediately raises the question where do T and t come from? That is easily answered. T is always the leasts 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 and t (i.e., 0, Tt, $t_{sign}t$, or T0, see Section 2.20.2). Thus, the complete specification would be

⟵〰〰⟶	$\frac{M}{D_4}$	F_4	⟵〰〰⟶	BISB	CISB	Condition Selection	A_f	A_t
	D	12		$t_{sign}t$		ALOV	A+1	RA+B

D) AS, VS, And DS Control Field

The dedicated bits for shifter control are interpreted as shown in Table 3.3.

Binary Notation	Shift/Load Control
00	Do Nothing
01	Shift Right
10	Shift Left
11	Load

Table 3.3

Shift/Load Control Bits

Thus, the specification

AS →, VS ←, DS ←

could be represented symbolically as

AS	VS	DS
→	←	←

.

The binary representation,

AS	VS	DS
01	10	10

does not interest us here. The specification

AS, LR := AL ; DS ←.

would be given by

	AS	VS	DS	BS	BD	SOURCE	BISB	CISB	Condition Selection	A_f	A_t
	L		←		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 2.21.1, 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) execute bus transport
b) execute microoperation
c) execute microinstruction based on the current conditions

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

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

A) Bus Transport

The following actions occur during this step:

0) if short cycle, delay the conditions of the previous microinstructions (this has been combined with Bus transport for convenience)

1) the SOURCE is selected

2) the SOURCE is masked by the BUS masks and gated onto the BUS

3) the BUS is shifted as required by the BUS Shifter

4) the output of the BS is masked by the Postshift masks to yield the Shifted Bus, SB.

5) at this point, both the BUS and the SB are stable and can be loaded into various destinations: call this time 1.

B) Microoperation Execution

The following actions occur during this step:

0) the microoperations are decoded and divided into two types, those which can be executed at time 1 and those which can be executed at time 2; this decoding is completed by time 1.

1) all SB, and BUS loads are executed together with AS, VS, and DS operations and time 1 microoperations.

2) when time 1 microoperations are completed, time 2 microoperations are executed.

C) Microinstruction Sequencing

0) the condition specified by the condition selection bits is selected. In short cycle this can happen immediately upon the completion of B, above, as one is testing delayed conditions. In long cycle this cannot happen immediately upon the completion of B, above, but must wait until all conditions are stable and can be tested. Thus, one sees that in long cycle the microinstruction sequencing is delayed and hence its name,

1) select the carry-in and B-input into the CUAL and the RA and RB adders,

2) select the next address using A_t if c=1 or A_f if c=0 unless a force 0 address condition has arisen;

3) fetch microinstruction go to A, above.

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.

Example

The microinstruction

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

means: take the element of WA pointed to by 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 and the microoperation both occur at time 1 and the microoperation WAP +1 occurs at time 2. 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. The only exception to this rule, as noted in Section 2.20.3, is RA and RB because they are driven as hardware stacks and not RG's; i.e., their address space is changed first and then loaded (the inverse of the above) when RA ↓ or RB ↓ is executed.

Those microoperations which are executed at time 1 are said to have begun at Clock Pulse 1 , $C_p = 1$, while those which are executed at time 2 are said to have begun at Clock Pulse 2, $C_p = 2$, This notation is used in Section 3.3 which follows. This notation, along with the description of microinstruction execution given in 3.2 above, completely define what a given microinstruction means. As an example

WB := AL, BS → BE ; SET ALF +, WBU := 9

means: store the output of AL in WB register pointed to by WBP after shifting it the amount specified by the BE; then change the ALF to AS + LR and change the WBU to 9; then go to the next microinstruction.

3.3 Comprehensive Tables of Microoperations for Individual Functional Units

The following tables (presented 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:

$$\begin{aligned} XX &= EX \mid SB \mid SG, \\ ZZ &= EX \mid S1 \mid S2 \\ WU &= EX \mid SB \mid WS \\ WG &= EX \mid SB \mid WS. \end{aligned}$$

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 it 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 microinstruction should not be specified in F_3.

d) Selection bits which determine the BS control source always come from S_3.

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

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

MICROOPERATIONS FOR Arithmetic Logical Unit, AL

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	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	ALS1 :=	M	ALS1 :=	ZZ CM	D	d d d d	Load the AL SG Save1 register from CM|EX|S1|S2
1	ALS2 := ALP									Load the AL SG Save2 register from the AL SG Pointer
1			M	ALSG := SB						Load the AL SG with SB(0:5)
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 LR + AS
2			M	SET ALF AS						Set AL Function to AS

MICROOPERATIONS FOR Accumulator Shifter, AS

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	AS(0)S :=	XX CM	D	d d d				M	AS(0)S :=	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 Bit Encoder, BE

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	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 interchenged 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 BE pointer from CM|EX|S1|S2
2					M	BEP +1				Increment BE pointer
2					M	BEP -1				Decrement BE pointer
2					M	BEPC				Clear BE pointer
2			M	BES1 :=	M	BES1 :=	ZZ CM	D	d d d d	Load BE Save1 register from CM|EX|S1|S2
1	BES2:=BEP									Load BE Save2 register from BE Pointer
1			M	BESG:=SB						Load BE SG from SB(0: 3)

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
0					D	d d d d d d	YY CM			THIS SELECTION IS REQUIRED WHENEVER THE BUS SHIFTER IS ENABLED *)
2	BSP :=	ZZ CM	D	d d d d						Load BS register group 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	BSS1	ZZ CM	D	d d d d						Load BS Save1 register from CM|EX|S1|S2
1					M	BSS2:=BSP				Load BS Save2 register from BS Pointer
1								M	BSSG:=SB	Load BS SG from SB(0:5)
										*) YY = EX|BE|BS|SG

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 CAS (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			M	CASP +1						Increment CAS Pointer
2			M	CASP -1						Decrement CAS Pointer
2			M	CASPC						Clear CAS Pointer
1			M	CAS := CA						Load CA Save RG from CA

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 CBS (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	CBSP +1				Increment CBS Pointer
2					M	CBSP -1				Decrement CBS Pointer
2					M	CBSPC				Clear CBS Pointer
1					M	CBS := CB				Load CB Save RG from CB
	*) VV=SB|BE|CBS									+) when BE is selected as the source, the high order 10 bits of CB are set to 0

MICROOPERATIONS FOR Condition Save Register, CR

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	CRS1 :=	M	CRS1 :=	ZZ CM	D	d d d d	Load CR RG Save1 buffer from CM|EX|S1|S2
1	CRS2 := 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
	S* = special depending on short or long cycle									

MICROOPERATIONS FOR Common Shifters (AS, VS, DS) Standard Group and parallel options, CS

7	2	1	7	1	7	2	1	7

Cp	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
2					M	CSP :=	ZZ CM	D	d d d d	Load the CS Pointer from CM|EX|S1|S2
2					M	CSP +1				Increment the CS Pointer
2					M	CSP -1				Decrement the CS Pointer
2					M	CSPC				Clear the CS Pointer
2			M	CSS1 :=	M	CSS1 :=	ZZ CM	D	d d d d	Load the CS Save1 register from CM|EX|S1|S2
1	CSS2 := CSP				M	CSS2:=CSP				Load the CS Save2 register from the CS Pointer
1			M	CSSG := SB						Load the CS SG from SB(0:5)
2								M	CSLL	Set AS, VS, and DS to logical left shift
2								M	CSLR	Set AS, VS, and DS to logical right shift
2					M	CS(V)SC				Clear AS, VS, and DS Variable Bit Selection register
2	CS(0)S :=	XX CM	D	d d d						Load AS(0), VS(0) and DS(0:1) Source register from CM|EX|SB|SG
2	CS(63)S :=	XX CM	D	d d d						Load AS(63), VS(63) and DS(62:63) Source register from CM|EX|SB|SG
2	CS(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 Control Unit, CU

7	2	1	7	1	7	2	1	7

Cp	F1	S1	M/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
1			M	SA := SB						Load Save Address register from SB(0:11)
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			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 + 4						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	INTON		M	INTON				M	INTON	Enable interrupt conditions to force 0 address
1	INTOFF		M	INTOFF				M	INTOFF	Disable interrupt conditions from forcing 0 address
1	SET CUALF B									Set CUAL Function register to B
1					M	RTCT OFF				Turn Real Time Clock overflow toggle off
	*) requires two microinstruction cycles to complete this action									

MICROOPERATIONS FOR Double Shifter, DS

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	DS(0:1)S :=	XX CM	D	d d d						Load DS(0:1) Source register from CM|EX|SB|SG
2	DS(62:63)S :=	XX CM	D	d d d						Load DS(62:63) 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/D	F2	M/D	F3	S3	M/D	F4	MICROOPERATION
1	IAD :=	EE CM	D	d d d d						Load IA Device register from CM|EX0|SB|EX1
2	IAA		M	IAA				M	IAA	Activate Port, i.e. read
1			M	IADC						Clear IA Device register
1			M	IAD +1						Increment IA Device register
1	IBD :=	EE CM	D	d d d d						Load IB Device register from CM|EX0|SB|EX1
2	IBA :=		M	IBA				M	IBA	Activate Port, i.e., read
1			M	IBDC						Clear IB Device register
1			M	IBD +1						Increment IB Device register

MICROOPERATIONS FOR Loading Mask Registers A, LA

F1	S1	M/D	F2	M/D	F3	S3	M/D	F4
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	LAP :=	ZZ CM	D	d d d d						Load LA Pointer from CM|EX|S1|S2
2	LAP +1		M	LAP +1				M	LAP +1	Increment LA Pointer
2	LAP -1		M	LAP -1				M	LAP -1	Decrement LA Pointer
2	LAPC							M	LAPC	Clear LA Pointer
2	LAS1 :=	ZZ CM	D	d d d d				M	LAS1 :=	Load LA Save1 register fromCM|EX|S1|S2
1					M	LAS2:=LAP				Load LA Save2 register from LA Pointer
1								M	LA := SB	Load LA from SB(0:63)

MICROOPERATIONS FOR Loading Mask Registers B, LB

F1	S1	M/D	F2	M/D	F3	S3	M/D	F4
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 LB Pointer from CM|EX|S1|S2
2			M	LBP +1	M	LBP +1		M	LBP +1	Increment LB Pointer
2			M	LBP -1	M	LBP -1		M	LBP -1	Decrement LB Pointer
2					M	LBPC		M	LBPC	Clear LB Pointer
2			M	LBS1 :=	M	LBS1 :=	ZZ CM	D	d d d d	Load LB Save1 register from CM|EX|S1|S2
1	LBS2:=LBP									Load LB Save2 register from LB Pointer
1			M	LB := SB						Load LB from SB(0:63)
			M	LPC						Clear both LA Pointer and LB Pointer

MICROOPERATIONS FOR Local AL Registers, LR

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

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	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	BMPP :=	ZZ CM	D	d d d d	Load BM Pointer SG Pointer from CM|EX|S1|S2
2					M	BMPP +1				Increment BMPSG Pointer
2					M	BMPP -1				Decrement BMP SG Pointer
2					M	BMPPC				Clear BMP SG Pointer
2			M	BMPS1 :=	M	BMPS1 :=	ZZ CM	D	d d d d	Load BMP SG Save1 register from CM|EX|S1|S2
1	BMPS2 := BMPP									Load BMP SG Save2 register from the BMPP
1			M	BMP := SB						Load BMP SG with SB(0:3)

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|EX0|SB|EX1
2	OAA	d			M	OAA		M	OAA	Activate Port, i.e., write OA(0:63)d
1					M	OADC				Clear OA Device register
1					M	OBD :=	EE CM	D	d d d d	Load OB Device register from CM|EX0|SB|EX1
2	OBA	d	M	OBA				M	OBA	Activate Port, i.e., write OB(0:63)d
1			M	OBDC						Clear OB Device register
1					M	OCD :=	EE CM	D	d d d d	Load OC Device register from CM|EX0|SB|EX1
2	OCA	d			M	OCA		M	OCA	Activate Port, i.e., write OC(0:63)d
1					M	OCDC				Clear IC Device register
1			M	OC:=BUS						Load OC from BUS(0:63)
1					M	ODD :=	EE CM	D	d d d d	Load OD Device register from CM|EX0|SB|EX1
2	ODA	d	M	ODA				M	ODA	Activate Port, i.e., write OD(0:63)d
1			M	ODDC						Clear OD Device register
1			M	OD:=BUS						Load OD from BUS(0:63)

MICROOPERATIONS FOR Postshift Masks, PA, PB, and 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	PGS1 :=	M	PGS1 :=	ZZ CM	D	d d d d	Load PG Save1 register from CM|EX|S1|S2
1	PGS2:=PGP									Load PG Save2 register from PGP
1			M	PGSG := SB						Load PG SG from SB(0:6)
2	PAP :=	XX CM	D	d d d d						Load PA Pointer from CM|EX|SB|RG
2	PAP +1							M	PAP +1	Increment PA Pointer
2	PAP -1							M	PAP -1	Decrement PA Pointer
2	PAPC							M	PAPC	Clear PA Pointer
1					M	PA:=BUS				Load PA RG from BUS(0:63)
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(0:63)
2					M	PAB +1				Increment PA and PB Pointer
2					M	PAB -1				Decrement PA and PB Pointer
2					M	PABC				Clear PA and PB Pointer
2					M	PABPP :=	ZZ CM	D	d d d d	Load PAB Pointers RG Pointer from CM|EX|S1|S2
2					M	PABPP +1				Increment PABP Pointer
2					M	PABPP -1				Decrement PABP Pointer
2					M	PABPPC				Clear PABP Pointer
2				PABPS1 :=	M	PABPS1 :=	ZZ CM	D	d d d d	Load PABP Save1 register from CM|EX|S1|S2
1	PABPS2 := PABPP									Load PABP Save2 register from PABP Pointer
1			M	PABP:=SB						Load PABP from SB(0:3)

MICROOPERATIONS FOR Variable Width Shifter, VS

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	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 :=	WU CM	D	d d d d						Load WA Unit pointer from CM|EX|SB|US
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 :=	WG CM	D	d d d d	Load WA Group pointer from CM|EX|SB|GS
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 :=	WU CM	D	d d d d			WG CM	D	d d d d	Load WA Unit pointer from CM|EX|SB|US AND load WA Group pointer from CM|EX|SB|GS
2	WAPC									Clear WA Unit pointer and WA Group pointer
1								M	COUPLE A	Couple WA Unit and Group pointers to form an 8 bit counter
1								M	UNCOUPLE A	Uncouple WA Unit and Group pointers to form two independent 4 bit counters

MICROOPERATIONS FOR WA Unit and Group Save Registers, 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								M	WAUS:=WAU	Load WA Unit Save RG with WAU
2								M	WAUSP +1	Increment WA Unit Save RG pointer
2								M	WAUSP -1	Decrement WA Unit Save RG pointer
2								M	WAUSPC	Clear WA Unit Save RG pointer
1	WAGS:=WAG									Load WA Group Save RG with WAG
2	WAGSP +1									Increment WA Group Save RG pointer
2	WAGSP -1									Decrement WA Group Save RG pointer
2	WAGSPC									Clear WA Group Save RG pointer
1								M	WAPS:=WAP	Load WA Unit and WA Group Save registers with WAU and WAG respectively
2								M	WAPSP +1	Increment WA Unit and WA Group Save pointers
2								M	WAPSP -1	Decrement WA Unit and WA Group Save pointers
2								M	WAPSPC	Clear WA Unit and WA Group Save pointers

MICROOPERATIONS FOR Working Registers, B, WB

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	WBU :=	WU / CM	D	d d d d	Load WB Unit pointer from CM|EX|SB|US
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 :=	WG / CM	D	d d d d						Load WB Group pointer from CM|EX|SB|GS
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 :=	WG / CM	D	d d d d			WU / CM	D	d d d d	Load WB Unit pointer from CM|EX|SB|US AND load WB Group pointer from CM|EX|SB|GS
2			M	WBPC						Clear WB Unit pointer and WB Group pointer
1					M	COUPLE B				Couple WB Unit pointer and Group pointers to form an 8 bit counter
1					M	UNCOUPLE B				Uncouple WB Unit pointer and Group pointer to form two independent 4 bit counters

MICROOPERATIONS FOR WB Unit and Group Save Registers, WBUS and WBGS

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	WBUS:=WBU						Load WB Unit Save RG from WBU
1			M	WBUS +1						Increment WB Unit Save RG pointer
2			M	WBUS -1						Decrement WB Unit Save RG pointer
2			M	WBUSPC						Clear WB Unit Save RG pointer
1					M	WBGS:=WBG				Load WB Group Save RG from WBG
2					M	WBGSP +1				Increment WB Group Save RG pointer
2					M	WBGSP -1				Decrement WB Group Save RG pointer
2					M	WBGSPC				Clear WB Group Save RG pointer
1					M	WBPS:=WBP				Load WB Unit and WB Group Save register with WBU and WBG respectively
2					M	WBPSP +1				Increment WB Unit and WB Group Save pointers
2					M	WBPSP -1				Decrement WB Unit and WB Group Save pointers
2					M	WBPSPC				Clear WB Unit and WB Group Save pointers

MICROOPERATIONS FOR Common WA and WB Operations, WC

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	WCU +1	Increment WA and WB Unit pointers
2								M	WCU -1	Decrement WA and WB Unit pointers
1			M	WCUS						Load WA Unit Save RG and WB Unit Save RG
1					M	WCGS				Load WA Group Save RG and WB Group Save RG

Table of First Occurrance of Abbreviations and Symbols

(not including conditions or microoperations)

Abbreviation	Interpretation	Page
A_t , A_f	Address Specifications	11
AL	Arithmetical Logical Unit	28
ALF	AL Function and Carry-in Register	30
ALP	ALRG Pointer	29
ALSG	AL Standard Group	30
ALS1	ALSG Save1 Pointer	30
ALS2	ALSG Save2 Pointer	30
AS	Accumulator Shifter	32
BD	Bus Destination	9
BE	Bit Encoder	15
BEF	Bit Encoder Function Selection Register	53
BEP	BESG Pointer	54
BESG	BEF Standard Group	54
BES1	BESG Save1 Register	54
BES2	BESG Save2 Register	54
BISB	B-Input Selection Bits	69
BM	Bus Masks	19
BMP	Bus Mask Pointer Standard Group	22
BMPP	BMP Pointer	22
BMPS1	BMP Save1 Register	22
BMPS2	BMP Save2 Register	22
BS	Bus Shifter	8
BSP	BS Standard Group Pointer	16
BSSG	Bus Shifter Standard Group	15
BSS1	BS Save1 Register	16
BSS2	BS Save2 Register	17
BUS	the BUS	8
CA	Counter A	6
CAS	Counter A Save Registers	7
CASP	Counter A Save Register Pointer	7
CB	Counter B	55
CBS	Counter B Save Registers	86
CBSP	Counter B Save Register Pointer	86

Abbreviation	Interpretation	Page
CISB	Carry-in Selection Bit	68
CR	Condition Save Registers	81
CRP	CR Pointer	82
CRS1	CR Save1 Register	82
CRS2	CR Save2 Register	82
CS	Common Shifter	46
CSB	Condition Selection Bits	81
CSP	CSSG Pointer	46
CSSG	Common Shifter Standard Group	45
CSS1	CSSG Save1 Register	46
CSS2	CSSG Save2 Register	46
CU	Control Unit	64
CUAL	Control Unit Arithmetical Logical Unit	67
CUALF	CUAL Function Register	66
DESTINATION	Bus Destination, BD	9
DS	Double Shifter	40
EX	External Register	7
EX0	External Register Byte 0	5
EX1	External Register Byte 1	59
EX2	External Register Byte 2	73
EX3	External Register Byte 3	73
IA	Input Port A	58
IAD	IA Device Register	59
IB	Input Port B	58
IBD	IB Device Register	74
IRA	Interrupt Recovery Address	75
LA	Loading Masks A	46
LAP	LA Pointer	48
LAS1	LA Save1 Register	48
LAS2	LA Save2 Register	48
LB	Loading Masks B	46
LBP	LB Pointer	48
LBS1	LB Save1 Register	48
LBS2	LB Save2 Register	48
LR	Local Registers	31

Abbreviation	Interpretation	Page
LRIP	Local Registers Input Pointer	31
LROP	Local Registers Output Pointer	31
LRP	LRIP and LROP	32
LSB	A Bit Pointer (available through BE)	51
MA	Mask A Registers	20
MAP	MA Pointer	20
MB	Mask B Registers	20
MBP	MB Pointer	22
MSB	A Bit Pointer (available through BE)	51
OA	Output Port A	61
OAD	OA Device Register	62
OB	Output Port B	61
OBD	OB Device Register	74
OC	Output Port C	61
OCD	OC Device Register	62
OD	Output Port D	62
ODD	OD Device Register	74
PA	Postshift Mask A Registers	23
PABP	Postshift AB Pointer	26
PAP	PA Pointer	26
PB	Postshift Mask B Registers	93
PBP	PB Pointer	93
PG	Postshift Mask Generator	24
PGP	PGSG Pointer	26
PGSG	Postshift Mask Generator Standard Group	26
PGS	Postshift Mask Generation Selection Reg.	25
PGS1	PGSG Save1 Register	26
PGS2	PGSG Save2 Register	26
PM	Postshift Masks	23
RA	Return Jump Stack A	65
RAP	Return Jump Stack A Pointer	71
RB	Return Jump Stack B	65
RBP	Return Jump Stack B Pointer	72
RG	Register Group	4
RGP	Register Group Pointer	4

Abbreviation	Interpretation	Page
RTC	Real Time Clock	75
RTCT	Real Time Clock Overflow Toggle	84
SA	Save Address Register	65
SB	Shifted Bus	7
SC	Selected Condition	82
SG	Standard Group	17
"Shifters"	AS, VS, and DS	41
SOURCE	the input to the BUS	8
SR	Snooper Registers	87
V	The Variable Bit	33
VS	Variable Shifter	58
WA	Working Registers A	8
WAG	Working Registers A Group Pointer	90
WAGS	WAG Save Registers	92
WAP	WA Pointer	9
WAPS	WA Pointer Save registers	10
WAPSP	WAPS Pointer	11
WAU	Working Registers A Unit Pointer	90
WAUS	WAU Save Registers	92
WB	Working Registers B	8
WBP	WB Pointer	11
WBPS	WB Pointer Save registers	11
WBPSP	WBPS Pointer	11

Symbol	Interpretation	Page
$\wedge$	Logical "and"	28
$\vee$	Logical "inclusive or"	19
$-$	Logical "negation"	28
$\equiv$	Logical "equivalence"	28
$\not\equiv$	Logical "nonequivalence"	28
$\rightarrow$	Right Shift	15
$\rightarrow\cdot$	Postshift Mask Generation Direction	25
$\leftarrow$	Left Shift	16
$\cdot\leftarrow$	Postshift Mask Generation Direction	25
[ ]	Option of Inclusion	64
$\vert$	Possible Alternate Sources	7
$\longrightarrow$	Input	19
$-\!\bigcirc$	Mask	19
$-\!\textcircled{$\ell$}$	Loading Mask	47

List of Figures

List of Tables

References

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

[2] "KAROLINE, a network computer project", by Ole Brun Madsen, RECAU, University of Aarhus, Aarhus, Denmark, 1972.

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

[4] "A Small Group of Research Projects in Machine Design for Scientific Computation", by Bruce D. Shriver, Department of Computer Science Report No. 14, University of Aarhus, Aarhus, Denmark, April 1973.

[5] "The Significance of Microprogramming", by R.F. Rosin, to be presented at the International Computing Symposium 1973 in Davos, Switzerland.

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

/km

A description of the MATHILDA system

Micro
Archives
7-45

Shriver, Bruce D.
A description of the MATHILDA system / by Bruce D. Shriver.-- Aarhus, Denmark: Department of Computer Science, Matematisk Institut, Aarhus University, 1973.
(DAIMI PB-13)

I. Title.