Introduction

to the

COMET Microarchitecture

Yale N. Patt

San Francisco State University
San Francisco, CA 94132

and

Corporate Research Group
Digital Equipment Corporation
Maynard, Ma. 01754

25 March 1980

PREFACE

This report is an introduction to the microarchitecture of a particular host computer, COMET. It is not a general introduction to microprogramming; other books will have to do that. It is also not a hardware reference manual. No attempt has been made to delineate an encyclopedic taxonomy of COMET's features. Instead, topics are discussed in the order that I think they fit. My goal is that the reader should be able to make sense of COMET and its parts.

This report was written from the perspective of the microprogrammer; it describes the microarchitecture,i.e., the architecture visible to the microprogrammer. It should be useful to those who want to know how COMET implements VAX, and also to those who need to get started so they can write user microcode for COMET.

It is assumed that the reader has some understanding of computer architecture in general, and the VAX architecture in particular. Nevertheless, certain VAX features like memory management and interrupt handling are discussed first, before their COMET implementations.

The report was written because a lot of people outside of the COMET group wanted to know how COMET works. I have been able to complete it because Paul Gilbault and Charlie McDowell were willing to patiently answer a lot of questions, and because Bob Glorioso and Don Gaubatz either agreed or were willing to accept my judgment that it was something we ought to be doing.

I must also acknowledge, with thanks, the excellent critical reading of an earlier draft of this report by Fernando Colón Osorio, Charlie McDowell, and Martin Minow. Thanks are also due to Serena Shields for typing the manuscript. She has patiently endured my many changes to this report.

The report is organized in six chapters. Chapter 1 provides an overview of COMET, both from the standpoint of it being a host microprogrammable computer, and from the standpoint of its VAX emulation. The two major parts of a microprogrammable computer are covered next: Chapter 2 treats the microsequencer and Chapters 3 and 4 deal with the Data Path. Chapter 5 is specific to the VAX emulation. It describes the COMET mechanisms for implementing the VAX interrupt and exception handling and memory management functions. The report concludes with two examples of COMET microcode. One was taken directly from the VAX emulation. It is the execution flow for the INDEX instruction. The other is a new (unsupported, unasked for, and perhaps unwelcomed!) special purpose instruction for matching bit patterns. The intent was to show how to go about designing your own new instruction.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

1.1 Overview of COMET

COMET is a microprogrammable computer which was designed specifically to emulate the VAX-11 architecture. It consists of up to 16K words of control store, one 80 bit microinstruction per word, a microsequencer, a 32 bit Data Path, a number of registers which are needed to access COMET's main memory, and a number of status and control registers which are needed to emulate VAX. Figure 1.1 is an overall block diagram of the COMET microarchitecture. Figure 1.2 shows the fields of a COMET microinstruction. The number associated with each field is the section in this report where the use of that field is discussed in detail.

There are three major internal buses in COMET: the WBUS*, the MBUS and the RBUS. The main bus is the WBUS. The output of the ALU, unless inhibited, goes on the WBUS. Data to be written into memory and data to be written into the Scratch Pad registers are taken from the WBUS. Status and control information are passed to and from their particular registers via the WBUS. The MBUS and RBUS provide sources for the Super Rotator and the ALU. Data on the MBUS is primarily taken from the VAX main memory interface registers and from the M scratch pad registers. Data on the RBUS is from the R scratch pad registers and the Long Literal Register.

The COMET Data Path consists of two sets of Scratch Pad registers, a LONLIT register, the Super Rotator, the ALU, and two special registers (D and Q). All are 32 bits wide. The ALU can perform 2's complement arithmetic, BCD arithmetic, and logical operations. There are two input ports to the ALU (A and B); both are multiplexed. The MUX field controls the sources of the ALU, the ALU field specifies the arithmetic or logical function to be performed, and the DQ field controls the destination of the ouput. The source of the carry input to the ALU is specified by the ALUCI field. Input to the ALU can come from the RBUS, the MBUS, the Super Rotator, and the D and Q registers. The output of the ALU can be shifted or rotated (by itself, or combined with the Q register); the type of shift is determined by the DQ and ALUSHF fields. The output of the ALU can go on the WBUS and it can go to the D or Q register. In addition, the ALU can perform certain special functions (for example, a FAST MULTIPLY) where the input and output are specified as part of the function. This

---

*The Appendix contains a list of the acronyms used in this report, often with additional commentary.

is done by coding the MUX, ALU and DQ fields as a single unit. We call this the ALPCTL field.

There are 56 Scratch Pad registers . Eight have two ports (to the MBUS and the RBUS), eight can be accessed by the MBUS only, and 40 can be accessed by the RBUS only. The particular registers accessed during any microcycle are specified by the MSRC and RSRC fields. Writing to the Scratch Pad registers is controlled by the SPW field.

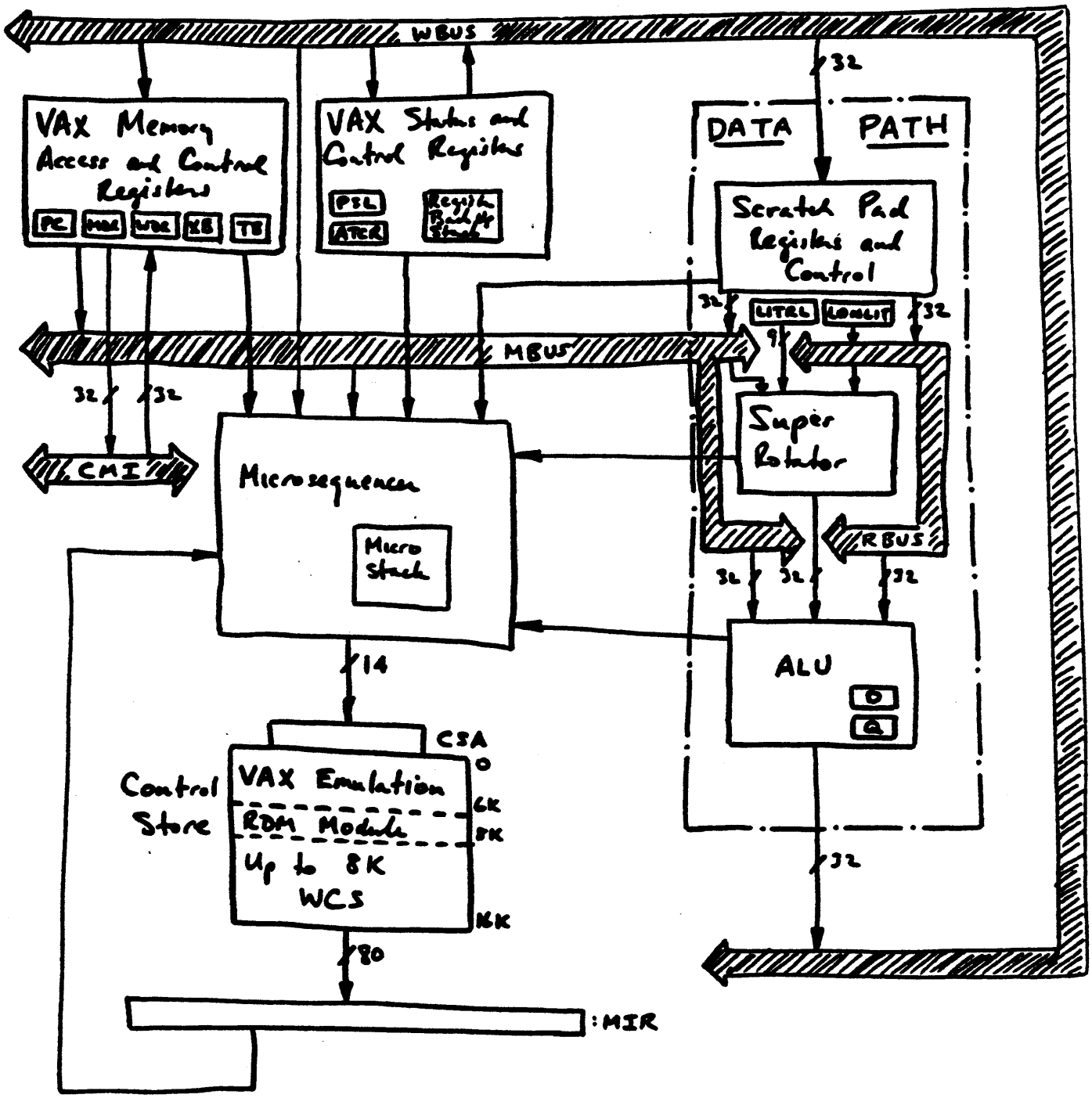


Figure 1.1 COMET Microarchitecture (Overview)

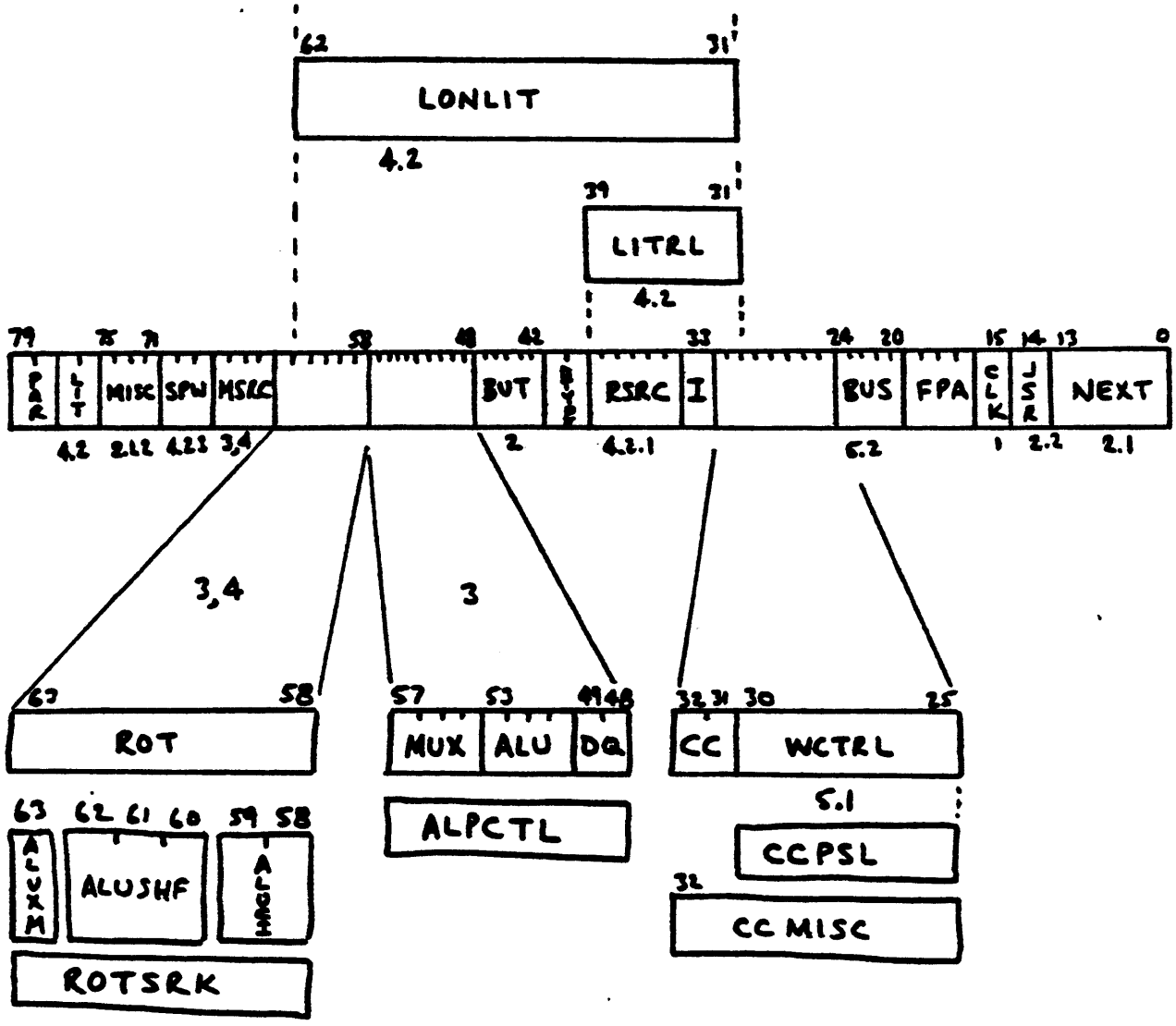


Figure 1.2 The COMET Microinstruction

The Super Rotator is a powerful combinational logic circuit. It can barrel shift a 64-bit data element, it can extract a desired field from a given piece of data, and it can construct a 32-bit data element according to a variety of specifications. This provides COMET with a very efficient bit manipulation capability. The Super Rotator is controlled by the ROT field.

Immediate data of 9 or 32 bits can be entered into the Data Path from the instruction stream under the control of the LIT field. The LIT/LONLIT* micro-order causes the LONLIT register to be loaded with Bits<62:31> of the microinstruction. This data is then available in the next microinstruction; access to it is controlled by RSRC. The LIT/LITRL micro-order causes Bits<39:31> of the microinstruction to be made available as input to the Super Rotator during the same microcycle.

The basic microcycle of the COMET architecture is 320 nsec. This is sufficient to read from the Scratch Pad registers, pass through the Super Rotator, perform an ALU operation, and write back to a Scratch Pad register. Certain activities can cause a microinstruction to need more than 320 nsec. For example, suppose the address of the next microinstruction is to be computed partly from the resuts of the ALU operation. We will see that the microrder BUT/WX.EQ.0 produces such a situation. When this happens, the CLK field can extend the microcycle to 480 nsec.

COMET has no microprogram counter. The address of the next microinstruction (CSA) is usually determined by the microsequencer in one of several ways. The microsequencer can generate a microbranch (up to 64-way conditional branch) based on the values of certain internal signals specified by the BUT field. The branch addresses are based on the contents of the NEXT field and the values of these specified signals. Or the CSA can be obtained by popping the microstack. Or the address can be obtained from the IRD1 or IRDX ROMs. The above schemes are all under the control of the BUT field. In addition, the COMET hardware can override these addressing schemes by forcing the CSA (by means of a microtrap) to a fixed address. This last technique is used for memory management and to initiate some of the VAX exceptions and interrupts.

Control store addressing supports up to 16K of control store. Actually, current hardware implementation contain only 9K of control store. The low order 6K is required to emulate VAX and the next 2K is dedicated to the Remote Diagnostic module (RDM). This leaves a possible 8K address space for WCS, of which 1K is actually implemented.

---

* The notation <field>/<code> is used extensively throughout this report. LIT/LONLIT represents the micro-order in which the LIT field contains the LONLIT code.

Finally, COMET contains a number of features which are specifically included to aid in the emulation of VAX. The PC, MDR, WDR, Translation Buffer, and Execution Buffer are a few of the registers which are used to control access through the CMI, a 32 bit wide synchronous bus, to the cache and VAX main memory. The BUS field controls this access. The PSL, Software IPR, Status Flags, Console and TU58 registers, ASTLVL register, and the internal next interval register are a few of status and control registers which are used to control interrupt and exception processing. The WCTRL field controls these functions.

1.2 The VAX Emulation.

The purpose of this section is to provide an overview of how COMET emulates VAX. The details of each mechanism are covered more thoroughly in later chapters of this report.

The VAX registers are, for the most part, implemented in the M and R Scratch Pads. In particular, R0 through R12, FP, and SP are implemented by R[10] through R[1E] in the R Scratch Pad. The stack pointers are R[20] through R[24], the memory management registers are R[28] through R[2D], the high order 16 bits of ICR and NICR is R[2E], the PCBB is R[25], and the SCBB and SISR are M[0E] and M[0F]. The rest of the VAX registers, including PC, PSL, and ASTLVL are implemented by COMET registers designed specifically for that purpose.

Emulation starts with the BUT/IRD1 micro-order. This is the signal to begin the emulation of the next VAX machine instruction. In the microcode which emulates each VAX instruction, this micro-order is present in the last microinstruction.

BUT/IRD1 causes two things to occur. It invokes a hardware routine DOSERVICE which checks for traps and interrupts. If any are pending, the processor will microtrap to the appropriate control store address to initiate the trap or interrupt. Also, it causes two bytes to be fetched from the instruction stream (i.e., from the XB) and loaded into the IR and OSR. (The opcode of the next VAX instruction is loaded into the IR; the first operand specifier is loaded into the OSR). The XB (execution buffer) is an eight byte register. The instruction stream is prefetched automatically four bytes at a time and stored in the XB. Each time the XB is accessed, the PC is automatically incremented.

The processor then begins the emulation of the VAX instruction. Usually it branches to common code to evaluate the address of the first operand. The branch address is obtained from a ROM (the IRD1 ROM) which is indexed by the opcode of the VAX instruction, by whether the current VAX instruction was

previously suspended (i.e., if the FPD bit is set), and by whether Floating Point Accelerator hardware is present. The common code terminates with a BUT/IRDX micro-order which causes the next control store address to be obtained from the IRDX ROM. The rest of the operand addresses are computed and the VAX instruction is emulated, terminating in a BUT/IRD1 micro-order, which starts the cycle again.

If the VAX instruction requires a memory access, a BUS read or write initiates the access. COMET maintains a TB (translation buffer) of PTE's, which are needed to map virtual addresses into physical addresses. If the PTE is not present in the TB or if the memory access is unaligned (VAX allows the user to disregard natural word and longword boundaries), COMET microtraps (forces the control store address) to a specific location to correct the problem. The executing microinstruction is not allowed to complete until the problem is corrected. If the problem is corrected, control returns to that microinstruction.

If the emulation of a VAX instruction requires a sufficiently long time that pending interrupts cannot be ignored, the VAX emulation tests for interrupts under microprogram control. If an interrupt is to be initiated, the processor is put into a consistent state by either undoing whatever processing has occurred, or if that is not possible, by setting FPD and following a prescribed procedure for "packing up" the relevant machine state so that the VAX instruction can be restarted at the point where it was suspended.

The initiation of exceptions and interrupts are emulated in microcode. The branch to the starting address is caused by either a microtrap in case the condition was detected by the hardware (for example, by DOSERVICE), or by a microbranch in case the condition was detected under microprogram control. In either case the microcode selects the appropriate stack to service the exception or interrupt, pushes the current PC and PSL as well as any necessary parameters on that stack, puts the processor into a consistent machine state, constructs a PSL for the service routine, performs any special tasks peculiar to that exception or interrupt, and loads PC with the starting VAX address of the service routine. The microcode terminates in BUT/IRD1, the signal to fetch the next VAX instruction, which is usually the first instruction in the VAX service routine.

1.3 User Microprogramming

There are three general uses for microprogramming: emulation of a target machine, instruction set enhancement, and fine tuning. By instruction set enhancement, we mean addding new machine language instructions to the machine instruction set. By fine tuning, we mean adding a routine in microcode in order to carry out a set of tasks (for example, an operating system subroutine) more efficiently than can be done in machine

language.

A number of COMET features do support writing your own microcode to augment the VAX instruction set or to fine tune some piece of software. These features are discussed below. It is not intended that COMET be used to emulate other target machines.

To support general microprogramming, the Data Path includes the following features: 18 general purpose 32 bit scratch pad registers, 8 of which have ports to both the RBUS and the MBUS; the Super Rotator, which allows very efficient (in hardware) bit picking operations; and a flexible ALU. As is the case with many microprogrammable computers, inputs to the ALU are multiplexed, the output of the ALU can be shifted or rotated (alone or in combination with the Q register), and the output can be applied to several alternative destinations.

The microsequencer supports general microprogramming in three important ways: conditional branching loop control and subroutine control. COMET has six independent flag bits (FLAG0 through FLAG5) which can be set or cleared under microprogram control (e.g., MISC/SET.FLAG0), then later used for conditional branching (e.g. BUT/FLAG0). COMET also has a five-bit step counter which can be initialized to any arbitrary value ($0 \leq n \leq 31$) by the microrder WCTRL/STEPC_WB. If this is followed by a loop which terminates with BUT/DBZ.SC, the loop will be performed n times. Each iteration will conclude with a "decrement the step counter and branch on zero." In the case of subroutine control, a 16-deep microstack is available for nested subroutine calls. The JSR/PUSH micro-order pushes the CSA (control store address) onto the microstack; the BUT/RETURN micro-order pops it. Chapter 2 discusses the functionality of the microsequencer in greater detail.

COMET provides two independent paths to memory, one for data (read/write) and one for instructions (read only). In the case of data, the VA is used as the storage address register, and the MDR (read) and WDR (write) are used as storage data registers. In the case of instruction, PC points to memory and the XB can be used as a storage data register. Both are available to the user microprogrammer, and in fact were used in the VAX firmware where two distinct Data Paths to memory were needed (cf. Section 5.2).

Finally, the COMET microinstruction (80 bits) provides a fair amount of parallelism. In a single microinstruction, one can introduce an immediate operand, perform an ALU function, push a control store address on the microstack, set or clear a flag bit, initiate a read or write to memory, and perform a multi-way conditional branch.

Access to WCS is via the opcode FC in the VAX instruction stream. As is described in Section 2.3 (branch on opcode) and 5.1 (initiate exceptions and interrupts), this causes an "opcode reserved to customers" fault. If WCS is present and if the

System Control Block vector specifies that the exception should be handled in WCS (i.e., SCB [14]<1:0> = 10 (binary), then a branch to a location in writeable control store occurs . From this point on, user microcode has control of the micromachine. The instruction stream (via XB) can be used as appropriate; the VAX firmware is also available.

Control can be returned to the VAX emulation by means of the BUT/IRD1 micro-order. We should note that when an exception or interrupt is to be handled in WCS, the PC and PSL of the suspended process are not pushed on the stack (c.f. Section 5.1). Therefore, before BUT/IRD1 is invoked, the PC must be set to the memory location of the VAX machine language program where you want the firmware to take over. The micro-order WCTRL/PC_WB (load the PC with the contents of the WBUS) can accomplish this.

CHAPTER 2. THE MICROSEQUENCER

The COMET microarchitecture contains no microprogram counter. Unless the hardware overrides the microprogram the address of the next microinstruction (i.e., the control store address-CSA) is obtained in one of three ways:

1. from the NEXT field of the current microinstruction, ORed with particular signals specified by the BUT field. (this is the multi-way conditional branch mechanism).

2. from the microstack.

3. from the VAX-specific ROMs.

All are under the control of the Branch-U-test (BUT) field. Figure 2.1 is an overview of the microsequencer operation.*

2.1 The Multi-way branch

For all but eight of the BUT codes, the CSA is obtained by performing the logical-OR of the NEXT field of the current microinstruction with the particular set of bits specified by the BUT field. Section 2.1.1 describes the branching mechanism. Section 2.1.2 delineates the sources of the signals to be ORed.

---

* We should point out that in the figures of this chapter, CSA is illustrated as a separate register. In the actual hardware, the CSA is really obtained from the microstack; i.e., from USTK[USTKP]. See Section 2.2.

2.1.1 The Branching Mechanism

The general mechanism for forming the multi-way branch is illustrated in figure 2.2.

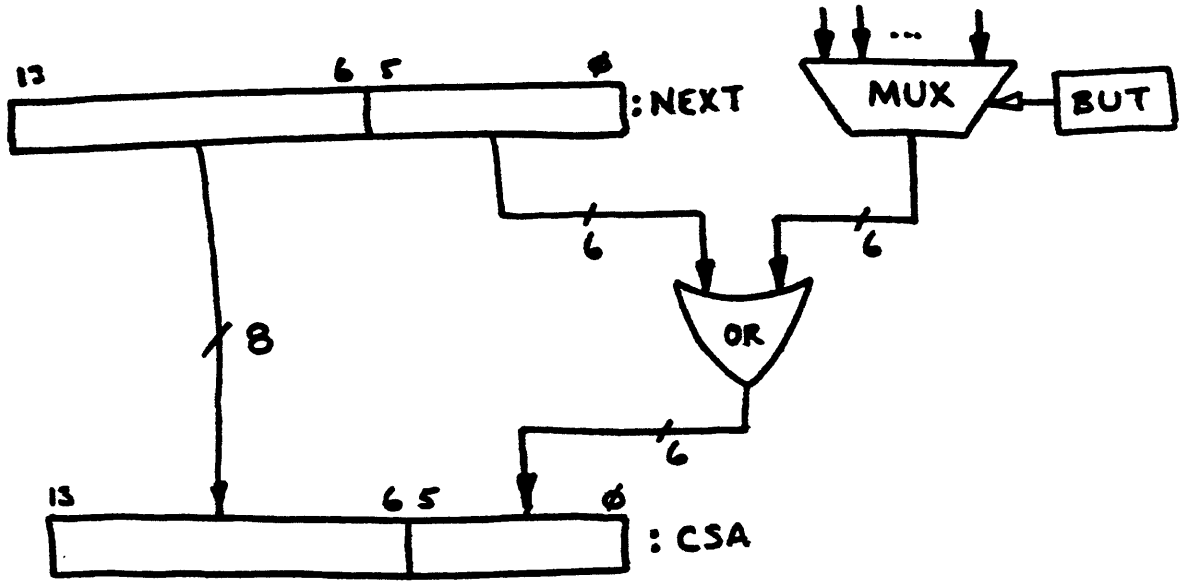


Figure 2.2

Bits <13:6> of the CSA are loaded directly from bits <13:6> of the NEXT field of the current microinstruction. The source of bits <5:0> of the CSA is determined by the particular BUT micro-order. For each of these 6 bits, if the BUT micro-order specifies a signal, then that bit of the CSA is the logical-OR of the signal specified and the corresponding bit of the NEXT field. If the BUT micro-order does not specify a signal, then that bit of the CSA is the logical-OR of "0" and the corresponding bit of the NEXT field. Table 2.1 is a complete description of this specification. As can be seen from figure 2.3, the signals to be ORed with bits from the NEXT field can come from a variety of sources. Some of the sources, such as the VA register, DSIZE latches,PSL,IR, and OSR are specific to the VAX emulation. Other sources, such as the WBUS, MBUS, and FLAG bits are more general microarchitecture structures.

Example 2.1 BUT/FLAG2TO0 specifies that three bits are to be ORed as follows:

NEXT<2> is ORed with FLAG2
NEXT<1> is ORed with FLAG1
NEXT<0> is ORed with FLAG0

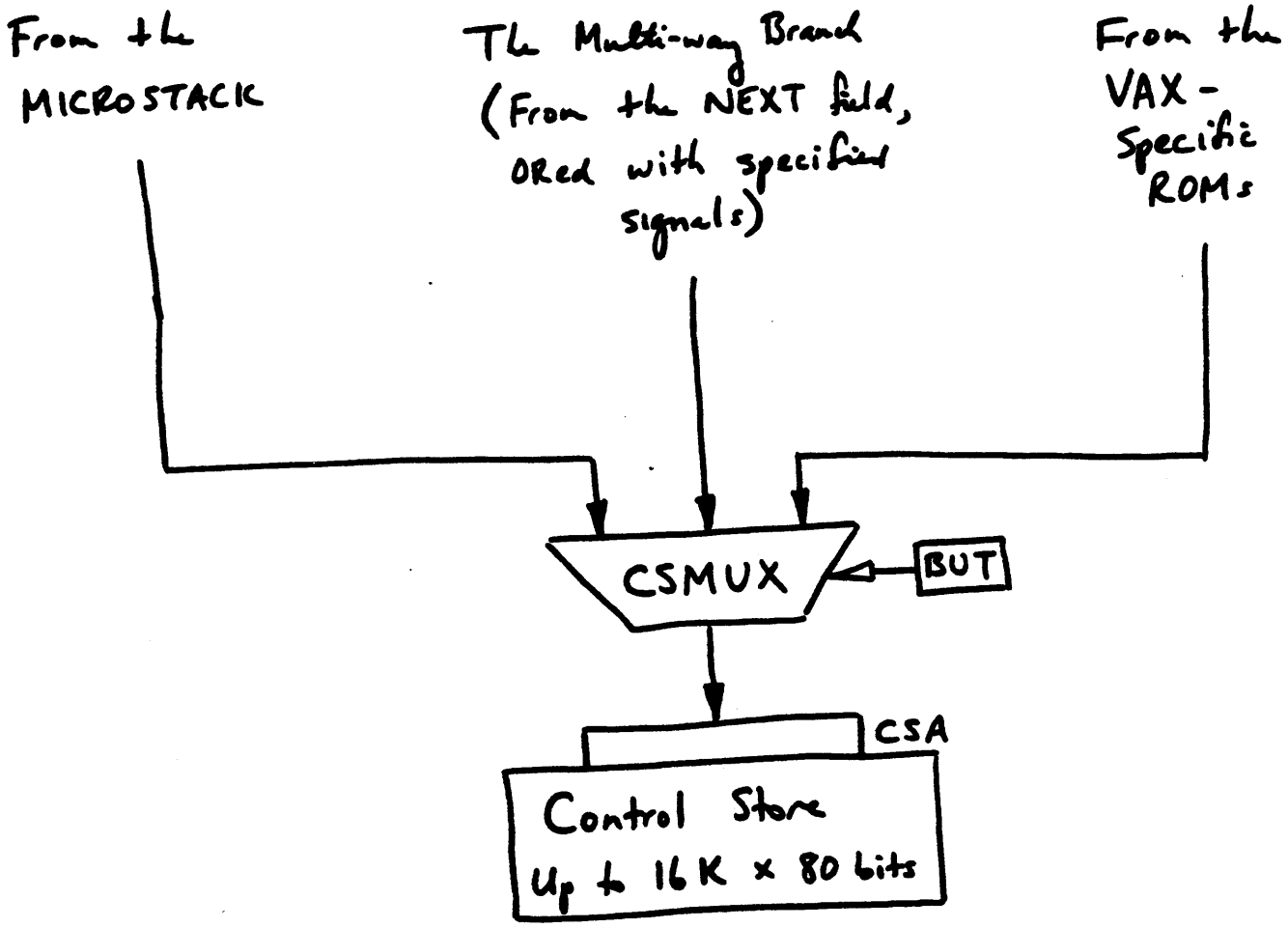
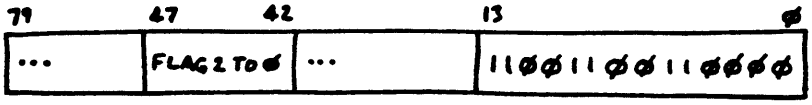


Figure 2.1 The Microsequencer (Overview)

Thus, if the current microinstruction had the following value:

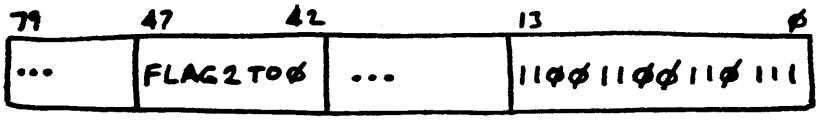


the CSA for the next microinstruction would be

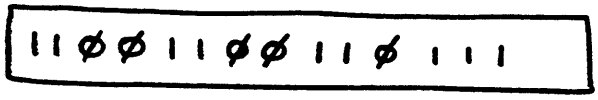
where x is the value of FLAG2, y is the value of FLAG1, and z is the value of FLAG0. In other words, an 8-way conditional branch has been produced.

In the example above, an 8-way branch was produced. If, however, NEXT <0> had been set to 1, then the effect of FLAG 0 would have been lost. The CSA of the next microinstruction would have been

resulting in a 4-way branch. In general, it is possible to produce a 0-,2-,4-,8-,16-,32-, or 64-way conditional branch, depending on the particular BUT code and the state of the six low-order bits of the NEXT field. The actual no. of possible branches is 2^k, where k is the number of "relevant" bits in the NEXT field which are cleared. A bit is relevant if it is designated for ORing by the BUT code. In the above example, if the current microinstruction had the value



the CSA of the next microinstruction would have been



Hence, a 0-way (unconditional) branch would have resulted.

Example 2.2 BUT/NOP specifies that 0 bits are to be ORed. The CSA of the next microinstruction is identical to the NEXT field of the current microinstruction. An unconditional branch is the result.

Figure 2.3 shows the flow of control for setting the CSA by the multi-way branch mechanism.

2.1.1 Sources of Signals to be ORed.

The microsequencer provides conditional branch capability based on a wide variety of relevant conditions in the microarchitecture. This is accomplished by the choices of signals to be ORed which are available to the BUT field. For example, the BUT/FPS1, BUT/FPS2, and BUT/FPS3 cause conditional branching based on settings of the front panel switches. The BUT/SPASTA micro-order causes conditional branching based on signals relating to the Scratch Pad registers (RNUM and Register Back Up Stack). Several micro-orders (e.g., BUT/WBUS1to0, BUT/WBUS31to30, and BUT/SRKSTA) provide branching based on signals on the WBUS. Most of them use the WBUS signals directly. However, the BUT/SRKSTA micro-order uses the combinational logic capacity of the Super Rotator to reduce WBUS <7:0> to one of four conditions, and provides those four conditions as SRKSTA <1:0> to the microsequencer. The BUT/UVCTR micro-order provides microbranch capability needed by the exception and interrupt handling and the memory management microcode.

In addition, COMET contains two sources which can be considered part of the microsequencer: the 5 bit step counter and the six independent flag flip-flops. The step counter can be set to any value from 0 to 31 (WCTRL/STEPC_WB) and then used to control the number of iterations through a loop. BUT/DBZ.SC decrements the step counter and then performs a conditonal branch based on whether or not the step counter equals 0. The six flag flip-flops can be set and cleared independently by micro-orders in the MISC field, then later used to perform conditional branching based on their state. For example, BUT/FLAG0 permits conditional branching based on the state of FLAG0. BUT/FLAG2TO0 permits an 8-way branch based on the states of FLAG2, FLAG1, and FLAG0.

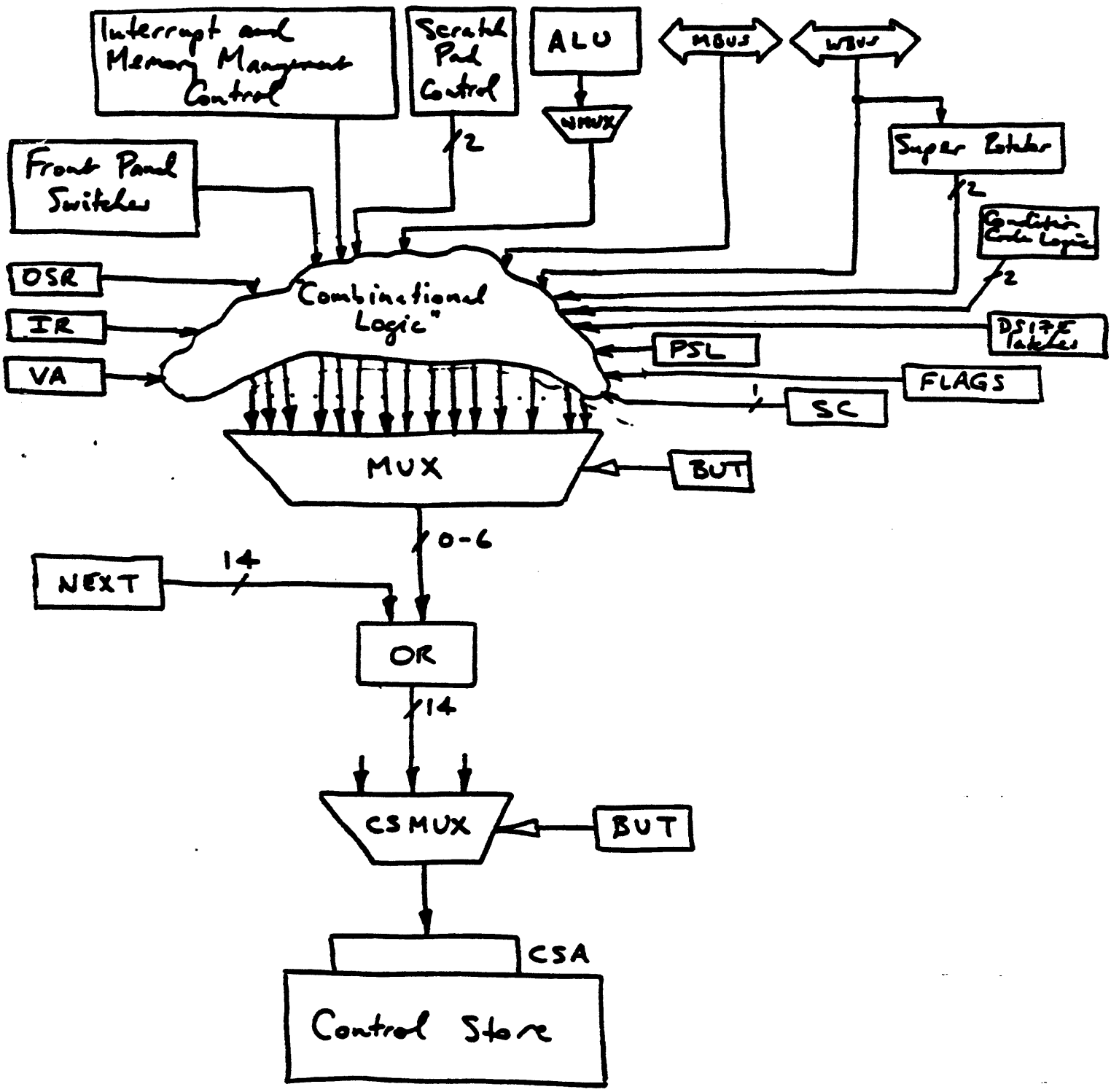


Figure 2.3 Microsequencer (The Multi-way Branch)

2.2 The Microstack

The microstack can be used to obtain the CSA of the next microinstruction. In particular, an address can be pushed onto the microstack (by means of JSR/PUSH) and later popped (by means of BUT/RETURN or BUT/RET.DINH). The push and pop mechanisms, and their usefulness in subroutine control, will be explained shortly.

The microstack is capable of storing 16 CSAs, each of length 14 bits. A microstack pointer USTKP always points to the first available word in the microstack. It is updated automatically as a consequence of the push and pop operations. Figure 2.4 shows the microstack with addresses 271, 345, and 18 stored in words 0,1, and 2, respectively. Word 3 is available for storing a CSA.

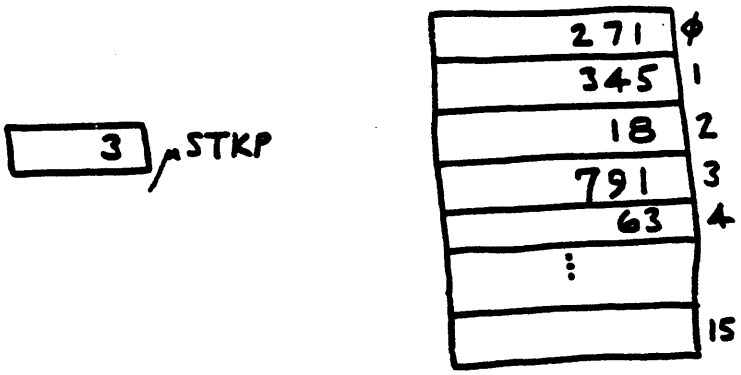


Figure 2.4. The microstack with valid entries in words 0,1,& 2.

Figure 2.5 shows the flow of control for pushing addresses onto the microstack and for popping them for use in obtaining the CSA of the next microinstruction.

2.2.1 The Push Mechanism

During the execution of every microinstruction, the following events occur relative to the microstack:

(1) During the first part of the microcycle, the JSR bit is examined. If it is set (i.e., JSR/PUSH), the microstack pointer is incremented.

(2) During the second part of the microcycle, the output of the CSMUX (i.e., the address of the next microinstruction) is stored on the microstack, in USTK[USTKP]. In COMET, USTK[USTKP] is the control store address register.

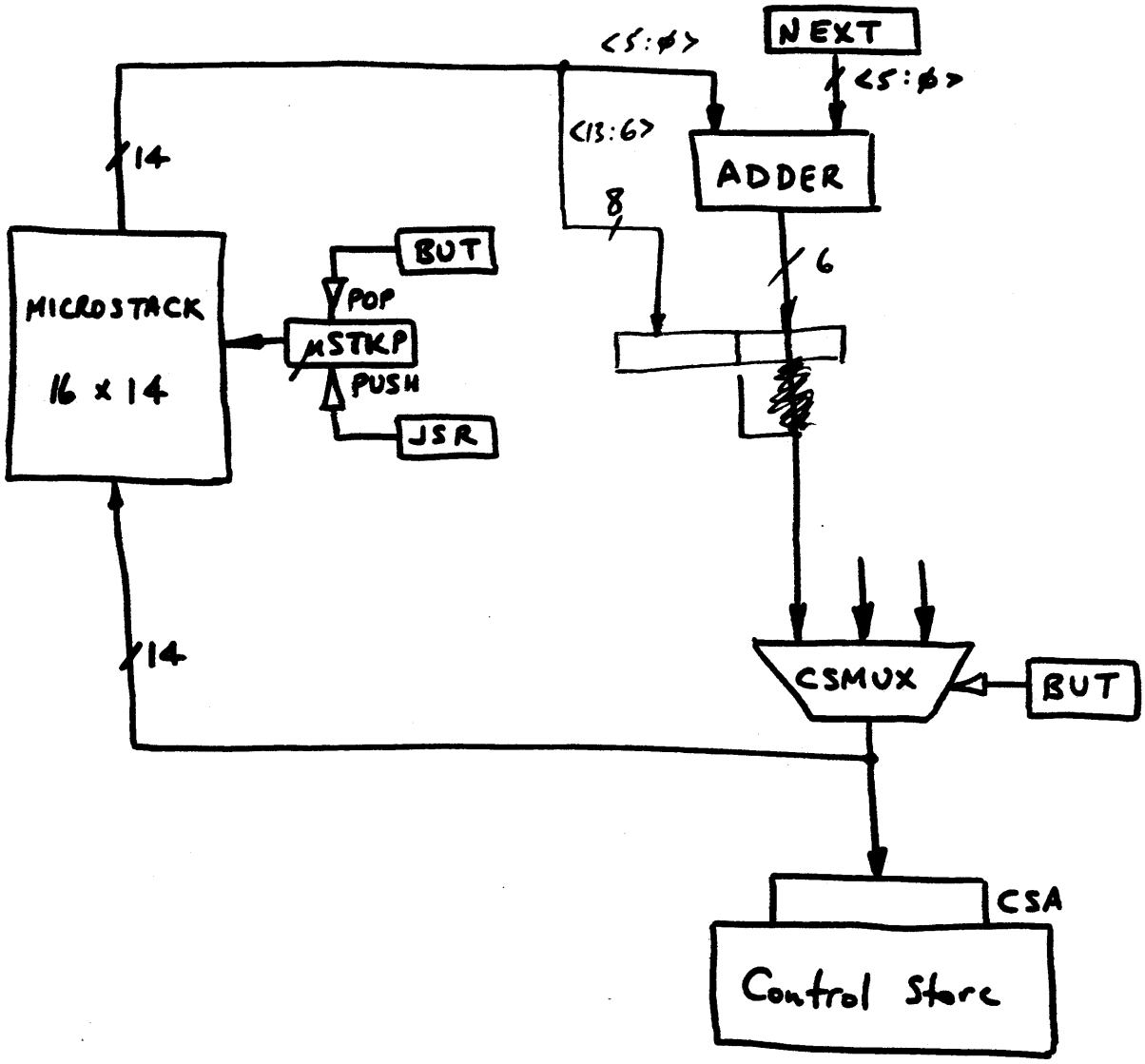
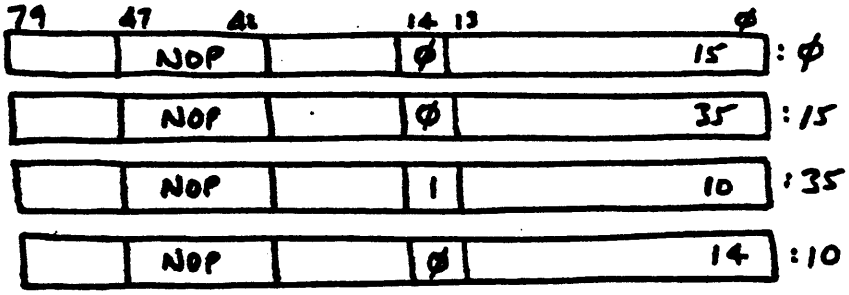


Figure 2.5 The Microsequencer (from the Microstack)

Therefore, for purposes of clarity in this report, we refer to USTK[USTKP] as the CSA register whenever we are discussing its function as the register containing the address of the next microinstruction. Thus, in figure 2.5, CSA is shown as a separate entity, although in reality, it is USTK[USTKP].

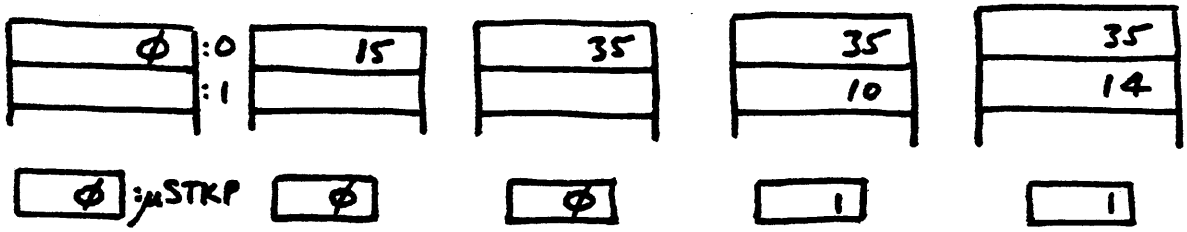
Note that loading the CSA register (i.e., storing the address of the next microinstruction is USTK[USTKP]) does not alter the microstack pointer. Thus, loading the CSA register does not push this address onto the microstack. In other words, although the address is "physically" stored in the microstack, it is stored just above the top of the microstack; i.e., the location in which it is stored is still "logically" part of available space. The following example should make this clear.

Example 2.3 Consider the sequencing of the following microcode:



The microstack behaves as follows:

Before 0 is fetched	Before 15 is fetched	Before 35 is fetched	Before 10 is fetched	After 10 is executed



Note that before 15 is fetched, the CSA 15 is physically stored in word 0 on the microstack. However, since the microstack pointer is 0, the microstack is still "logically" empty. Note, further, that the JSR bit is set (i.e., JSR/PUSH) in the microinstruction at CSA 35. This calls for pushing CSA 35 onto the microstack. The CSA 35 is stored on the microstack during the execution of the microinstruction at 15. However, the microstack remains empty until the first part of the microcycle

in which the microinstruction at CSA 35 is executed. Since the JSR bit is set, the microstack pointer is incremented, thereby completing the operation of pushing CSA 35 onto the stack. In the second half of that same microcycle, CSA 10 is stored in word 1 of the microstack. But the microstack pointer is not altered. Thus, at the end of execution of that microinstruction (i.e., before 10 is fetched), the microstack pointer contains the value 1, signifying that word 1 is the first available space and that word 0 contains a stacked CSA.

One final remark should be made with respect to the push mechanism. Since every CSA is stored on the stack, indeed that store operation is, in fact, the loading of the CSA register, no additional time is required for pushing an address on the microstack. That is, the machine does not wait until the JSR/PUSH code is detected before stacking the CSA of the current microinstruction. Thus, the push operation does not slow down the processing.

2.2.2 The Pop Mechanism

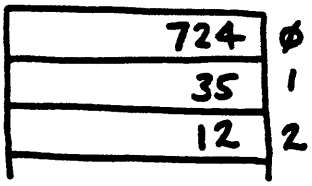
The top of the microstack is popped and used to obtain the CSA of the next microinstruction by the following sequence of operations:

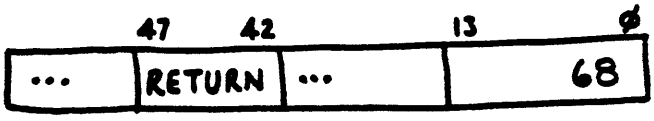
(1) USTKP <-- STKP - 1.

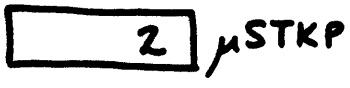
(2) CSA<13:6> <-- USTK[USTKP]<13:6>
CSA<5:0> <-- USTK[USTKP]<5:0> + NEXT<5:0>

This sequence is caused by BUT/RETURN or BUT/RET.DINH.

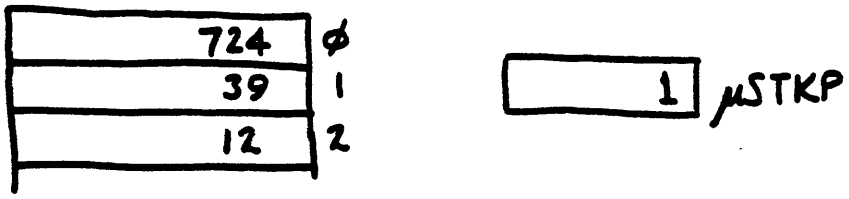
Example 2.4 Consider the following current microinstruction and contents of the microstack (all numbers in this example are decimal representations):







After execution, the CSA of the next microinstruction will be 39. The microstack will be as shown:

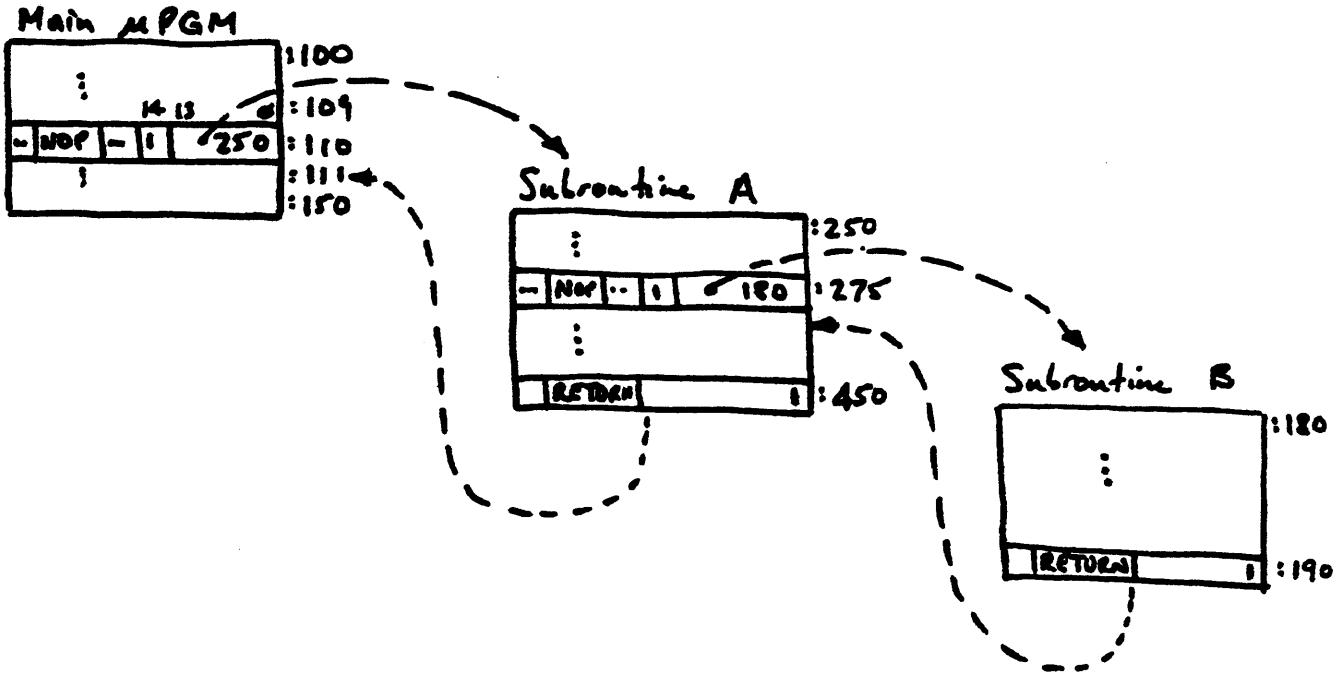


Only the address stored in word 0 is "logically" on the stack.

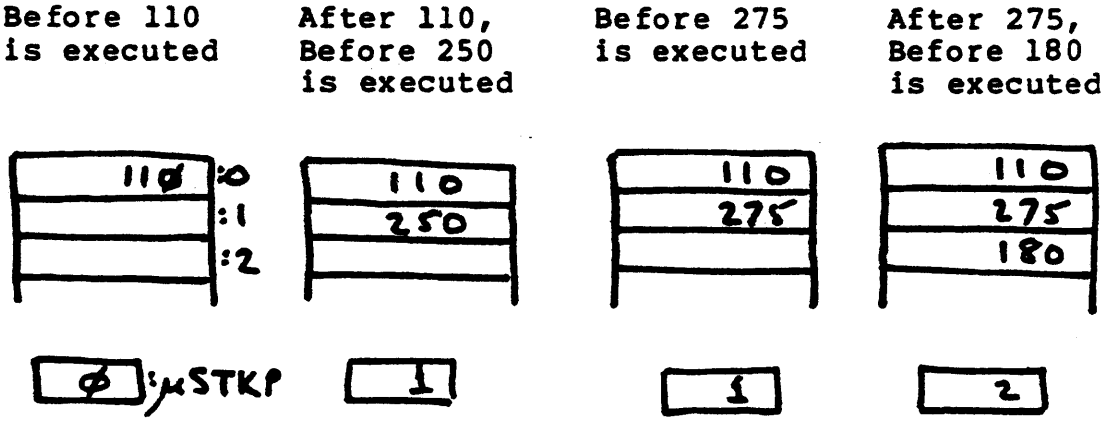
2.2.3 Subroutine Control.

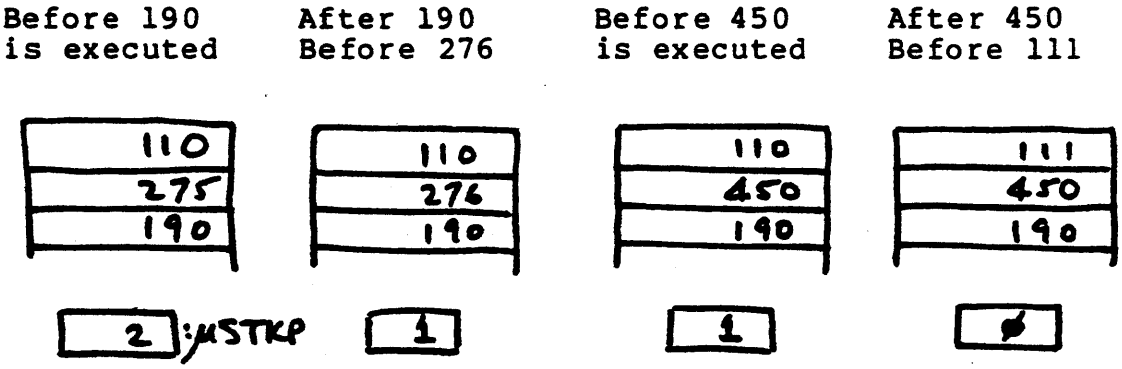
We conclude this section with an example of several nested subroutines, and a demonstration of how the flow of control is handled using the microstack.

Example 2.5 Consider a main microprogram which at CSA 110 invokes subroutine A, which in turn at CSA 275 invokes subroutine B. Assume each microprogram has its microinstructions executing in sequential order. A pictorial representation is shown below:



The contents of the microstack at critical places in the execution of the microprogram are shown below:





2.3 The VAX-Specific ROMs

In the emulation of a VAX machine instruction, two places in the microprogram flow of control stand out:

(1) the initiation of the microsequence to emulate the next machine instruction, and

(2) the initiation of the microsequence to evaluate the next operand for the current machine instruction.

Recall that a VAX instruction has variable length (each opcode can have from 0 to 6 operands), and further that the access type and data type of each operand can differ depending on whether it is the first, second, third, etc. operand of that opcode. For these two reasons, the flow of control to initiate the emulation of each machine instruction and the flow of control to evaluate each operand must be specified individually for each opcode and for each operand. The COMET microarchitecture provides three ROMs (IRD1, IRDX, and DSIZE) to assist in this specification.

Figure 2.6 shows the use of the ROMs to obtain the CSA of the next microinstruction.

Six BUT codes use the ROMs for obtaining the CSA of the next microinstruction. The BUT/IRD1 code, present in the last microinstruction of the microprogram which is emulating the current machine instruction, is the signal to begin the emulation of the next machine instruction. It uses the IRD1 ROM to obtain the starting address of the microcode to do this job. The BUT/IRDX code, present in the last microinstruction of the microprogram which is evaluating the current operand, is the signal to begin the evaluation of the next operand. It uses the IRDX ROM or the microstack to obtain the starting address of the microcode to do that job. The purpose of the four other BUT codes (BUT/IRD1TST, BUT/BRA.ON.ADD, BUT/LOD.INC.BRA., and BUT/LOD.BRA) will be explained after we describe the ROMs.

2.3.1 Organization of the ROMs

THE IRD1 ROM.

The IRD1 ROM is used to compute the starting address of the microcode which is to emulate the next VAX instruction. It consists of 1K words, each containing 8 bits. Thus, 10 bits are required to address this ROM; that is, to determine the starting address of the particular emulation microcode to be executed next. They are:

(1) The opcode of the VAX instruction (8 bits),

(2) Whether or not the FPD bit is set, and

(3) Whether or not the Floating Point Accelerator hardware is present.

The eight opcode bits are obtained directly from the XB, rather than from the IR. The BUT/IRD1 code initiates the loading of the IR from the XB. However, to wait for the IR to be loaded before obtaining the IRD1 ROM address would be to unnecessarily slow down the processing.

We need to distinguish the case when FPD is set from the case where it is cleared because the emulation proceeds differently for the two cases. If FPD is set, this means that the VAX instruction was suspended in order to service an exception or higher priority process. When that happened, the state of the machine was saved (we say, "packed"). Before the instruction can resume execution, it must be "unpacked." Ergo, two different branch addresses out of the IRD1 ROM. Section 5.1 discusses this in greater detail.

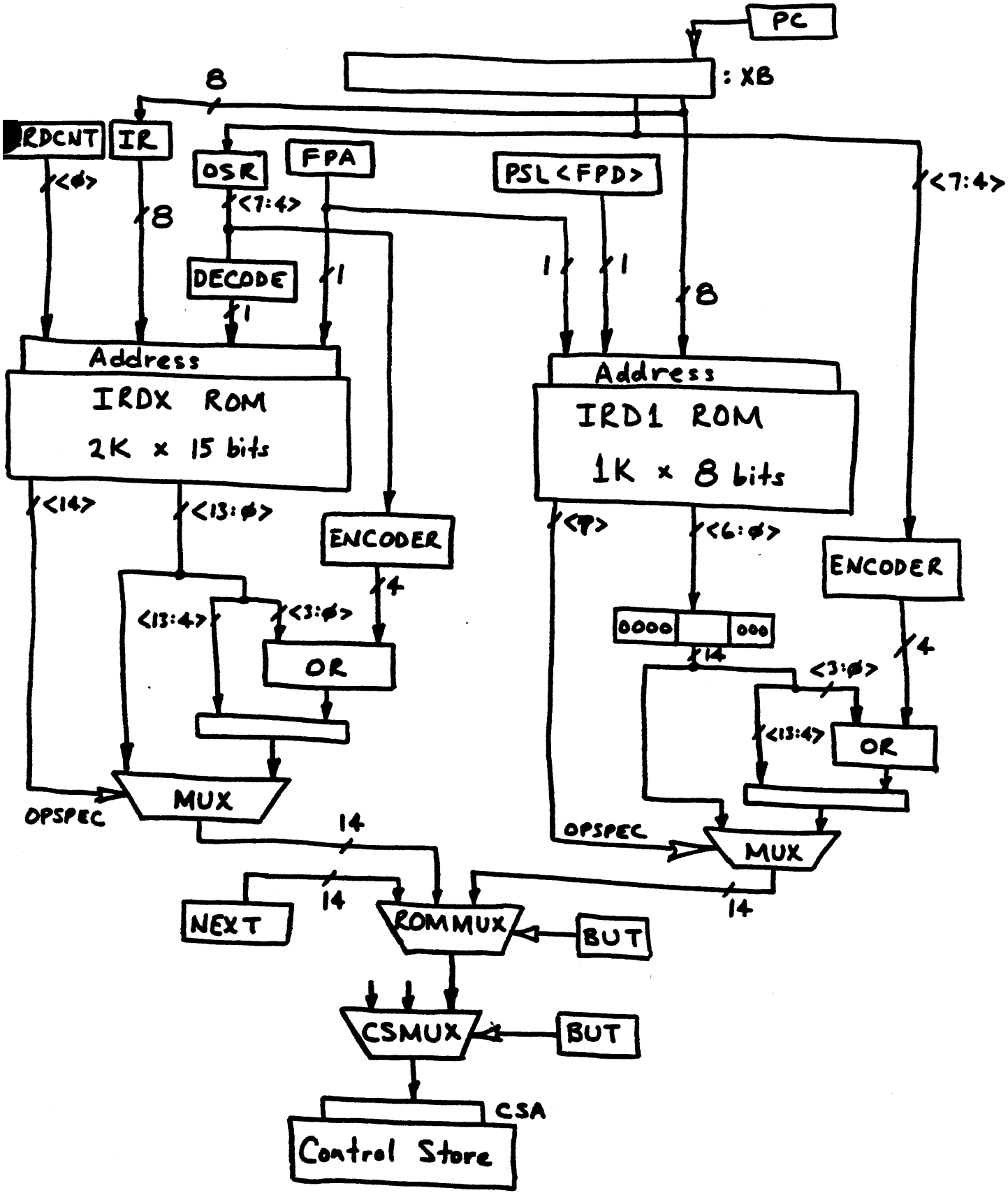
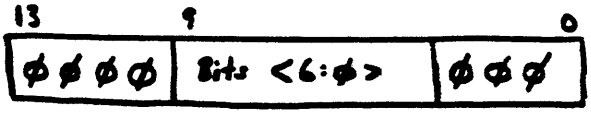


Figure 2.6 The Microsequencer (from the ROMs)

Also, we need to distinguish between the case where the FPA hardware is present and the case where it is not, since the difference in hardware will result in different microcode to emulate the instruction.

As we said, eight bits of information are stored at each IRD1 ROM address. The seven low order bits form a 14 bit address as follows:



The remaining bit, the high order bit, is called the OPSPEC bit. Its function is to prepare for the evaluation of the next operand. In the case of the IRD1 ROM, if OPSPEC is set, it performs the following functions:

(1) It causes the OSR to be loaded (from the XB) with the first operand specifier of the current VAX instruction.

(2) It loads the DSIZE latches from the DSIZE ROM. The data type of the current operand being evaluated is contained in the DSIZE latches. Since an opcode with several operands can have several different data types, the DSIZE ROM specifies the data type of each operand of each opcode. The DSIZE ROM is indexed by opcode and by the IRDCNT register, which keeps track of which operand is being evaluated.

(3) It specifies that the four low order bits of the 14 bit CSA formed above are to be ORed with a four bit encoding of the addressing mode of the first operand specifier. Since the operand is to be evaluated, and since that evaluation is a function of the addressing mode of the operand specifier, the branch address must take the addressing mode into account.

THE IRDX ROM

The IRDX ROM consists of 2K words, each containing 15 bits. As in the case of the IRD1 ROM, the high order bit is the OPSPEC bit. It provides all the functions that the OPSPEC bit does in the IRD1 ROM, and in addition it increments the IRDCNT register. The remaining 14 bits constitute a 14 bit address which is used to obtain the CSA of the next microinstruction. As with the IRD1 ROM, this address is first modified by an encoding of the addressing mode of the operand specifier if the OPSPEC bit is set.

The IRDX ROM itself is addressed by 11 bits, as follows:

1. The opcode (this time taken from the IR where it has been present since the last BUT/IRD1 micro-order). 8 bits.

2. Whether or not register mode.

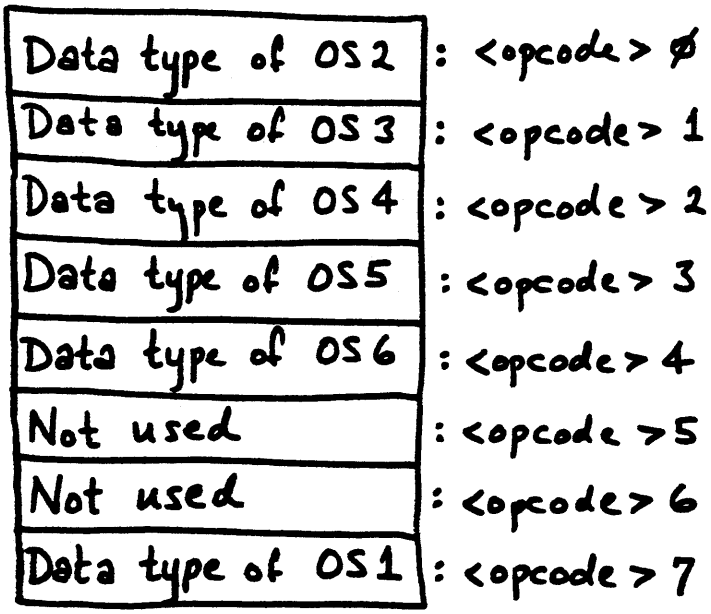
3. Whether or not the Floating Point Accelerator is present.

4. The low order bit of IRDCNT.

As will be seen, the IRDX ROM is used to obtain a CSA only if the second operand is being evaluated or if a branch to opcode-specific execution code is to be taken. The low order bit of the IRDCNT is used to distinguish between these two cases.

DSIZE ROM

The DSIZE ROM consists of 2K words, each containing two bits. The two bits specify the data type (whether byte, word, longword, or opcode dependent) of each operand of each opcode. Eleven bits are needed to address the DSIZE ROM. The eight high order bits come from the opcode; the three low order bits come from the IRDCNT register. The figure below shows the contents of the DSIZE ROM pertaining to a typical <opcode>.



2.3.2 Processing of the BUT codes.

BUT/IRD1

The detection of BUT/IRD1 in the current microinstruction is the signal that this is the last microinstruction in the emulation of the current machine instruction, and that the microarchitecture is to begin processing the next machine instruction. The hardware routine DOSERVE is invoked to initiate the service of any interrupts which may be pending. If there are no higher priority interrupts pending (or after they are serviced), two bytes are fetched from the XB; the first is loaded into the IR, the second is loaded into the OSR. Simultaneously, the IRD1 ROM is addressed. If the OPSPEC bit is not set, the loading of the OSR is inhibited. In either event, the CSA is specified as described in Section 2.3.1 above. Whether or not the OPSPEC bit is set, IRDCNT is forced to 7 at the beginning of the microcycle, which allows the DSIZE latches to be set from the DSIZE ROM (in case the OPSPEC bit is set), and then cleared to 0 at end of the microcycle.

BUT/IRD1TST

The BUT/IRD1TST code is used to test the hardware. It functions exactly like the BUT/IRD1 code except that the CSA is taken directly from the NEXT field of the microinstruction. This can be used, for example, to test if the IR and OSR have been loaded properly, the IRDCNT cleared, etc. without losing control of the microinstruction flow. The next microinstruction executed is the one at the address specified by the NEXT field, rather than the one addressed by the ROM. Note that since the address specified by the NEXT field would have NEXT<3:0> ORed with an encoding of the addressing mode if OPSPEC is set, it may be desirable to specify NEXT <3:0>=1111 to avoid that multi-way branch.

BUT/IRDX

The detection of BUT/IRDX in the current microinstruction is the signal that this is the last microinstruction in the evaluation of the current operand, and that the microarchitecture is to begin its next step. IRDCNT is examined. If IRDCNT is 0 or 1, the IRDX ROM is addressed and the CSA is formed as described in Section 2.3.1. This is the mechanism used for branching to the microcode to evaluate the second operand and to begin execution of opcode-specific microcode. If IRDCNT is greater than 1, the CSA is obtained by popping the microstack. In this case, the loading of the OSR, incrementing the IRDCNT and further addressing mode branching for the purpose of evaluating the remaining operands is controlled in the subsequent microcode by means of BUT/BRA.ON.ADD, BUT/LOD.INC.BRA, and BUT/LOD.BRA codes.

BUT/LOD.INC.BRA.

This code loads the OSR, increments IRDCNT, and determines the CSA of the next microinstruction in the same way that BUT/IRD1 does with OPSPEC set. This code is used to evaluate operands when the OSR has not been previously loaded.

BUT/BRA.ON.ADD

This code does not load the OSR, and does not increment the IRDCNT. This code is used when the OSR has been previously loaded and IRDCNT has been properly set. The CSA of the next microinstruction is formed as in BUT/LOD.INC.BRA.

BUT/LOD.BRA.

This code loads the OSR and then forms the CSA as in BUT/LOD.INC.BRA. The IRDCNT is not updated. This code is used in evaluating the Base Operand Address in index addressing mode. Since the BOA is the second operand address to be evaluated for the one operand, the IRDCNT must not be changed.

2.3.3 AN EXAMPLE

We conclude this section with an example, showing how the BUT codes are used in emulating a VAX machine instruction.

Example 2.6 Consider a VAX machine instruction having five operands, with the fourth operand specifier designating index mode. Figure 2.7 illustrates the flow of control to emulate the machine instruction.

Emulation starts at (1), the last microinstruction of the microcode which emulates the previous machine instruction. BUT/IRD1 is a signal to begin the emulation of this machine instruction. The IRD1 ROM is addressed. Since OPSPEC=1 the OSR is loaded with the first operand specifier. The DSIZE latches are set from IRDCNT=7,?? IRDCNT is set to 0 and a branch is taken to B, the microcode to evaluate the first operand. Note that B (as well as C,E,F,G, and H) is common code. It is independent of the opcode. It depends only on the addressing mode of the operand specifier, and that addressing mode was used in computing the branch address.

The last microinstruction in the evaluation of the first operand (2) contains BUT/IRDX. Since IRDCNT=0, the IRDX ROM is addressed, using IRDCNT <0> as part of its index. Since OPSPEC=1, the OSR is loaded with the second operand specifier, the DSIZE latches are set according to IRDCNT=0, IRDCNT is incremented, and a branch is taken to C, the microcode to evaluate the second operand.

The last microinstruction in C contains BUT/IRDX. Since IRDCNT is still less than 2 (IRDCNT=1), the IRDX ROM is again addressed. Since OPSPEC=1, the OSR is loaded with the third operand specifier, the DSIZE latches are set according to IRDCNT=1, IRDCNT is incremented, and a branch is taken to D. The four low-order bits of the branch address would be obtained by ORing the four low-order bits in the IRDX ROM with an encoding of the addressing mode contained in OSR. Thus, to insure that the branch taken is to D, it is necessary to insist that the four low-order bits of D be 1111, and that the four low-order bits in IRDX ROM also be 1111. (Alternatively, we could have set OPSPEC=0, in which case there would be no branching on addressing mode, and there would be no such restriction on the nature of the control store address D. In that case, the BUT micro-order at (4) would have to be LOD.INC.BRA in order to load OSR and update IRDCNT.)

The microcode starting at D is specific to the VAX machine instruction being emulated.* After executing some number of microinstructions (perhaps none), the microinstruction at 4 is executed. This is a branch to E, the common microcode to evaluate the third operand. The BUT/BRA.ON.ADD code is used since the OSR has already been loaded and the IRDCNT has already been incremented. Both occurred at (3). The JSR/PUSH code is included to store the CSA of (4) on the microstack.

The last microinstruction in E contains BUT/IRDX. Since IRDCNT=2, the net effect is to pop the microstack, causing a branch to (6) due to NEXT/1. In order to evaluate the fourth operand, BUT/LOD.INC.BRA is used. This causes the OSR to be loaded with the fourth operand specifier, IRDCNT to be incremented, and a branch to the common code at F.

We have assumed that the fourth operand specifier designated index mode. Index mode requires a second operand specifier (called the base operand specifier) for this one operand. At some point, therefore, we must load that operand specifier and branch to the common code to evaluate it. We do not increment IRDCNT since we are still dealing with the fourth operand. This is accomplished by BUT/LOD.BRA at (7).

---

* If that were not the case, for example, if the microcode at D were common code used to evaluate the third operand, there would be no way to return control to the emulation of the current machine instruction.

Processing continues in this vein until (11), at which point all five operands have been evaluated. BUT/IRDX pops the stack (IRDCNT=4) and control goes to (12), the microcode to complete the emulation of the current VAX machine instruction.

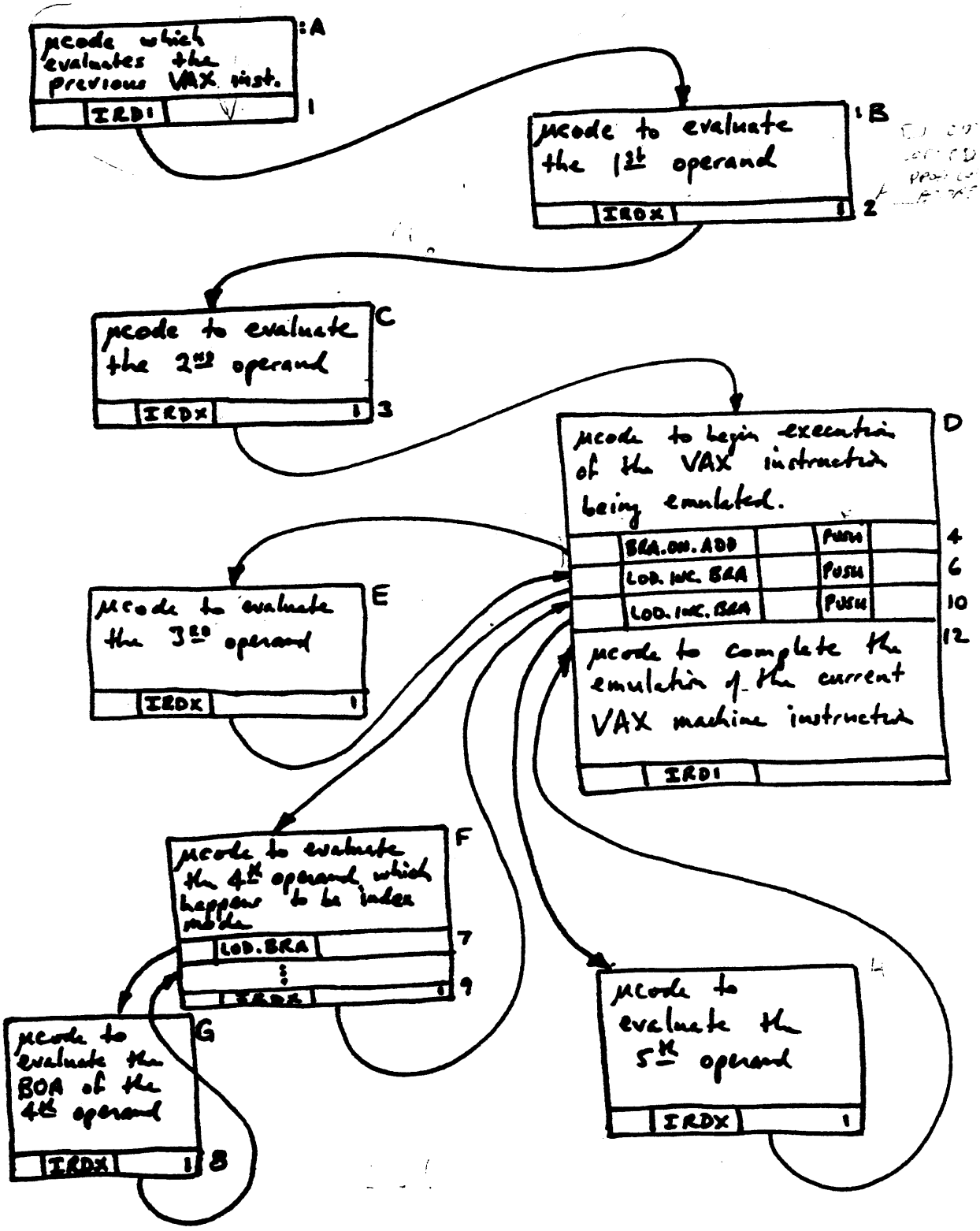


Figure 2.7 Microsequence Flow Using ROMs. (Example 2.6)

Table 2.1 BUT-OR[redacted]gnals for Microsequencing

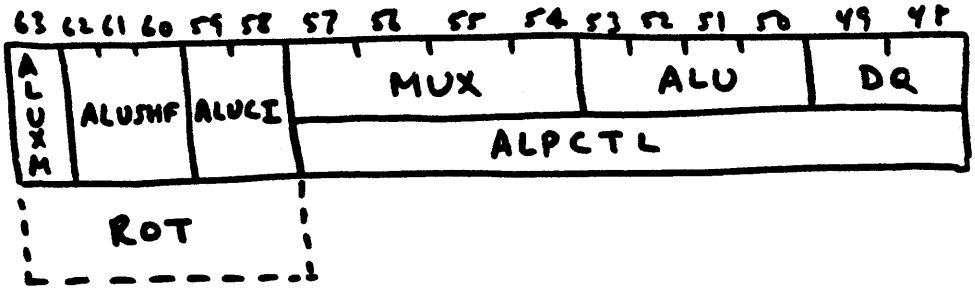
BUT FIELD	CSA<5>	CSA<4>	CSA<3>	CSA<2>	CSA<1>	CSA<0>
BUTXB=20						BUT XB UTRAP = 1, ELSE 0
CM.ODD.ADD=21						NOTE 2
IR.2TO0=19				IR<2>	IR<1>	IR<0>
IR5=23						IR<5> (NOTE 8)
IR2=22						IR<2> (NOTE 8)
REGMODE=24						OSR<7:4>=5
FRO.FLTZ=2A					NOT.MBUS<15>	WX<31:0>.NE.0
WBUS1TO0=9					WBUS<1>	WBUS<0>
WBUS1TO0.NE.0=0E						WBUS<1:0>.NE.0
WBUS5TO0=08	WB<5>	WB<4>	WB<3>	WB<2>	WB<1>	WB<0>
WBUS0=0A						WB<0>
WBUS31TO30=1B					WB<31>	WB<30>
WX.EQ.0=28						WMUX<31:0>.EQ.0
WX.NE.0=29						WMUX<31:0>.NE.0
BCDCHK=26						NOTE 1
SRKSTA=37					SRKSTA<1>	SRKSTA<0> (NOTE 6)
SPASTA=2E					SPASTA<1>	SPASTA<0> (NOTE 5)
CCBR=2D					CCBR<1>	CCBR<0> (NOTE 7)
CCBR1.CCBR0.IR0=35				CCBR<1>	CCBR<0>	.NOT.IR<0> (NOTE 8)
CCBR0.SRKSTA0=36					CCBR<0>	SRKSTA<0>
DSIZE=31					DSIZE<1>	DSIZE<0> (LATCHES)
DBZ.SC=0C						SC.EQ.0
WCSENA=27						0=WCS PRESENT AND ENABLED 1=WCS NOT PRESENT OR WCS DISABLED
BR.SC-4.INT-TS=0D				NOTE 3	NOTE 3	NOTE 3
MM.ALLOW.INT=08					NOTE 4	
INT-TIMSERV=2B					.NOT.INT	TIMSERV
CCBR1.INT-TS=2C					CCBR<1>	(INT.OR.TIMSERV).AND.NOT.CCBR<1>
FPD=0F						PSL<27>
PSLC=25						PSL<C>
PSLTP=2F						PSL<TP>
UVCTR=1E			UVCTR<3>	UVCTR<2>	UVCTR<1>	UVCTR<0> (NOTE 9)
FPS1=34				NOT.HALT	START<1>	START<0> (NOTE 10)
FPS2=33					BOOT<1>	BOOT<0> (NOTE 11)
FPS3=32					ACLO	FPLOCK
FLAG0=10						FLAG0
FLAG1=17					FLAG1	
FLAG2=12				FLAG2		
FLAG3=13						FLAG3
MM.NOINT=11						MM.NOINT (FLAG4)
STACKFLG=1A						PSHSTACK (FLAG5)
FLAG2TO0=16				FLAG2	FLAG1	FLAG0
FLAG1TO0=14					FLAG1	FLAG0
F1.XOR23=15					FLAG1	FLAG2.XOR.FLAG3

CHAPTER 3. THE DATA PATH, PART I: THE ALU

The computational element of the COMET microarchitecture is the Data Path. In this chapter, we discuss that part of it consisting of the ALU, the D and Q registers, the ALKC, ALUSO, and LOOP flags, and the necessary logic to support the relevant fields of the microinstruction. We call this part of the Data Path the ALU system. In Chapter 4, we will discuss the rest of the Data Path, in particular, the Super Rotator and the Scratch Pad registers.

Inputs to the ALU come from the RBUS, the MBUS, the Super Rotator, and the D and Q registers. Output of the ALU goes to the WBUS, the D register and/or the Q register. The ALKC flag is set during add and subtract operations to reflect a carry out of the ALU during addition or a borrow for the most significant bit during subtraction. The ALUSO and LOOP flags are used in the multiply and divide operations. Figure 3.1 is an overall block diagram of the ALU system.

The fields of the microinstruction that control the functioning of the ALU are shown below:



We will study the ALU in several parts. In section 3.1, we identify the basic functionality of the ALU system; i.e., that involving the ALUXM, ALUCI, MUX, ALU and DQ fields. In this case, the inputs to the ALU are specified by the MUX, ALUCI, and ALUXM fields. The function performed by the ALU is specified by the ALU field, and the destination of the output of the ALU is controlled by the MUX and DQ fields. In section 3.2, we show how the ALUSHF field provides for shifting and rotating the output of the ALU, the Q register, and both. In section 3.3 we discuss the ALU special functions, wherein the 10 bit field ALPCTL specifies as a unit the entire ALU operation (i.e., inputs, function, and destination of output).

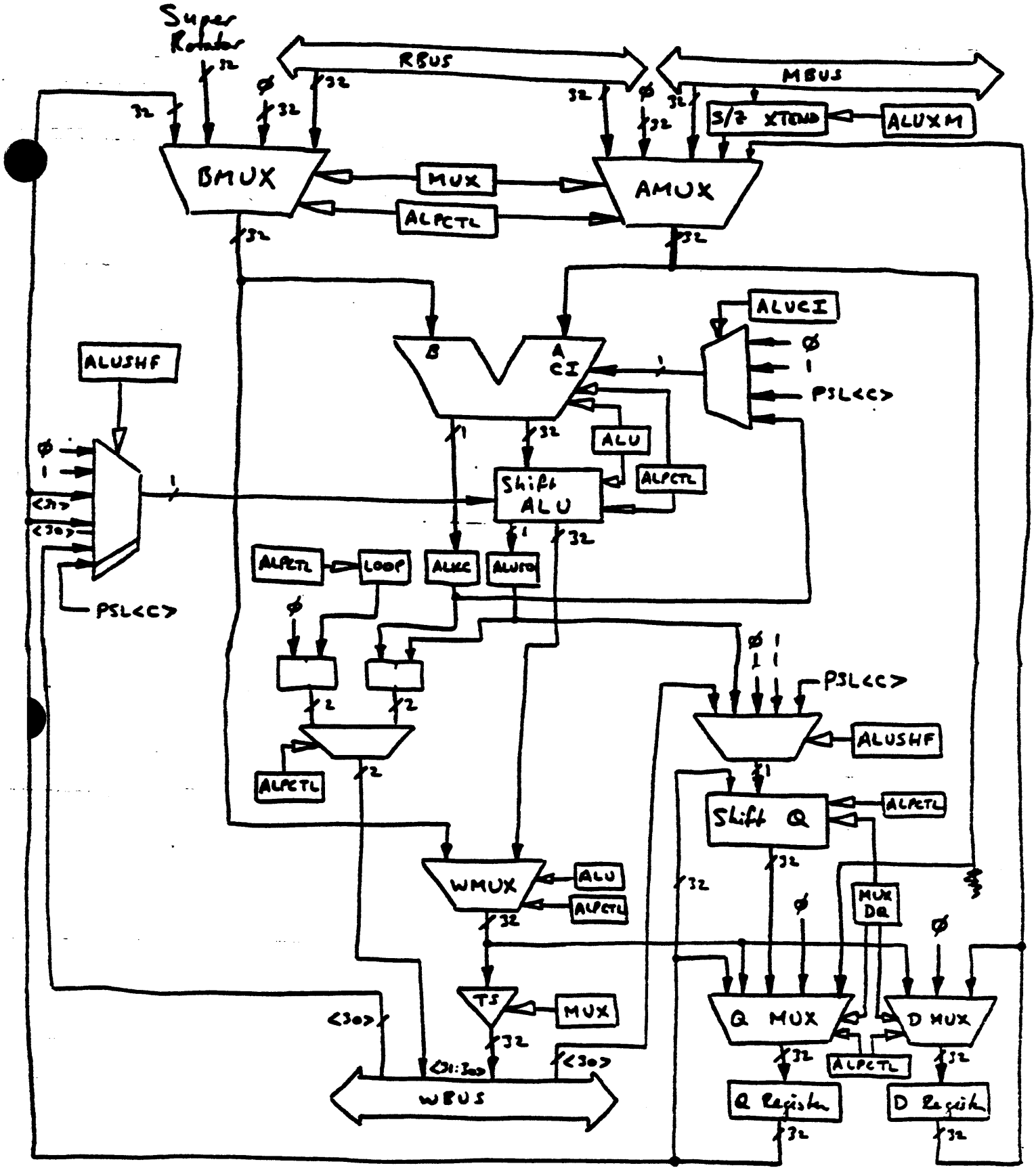
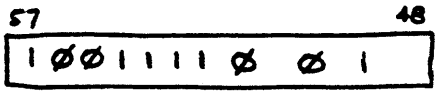


Figure 3.1 ALU System - Overall Block Diagram

Note that the above field specifications provide opportunity for two sets of conflicts. The first set of conflicts involves bits <57:48>. It is resolved in the following way. If the ALPCTL field specifies one of the 50 special functions (from a set of 1024 possible codes), then that special funtion is performed rather than the separately decoded MUX, ALU, and DQ fields. For example, if bits <57:48> are specified as



then the hardware performs the fast multiply operation (ALPCTL/MULFAST), instead of setting the D register to the logical-AND of the RBUS and the complement of the D register and shifting the Q register one bit to the right (MUX/D.R2, ALUOD/ANDNOT.OD, DQ/SQR.D..WX).

The second set of conflicts involve bits <63:58>. This field is also the ROT field which controls the Super Rotator (see Section 4.1). This conflict is resolved as follows: If the MUX field specifies that the output of the Super Rotator is to be an input of the ALU or if the ROT field specifies the loading of either of its two (P or S) latches (again, see Section 4.1), then the ALUSHF field and the ALUCI field, are both disabled. Their control functions operate as if the codes specified were 0. The ALUXM field, on the other hand, is not disabled. It continues to function on the basis of the value in bit <63>.

3.1 Basic Functioning

The basic funtioning of the ALU is shown in Figure 3.2. The ALU has three inputs: 32 bit A and B inputs and in the case of arithmetic operations, a single bit carry input (CI). The A and B inputs are both multiplexed under the control of the MUX field. The CI input is multiplexed under the control of the ALUCI field. The function performed by the ALU is specified by the ALU field. The ALU generates a 32-bit ouput and in the case of arithmetic operations, a one bit carry out (ALKC). The destination of the output of the ALU is controlled by the MUX field in conjunction with the DQ field.

Table 3.1 shows the multiplexing of the A and B inputs to the ALU. Sources for the AMUX are the MBUS, the RBUS, the constant 0, and the D register. In those cases where the MBUS source as specified by the MSRC field is less than 32 bits, the MBUS is sign or zero extended to 32 bits before it is applied to the AMUX. The ALUXM field specifies whether the extension should be sign extend or zero-extended. Sources for the BMUX are the RBUS, the output of the Super Rotator, the Q register and the constant 0.

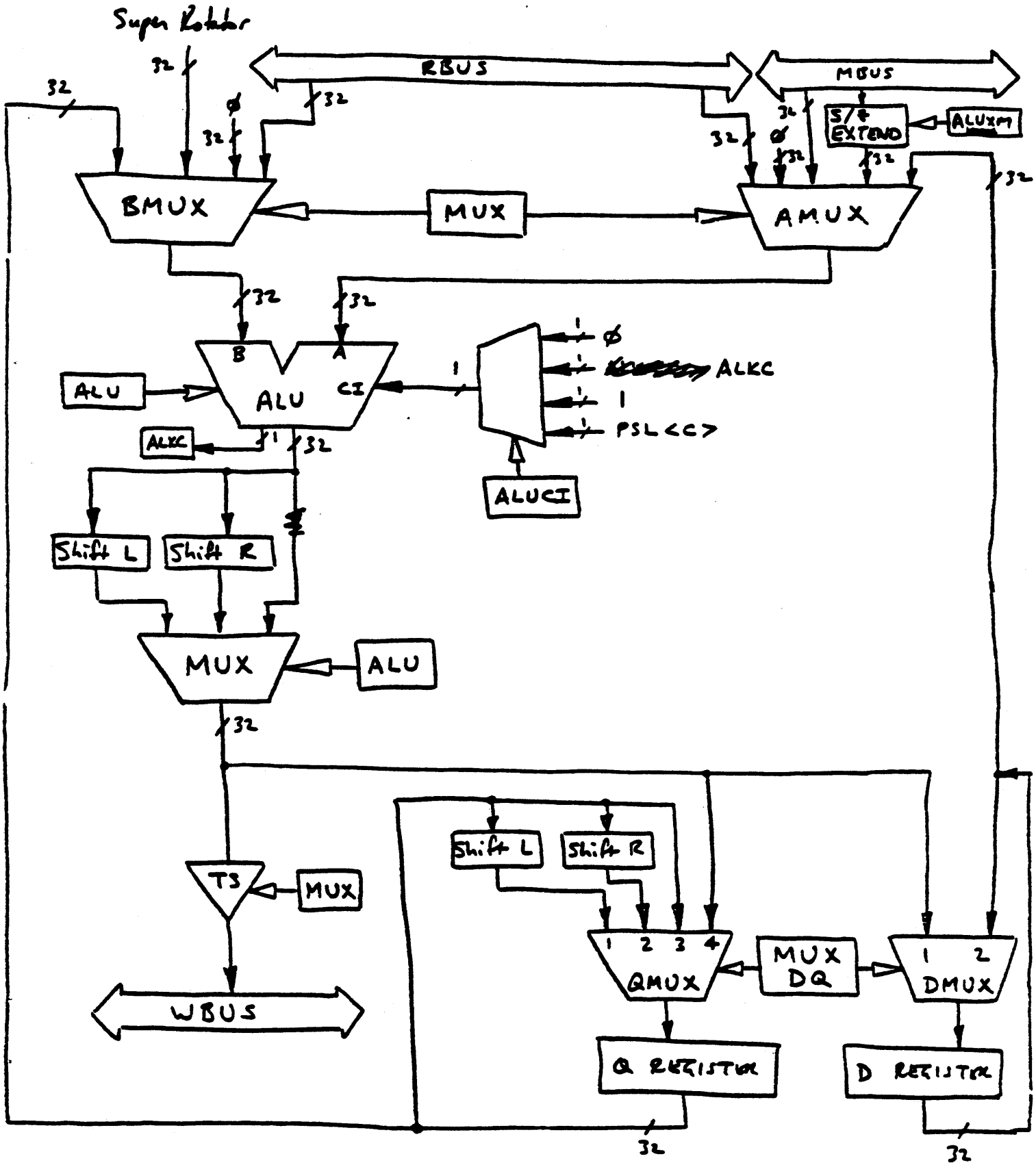


Figure 3.2 Basic Functioning of the ALU

Table 3.2 shows the functions performed by the ALU as specified by the ALU field. Note that certain functions specify that the output of the ALU should be shifted one bit right or left before being output to the WBUS, D register, or Q register. The details of that shifting mechanism will be covered in the next section.

The destination of the output of the ALU is controlled by the MUX field in conjunction with the DQ field. The MUX field alone controls whether or not the ouput of the ALU is to go on the WBUS. Note the tri-state device between the output of the ALU and the WBUS (figure 3.2). If the MUX field specifies a binary code of 1001 or 1101, the output is inhibited. For all other MUX codes, the output of the ALU goes on the WBUS. In the case of the D and Q registers, the MUX field acts in a bit steering capacity for the DQ field. Table 3.3 shows the details of that control. Note that in certain cases, the Q register is shifted one bit to the left or right. Details of that shift mechanism will be covered in the next section.

3.2 The Single Bit Shift Operation

As was stated in Section 3.1, the Q register and the output of the ALU can be shifted one bit left or right before being applied to their respective destinations. Four fields are involved in the control of the shift operation. The ALU field (cf. Table 3.2) specifies whether the output of the ALU is to be shifted right, left, or not at all. The MUX and DQ fields (cf. Table 3.3) specify whether the Q register is to be shifted right or left or not at all. The ALUSHF field specifies the bits to be shifted into the Q register and to the output of the ALU. Table 3.4 delineates the ALUSHF specification.

Two of the codes in Table 3.4 (ALUSHF/SHF and ALUSHF/ROT) require some explanation. In these two cases the ALU output and the Q register can be treated as if they were a 64 bit register. Figure 3.3 illustrates the shift operation for each of the 16 cases.

3.3 Special Functions - The ALPCTL field

The field <57:48> has 1024 possible codes. For all but 50 of them, the MUX, the ALU, and DQ fields are decoded as described in Section 3.1 to determine the functioning of the ALU system. In the remaining 50 cases, the 10 bits are decoded as a

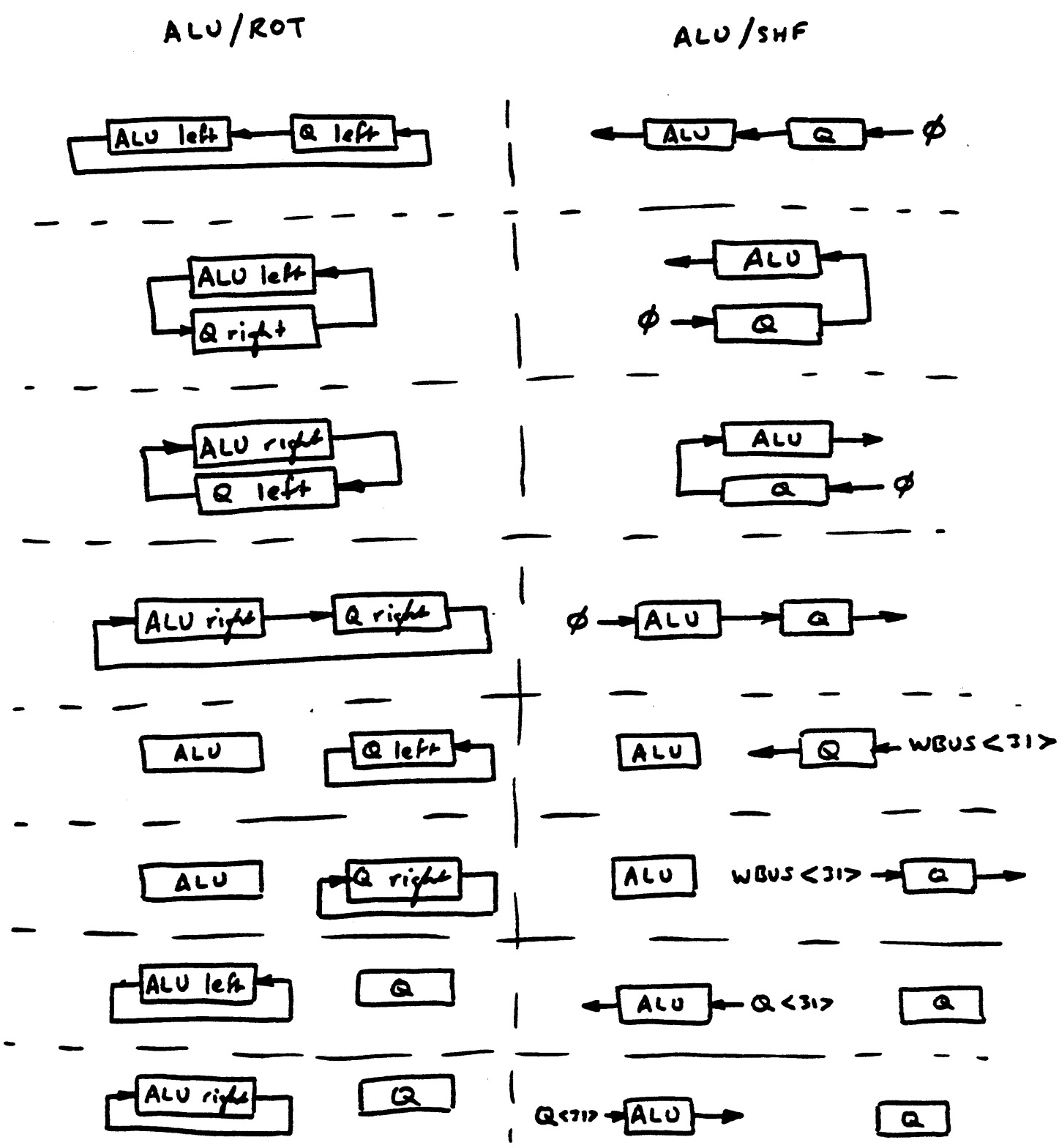


Figure 3.3 Shift Operation for ALUSHF/SHF and ALUSHF/ROT Micro-orders

unit (i.e., the ALPCTL field) to specify the functions to be performed. Figure 3.4 is a block diagram of the data flow for the ALPCTL functions. Table 3.5 lists the 50 functions.

Note in particular the multiply and divide operations which are implemented respectively as sequences of shifts and adds and subtracts. The ALUSO and LOOP flags are provided to aid in the microprogramming of these operations. Consider, for example, the multiply routine. LOOP is set during the first iteration of a multiply routine and then used to control subsequent iterations. ALUSO is the bit shifted out of the ALU. Since multiply is implemented as a sequence of shifts and adds, ALUSO contains the low order bit of the multiplier which is used to determine whether or not the multiplicand should be added to the partial product.

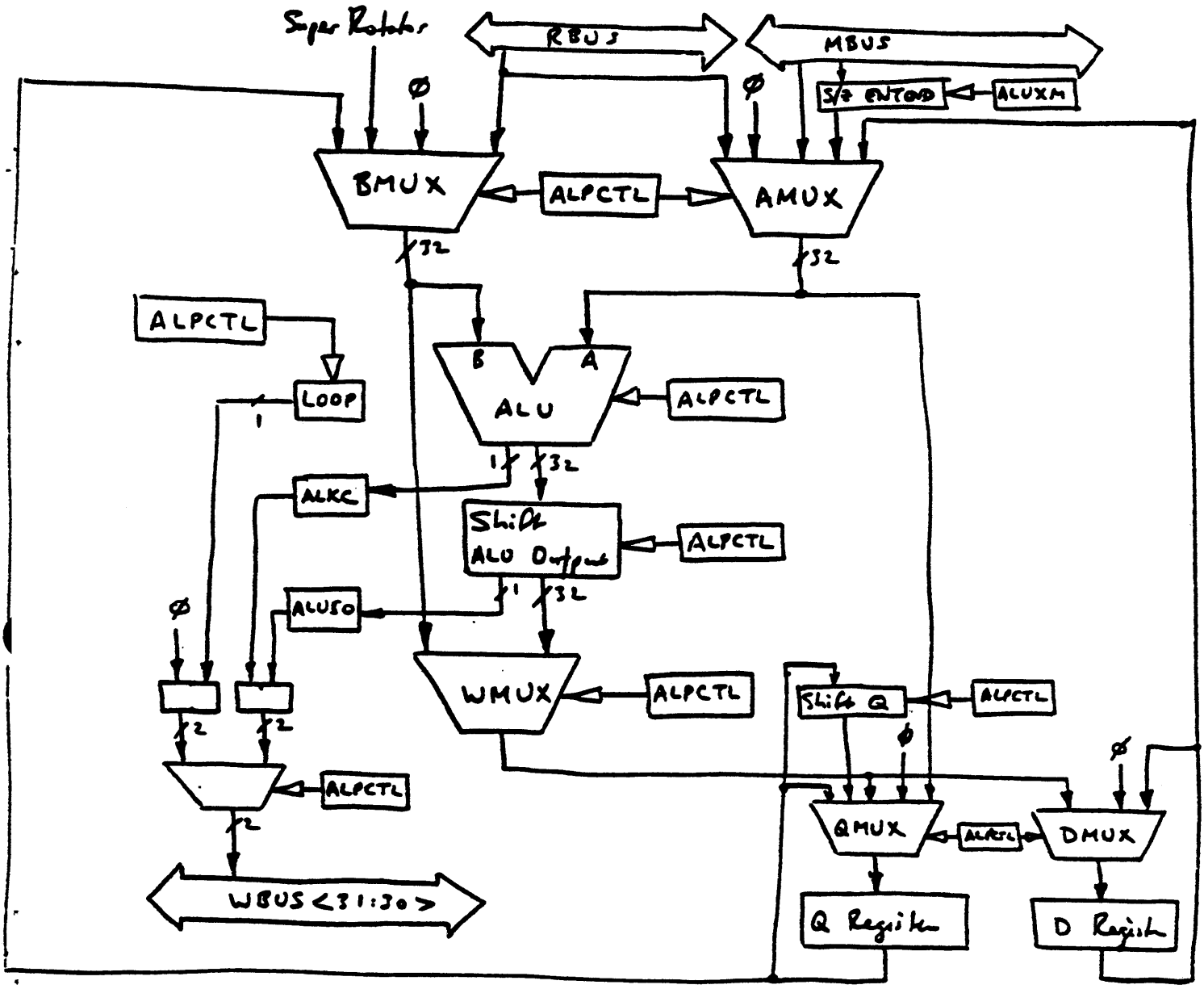


Figure 3.4 Data Flow for the ALPCTL functions

Table 3.1 Sources for the A and B inputs to the ALU

MUX	A input	B input
0000	MBUS	RBUS
0001	MBUS	RBUS
0010	MBUS	Q REGISTER
0011	MBUS	Q REGISTER
0100	MBUS	Super Rotator
0101	Ext. MBUS	RBUS
0110	Ext. MBUS	Q Register
0111	Ext. MBUS	Super Rotator
1000	D Register	RBUS
1001(0)*	D Register	RBUS
1001(1)*	D Register	Constant 0
1010	D Register	Q Register
1011	D Register	Q Register
1100	D Register	Super Rotator
1101	Constant 0	Super Rotator
1110	RBUS	Q Register
1111	RBUS	Super Rotator

* Bit <49> is steering bit for MUX/1001

Table 3.2 ALU Function Control of the ALU

ALU	FUNCTION
0000	A - B - CI
0001	A - B - CI in BCD
0010	A - B - CI and shift the result right one bit
0011	A - B - CI and shift the result left one bit
0100	A + B + CI
0101	A + B + CI in BCD
0110	A + B + CI and shift the result right one bit
0111	A + B + CI and shift the result left one bit
1000	logical-AND (A,B)
1001	logical-OR (A,B)
1010	logical-AND (A,B) and shift the result right one bit
1011	logical-AND (A,B) and shift the result left one bit
1100	B - A - CI
1101	exclusive-OR (A,B)
1110	logical-AND (A, not B)
1111	logical-AND (not A, B)

Table 3.3 MUX, DQ Control of D and Q Registers

If MUX/1001 (Binary)

DQ field	Q	D
00	1*	1**
01	2	1
10	1	1
11	2	1

If MUX/(0001 or 0011 or 1011)

DQ field	Q	D
00	1	2
01	2	2
10	1	1
11	2	1

If MUX/anything else

DQ field	Q	D
00	3	2
01	4	2
10	3	1
11	4	1

* Numbers in this column are taken from the QMUX inputs in Figure 3.2

** Numbers in this column are taken from the DMUX inputs in Figure 3.2

Table 3.4 ALUSHF control for the Single bit Shift Operations

ALUSHF	ALU	Q
ZERO	0	0
ONE	1	1
SHF	Shift ALU'Q together (see figure 3.3)	
ROT	Rotate ALU'Q together (see figure 3.3)	
ALU0.Q1	0	1
ALU1.Q0	1	0
WBUS30	WBUS<30>	WBUS<30>
PSLC	PSL<C>	PSL<C>

Table 3.5 ALPCTL Special Functions

ALPCTL code	RESULTS
WX_D_Q.Q_D	WMUX,D <-- Q OLD Q <-- D OLD
WX_D_Q.Q_M	WMUX,D <-- Q OLD Q <-- MBUS
WX_D_R.Q_D	WMUX,D <-- RBUS Q <-- D OLD
WX_D_R.Q_M	WMUX,D <-- RBUS Q <-- MBUS
WX_D_R.Q_XM	WMUX,D <-- RBUS Q <-- S/Z MBUS
WX_D_S.Q_0	WMUX,D <-- SUP ROT Q <-- 0
WX_D_S.Q_R	WMUX,D <-- SUP ROT Q <-- RBUS
WX_D_S.Q_XM	WMUX,D <-- SUP ROT Q <-- S/Z MBUS
WX_Q.Q_D	WMUX <-- Q OLD Q <-- D
WX_Q.Q_M	WMUX <-- Q OLD Q <-- MBUS
WX_R.Q_D	WMUX <-- RBUS Q <-- D
WX_R.Q_M	WMUX <-- RBUS Q <-- MBUS
WX_R.Q_XM	WMUX <-- RBUS Q <-- S/Z MBUS
WX_S.Q_0	WMUX <-- SUP ROT Q <-- 0
WX_S.Q_R	WMUX <-- SUP ROT Q <-- RBUS
WX_S.Q_XM	WMUX <-- SUP ROT Q <-- S/Z MBUS
WX_D_Q_S	WMUX,D&Q <-- SUPER ROTATOR
WX_D_S	WMUX,D <-- SUPER ROTATOR
WX_Q_S	WMUX,Q <-- SUPER ROTATOR
WX_S	WMUX <-- SUPER ROTATOR
WX_D_Q_.NOT.S	WMUX,D&Q <-- .NOT.(SUPER ROTATOR)
WX_D_.NOT.S	WMUX,D <-- .NOT.(SUPER ROTATOR)
WX_Q_.NOT.S	WMUX,Q <-- .NOT.(SUPER ROTATOR)
WX_.NOT.S	WMUX <-- .NOT.(SUPER ROTATOR)
WX_D_DSL.SQL	WMXU,D <-- D SHF LEFT Q <-- SHF LEFT
WX_D_DSL.SQR	WMXU,D <-- D SHF LEFT Q <-- SHF RIGHT
WX_D_DSR.SQL	WMXU,D <-- D SHF RIGHT Q <-- SHF LEFT
WX_D_DSR.SQR	WMXU,D <-- D SHF RIGHT Q <-- SHF RIGHT
WB_LOOPF	WB<31:30> <-- 0'LOOP FLAG
WB_LOOPF.Q_0	WB<31:30> <-- 0'LOOP FLAG Q<-- 0
WB_LOOPF.D_0	WB<31:30> <-- 0'LOOP FLAG D<-- 0
WB_LOOPF.Q_D_0	WB<31:30> <-- 0'LOOP FLAG Q&D <-- 0
WB_ALUF	WB<31:30> <-- ALUSO'ALKC
WB_ALUF.Q_S	WB<31:30> <-- ALUSO'ALKC Q <-- S
WB_ALUF.D_S	WB<31:30> <-- ALUSO'ALKC D <-- S
WB_ALUF.Q_D_S	WB<31:30> <-- ALUSO'ALKC Q&D <-- S

MULFAST+	MULTIPLY +RBUS BY Q	(2 ITERATIONS PER CYCLE)
MULSLOW+	MULTIPLY +RBUS BY Q	(1 ITERATION PER CYCLE)
MULFAST-	MULTIPLY -RBUS BY Q	(2 ITERATIONS PER CYCLE)
MULSLOW-	MULTIPLY -RBUS BY Q	(1 ITERATION PER CYCLE)
DIVFAST+	DIVIDE Q BY +RBUS	(2 ITERATIONS PER CYCLE)
DIVSLOW+	DIVIDE Q BY +RBUS	(1 ITERATION PER CYCLE)
DIVFAST-	DIVIDE Q BY -RBUS	(2 ITERATIONS PER CYCLE)
DIVSLOW-	DIVIDE Q BY -RBUS	(1 ITERATION PER CYCLE)
REM	UNSHIFT REMAINDER	(RBUS MUST BE 0)
DIVDA	DIVIDE DOUBLE ADD	
DIVDS	DIVIDE DOUBLE SUB	

CHAPTER 4. THE DATA PATH, PART II: THE SUPER ROTATOR AND THE SCRATCH PAD REGISTERS

This chapter continues the description of COMET's Data Path with discussions of the Super Rotator and the Scratch Pad registers. The efficient bit manipulation capability of the Super Rotator and the easy accessibility of the Scratch Pad registers make these features very useful to the user microprogrammer.

4.1 The Super Rotator

The Super Rotator consists of two powerful combinational logic circuits and the six bit POSITION and SIZE latches. The purpose of the Super Rotator is to generate two outputs: a 32 bit data element and a two bit status code (SRKSTA <1:0>). Figure 4.1 is an overall block diagram of the Super Rotator.

Inputs to the Super Rotator are obtained from the three microarchitecture buses (MBUS, RBUS, and WBUS) and from the DSIZE latches. In addition immediate input data is available from the LITRL field of the current microinstruction. The 32 bit data output is applied to the B input of the ALU (cf. Section 3.1). The two bits of status information is applied to the microsequencer for use as a four-way branch (cf. Section 2.1). The Super Rotator is controlled by the ROT field (Bits <63:58>) of the current microinstruction.

As will be seen in the examples of tis section, the primary usefulness of the Super Rotator comes from the fact that the large combinational circuits provide a great deal of bit-manipulation capability at a much faster speed than could be done in microcode.

4.1.1 32 Bit Data Output

There are 64 ways in which the Super Rotator can produce its 32 bit data output, one for each of its 64 ROT micro-orders. They are listed in Table 4.1. Several are explained below, along with examples.*

---

* In each of the examples of this section, MBUS=31323334 (hex), RBUS=35363738 (hex), POSITION latch = 22 (decimal), SIZE latch = 17 (decimal), DSIZE latches = 10(Binary), and LITRL = 032 (hex).

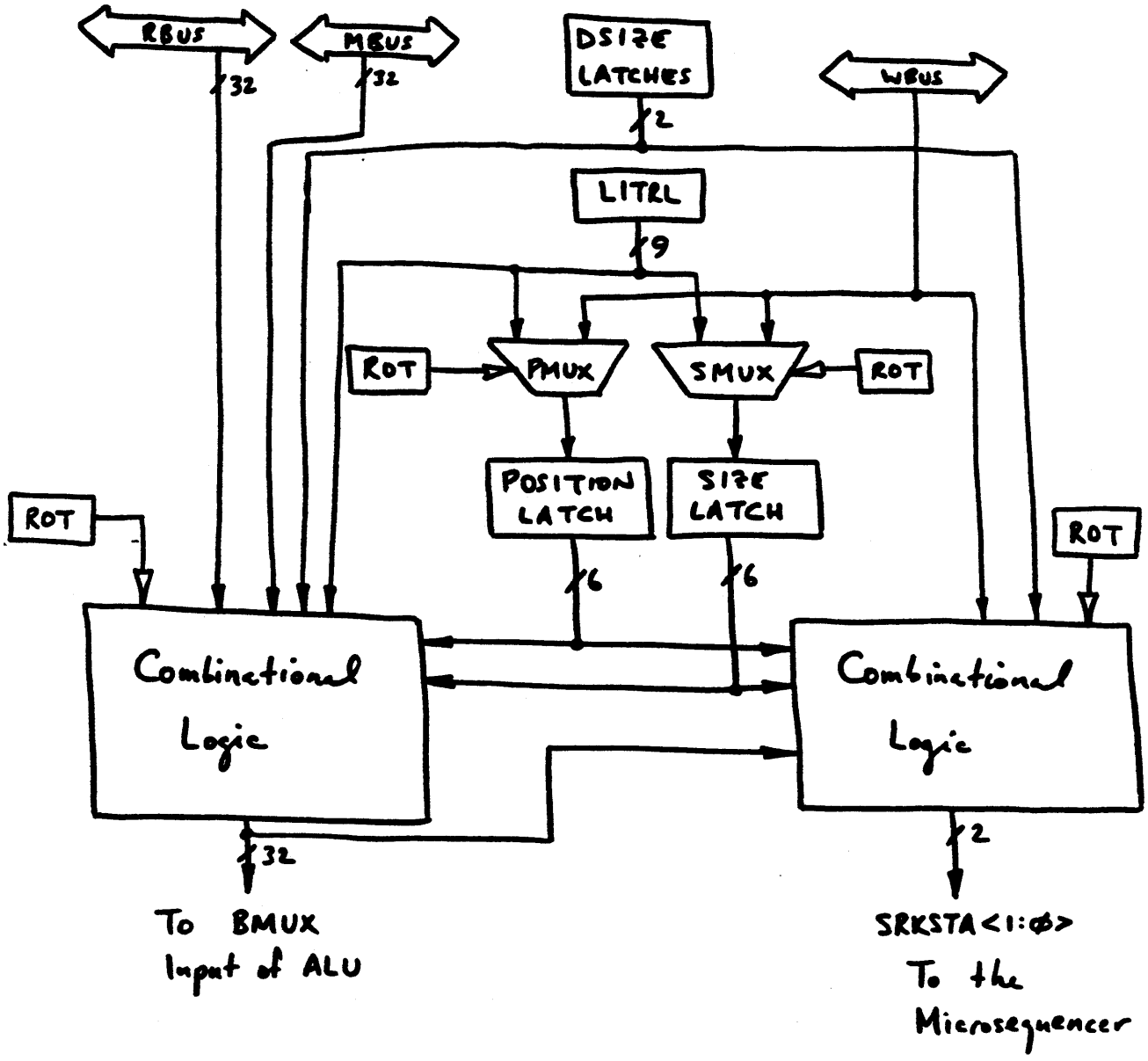
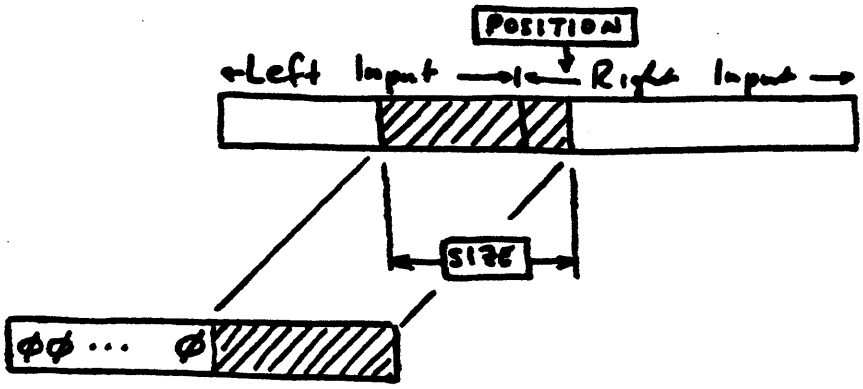


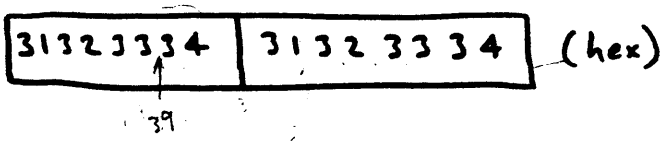
Figure 4.1 Super Rotator - Overall Block Diagram

(1) Extract and zero extend. The two 32 bit inputs specified by the ROT code are concatenated, forming a 64 bit element. From this, s bits (the SIZE), starting at bit p (the POSITION) are extracted. The 32 bit output is formed by adding 32-s high order 0's. The general mechanism is shown below:



Specific cases are shown in Examples 4.1 and 4.2.

Example 4.1. If ROT/XZ.MM is specified, the Super Rotator concatenates M'M (recall from the footnote on page 4-1 that the MBUS contains the hex number 31323334)

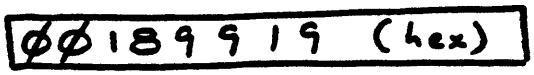


extracts

and outputs

As is the case in many of the ROT codes, the SIZE latch specifies the number of bits to be extracted, and the POSITION latch specifies the position of the low order bit.

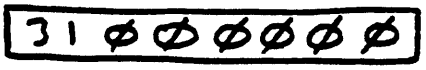
Example 4.2 If ROT/XZ.VPN is specified, the Super Rotator outputs



Note that in example 4.1, the size and position of the fields to be extracted are specified by the SIZE and POSITION latches respectively. In example 4.2, the size and position are constants specified by the ROT/XZ.VPN code; i.e., size is 21, position is 09. The ROT code specifies the number of bits to be extracted (or shifted or rotated) and the position of the low-order bit. The ROT code can specify these numbers as constants, as in ROT/XZ.VPN, or as quantities to be evaluated, as in ROT/XZ.MM.

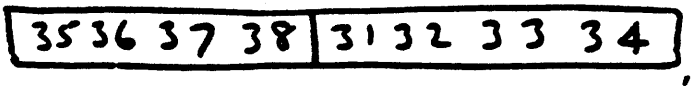
(2) Clear bytes. The MBUS is used as the input, the specified number of low order bytes are cleared, and the result is output.

Example 4.3. If ROT/CLR3BM is specified, the Super Rotator outputs

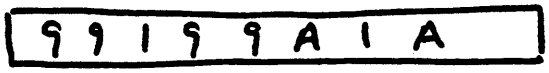


(3) Rotate. The two 32 bit inputs specified by the ROT code are concatenated, forming a 64 bit element. The result is rotated (i.e., shifted end-around) the specified number of bits, and the low order 32 bits are output.

Example 4.4. If ROT/RL.RM.PS is specified, the Super Rotator concatenates R'M

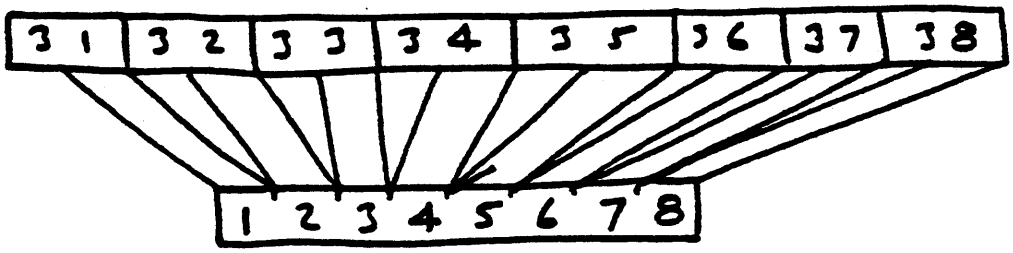


rotates left seven bits - since (22+17) mod 32 = 7, and outputs

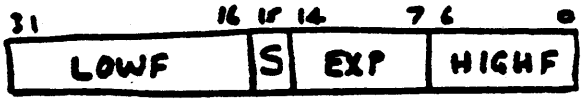


(4) Convert Numeric to Packed. The VAX-11 architecture provides a numeric data type where each decimal digit is stored in one byte, and a packed data type where two decimal digits are stored in one byte. The Super Rotator provides the capability for converting data from one type to the other.

Example 4.5. The ROT/CVTNP code takes the 8 bytes specified by M'R and produces the 32 bit output shown



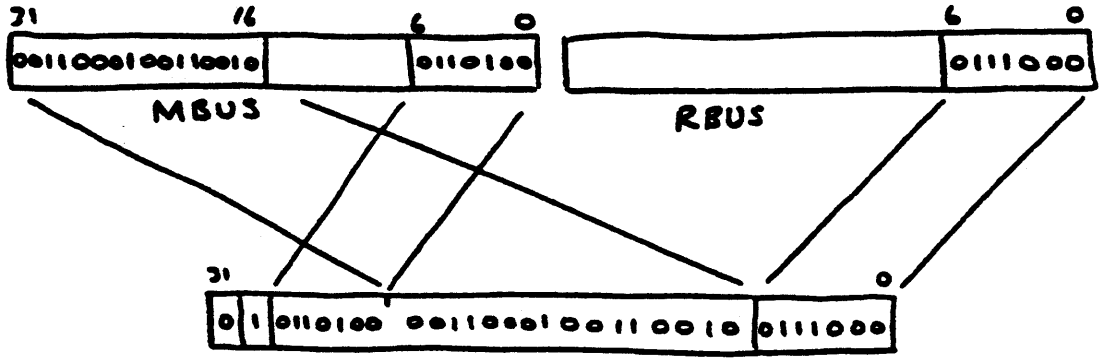
(5) Pack and Unpack Floating Point Fraction. The VAX-11 architecture stores a floating point number in four bytes, as follows:



where S = the sign bit, EXP = the exponent in excess-120 code, HIGHF = the high order bits of the fractional part, LOWF = the low bits of the fractional part. The fractional part consists of 24 bits. The redundant most significant bit is not stored. The next 7 bits, in decreasing order of significance, are stored in bits 6 through 0. The next 16 bits, in decreasing order of significance, are stored in bits 31 through 16.

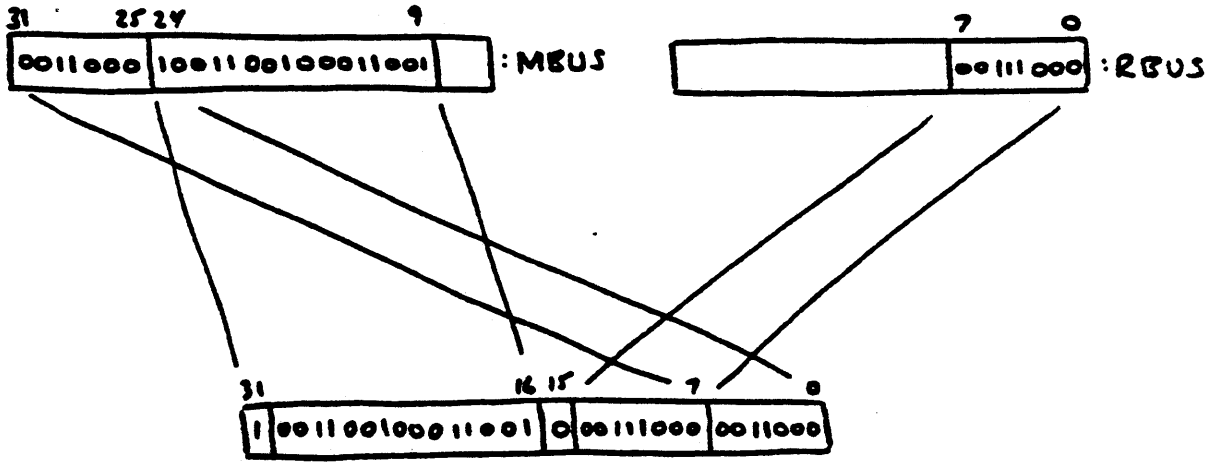
The Super Rotator provides the capability for recombing the fractional part in a more useable form.

Example 4.6 If ROT/GETFPF is specified, the Super Rotator takes the data on the MBUS and RBUS and produces the 32 bit output shown below:



Note that the redundant most significant bit is now present (Bit <30> of the output), and that the 24 bit fraction is combined in its useable form (Bits <30:7>). Note also that the low order bits (<6:0>) contain the exponent, no longer in excess-code.

Example 4.7. To recombine the floating point number into its VAX-11 floating data type, ROT/FPACK is used. The Super Rotator takes the data on the MBUS and RBUS and produces the 32 bit output shown below



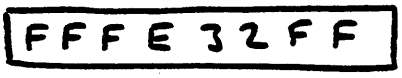
Note that before ROT/FPACK can be used, the fractional part (on

the MBUS) first must be shifted left until the most significant bit is shifted out, and that the exponent (on the RBUS) first must be expressed in excess-128 code.

(6) Constants. The ROT field can specify that the 32 bit output be one of the following constants: 0, -1, 1, 2, 4, or 8. Specification of the constant 1, 2, 4, or 8 is by means of ROT/CONX.SIZE. Which one is output is determined by the state of the DSIZE latches, i.e., by 0, 1, 2, or 3 respectively.

(7) Literal. The ROT field, together with the LITRL field can specify a 32 bit field which is particularly useful for masking operations. The nine bit LITRL field is extended to 32 bits by 23 0's or 23 1's and then rotated a specified number of nibbles. The result is a 32 bit mask such that the nine-bit LITRL field is properly aligned to make the desired test, and the other 23 bits are all 0's or all 1's as is necessary for the test.

Example 4.8 If ROT/OLIT8 is specified, the Super Rotator produces



4.1.2 SRKSTA Status Bits.

The Super Rotator also produces a two bit code containing status information relating to the state of certain signals in the microarchitecture. Recall that these two bits SRKSTA <1> and SRKSTA <0> are applied to the microsequencer for use as a four-way branch.

Table 4.2 shows the specification of the two status lines. Note that the ROT field has been relabeled ROTSRK, and the 64 micro-orders have been relabeled with more relevant mnemonics, as well.* The specification of SRKSTA <1:0> for absolute value check, ASCII sign check, and for WBUS range check will be described below. The other micro-orders are more straightforward, as shown by examples 4.9 and 4.10.

Example 4.9. ROTSRK/DSIZE.020 specifies that the SRKSTA <1:0> will be formed as follows:

SRKSTA<1> = 1 iff DSIZE <1> = 1
SRKSTA<0> = 1 iff DSIZE <0> = 1

with the result that the SRKSTA <1:0> code conveys the status of the DSIZE latches.

Example 4.10. ROTSRK/PL.EQ.0.SIGN.120 specifies that the SRKSTA <1:0> code conveys the state of the POSITION latch, as follows:

SRKSTA<1:0>	POSITION LATCH
00	POSITION LATCH = 0
01	POSITION LATCH = 16
10	1≤ POSITION LATCH ≤ 15
11	POSITION LATCH > 16

---

* Recall from the introduction to Chapter 3 that a possible conflict can exist between the ROT field (Bits <63:58>) and the ALUSHF (Bits <62:60>) and ALUCI (Bits <59:58>) fields. In particular, if the ROT field specifies loading either POSITION or SIZE latch, or if MUX specifies the output of the Super Rotator as the B input to the ALU, the ALUSHF and ALUCI default to 0. However, if the ROT field is only concerned with SRKSTA<1:0>, then the ALUSHF and ALUCI are available. The relabeling of Bits <63:58> in Table 4.2 is intended as a convenience to the microprogrammer, allowing the selection of the appropriate ROTSRK micro-order to control both SRKSTA<1:0> and ALUSHF-ALUCI, as well.

(1) **Absolute Value Check.** The condition tested is the absolute value of the low order byte on the WBUS.

Example 4.11. ROTSRK/ABSVAL.163.D specifies that SRKSTA <1:0> will be formed as follows:

```
SRKSTA<1> = 1 iff WBUS<7> = 0
SRKSTA<0> = 1 iff the absolute value of WBUS<7:0>
            is greater than or equal to 32.
```

The SRKSTA<1:0> code conveys the following information:

SRKSTA<1:0>	WBUS<7:0>
00	-31 ≤ WBUS<7:0> ≤ -1
01	WBUS<7:0> ≤ -32
10	0 ≤ WBUS<7:0> ≤ 31
11	WBUS<7:0> ≥ 32

(2) **ASCII Sign Check.** The condition tested is whether or not the low order byte on the WBUS is an ASCII sign.

Example 4.12. ROTSRK/ASCIISIGN.050 specifies that SRKSTA <1:0> will be formed as follows:

```
SRKSTA<1> = 1 iff WBUS<7:0> .NE. (32,43,45)
SRKSTA<0> = 1 iff WBUS<7:0> .NE. 45
```

The SRKSTA<1:0> code conveys the following information:

SRKSTA<1:0>	WBUS<7:0>
00	ASCII "-"
01	ASCII "+" or "space"
10	not possible - machine error
11	not an ASCII sign

(3) **WBUS Range Check.** The condition tested is the unsigned value of the low order byte on the WBUS.

Example 4.13. ROTSRK/WBRANGE.131D specifies that SRKSTA <1:0> will be formed as follows:

SRKSTA<1> = 1 iff WBUS<7:0>, as an unsigned integer, is greater than 31.

SRKSTA<0> = 1 iff WBUS<7:0> .NE. (1,32)

The SRKSTA<1:0> code conveys the following information:

SRKSTA<1:0>	WBUS<7:0>
00	1 ≤ WBUS<7:0> ≤ 31
01	WBUS<7:0> = 0
10	WBUS<7:0> = 32
11	WBUS<7:0> > 32

4.2 The Scratch Pad Registers

A register file is a set of very fast access storage registers. Almost every microprogrammable computer has one. Efficient emulation requires that the host machine have one available both for the purpose of storing frequently used constants and intermediate results and for the purpose of identifying the target machine's processor registers. COMET is no exception. Its register file consists of 48 R Scratch Pad registers, 16 M Scratch Pad registers, and a Long Literal register.* All are 32 bits wide.

Two other structures are associated with the Scratch Pad registers, a four-bit RNUM register and a six-deep Register Back-Up Stack. RNUM is used for addressing both R and M Scratch Pad registers. The Register Back-Up Stack is used to restore the contents of the VAX general purpose registers (implemented with Scratch Pad registers) if it is necessary to undo the partial emulation of a VAX machine instruction in order to service an interrupt or exception. The state of RNUM or the state of the Register Back Up Stack is available as a two-bit status code (SPASTA<1:0>) for use by the microsequencer in performing a four-way branch.

---

* Actually this is not quite correct; there are really eight fewer registers. This is because eight (i.e., RSP[00] through RSP[07] and eight of the 16 M Scratch Pad registers (i.e., MSP[0] through MSP[7]) are actually the same eight registers, accessible to both MBUS and RBUS.

Addressing the Scratch Pad registers is controlled by the RSRC and MSRC fields of the current microinstruction. Writing is controlled by the SPW field. Figure 4.2 is an overall block diagram of the Scratch Pad registers.

4.2.1 Uses of the Registers

The 16 M Scratch Pad registers (MSP[0] through MSP[F]) are used as follows:

(1) MSP[0] - MSP[A] are general temporaries; i.e., they are available for storing intermediate results. The first eight of them are dual port registers. They can be referenced as M registers or as R registers; that is, MSP[0]=RSP[0], MSP[1]=RSP[1],...MSP[7]=RSP[7].

(2) MSP[B] and MSP[C] have been designated for storing special values. MSP[B] stores the error code which is needed in the subsequent processing of VAX memory faults and arithmetic traps (see chapter 5). MSP[C] stores the FPD pack routine offset which is used in the initiation of an interrupt or exception if it is necessary to suspend the emulation of a VAX instruction and if it is not possible to undo the processing which has already occurred. (See Section 5.1.3.3).

(3) MSP[D] is one of six temporary registers allocated to the memory management microcode.

(4) MSP[E] and MSP[F] have been designated as VAX internal processor registers. MSP[E] is the System Control Block Base register. MSP[F] is the Software Interrupt Summary register. Both are used in the servicing of interrupts (see Section 5.1).

The 48 R Scratch Pad registers (RSP[0] through RSP[2F]) are used as follows:

(1) RSP[0] through RSP[D] and RSP[1F] are general temporaries. Recall that the first eight of them are dual port registers and correspond to MSP[0] through MSP[7].

(2) RSP[E], RSP[F], RSP[26], RSP[27], and RSP[2F] are five of the six temporary registers specifically allocated to the memory management microcode. The remaining one is MSP[D].

(3) RSP[10] through RSP[1E] have been designated as VAX general purpose registers R0 through R14. VAX uses R13 as its frame pointer (FP) and R14 as its active stack pointer (SP).

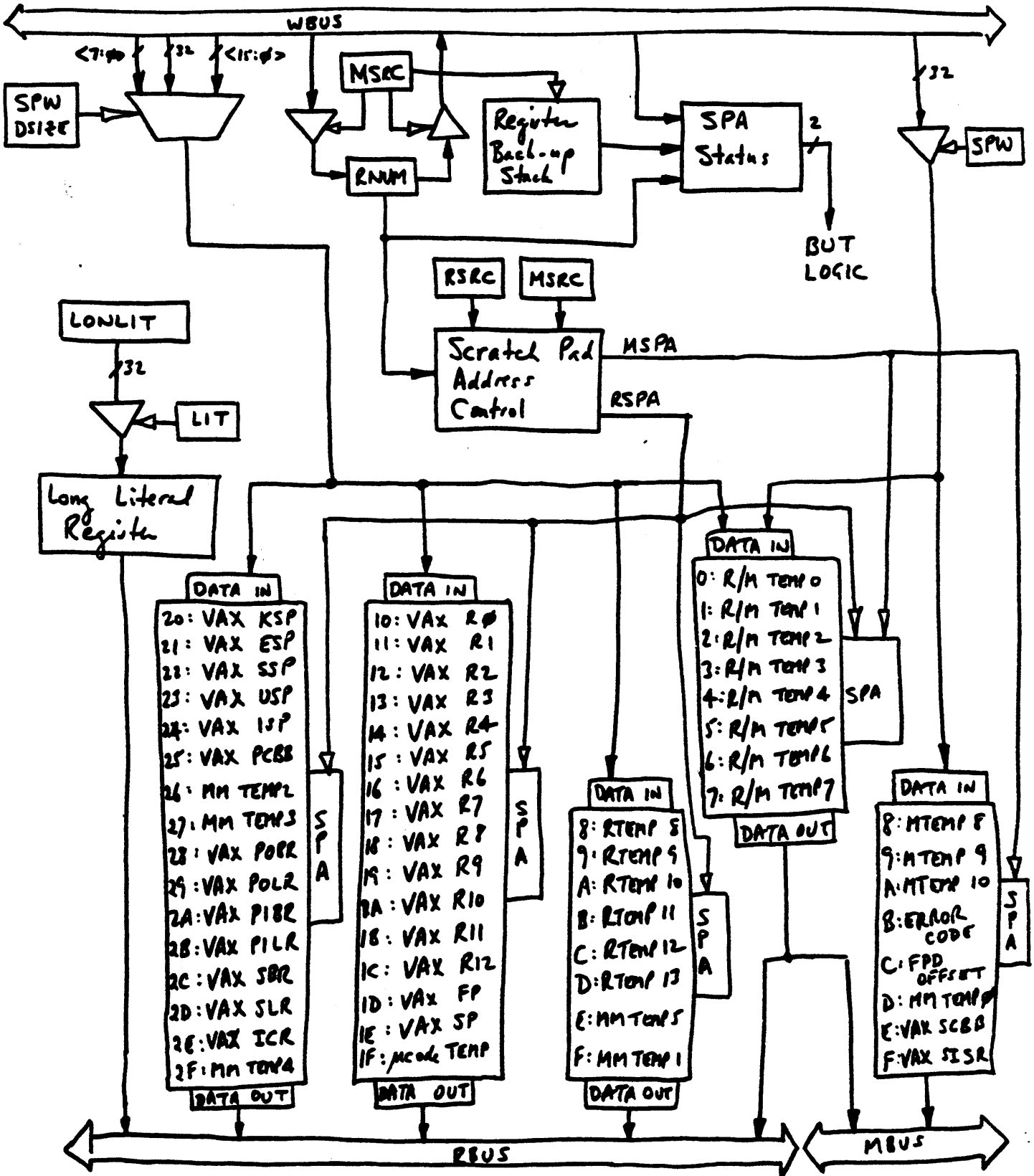


Figure 4.2 Scratch Pad Registers - Overall Block Diagram

(4) RSP[20] through RSP[25] and RSP[28] through RSP[2E] have been designated as VAX internal processor registers.

The Long Literal Register is used to store 32 bits of immediate data obtained from the LONLIT field of the current microinstruction. If LIT/LONLIT is specified, the Long Literal register is loaded with the contents of <62:31> of the current microinstruction.

Table 4.3 delineates the uses of the 48 R Scratch Pad and 16 M Scratch Pad registers.

4.2.2 Address Control

The Scratch Pad registers are addressed by the MSRC or RSRC field of the current microinstruction either directly, or in conjunction with the RNUM register.

Example 4.14 If MSRC/TEMP6 is specified, MSP[6], the dual port M Scratch Pad register TEMP6 is addressed. If MSRC/SCBB is specified, the System Control Block Base register is addressed. If MSRC/TEMP.R+1 is specified and if RNUM contains the value E(hex), then MSP[F], the Software Interrrupt Summary Register is addressed.

Example 4.15 If RSRC/DST.ROR1 is specified, the R Scratch Pad register addressed is determined as follows: If RNUM is odd, say it contains the value 2k+1, then RSP[2k+1] is addressed. If RNUM is even, say it contains the value 2k, then RSP[2k+1] is addressed. If RSRC/IP2.R is specified, the R Scratch Pad register addressed is determined as follows: For purposes of indexing on RNUM, RSP[20] through RSP[2F] are designated IPR[0] through IPR[F]. Therefore, if RNUM contains the value A, for example, then RSP[2A], the P1 Base Register, is the register being addressed. If RSRC/LONLIT is specified, the Long Literal Register is addressed.

4.2.3 Write Control

The SPW field of the current microinstruction controls writing into the R and M Scratch Pad registers. The address of the register to be written is determined as discussed in Section 4.2.2. The LIT field controls writing into the Long Literal Register, as described in Section 4.2.1.

Example 4.16. If SPW/NOP is specified, no writing into R or M Scratch Pad occurs. If SPW/RSIZE is specified, the low order 1 or 2 bytes or the entire 4 bytes of information on the WBUS is written into the corresponding field of the R Scratch Pad register specified by RSRC. The number of bytes written is determined by the DSIZE latches.

Certain MSRC and RSRC codes do not specify R or M Scratch Pad registers. For example, MSRC/TB specifies that the input to the MBUS is from the Translation Buffer. In those cases, the register written is TEMP0.

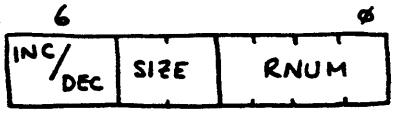
Finally, three RSRC micro-orders (RSRC/DST.R, RSRC/DST.R+1, and RSRC/DST.ROR1) provide for conditional writing, useful to the VAX emulation. For those micro-orders, if the operand specifier in the VAX machine instruction is register mode, the write occurs. If not, the write is inhibited. This construct is used in conjunction with BUS/WRITE.NOREG (i.e., write into memory unless register mode) to provide the capability to write a value into a destination operand in a single microinstruction, independent of whether the operand address is a memory location or a register. If the destination is a register (i.e., register mode), the value is written into the Scratch Pad register and the memory write is inhibited. If the destination is a memory location (i.e., not register mode), the value is written into memory and the Scratch Pad write is inhibited.

4.2.4 The Register Back Up Stack

In this section, the functionality of the Register Back Up Stack will be described. Its use in the processing of VAX interrupts and exceptions will be treated in Section 5.1.

The Register Back Up Stack stores certain information about the VAX general purpose registers which can be used to restore them to their original values if an interrupt or exception is to be taken and the VAX machine instruction is to be restarted from the beginning at a later time. In particular, during the evaluation of a VAX operand specifier, if the addressing mode is autoincrement or autodecrement, the register is updated. In order to return the register to its original value, we must save the register number (RNUM), the amount of the update (4*SIZE), and whether it was autoincremented or autodecremented (1 or 0).

The Register Back Up Stack consists of six registers, each 7 bits wide, as shown below:



Access is through a three-bit stack pointer RBSP. Reading and writing is controlled by the MSRC field. In addition, the Register Back Up Stack is always cleared by the BUT/IRD1 code since it is unnecessary to save its contents after the emulation of the current VAX machine instruction has been completed.

Four MSRC codes deal with the Register Back Up Stack. MSRC/PSHADD pushes RNUM, SIZE, and +1 onto the stack: i.e.,

RBS[RBSP] <--- RNUM'SIZE'+1

RBSP <--- RBSP + 1 .

MSRC/PUSHSUB pushes RNUM, SIZE, and 0 onto the stack. These two operations are performed during operand specifier evaluation. MSRC/READRBS pops the stack. This is done during the initiation of an interrupt or exception in order to restore the register to its original value. Finally, MSRC/WB_RBSP outputs the RBSP onto the WBUS.

4.2.5 SPASTA Status Bits

COMET provides a two-bit code containing status information about RNUM or the Register Back Up Stack, see Table 4.4. The code is available to the microsequencer for use (BUT/SPASTA) in performing a four-way branch. For example, if RSRC does not specify a VAX general purpose register, and if MSRC specifies that RNUM is to get the low order four bits of WBUS, then a three-way branch can be produced by BUT/SPASTA, based on the value on WBUS<3:0>. On the other hand, if RSRC does in fact specify a VAX general purpose register, and MSRC does not specify RNUM_WBUS or READRBS or WB_RBSP, then a four-way branch can be effected by BUT/SPASTA based on the value in RNUM.

TABLE 4.1 THE ROT MICRO-ORDERS

ROT	FUNCTION
XZ.MR	EXTRACT & ZERO EXTEND M'R, POS = PL, SIZE = SL
XZ.MM	EXTRACT & ZERO EXTEND M'M, POS = PL, SIZE = SL
XZ.RR	EXTRACT & ZERO EXTEND R'R, POS = PL, SIZE = SL
XZ.VPN	EXTRACT & ZERO EXTEND M'M, POS = 09, SIZE = 21
XZ.PTX	EXTRACT & ZERO EXTEND M'M, POS = 07, SIZE = 23
CLR1BM	CLR M<07:0>
CLR2BM	CLR M<15:0>
CLR3BM	CLR M<23:0>
RL.RM.P	ROT LEFT R'M, NO. BITS = PLATCH<4:0> (NOTE 1)
RL.RM.P	ROT LEFT R'M, NO. BITS = (PL+SL)<4:0> (NOTE 1)
RL.RM.4	ROT LEFT R'M, NO. BITS = 4
RL.MM.P	ROT LEFT M'M, NO. BITS = PLATCH
RL.MM.PTE	ROT LEFT M'M, NO. BITS = 9
RL.RR.P	ROT LEFT R'R, NO. BITS = PLATCH
RR.MR.P	ROT RIGHT M'R, NO. BITS = PLATCH<4:0>
RR.MR.PS	ROT RIGHT M'R, NO. BITS = (PL+SL)<4:0>
RR.MR.4	ROT RIGHT M'R, NO. BITS = 4
RR.MR.S	ROT RIGHT M'R, NO. BITS = SLATCH<4:0>
RR.MR.9	ROT RIGHT M'R, NO. BITS = 9
RR.MM.P	ROT RIGHT M'M, NO. BITS = PLATCH
RR.MM.PS	ROT RIGHT M'M, NO. BITS = PLATCH + SLATCH
RR.MM.SIZ	ROT RIGHT M'M, NO. BITS = 8,16,24,0
RR.RR.P	ROT RIGHT R'R, NO. BITS = PLATCH
RR.RR.PS	ROT RIGHT R'R, NO. BITS = PLATCH + SLATCH
RR.RR.SIZ	ROT RIGHT R'R, NO. BITS = 8,16,24,0
ASL.R.P	ARITH SHF LEFT R, NO. BITS = PLATCH (NOTE 2)
ASL.R.SIZ	ARITH SHF LEFT R, NO. BITS = 0,1,2,3
ASL.R.7	ARITH SHF LEFT R, NO. BITS = 7
ASL.M.P	ARITH SHF LEFT M, NO. BITS = PLATCH (NOTE 2)
ASR.M.P	ARITH SHF RIGHT M, NO. BITS = PLATCH
ASR.M.-P	ARITH SHF RIGHT M, NO. BITS = -PLACTCH
ASR.M.3	ARITH SHF RIGHT M, NO. BITS = 3
GETNIB	GET LEAST SIGNIFICANT NIBBLE FROM MBUS
BCDSWP	BCD SWAP, MBUS
CVTPN	CONVERT PACKED TO NUMERIC, 4NIB TO 4BYTE, MBUS RBUS MUST = 3XX33 (HEX)
CVTNP	CONVERT NUMERIC TO PACKED, 8BYTE TO 8NIB, M'R

```
PL_MSS    FIND MOST SIGNIFICANT BIT SET MBUS, WBUS
GETEXP    EXTRACT & ZERO EXTEND M'M POS = 7, SIZE = 8
GETFPF    UNPACK FLOATING POINT FRACTION, M'R
FPLIT     EXPAND FLOATING POINT LITERAL, MBUS
FPACK     S ROT<31:16,15,14:7,6:0> <-
     MB<24:9>,0,RB<7:0>,MB<31:25>


PL  SUP ROT <- PLATCH
SL  SUP ROT <- SLATCH
SL,PL_WB S ROT <- SLATCH . PLATCH <- WB<5:0>
OLITO.PL43_WB S ROT <- OLITO . PL<4:3> <-WB<1:0>
OLITO.PL_LIT  S ROT <-OLITO . PLATCH <- SHORT LITERAL
PL.SL_WB S ROT <- PLATCH . SLATCH <- WB<5:0>
OLITO.SL_LIT  S ROT <-OLITO . SLATCH <- SHORT LITERAL


ZERO      CONSTANT 0
MINUS1    CONSTANT -1
CONX,SIZ  CONSTANT 1,2,4,8 DEPENDNG ON SIZE (-(R)+)
ZLITO     0 EXTEND LITERAL & ROT LEFT 00 BITS
ZLIT4     0 EXTEND LITERAL & ROT LEFT 04 BITS
ZLIT8     0 EXTEND LITERAL & ROT LEFT 08 BITS
ZLIT12    0 EXTEND LITERAL & ROT LEFT 12 BITS
ZLIT16    0 EXTEND LITERAL & ROT LEFT 16 BITS
ZLIT20    0 EXTEND LITERAL & ROT LEFT 20 BITS
ZLIT24    0 EXTEND LITERAL & ROT LEFT 24 BITS
ZLIT28    0 EXTEND LITERAL & ROT LEFT 28 BITS
ZLITPL    0 EXTEND LITERAL & ROT LEFT PL BITS
OLITO     1 EXTEND LITERAL & ROT LEFT 00 BITS
OLIT8     1 EXTEND LITERAL & ROT LEFT 08 BITS
OLIT16    1 EXTEND LITERAL & ROT LEFT 16 BITS
OLIT24    1 EXTEND LITERAL & ROT LEFT 24 BITS
```

Table 4.2 THE ROTSRK MICRO-ORDERS

ROTSRK	SRKSTA<1>	SRKSTA<0>	ALUSHF	ALUCI
ABSVAL.163.D	ABS VAL CHECK		ZERO	ZERO
ABSVAL.171.D	ABS VAL CHECK		ZERO	ZERO
ABSVAL.173.D	ABS VAL CHECK		ZERO	ZERO
ABSVAL.140	ABS VAL CHECK		ALUO.Q1	ZERO
ABSVAL.141	ABS VAL CHECK		ALUO.Q1	ALKC
ABSVAL.142	ABS VAL CHECK		ALUO.Q1	ONE
ABSVAL.143	ABS VAL CHECK		ALUO.Q1	PSLC
ABSVAL.150	ABS VAL CHECK		ALU1.Q0	ZERO
ABSVAL.151	ABS VAL CHECK		ALU1.Q0	ALKC
ABSVAL.152	ABS VAL CHECK		ALU1.Q0	ONE
ABSVAL.153	ABS VAL CHECK		ALU1.Q0	PSLC
ABSVAL.160	ABS VAL CHECK		WBUS30	ZERO
ABSVAL.161	ABS VAL CHECK		WBUS 30	ALKC
ABSVAL.162	ABS VAL CHECK		WBUS 30	ONE
ABSVAL.170	ABS VAL CHECK		PSLC	ZERO
ABSVAL.172	ABS VAL CHECK		PSLC	ONE
ASCIISIGN.050	ASCII SIGN CHECK		ALU1.Q0	ZERO
ASCIISIGN.051	ASCII SIGN CHECK		ALU1.Q0	ALKC
ASCIISIGN.052	ASCII SIGN CHECK		ALU1.Q0	ONE
rASCIISIGN.053	ASCII SIGN CHECK		ALU1.Q0	PSLC
ASCIISIGN.070	ASCII SIGN CHECK		PSLC	ZERO
ASCIISIGN.071	ASCII SIGN CHECK		PSLC	ALKC
ASCIISIGN.072	ASCII SIGN CHECK		PSLC	ONE
ASCIISIGN.073	ASCII SIGN CHECK		PSLC	PSLC
DSIZE.020	DSIZE<1>	DSIZE<0>	SHF	ZERO
DSIZE.021	DSIZE<1>	DSIZE<0>	SHF	ALKC
DSIZE.022	DSIZE<1>	DSIZE<0>	SHF	ONE
DSIZE.023	DSIZE<1>	DSIZE<0>	SHF	PSLC
DSIZE.030	DSIZE<1>	DSIZE<0>	ROT	ZERO
DSIZE.031	DSIZE<1>	DSIZE<0>	ROT	ALKC
DSIZE.032	DSIZE<1>	DSIZE<0>	ROT	ONE
DSIZE.033	DSIZE<1>	DSIZE<0>	SHF	ONE
PL.EQ.0.SIGN.120	PL<4:0>.EQ.0	PL<5>	SHF	ZERO
PL.EQ.0.SIGN.121	PL<4:0>.EQ.0	PL<5>	SHF	ALKC
PL.EQ.0.SIGN.122	PL<4:0>.EQ.0	PL<5>	SHF	ONE
PL.EQ.0.123=2B	PL<4:0>.EQ.0	0	SHF	PSLC

BCDSIGN.040	S<3:0>.NE.0	S<3:0>.NE.(11,13)	ALUO.Q1	ZERO
BCDSIGN.041	S<3:0>.NE.0	S<3:0>.NE.(11,13)	ALUO.Q1	ALKC
BCDSIGN.042	S<3:0>.NE.0	S<3:0>.NE.(11,13)	ALUO.Q1	ONE
BCDSIGN.043	S<3:0>.NE.0	S<3:0>.NE.(11,13)	ALUO.Q1	PSLC
BCDSIGN.060	S<3:0>.NE.0	S<3:0>.NE.(11,13)	WBUS30	ZERO
BCDSIGN.061	S<3:0>.NE.0	S<3:0>.NE.(11,13)	WBUS30	ALKC
BCDSIGN.062	S<3:0>.NE.0	S<3:0>.NE.(11,13)	WBUS30	ONE
BCDSIGN.063	S<3:0>.NE.0	S<3:0>.NE.(11,13)	WBUS30	PSLC

VIELD.000	SL.EQ.0	(PL<4:0>+SL).GT.32	ZERO	ZERO
VIELD.001	SL.EQ.0	(PL<4:0>+SL).GT.32	ZERO	ALKC
VIELD.002	SL.EQ.0	(PL<4:0>+SL).GT.32	ZERO	ONE
VIELD.010	SL.EQ.0	(PL<4:0>+SL).GT.32	ONE	ZERO
VIELD.011	SL.EQ.0	(PL<4:0>+SL).GT.32	ONE	ALKC
VIELD.012	SL.EQ.0	(PL<4:0>+SL).GT.32	ONE	ONE
VIELD.110	SL.EQ.0	(PL<4:0>+SL).GT.32	ONE	ZERO
VIELD.111	SL.EQ.0	(PL<4:0>+SL).GT.32	ONE	ALKC
VIELD.112	SL.EQ.0	(PL<4:0>+SL).GT.32	ONE	ONE

SL.EQ.0.SIGN.101	SL.EQ.0	PL<5>	ZERO	ALKC
SL.EQ.0.SIGN.102	SL.EQ.0	PL<5>	ZERO	ONE
SL.EQ.0.100	SL.EQ.0	UNDEFINED	ZERO	ZERO

WBRANGE.131.D	WBUS RANGE CHECK		ZERO	ZERO
WBRANGE.133.D	WBUS RANGE CHECK		ZERO	ZERO
WBRANGE.130	WBUS RANGE CHECK		ROT	ZERO
WBRANGE.132	WBUS RANGE CHECK		ROT	ONE

WX.NE.0113.D	WX<31:16>.NE.0	WX<15:0>.NE.0	ZERO	ZERO
WX.NE.0.103	WX<31:16>.NE.0	WX<15:0>.NE.0	ZERO	PSLC

PL5.003 0	PL<5>	ZERO	PSLC
PL5.013 0	PL<5>	ONE	PSLC

Table 4.3 USES OF THE SCRATCH PAD REGISTERS

RSP[0] - RSP [7]	General temporary registers (RTEMP0-RTEMP7) Dual Port; i.e., also MSP[0] - MSP[7].
RSP[8] - RSP [D]	General temporary registers (RTEMP8-RTEMP13)
RSP[10] - RSP [1E]	VAX general purpose registers (R0-R12,FP,SP)
RSP [1F]	Microcode temporary
RSP[20]	VAX internal processor register (KERNEL STACK POINTER)
RSP[21]	VAX internal processor register (EXECUTIVE STACK POINTER)
RSP[22]	VAX internal processor register (SUPERVISOR STACK POINTER)
RSP[23]	VAX internal processor register (USER STACK POINTER)
RSP[24]	VAX internal processor register (INTERRUPT STACK POINTER)
RSP[25]	VAX Internal processor register (PROCESS CONTROL BLOCK BASE)
RSP[26]	Memory Management temporary register (MMTEMP 2)
RSP[27]	Memory Management temporary register (MMTEMP 3)
RSP[28]	VAX internal processor register (P0 BASE REGISTER)
RSP[29]	VAX internal processor register (P0 LENGTH REGISTER)
RSP[2A]	VAX internal processor register (P1 BASE REGISTER)
RSP[2B]	VAX internal processor register (P1 LENGTH REGISTER)
RSP[2C]	VAX internal processor register (SYSTEM BASE REGISTER)
RSP[2D]	VAX internal processor register (SYSTEM LENGTH REGISTER)

RSP[2E]	VAX internal processor register (NEXT INTERVAL REGISTER)
RSP[2F]	Memory Management temporary register (MMTEMP 4)
MSP[0]-MSP[7]	General temporary register (MTEMP0-MTEMP7). Dual port; i.e., also RSP[0]-RSP[7].
MSP[8]-MSP[A]	General temporary registers (MTEMP8-MTEMP10)
MSP[B]	Error code for Memory faults and Arithmetic traps
MSP[C]	FPD Pack Routine Offset
MSP[D]	Memory Management temporary register (MMTEMP0)
MSP[E]	VAX internal processor register (SYSTEM CONTROL BLOCK BASE)
MSP[F]	VAX internal processor register (SOFTWARE INTERRUPT SUMMARY REGISTER).

Table 4.4 SPECIFICATION OF SPASTA STATUS BITS

CONDITION					SPASTA <1:0>
RSC	**MSRC**	**WBUS<3:0>**	**RNUM**	**RBUS**	
not GPR*	RNUM-WBUS	[8,13]**	-	-	00
not GPR*	RNUM-WBUS	[5,7], 14 or 15	-	-	10
not GPR*	RNUM-WBUS	[0,4]	-	-	11
not GPR*	READRBS	-	-	Bit<6>=1	01
not GPR*	READRBS	-	-	Bit<6>=0	00
not GPR*	WB_RBSP	-	-	RBSP=0	01
not GPR*	WB_RBSP	-	-	RBSP>0	00
not GPR*	other***	-	-	-	00
GPR*	RNUM_WBUS	-	-	-	Undef
GPR*	READRBS	-	-	-	Undef
GPR*	WB_RBSP	-	-	-	Undef
GPR*	other***	-	[0,5], [8,13], or 15	-	00
GPR*	other***	-	14	-	01
GPR*	other***	-	7	-	10
GPR*	other***	-	6	-	11

---

* not GPR: RSRC does not specify a VAX general purpose register.

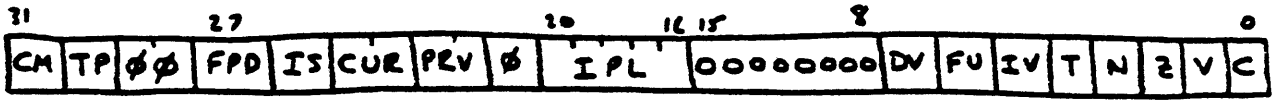
** [8,13]: 8,9,10,11,12, or 13

*** other: MSRC does not specify RNUM_WBUS,READRBS, or WB_RBSP

CHAPTER 5. COMET IMPLEMENTATION OF VAX'S SYSTEM ARCHITECTURE

This chapter describes the COMET implementation of the two major components of VAX's system architecture the interrupt and exception handling mechanism and the hardware memory management facility. Other parts of the system architecture (for example, the notion of process structure, the internal processor registers, and the five privileged machine instructions) are not treated explicitly here since their implemenation is straightforward and requires no additional understanding of COMET.

One element of the VAX process structure must be mentioned, however, because it is used extensively to implement the system architecture features. That is the PSL (processor status longword). Each process has one; it contains important status and control information about the process. It is shown below without explanation; its fields will be identified as they are needed in the various sections of this chapter.



The exception and interrupt handling mechanism and the memory management facility are controlled mainly by the BUS and WCTRL fields. The BUS field is used to intitiate bus cycles, i.e., reads and writes to memory. The virtual address is placed in the VA register. On a read, the value read ends up in the MDR register; on a write, the value to be written is loaded into the WDR register. VA, MDR, and WDR are all COMET accessible registers. The WCTRL field is used to pass control information between the WBUS and several COMET registers which are important to the handling of interrupts and exceptions. Figure 5.1 shows the WBUS and the relevant COMET registers.

5.1 Interrupts and Exceptions

5.1.1 VAX Interrupts and Exceptions

During the execution of a process, it is often the case that an event occurs which requires the normal flow of execution to be suspended in order to execute another piece of software. Sometimes the cause of this event is external to and independent of the executing process. One example is the detection of a memory parity error. Such an event we call an interrupt. Other times the event is caused by the executing process itself. An example of this is the reserved addressing mode fault, which is caused by a VAX instruction attempting to use an addressing mode in a way that is not allowed (e.g., use of immediate mode to specify a destination operand). Such an event we call an exception.

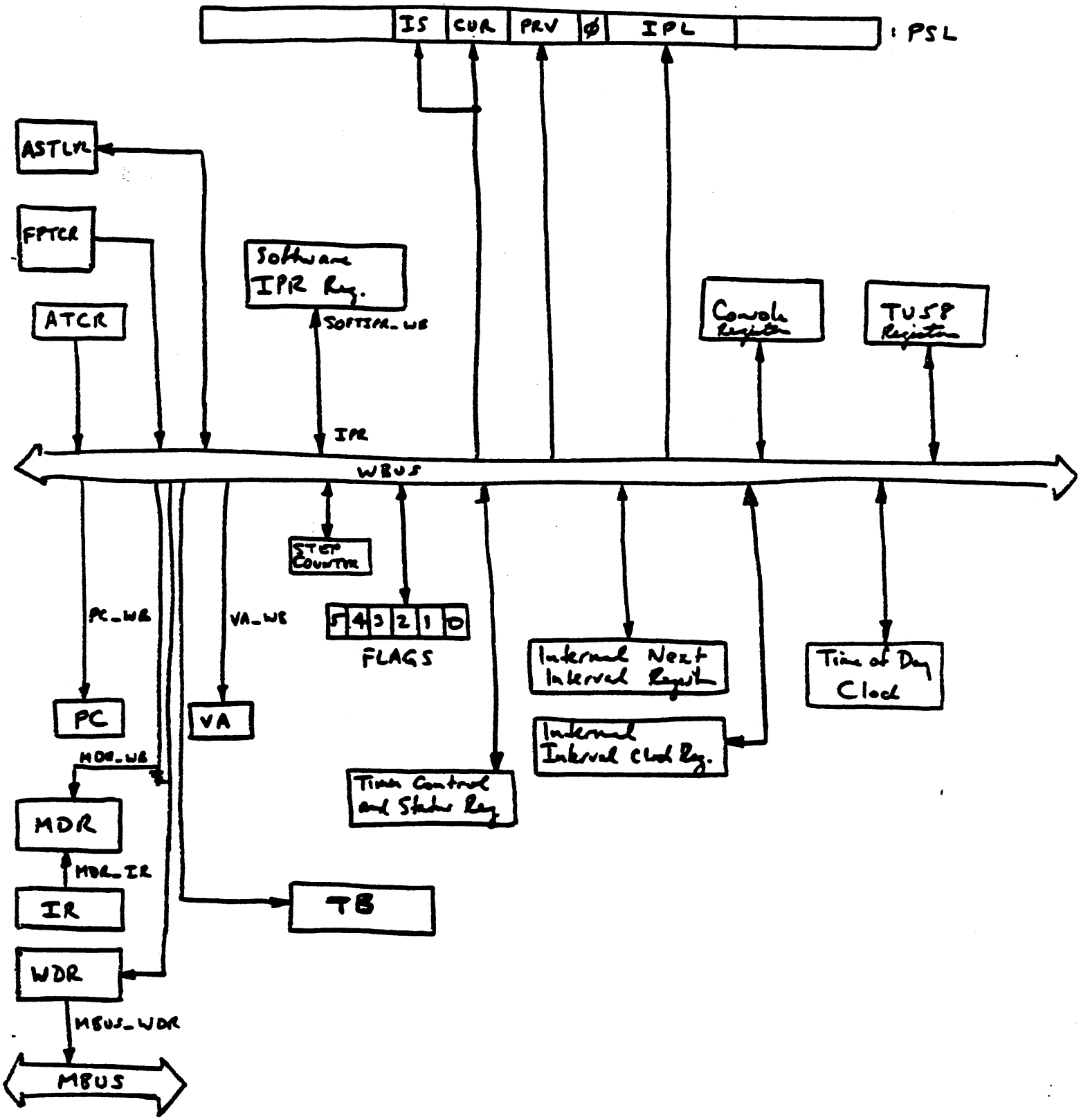


Figure 5.1 WBUS and COMET registers

In both cases, the currently executing process must suspend execution and transfer control to a routine which services the interrupt or exception. In the case of an exception, this transfer of control usually occurs at the time the exception is detected. For example, the reserved addressing mode fault described above would occur at the time the processor attempted to compute the operand address. In the case of an interrupt, however, this transfer of control can occur only at pre-specified points in the execution of the VAX program; either between the execution of VAX machine instructions, or in the case of certain time-consuming instruction executions, at specific well-defined points within the execution of an instruction. Furthermore, this transfer of control can occur only if the urgency (i.e., priority) of the interrupting event is greater than the urgency (i.e., priority) of the executing process.

Associated with each process is its IPL (interrupt priority level) which is a measure of its degree of urgency. There are 32 priority levels, ranging from IPL 00 to IPL 1F (hex). For example, user programs usually execute at IPL 00; power failure interrupts at IPL 1E. The IPL of a process is stored in the IPL field of its PSL (processor status longword). Interrupts execute at an IPL which has been specifically designated for that interrupt. Exceptions (generally, although not always, as will be described momentarily) execute at the same IPL as the process which caused the exception.

The VAX System Reference Manual identifies eight types of interrupts (page 6-8) and six classes of exceptions (page 6-13). Table 5.1 lists the interrupts, their IPL's, the corresponding COMET equivalent, and the CSA for each.* One of the interrupts, AST delivery, requires special mention. AST's (asynchronous system traps) represent a way for notifying a process than an event which is relevant to the process, but not synchronized with it, has occured. This notification takes the form of a formal procedure. It is called an AST service routine. It is specified by the process at the time the AST is requested. If the event has occurred, the notification is said to be a pending AST. It will cause an interrupt at IPL2 if the process to be notified is currently executing and if the event is associated with an access mode which is at least as privileged as the current access mode of the process. This determination is made during execution of an REI instruction. The IPL2 interrupt initiates the AST Delivery routine which subsequently passes control to the AST service routine.

---

* The use of the CSA is described in Section 5.1.3.

Table 5.2 lists the classes of exceptions, along with the COMET mechanism for detecting each. Section 5.1.3 describes the COMET mechanisms. Section 5.2 discusses the memory management exceptions. We should also point out that although most of the exceptions execute at the same IPL as the process causing the exception, two do not: Kernel Stack Not Valid (KSNV) and Machine Check. (Many conditions can cause a machine check, among them a bus error, TB error, and Control Store parity error). Due to the serious nature of KSNV and Machine Check, these exceptions are serviced at IPL 1F in order to lock out all other processing until they are handled.

5.1.2 VAX I/E Handling Mechanism

VAX interrupts and exceptions are serviced in the following way. Once it has been determined that an interrupt or exception should be initiated, the PC and PSL of the executing process are pushed onto the appropriate stack (either kernel or interrupt stack - we will discuss which, momentarily), a new PSL is constructed for the service routine, any parameters needed by the service routine are pushed onto the stack, and control is transferred to the starting address of the service routine for that particular interrupt or exception. The last VAX machine instruction in a service routine is REI, Return from Exception or Interrupt. Execution of REI causes several things to happen. The old PC and PSL are popped from the stack. If there are no ASTs pending and no higher priority interrupts pending, then the interrupted process can resume execution at the address specified by PC. A test for pending ASTs is made by the REI instruction by comparing the Current Mode field of the popped PSL with the contents of the ASTLVL register.

The starting address of the service routine and information needed for the decision as to whether to process the interrupt or exception from the kernel stack or from the interrupt stack are contained in the System Control Block. The System Control Block consists of one page (128 contiguous longwords) of physical memory. Its physical base address is contained in the SCBB (System Control Block Base), an internal processor register. Each longword in the System Control Block corresponds to exactly one interrupt or exception. Bits <31:2>'00 form the virtual starting address of the service routine for that interrupt or exception. Bits <1:0> specify that the event should be serviced on the interrupt stack (if 01), or on the kernel stack unless the process is already running on the interrupt stack (if 00), or in writable control store if such exists (if 10). If the event is to be serviced in writable control store, control is passed to the microcode starting at CSA 2001 (hex).

5.1.3 COMET Implementation

One can consider the COMET implementation of the interrupts and exception handling mechanism as two microcoded routines, one for initiating an exception or interrupt and one for returning from an exception or interrupt. The return routine is straightforward. It is simply the emulation of the VAX REI instruction. The initiation routine, however, is more complicated. The VAX System Reference Manual does describe the routine "initiate interrupt or exception". But there is no corresponding VAX opcode which will transfer control as the result of an IRD1 ROM decode. On the contrary, the need to execute the "initiate" routine can be detected in one of several ways. The entry point to the microcode and the specific tasks which must be performed depend on the particular interrupt or exception which is detected. The various entry points and tasks will be discussed below. In all cases, however, like the emulation of any other VAX instruction, the "initiate" microcode terminates with BUT/IRD1 in the last microinstruction (Fetch the next instruction!). In this case, the next instruction is the first machine instruction of the VAX service routine.

5.1.3.1 Detection and Branching

The detection of an interrupt or exception and the resulting branch to the appropriate microcode can occur as a result of a microtrap, a microbranch, DOSERVICE, or a ROM decode.

Microtraps. A microtrap is effectively a fault to the microcode. It is caused by the hardware upon detection of a condition which would not allow the current microinstruction to complete execution successfully. The hardware forces the control store address to a fixed location depending on the particular condition, overriding the address specified by the BUT field of the current microinstruction. This location is the starting address for the microcode to initiate that particular interrupt or exception. In general, the current microinstruction is prevented from writing to any destination. The CSA of the current microinstruction is pushed on the microstack for re-execution if the condition causing the microtrap is corrected. Microtraps are used extensively by the memory management system, as is described in Section 5.2. They are also caused by serious system faults (machine checks) such as control store parity error and bus errors, for example. The DOSERVICE routine described below is a special case of the microtrap mechanism.

DOSERVICE. DOSERVICE is a hardware routine invoked by the presence of the BUT/IRD1 micro-order. It is used to test for traps and interrupts after completing the emulation of each VAX machine instruction. If a trap or interrupt is present, the hardware (DOSERVICE) forces a microtrap to a specific CSA depending on the trap or interrupt for the purpose of initiating the exception or interrupt. Tables 5.1 and 5.2 list the CSA's

for each DOSERVICE microtrap. Two other microtraps not listed in the table which can occur during DOSERVICE are Timer Service (CSA is set to 0014) and Console ^P Trap (CSA is set to 0016).

The microtrap occurs during the exeuction of the microinstruction following the one containing the BUT/IRD1 micro-order. Two comments are worth making with respect to this fact. First, like other microtraps, if DOSERVICE detects a trap or condition which results in a microtrap, the currently executing microinstruction is prevented from completing, no destinations are written, and no bus cycles are performed. Instead, the microtrap is taken. Second, if the currently executing microinstruction also contains a microtrap condition, that microtrap is lost; the DOSERVICE microtrap takes precedence. All three statements make sense when we consider that the current microinstruction is the first microinstruction of the microcode which emulates the next VAX machine instruction. Its execution should be deferred until after all interrupts and traps relating to the previous VAX instruction have been taken.

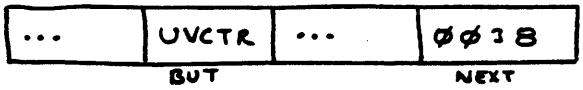
It is possible that more than one trap or interrupt could be pending when DOSERVICE is called. In such a case, they are handled one at a time, each DOSERVICE test resulting in a single microtrap. Each initiation routine eventually ends in BUT/IRD1 which again calls DOSERVICE. The order in which DOSERVICE traps and interrupts are initiated is as follows:

Arithmetic Trap
Timer Service
Console Control P
Interrupt at IPL 1E
.
.
Interrupt at IPL 01
T Bit Trap

Microbranch. A microbranch is a microprogrammed conditional branch which transfers control to an "initiate interrupt or exception" routine if the interrupt or exception is present. It uses the BUT field for multi-way branch control. In the case of exceptions, it is programmed into the microcode which handles the situation which could result in an exception. For example, a reserved operand microbranch is programmed into the microcode which evaluates the operand address. In the case of interrupts, it is programmed into the microcode at strategic locations in the emulation of very time-consuming VAX instruction in order to keep interrupt latency within the specified limits. This is done by means of the BUT/UVCTR micro-order, as follows:

The COMET microarchitecture includes four microvector lines which are set according to certain conditions and available to the interrupt and memory management systems for conditional branching. Table 5.3 delineates the meaning of the microvector

lines as a function of the micro-order which uses them. Note (from Table 5.3) that if none of the specific BUS or WCTRL microrders are present in the current microinstruction, then the microvector lines UVCTR<2:0> contain the code for the highest priority interrupt pending. The following microinstruction



will cause a microbranch to the starting address of the "initiate interrupt or exception" microcode for that interrupt.

ROM Decode. Recall (from Section 2.3) that the CSA of the first microinstruction in the emulation of a VAX machine instruction is obtained from the IRD1 ROM. Part of the index into the ROM is the opcode of the VAX instruction. Consequently, if a reserved opcode or the breakpoint (BKPT) fault is present in the instruction stream, it is detected because these eight bits are used to address the IRD1 ROM. The contents of that ROM address provide bits <9:3> of the CSA of the next microinstruction, i.e., the entry point of the corresponding "initiate" microcode.

5.1.3.2 The "Initiate" microcode.

The microcode to initiate an exception or interrupt must do several things, some of which are specified in the VAX System Reference Manual under "initiate exception or interrupt" (page 6-37), most of which are not. First (not specified), it must put the machine in a consistent state. If the emulation of a VAX machine instruction is suspended in the "middle," either because an interrupt must be serviced, or because some fault must be handled, then the contents of the general purpose registers and memory are unpredictable; they depend on just how far along in the emulation COMET was when the process was suspended. Thus, if the emulation of a machine instruction is to be suspended in the middle, then before control is transferred to the appropriate VAX service routine, it must be possible to do one of two things. Either undo the processing which has already been done for the VAX instruction being emulated, or for those VAX instructions which can be suspended and later restarted at the point of suspension, save the machine state. How this is accomplished is the subject of section 5.1.3.3.

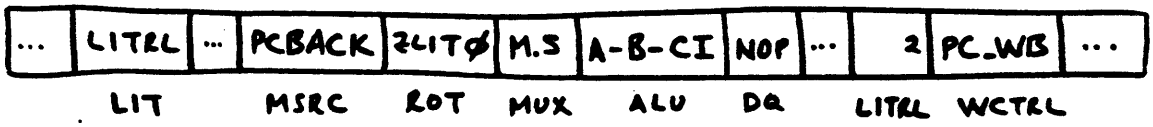
Second, the microcode must do the several tasks common to the initiation of all exceptions and interrupts. This includes selecting the appropriate stack for processing the exception or interrupt (the kernel or interrupt stack), pushing the PC and PSL of the suspended process on that stack, getting the new PC from the System Control Block, and constructing the PSL for the incipient VAX service routine.

Finally, the microcode must do those tasks specific to the particular exception or interrupt being initiated. For example, the trap code obtained from the Arithmetic Trap Code Register must be pushed on the stack before control is transferred to the Arithmetic Trap Service Routine. (This is done by the micro-order CCMISC/WB_ATCR.CCBR_SIGND.) Each exception and interrupt has associated with it its own set of specific tasks. In section 5.1.3.4 we describe the specific tasks for two of them: the Timer Service trap and the Software interrupt.

5.1.3.3 A Consistent Machine State

As discussed above, if the emulation of a VAX machine instruction is to be suspended in the middle, it is important to put the machine in a consistent state before transferring control to an interrupt or exception service routine. Several mechanisms exist in COMET for doing so.

First, before the PC is pushed on the stack, it must be backed up to point to the opcode of the VAX machine instruction being emulated. COMET has a register PCBACK. One of the effects of BUT/IRD1 is that PCBACK is loaded with PC+2. Thus, the following microinstruction is one way to back up the PC:



Second, if no general purpose registers or memory locations have been written into, the PC can be backed up, the interrupt or exceptions taken, and the suspended VAX instruction restarted from the beginning at some future time. If some of the general purpose registers have been altered during operand address calculation due to autoincrement and autodecrement addressing modes, the PC can still be backed up and the suspended VAX instruction restarted from the beginning at a later time. In this case, it is necessary to restore the general purpose registers to their values at the start of the VAX instruction before transfering control to the interrupt or exception service routine. To do this, COMET uses the Register Back Up Stack (RBS), described in Section 4.2.4. Recall that each entry in the RBS consists of a register number, a data size, and a 1 or 0 depending on whether the register was autoincremented or

autodecremented. Before transfering control to a service routine, the RBS pointer is examined. If it is non-zero, each entry in the RBS is popped, and the proper register is incremented or decremented the appropriate amount.

Finally, we consider the situation where some action has been taken which cannot be undone; for example, a write to memory. VAX provides a feature for certain instructions, notably the character string instructions, which allows them to be suspended, and later restarted from the point of suspension. The mechanism is the FPD bit (First part done) in the PSL. If the microcode emulating a VAX instruction performs an action which cannot be undone, PSL<FPD> is set. Subsequently, if it is necessary to suspend the emulation of that VAX instruction, PSL<FPD> is tested (BUT/FPD). If PSL<FPD> is set, it is necessary to pack the information into a consistent state before transfering control to the VAX service routine. A pointer to the appropriate packing routine is contained in one of the M Scratch Pad Registers (M[0C], FPDOFFSET). It was loaded there by the execution microcode of the VAX instruction being emulated. At a later time, when the emulation of the VAX machine instruction is to be resumed, BUT/IRD1 causes a branch not to the microcode to begin emulating the instruction, but instead to the microcode which first unpacks and then resumes the emulation at the point where it left off. This is accomplished by including the FPD bit as part of the index into the IRD1 ROM.

5.1.3.4 More detail: Timer Service and Software Interrupts.

In this section, we describe the specific tasks which COMET must perform in initiating a timer service trap and a software interrupt. The specific tasks associated with the other exceptions and interrupts will not be covered. These two have been chosen because they are a little more interesting (to the author) than the others, and they illustrate the most important procedures COMET goes through in initiating an exception or interrupt.

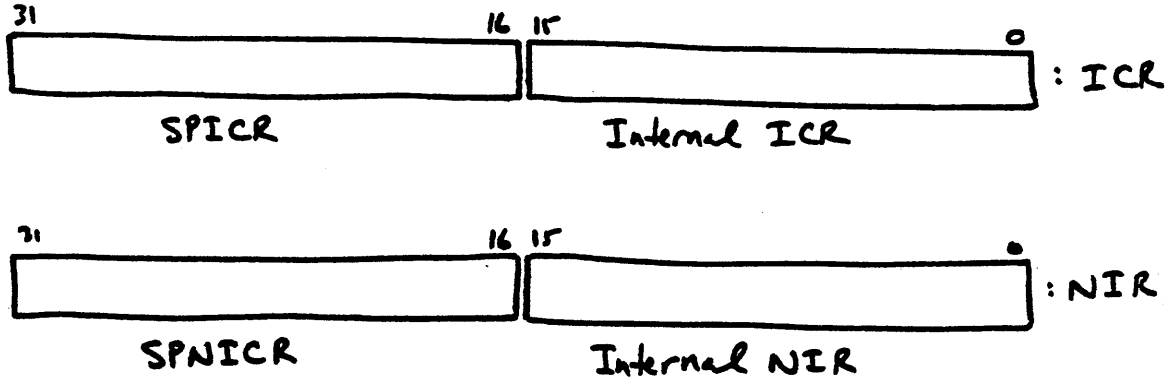
Timer Service.

VAX keeps track of the amount of time allocated to a process by means of two 32 bit registers, the Interval Count Register (ICR) and the Next Interval Register (NIR). The ICR contains the negative (2's complement) of the number of microseconds remaining in the current interval. The NIR contains the negative of the number of microseconds to be allocated to the next interval. At the start of an interval, ICR is loaded with the contents of NIR and starts incrementing at the rate of one count per microsecond. When ICR reaches 0, the interval has passed. The next interval is loaded into ICR, and the INT bit of the Interval Clock Control

and Status Register (ICCS) in set. If interrupts are enabled (i.e., if the IE bit of ICCS is set), an interrupt request at IPL 18 (hex) is generated. IPL 18 is the interval timer interrupt.

COMET implements this timing mechanism by means of a combination of microcode and hardware. The microcode is invoked by the Timer Service trap. Its function will be described momentarily. The hardware consists of five bits* in the Timer Control and Status Register (TCSR), which are labeled IR, SR, TR, VP, and TVP, four 16 bit registers (to implement ICR and NIR), and the associated logic and circuitry. IR is the COMET implementation of the INT bit of the VAX ICCS. SR, TR, VP, and TVP are internal bits required by the COMET microarchitecture.

The four 16 bit registers are shown below:



The high order 16 bits of the ICR and NIR together comprise R[2E] of the R Scratch Pad register file (cf. section 4.2). SPICR (which stands for Scratch Pad ICR) is implemented as R[2E]<31:16>, and SPNICR (which stands for Scratch Pad NIR) is R[2E]<15:0>. The low order 16 bits of the ICR and NIR are internal COMET registers. We refer to them in this discussion as IICR and INIR, respectively.

At the start of an interval, ICR contains the negative (2's complement) of the interval in microseconds. The low-order 16 bits of the ICR (i.e., IICR) is really a hardware up-counter which continues to increment, one count per microsecond, one cycle per 65 msec. Each time IICR cycles (except the "last" time, when ICR=0), the high order 16 bits of ICR (i.e., the SPICR) must be incremented.

---

* Actually, there are more than five bits, but the other bits are not relevant to this discussion.

The "last" time IICR cycles (i.e., ICR=0), signifying the end of the current interval, ICR must be loaded with the contents of NIR and the IR bit of the TCSR must be set. Loading ICR from NIR involves loading SPICR with the contents of SPNICR and loading IICR with the contents of INIR. The hardware controls the loading of IICR from INIR and the setting of IR. The Timer Service trap service routine controls the loading and incrementing of SPICR.

The Timer Service trap is invoked whenever a carry out of the IICR (i.e., the IICR has cycled) requires that the SPICR must be updated. The Timer Service trap works in conjunction with bits SR, TR, VP, and TVP of the TCSR register. The Timer Service trap is invoked by DOSERVICE if either SR or TR is set. If SR is set, this signifies that IICR has overflowed, and that it is not the last time this is to happen in the current interval. If TR is set, this signifies that IICR has overflowed and it is the last time this is to happen in the current interval, i.e., the interval is over. Whether or not it is the last time is specified by the state of VP. In other words, the end of an interval is specified by VP=1 and IICR overflowing.

The specific tasks performed by the Timer Service trap service routine can now be stated. Note that unlike the other exceptions and interrupts which are only initiated by the microcode the specific Timer Service trap microcode services the exception. The Timer Service trap microcode does the following:

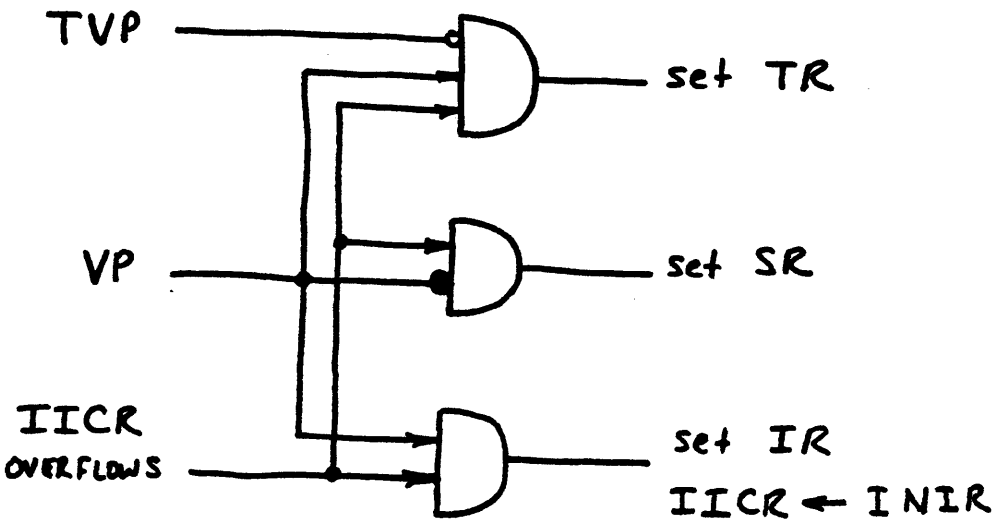
(1) If SR=1, then SPICR <-- SPICR +1, SR <-- 0.

(2) If TR=1, then SPICR <-- SNICR, TR <-- 0.

(3) If incrementing SPICR causes it to contain all 1's, then VP <-- 1, signifying that the interval has one more cycle of IICR (65 msec) remaining.

To complete the picture, we need to describe the functioning of the TVP bit and the gating. First the TVP bit. Usually, the length of an interval is less than 65 msec. Consequently, the next interval to be loaded into ICR usually contains all 1's in SPNICR. When this is the case, since at the end of the current interval SPICR already contains all 1's, it is not necessary to load SPICR.

The TVP reflects this situation. It is set and cleared by the microcode to reflect whether or not SPNICR contains all 1's. As a result, if TVP is set and IICR has overflowed for the last time in the current interval VP=1, the Timer Service trap is not invoked. SPICR already contain the contents of SPNICR. The gating which controls all this is summarized below. Not that at the end of an interval, IR is set and IICR is loaded from INIR, whether or not the Timer Service trap is invoked.



Software Interrupts.

A software interrupt microtraps or microbranches to 0038, depending on whether it was detected during DOSERVICE or during a microprogrammed branch (BUT/UVCTR). The microcode initiated at CSA 0038 performs the following actions:

(1) The base address of the Software Interrupt Vector in the System Control block is saved.

(2) The IPL of the highest pending software interrupt is obtained from the SISR (software interrupt summary register).

(3) The address of the SCB vector is computed from the base address (obtained in 1) and the IPL (obtained in 2).

(4) SISR<IPL> is cleared. Note: this means that it is the operating system's responsibility to not perform an REI until all software interrupts at that IPL have been serviced.

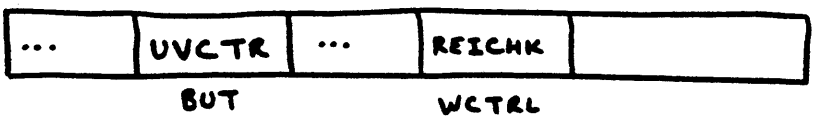
(5) The next highest IPL present in the SISR is loaded into the COMET internal Software IPR.

(6) A microbranch is taken to the common microcode for all exceptions and interrupts (recall Section 5.1.3.2)

5.1.3.5 Return from Exception or Interrupt (REI)

The final instruction in a VAX exception or interrupt sevice routine is the instruction REI. COMET emulates this instruction by popping the PC and PSL of the suspended process, switching to the proper stack, and then testing (by means of the hardware) whether the return is "legal," and also, whether there is an AST pending which can be delivered. *

The microinstruction to test for legal returns and for ASTs pending is



The microvector lines are as specified in Table 5.3 for the WCTRL/REICHK microrder; that is a three way microbranch occurs depending on whether the REI is legal, the REI is legal and ther is an AST pending which can be delivered, or the REI is not legal.

5.1.4 An Example

We conclude this section on interrupt and exception handling with an example. Suppose a user process is executing. Suppose COMET has just completed the emulation of one VAX machine instruction and is about to start the next. Suppose the next VAX instruction is BKPT. Suppose the following traps and interrupts are pending: integer overflow trap, a Timer Service trap, a UNIBUS device request at UNIBUS BR6, and a T-bit trap. Finally, suppose a kernel mode AST becomes available during the execution of the last microinstruction. What happens?

Figure 5.2 shows the flow of control of the microcode to handle the above situation. Figure 5.3 shows the contents of PC, PSL and the relevant VAX stacks at each point in the execution flow. We begin the discussion with the last microinstruction of the VAX machine instruction just completed. This microinstruction contains the BUT/IRD1 micro-order, which does two things. It causes the microsequencer to obtain the CSA of the next microinstruction from the IRD1 ROM. It also signals DOSERVICE to check for traps and interrupts during the next microcycle.

---

* There are several reasons why a return could be "illegal." For example, the access mode of the service routine might be of a lower privilege than that of the suspended process. Or the suspended process might have an IPL greater than 0 and not have kernel mode privileges.

The IRD1 ROM, indexed by the BKPT opcode, branches to microcode (1) to initiate the BKPT fault. During the first microinstruction of that routine, DOSERVICE detects the arithmetic trap, prevents the current microinstruction from executing, pushes its address onto the microstack, and microtraps to CSA 0011 to initiate the arithmetic trap.

The processor switches to the kernel stack*, pushes the PC and PSL of the suspended user process on the kernel stack, gets the trap code (in this case the value 1 for integer overflow) from the ATCR and pushes it onto the kernel stack, specifies a PSL for the Arithmetic Trap service routine and loads PC with the starting address of that routine. The microcode terminates with BUT/IRD1, causing the IRD1 ROM to branch (2) to the microcode which emulates the first VAX instruction of the Arithmetic Trap service routine. Again DOSERVICE is signaled to check for traps and interrupts during the next microcycle.

Again DOSERVICE detects a trap (Timer Service), inhibiting the current microinstruction, this time causing a microtrap to CSA 0014. Because Timer Service requires very little processing, it is serviced immediately,transparent to the PC, PSL, and the rest of the VAX architecture. Timer Service terminates with a BUT/IRD1 micro-order, causing the Arithmetic Trap service routine to again begin execution (3).

Again DOSERVICE inhibits the first microinstruction from completing, this time detecting the interrupt from the UNIBUS device (IPL 16). COMET microtraps to CSA 003A to initiate the interrupt. The processor switches to the interrupt stack, pushes the PC and PSL of the suspended process (in this case the VAX Arithmetic Trap service routine) onto the stack, specifies a new PSL for the device interrupt service routine, and loads PC with the starting address of that service routine.

The interrupt service routine executes (4), terminating with the REI instruction. The REI pops the stack, loading the PC and PSL registers with the PC and PSL of the Arithmetic Trap service routine, and since the Arithmetic Trap service routine executes on the kernel stack, switches to the Kernel Stack. The emulation of REI contains the WCTRL/REICHK micro-order which provides

---

* This example assumes that traps and AST Delivery are handled on the kernel stack, and higher priority interrupts are handled on the interrupt stack. Whether to use the kernel stack or interrupt stack to handle a particular interrupt or exception is a system software parameter and is specified by bits <1:0> of its SCB vector.

microbranching on the microvector lines depending on whether or not there is an AST pending which can be delivered. In this case there is an AST which can be delivered (indicated by UVCTR <1:0> = 01), so a microbranch is taken in order to post the delivery of the AST. Bit 2 of the Software Interrupt Summary Register (SISR) is set, since AST Delivery is initiated by an interrupt at IPL 2. If it were necessary (in this case it isn't), the Software IPR would be updated at this time (cf., Section 5.1.3.4). The microcode for the REI instruction is then allowed to complete, terminatng with a BUT/IRD1 micro-order.

Again, the Arithmetic Trap service routine attempts to execute, and again DOSERVICE inhibits the first microinstruction, this time because of the IPL 2 interrupt. COMET microtraps to CSA0038 to initiate the interrupt. The address of the AST Delivery service routine is computed and Bit 2 of the SISR is cleared. since the AST Delivery service routine executes on the kernel stack, the processor does not need to switch stacks. It pushes the PC and PSL of the Arithmetic Trap service routine onto the stack, specifies the new PSL and loads the PC with the starting address of the AST Delivery service routine.

The AST Delivery service routine executes (5), causing the pending AST to be delivered. That is, control passes (by means of the formal CALLG VAX instruction) to the address of the service routine specified by the AST. The AST service routine executes, terminating in a RET instruction. Control returns to the AST Delivery routine. After completing certain bookkeeping functions (not relevant to this discussion), the AST Delivery routine terminates; the final VAX instruction is REI. The REI instruction pops the kernel stack, loading PC with the starting address of the Arithmetic Trap service routine, and loading PSL with the PSL of the Arithmetic Trap service routine. This time there is no AST pending, so the microcode is allowed to terminate with a BUT/IRD1, which signals the emulation of the next VAX instruction, the first instruction of the Arithmetic Trap service routine (6). Since the Arithmetic trap service routine also executes on the kernel stack, the emulation of the REI instruction does not cause any switching of stacks.

The Arithmetic Trap service routine is now able to execute. It also terminates with an REI instruction. Again the PC and PSL are popped, this time containing the address of the BKPT opcode and the PSL of the user process. Since the user process executes on the user stack, the processor switches stacks. Once again (7), an attempt is made to emulate the BKPT instruction. However, since the PSL is that of the user process, the T-bit trap is detected, causing a microtrap to CSA 0015 to initiate the T-bit trap.

The processor switches to the kernel stack, pushes the PC and PSL of the suspended user process onto the kernel stack, specifies a new PSL for the T-bit trap service routine, and loads PC with its starting address, terminating with the micro-order BUT/IRD1.

Control passes (8) to the T-bit trap service routine, which executes, terminating in the VAX instruction REI. The PC and PSL of the user process are again popped and loaded into the PC and PSL registers. The processor switches to the user stack. The microcode terminates with BUT/IRD1.

Once again control passes (9) to the user process which attempts to initiate the BKPT fault. This time there are no traps or interrupts for DOSERVICE to detect, and the BKPT microcode is allowed to execute.

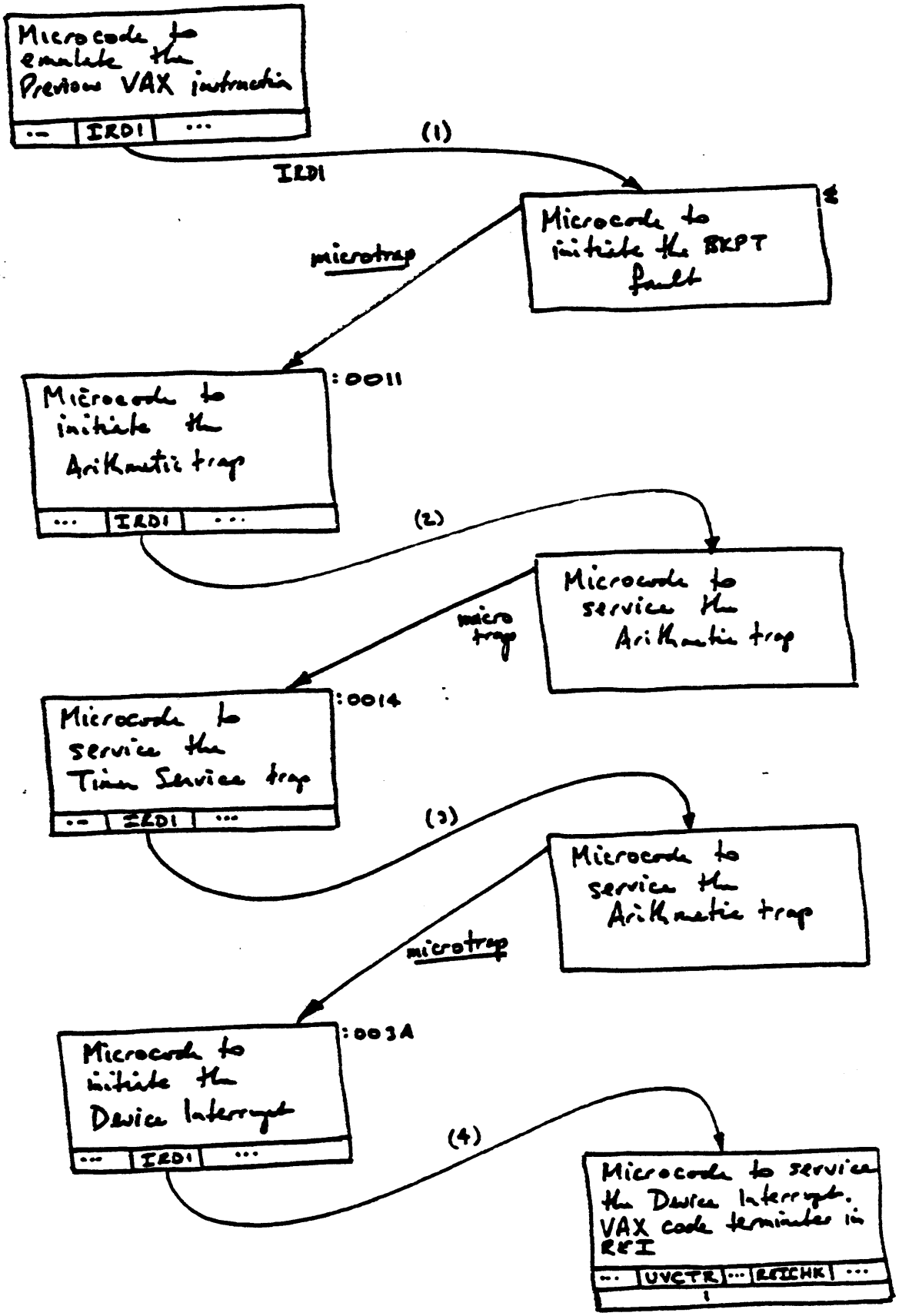


Figure 5.2 Microprogram Flow of Control (Example of Section 5.1.4)

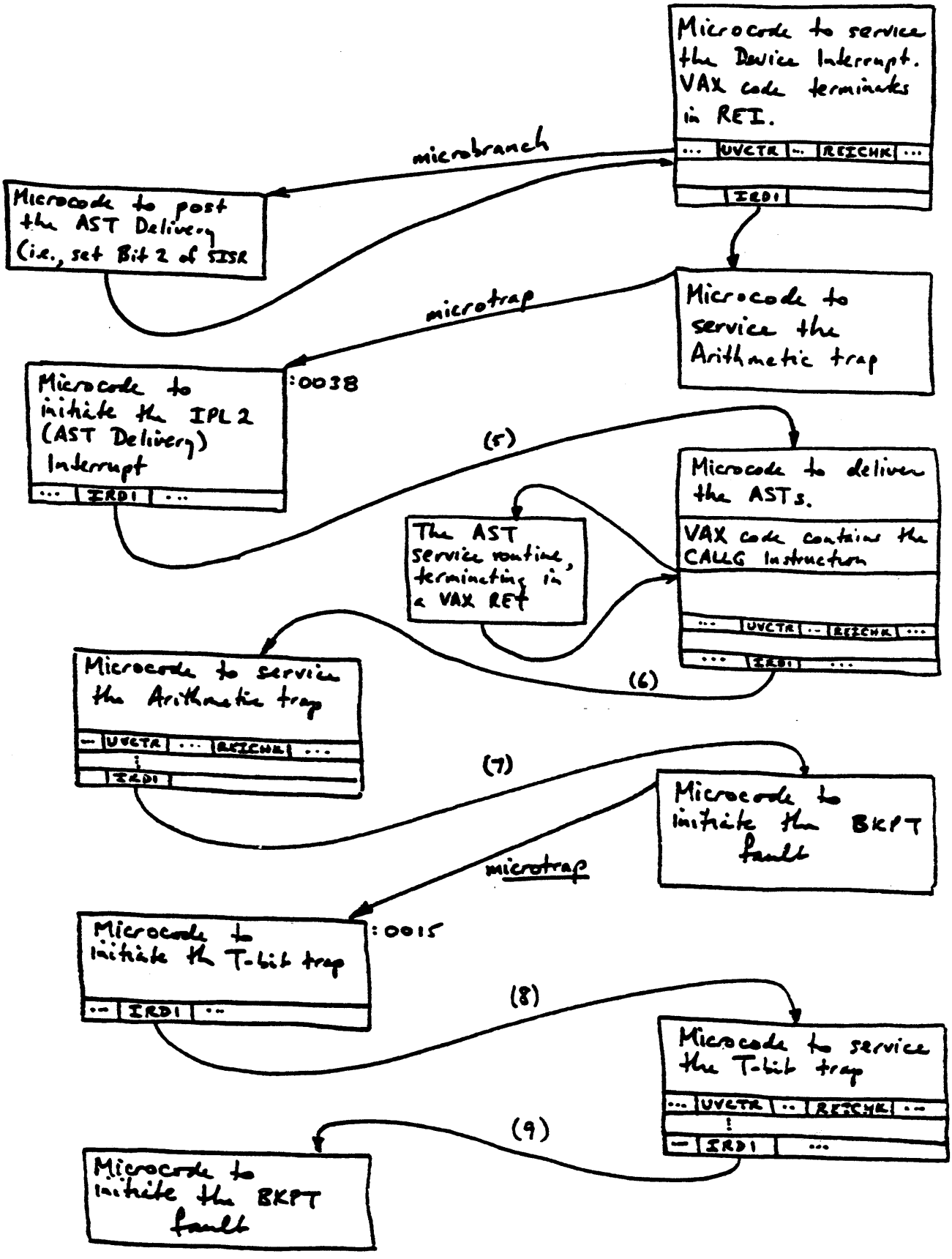
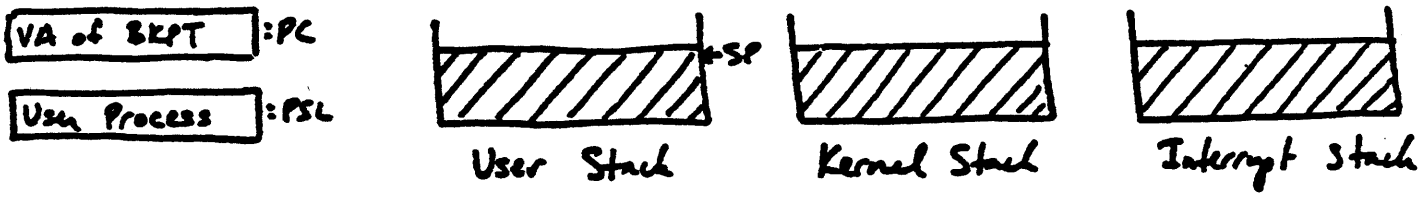
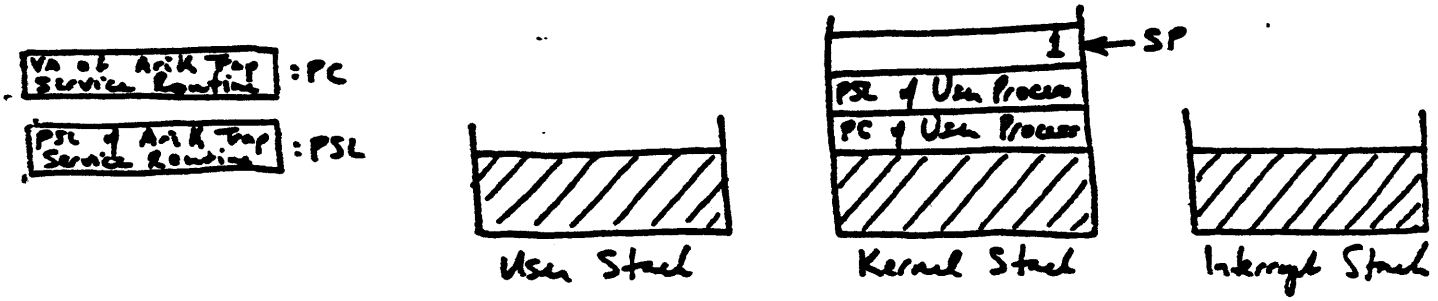


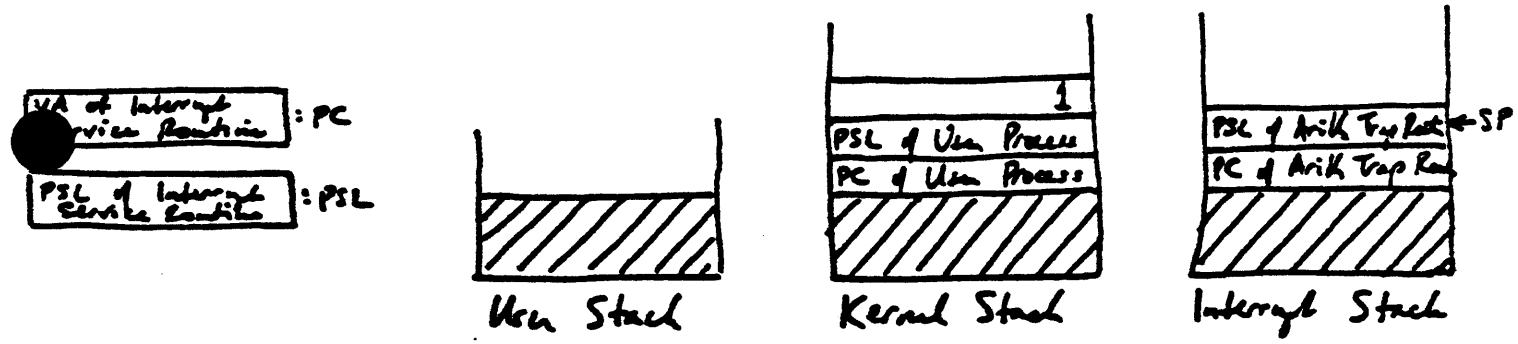
Figure 5.2 (continued). Microprogram Flow of Control (Example of Section 5.1.4)



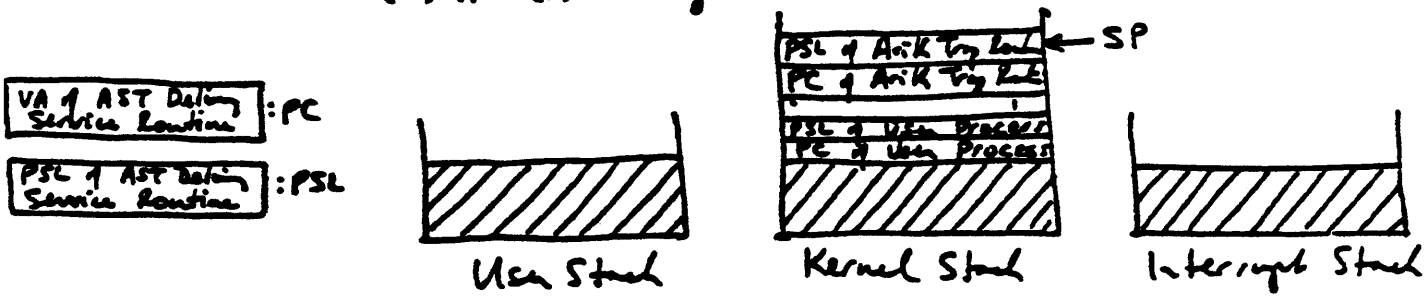
(a.) At (1) on Figure 5.2.



(b.) At (2) and (3) on Figure 5.2.

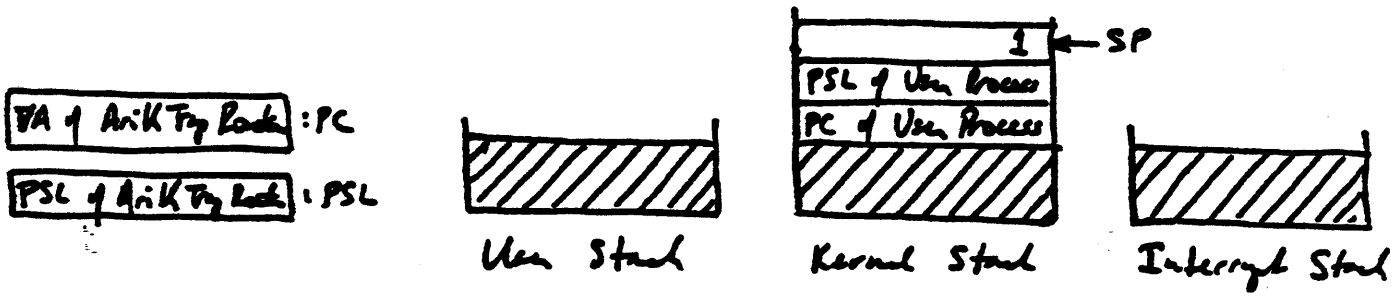


(c.) At (4) on Figure 5.2.

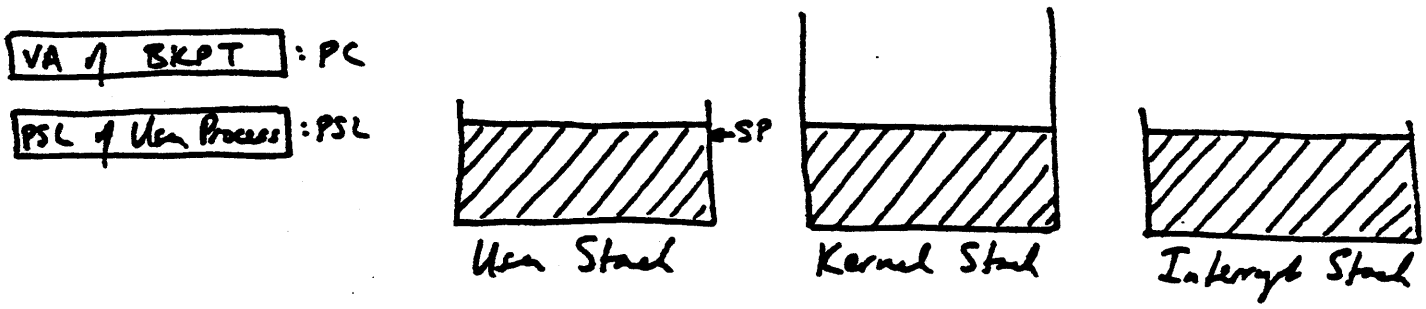


(d.) At (5) on Figure 5.2

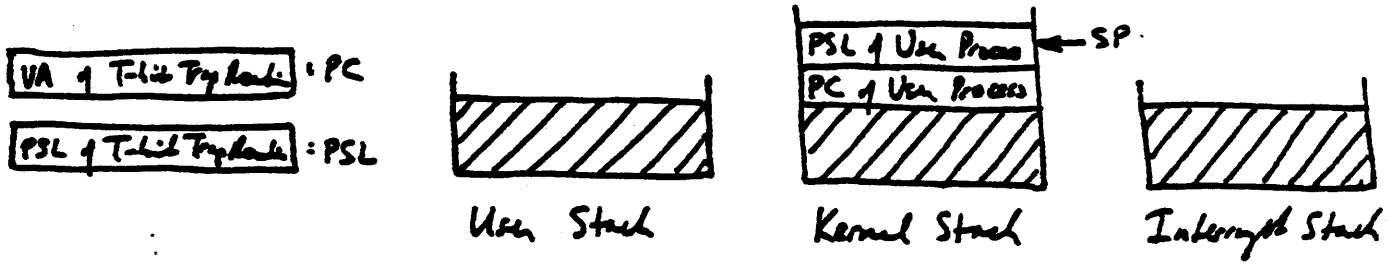
Figure 5.3 State of the VAX Stacks (Example of Section 5.1.4)



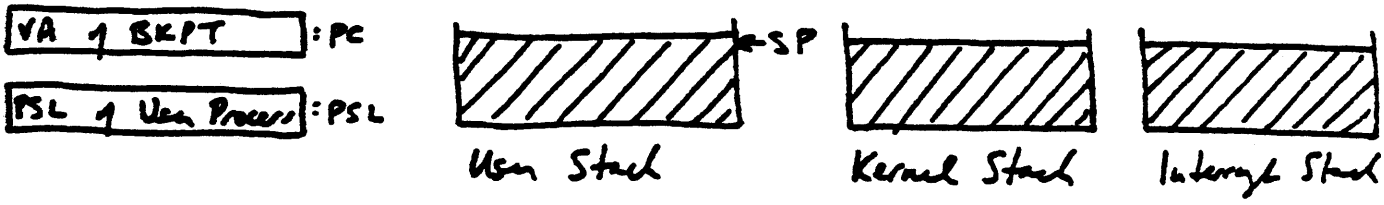
(e.) At (6) on Figure 5.2.



(f.) At (7) on Figure 5.2.



(g.) At (8) on Figure 5.2.



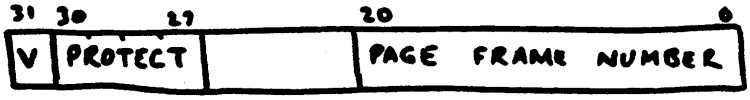
(h.) At (9) on Figure 5.2.

Figure 5.3 (continued). State of VAX Stacks (Example of Section 5.1.4)

5.2 Memory Management

5.2.1 VAX Memory Management

Vax provides each user with 4 Billion bytes of virtual storage, - 2 Billion bytes of process space (P0 Space and P1 Space) and 2 Billion bytes of system space. Virtual storage is partitioned into 512 byte units called pages. Corresponding to each page of virtual memory is a 32 bit Page Table Entry (PTE), shown below:



The PAGE FRAME NUMBER is the physical address of the page in main memory. The V (Valid) bit is set if the page frame number is valid, i.e., if the page actually does reside in physical memory and has not been swapped out. The PROTECTION code specifies the access rights to the information on the page for processes with different levels of privilege. VAX maintains four levels of privilege: Kernel, Executive, Supervisor, and User.

Page Table Entries for system space are stored in contiguous longwords of physical memory called the System Page Table. The base address of the System Page Table is stored in the System Base Register, one of VAX's interval processor registers (c.f., Section 4.2). Page Table Entries for P0 Space and P1 Space are stored in the P0 Page Table and the P1 Page Table which are located in virtual system space. Each page table consists of contiguous longwords. The base address (virtual) of the P0 Page Table and the P1 Page Table are stored in the P0 Base Register and P1 Base Register. Both are VAX internal processor registers. Figure 5.4 is a snapshot of VAX virtual memory, physical memory, and the corresponding page table.

A read or write to virtual memory involves several steps. First, VAX permits unaligned memory references. For example, a longword need not start on a longword boundary. However, since COMET's physical memory is always accessed on longword boundaries, a process which requests an unaligned memory access could require an extra physical memory access. Second, the hardware determines the physical address from the virtual address and the appropriate PTE, which may or may not be available in the Translation Buffer. (The Translation Buffer is effectively a cache of PTE's). If the required PTE is not in the Translation Buffer, then additional memory accesses must be made to bring the PTE into the Translation Buffer before the physical address of the desired memory location can be obtained. Finally, the presence of the PTE in the Translation Buffer does not guarantee that the memory access can be made. If the protection code associated with the page of memory specifies for the access desired a higher level of privilege than that of the process

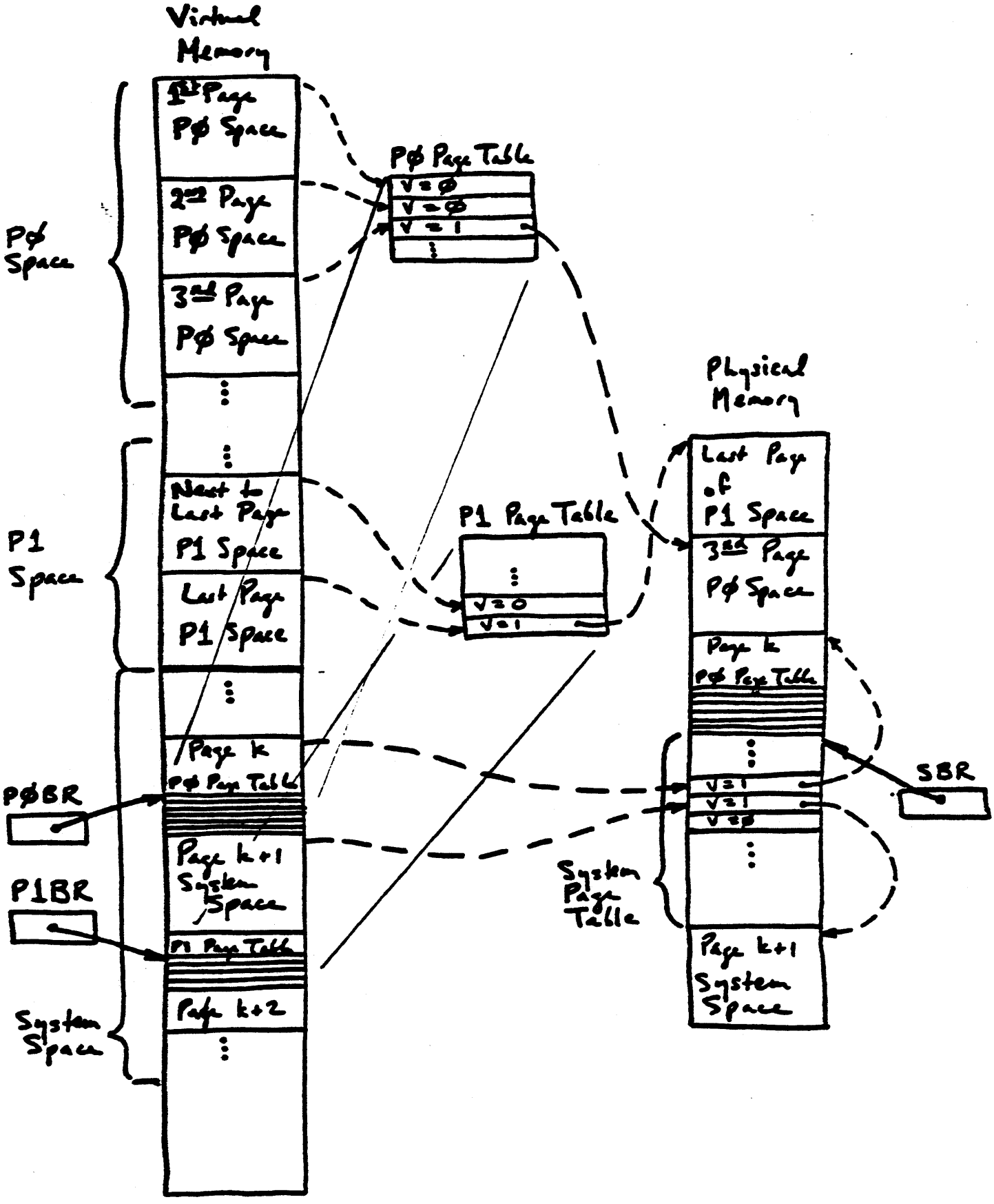


Figure 5.4 VAX Memory Management - An Overview

requesting the access, then an Access Control Violation (ACV) fault results. If the V bit is not set, designating that the page of virtual memory is not in physical memory, then a Translation Not Valid (TNV) fault results.

5.2.2 COMET Implementation

5.2.2.1 Memory Management Microtraps.

To implement the VAX memory management functions, the COMET microarchitecture must handle the conditions described above; in particular, unaligned memory references, TB misses (i.e., the required PTE is not in the Translation Buffer), and ACV and TNV faults. The conditions occur as a result of memory reads and writes, instruction fetches, and data reads from the instruction stream. To handle unaligned accesses, TB misses, and ACV faults, COMET uses the microtrap mechanism. To handle TNV faults (and ACV faults if the PTE is not in the TB), COMET uses the microbranch mechanism discussed in Section 5.1.3.1. In this discussion, we are most concerned with microtraps.

COMET identifies 13 microtraps associated with the memory management system. They are listed in Table 5.4. Recall that a microtrap is the result of a condition detected by the hardware and that the condition is such that the current microinstruction would not be able to complete its execution successfully. The microtrap pushes the address of the current microinstruction on the microstack and forces a branch to a fixed address in Control Store, usually for the purpose of correcting the condition which caused the microtrap, so that the microinstruction can be re-executed. The fixed CSA for each microtrap is also shown in Table 5.4. A single microinstruction could have more than one of the conditions listed in Table 5.4. In such a case microtraps occur sequentially, according to the priority scheme shown. Each microtrap corrects its condition, an attempt is made to re-execute the faulting microinstruction, and the next microtrap occurs. The process is illustrated in the example in Section 5.2.3.

We should, at the outset, differentiate the ACV microtraps from the others. The ACV microtraps are caused by faults in the VAX machine language program. They are exceptions which must be serviced by VAX exception handling routines, not unlike the way other interrupts and exceptions are serviced (see Section 5.1). Consequently, the microprogram flow never returns to the microinstruction causing the microtrap. On the other hand, the remaining microtraps are caused by conditions in the hardware implementation of the VAX architecture. Consequently, in these cases, the effect of the microtrap is to branch to a routine which corrects the condition (if possible), then pops the microstack, re-executing the faulting microinstruction.

There are fundamentally two microtrap service routines for memory management, with variations on each, depending on the particular microtrap. One routine handles unaligned memory accesses; the other handles TB misses. They are described in sections 5.2.2.3 and 5.2.2.4 below.

5.2.2.2 Re-execution of a Faulting Microinstruction

The re-execution of the faulting microinstruction takes one of three forms, depending on the microtrap.

If the microtrap prevents the faulting microinstruction from writing values to any destination, and if the microtrap service routine did not complete the memory access, then the faulting microinstruction is simply re-executed. This is accomplished by BUT/RETURN and NEXT/0 in the last microinstruction of the routine which corrects the condition.

If the routine which corrects the condition also completes the memory access, then the faulting microinstruction is re-executed, but bus cycles are suppressed so as not to repeat the memory access. This is accomplished by BUT/RETURN, NEXT/0, and MISC/RSBC in the last microinstruction of the routine which corrects the condition.

Finally, there is the case where the microtrap does not prevent the faulting microinstruction from writing values to its destinations. In such cases, clearly, the re-execution of the faulting microinstruction must not allow any destinations to be written. Only one microtrap produces this situation--when BUT/IRD1 results in a XBTB miss. Recall that BUT/IRD1 is contained in the last microinstruction of the microcode which emulates a VAX machine instruction. It causes the IR and OSR to be loaded from XB with the opcode and first operand specifier of the next VAX machine instruction to be emulated. If attempting to load IR and OSR results in a TB miss, the rest of the microinstruction is allowed to complete execution before the microtrap takes place. This is done since the microtrap really

involves fetching the next VAX machine instruction and has nothing to do with the successful completion of the emulation of the current VAX machine instruction. After the microtrap is taken and the TB miss is corrected, the faulting microinstruction is re-executed. This time only the BUT/IRD1 activity occurs. All destinations are inhibited, any bus cycles are suppressed. This is accomplished by BUT/RET.DINH and NEXT/0 in the last microinstruction of the routine which corrects the XBTB miss condition.

5.2.2.3 Unaligned memory access service routines.

An unaligned memory access is detected by the hardware when an attempt is made to read or write information which crosses a longword boundary. The hardware detects five different unaligned memory access conditions and produces a microtrap for each. Figure 5.5 shows the activity of the corresponding service routines.

The memory access is performed in two steps, one for each longword to be accessed: BUS/READ.NT and BUS/READ.SEC in the case of a read, BUS/WRITE.NT and BUS/WRITE.SEC in the case of a write, and BUS/WRITE.UL and BUS/WRITE.UL.SEC in the case of a write which also releases a lock set by a read lock microcode. The VA register is adjusted (VA <-- VA+4) which is necessary for the second access and readjusted (VA <-- VA - 4) after the second access so that when the faulting microinstruction is re-executed, VA contains the virtual address of the element read or written. ACV traps are not necessary in the first memory access. If the first memory access had resulted in a ACV fault, it would have been detected by the read or write ACV microtrap, which has a higher priority than the unaligned data microtrap.

The second memory access does not suppress the ACV microtrap. In the case of a read, this is no problem; if an ACV fault occurs on the second access, ACV READ microtrap is taken and eventually the ACV exception service routine is invoked. In the case of a write, it is important to detect the ACV fault before the first write occurs. This is accomplished via the WRITE CROSSING PAGE BOUNDARY and WRITE UNLOCK CROSSING PAGE BOUNDARY microtraps. In both cases BUS/PRB.WR and BUT/UVCTR are used to check the access rights of the second page before the first BUS/WRITE.NT is performed. BUS/PRB.WR produces the signals on the microvector lines, depending on the state of the page of memory being probed, as shown in Table 5.3.

Since the microtrap routine completes the desired memory access, it is important not to attempt the memory access again, when the faulting microinstruction is re-executed. Bus cycles are suppressed by coding BUT/RETURN and MISC/RSBC in the last microinstruction of this service routine.

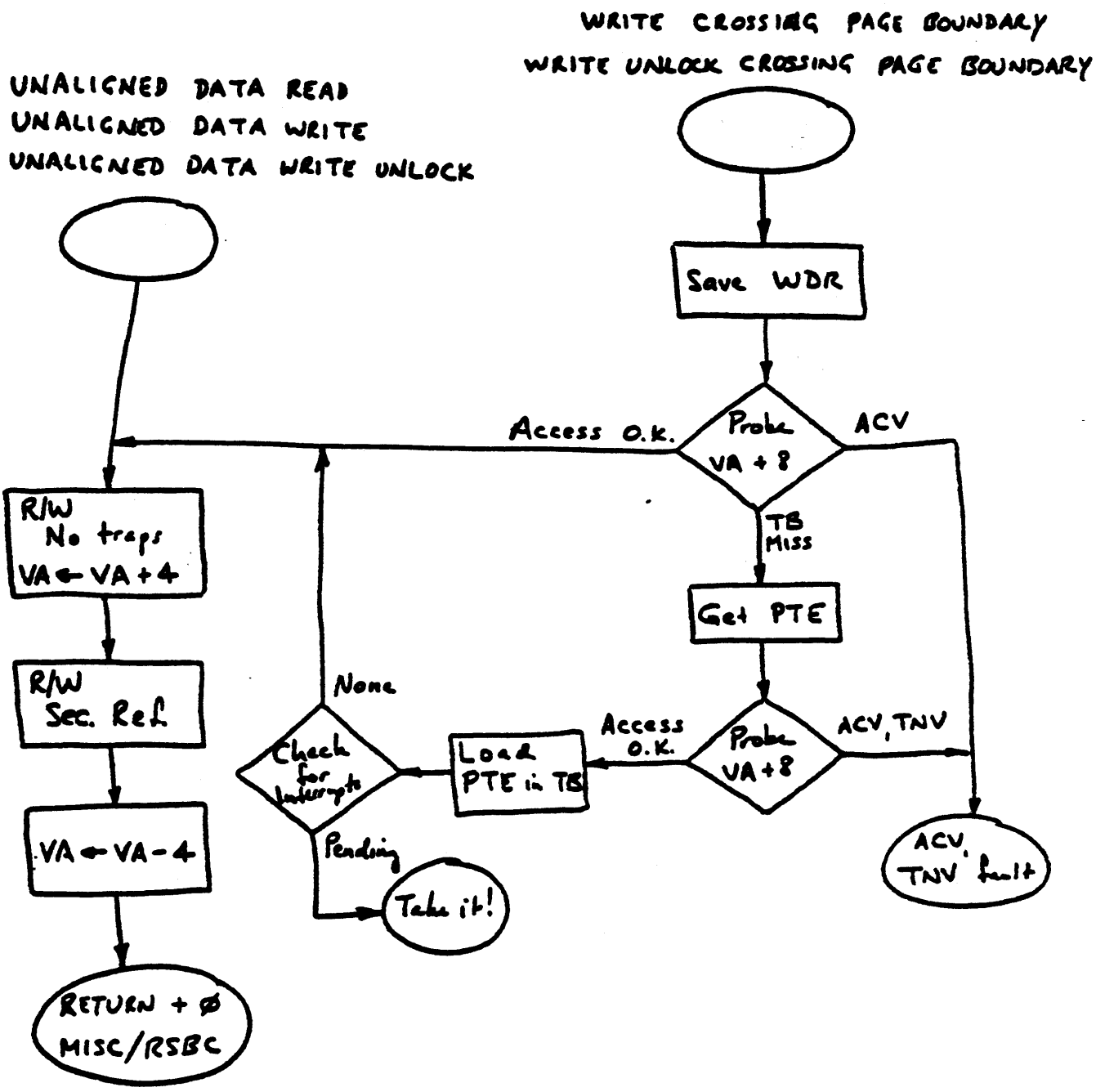


Figure 5.5. Flow of Control - Unaligned Memory Access Service Routines

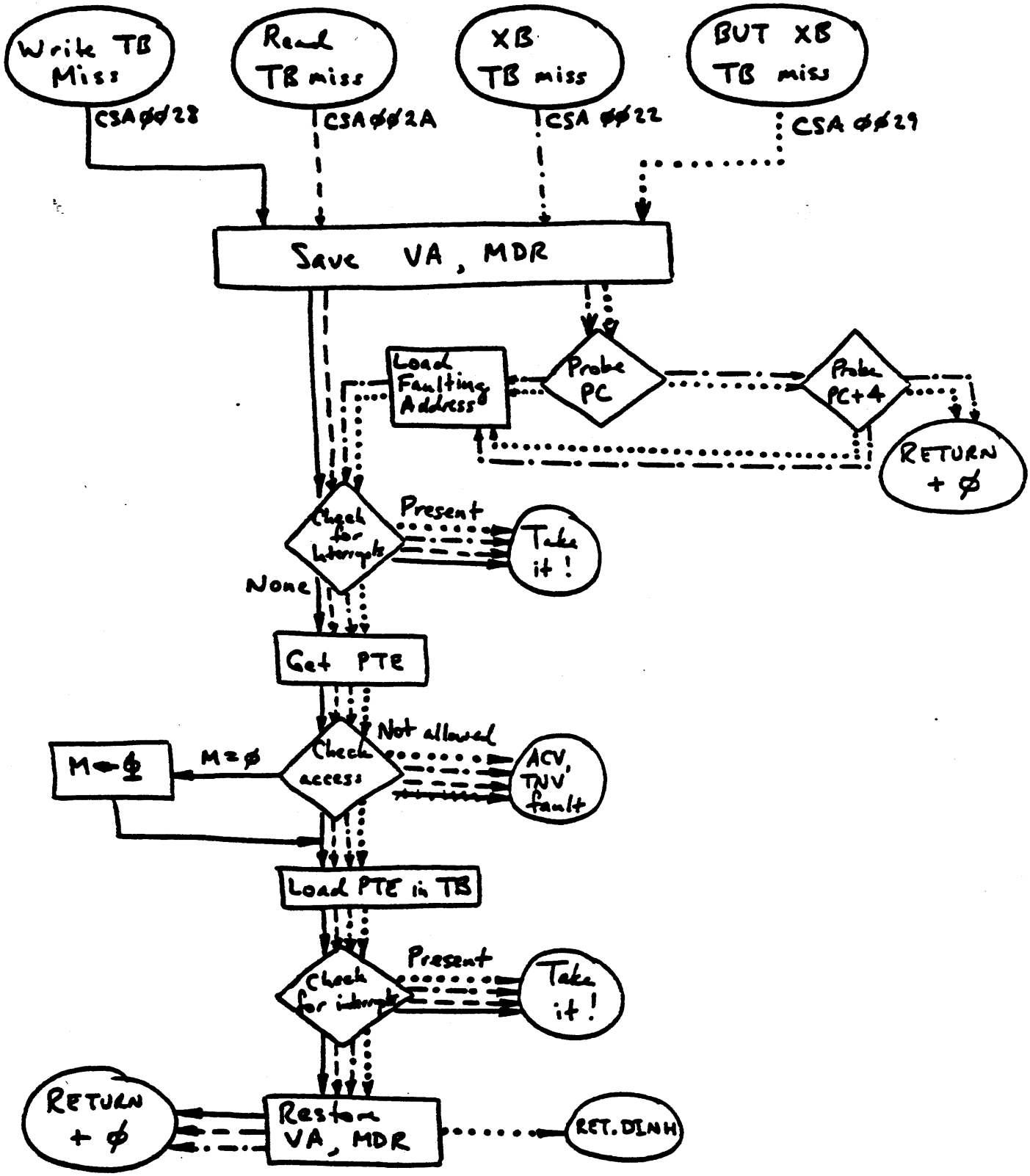


Figure 5.6. Flow of Control - TB Miss Service Routine

5.2.2.4 Translation Buffer (TB) miss service routines

The hardware detects four distinct cases where the required PTE is not in the Translator Buffer (TB). They are called TB miss conditions; each causes a microtrap. Figure 5.6 shows the activity of the corresponding service routines. READ TB MISS and WRITE TB MISS are caused by memory accesses where the memory location is addressed by VA. VA is the usual location of the memory address when a memory read or memory write is required. XBTB MISS is caused by a read access when the source of the data is addressed by PC(i.e., contained in the XB.) This is the case when the source of the data is the instruction stream (as in the case of immediate operands, for example). It is also the case in the COMET implementation of certain character string instructions where it was decided to use the PC as a pointer to the source string, rather than use the VA as a pointer to both source and destination strings (which would require changing the contents of VA twice for each character operated on). Finally, BUT XB TB MISS is caused by a TB MISS resulting from a BUT/IRD1 micro-order. Like XB TB MISS, the memory location addressed is specified by PC. This microtrap is the only one where destination writes are not inhibited, and the microinstruction is allowed to complete execution before the microtrap is taken.

Two things need to be said about the TB miss service routines. First, each routine includes two tests for the presence of pending interrupts. Since "Get PTE" could involve two physical memory accesses, this is the memory management system's contribution to protecting against an interrupt latency error caused by a TB miss. Second, there is no hardware detection of a TNV fault. A TNV fault is detected by examining the V bit in the PTE after the PTE has been obtained from memory. On a TB miss ACV and TNV faults are indicated on the microvector lines in response to the appropriate BUS/PRB micro-order.

5.2.3 An example

We conclude this section on memory management with an example. Suppose CSA "A" contains the last microinstruction in a routine to emulate a particular VAX machine instruction. Suppose this last microinstruction initiates a write to memory which crosses a page boundary; i.e., the write is to two separate pages. This microinstruction, then, includes the BUT/IRD1 and the BUS/WRITE micro-orders. Suppose, finally, that none of the three relevant PTE's (the one relating to the next opcode and the two relating to the destination to be written) are in the TB. What happens?

Figure 5.7 shows the flow of control for the execution of the microinstruction at A. The microinstruction attempts to execute (1), but potentially three microtraps could occur. The BUS/WRITE could cause TB miss and Write Crossing Page Boundary microtraps. The BUT/IRD1 could cause a BUT/XB/TB miss. Only one microtrap can occur at a time; the TB miss takes precedence. Destinations are inhibited, the address A is pushed on the microstack, and a forced branch (2) to CSA 002B occurs. COMET microcode then gets the PTE and loads it into the TB, and terminates with a microinstruction containing BUT/RETURN and NEXT/0. This pops the microstack, and the microinstruction at A is attempted again (3).

This time the Write Crossing Page Boundary microtrap takes precedence. Again destinations are inhibited, again A is pushed onto the microstack. This time a forced branch (4) to CSA 0027 occurs. Since the memory access is write, the access privilege of the second page is checked (BUS/PRB.WR) before the first page is written. The attempt to check the access of the second page results in a TB miss, and a microbranch (5) to the microcode to get the needed PTE results. After the PTE is loaded into the TB, the microstack is popped, returning to the Write Crossing Page Boundary routine (6).

Since the write access is allowed, the microcode proceeds to perform the two writes, terminating with a microinstruction which includes BUT/RETURN, MISC/RSBC, and NEXT/0 micro-orders. This pops the microstack, which causes COMET to again attempt to execute the microinstruction at A (7); this time, however, with the bus cycle suppressed since the write was performed by the microtrap routine.

This time, the BUT XB TB miss microtrap is detected. Unlike the other microtraps, BUT XB TB does not inhibit destinations. The microinstruction is allowed to complete before the microtrap is taken. After execution of the microinstruction, the hardware forces a microtrap (8) to CSA 0029 to get the PTE specified by the PC. The PTE is loaded into the TB and the microtrap routine terminates with BUT/RET.DINH. This pops the microstack, which again (9) causes the microinstruction at A to be executed. This time, however, all destinations are inhibited. Effectively, the only actions that occur are those caused by BUT/IRD1. The next VAX instruction is fetched.

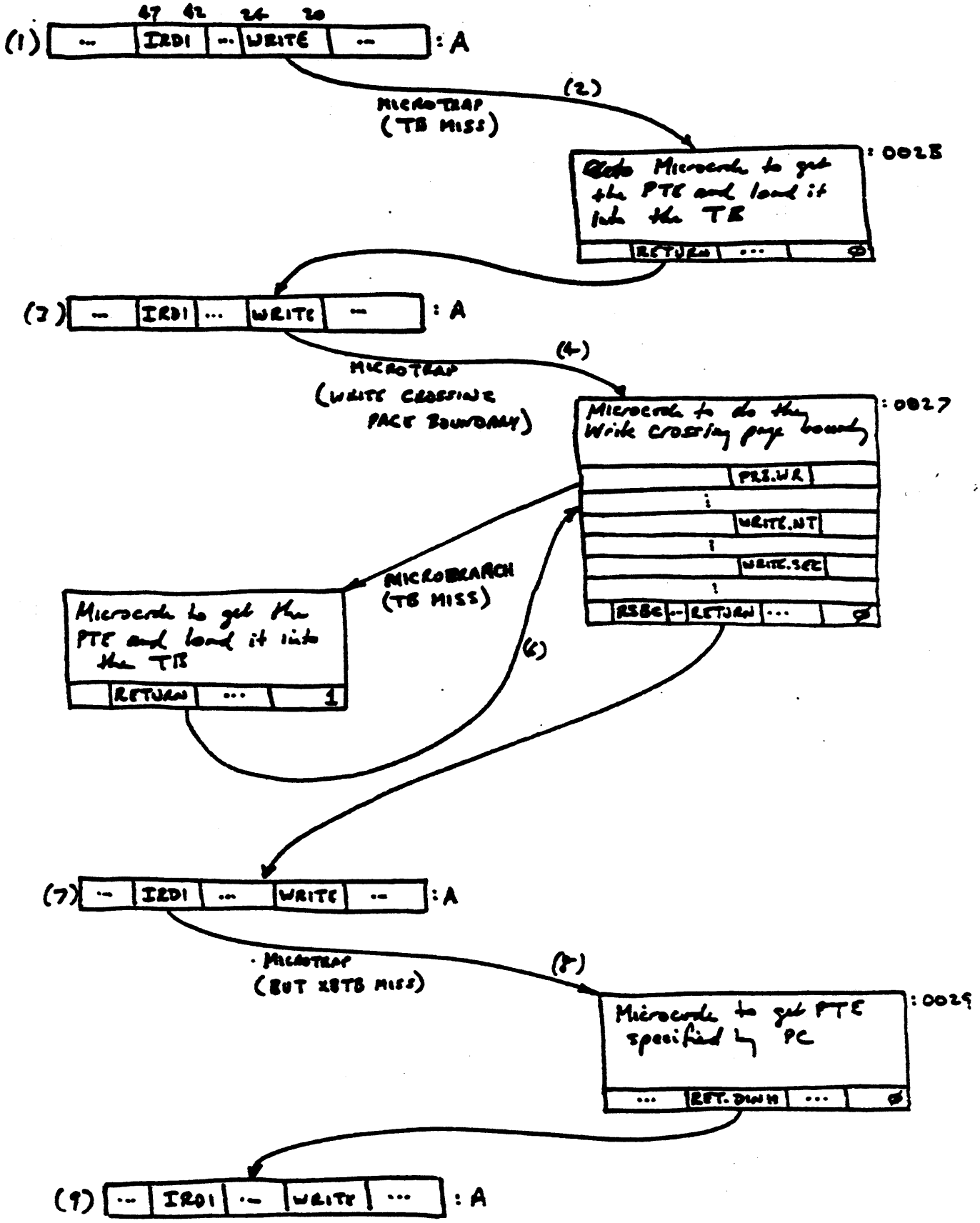


Figure 5.7 Flow of Control (Memory Management Example of Section 5.2.3)

TABLE 5.1 INTERRUPTS

	VAX	IPL(hex)	COMET	IPL	CSA
1.	Device interrupts	10-17	UNIBUS interrupts	14-17	003A
2.	Console	14	Console	14	0039
3.	Interval Timer	18	Interval Timer	18	003B
4.	Recovered errors (Implemenation specific)	18-1D	Corrected Memory Data	1A	003C
5.	Unrecovered errors (Implementation specific)	18-1D	Write Bus Error	1D	003E
6.	Power fail	1E	Power fail	1E	003F
7.	Software	01-0F	Software	01-0F	0038
8.	AST Delivery	02	(see sec. 5.1.3)	02	0038

TABLE 5.2 EXCEPTIONS

VAX	DETECTION IN COMET
1. Arithmetic Traps/Faults	
Integer Overflow	DOSERVICE (0011)
Integer Divide by Zero	MICROBRANCH
Floating Overflow	MICROBRANCH
Floating Divide by Zero	MICROBRANCH
Floating Underflow	MICROBRANCH
Decimal Overflow	DOSERVICE (0011)
Decimal Divide by Zero	MICROBRANCH
Subscript Range	MICROBRANCH
2. Memory Management	
Access Control Violation	MICROTRAP or MICROBRANCH
Translation not Valid	MICROBRANCH
3. During Operand Reference	
Reserved Addressing Mode	MICROBRANCH
Reserved Operand	MICROBRANCH
4. As a consequence of an Instruction	
Opcode Reserved to Digital	IRD1 ROM
Opcode Reserved to Customer	IRD1 ROM
Compatibility Mode	Comp. Mode ROM
Breakpoint	IRD1 ROM
5. Tracing	
T-Bit Trap	DOSERVICE (0015)
6. Serious System Failures	
Kernel Stack Not Valid	MICROBRANCH
Interrupt Stack Not Valid	MICROBRANCH
Machine Check	MICROTRAP (0028)

Table 5.3 Microvector Chart

	UVCTR<3>	UVCTR<2>	UVCTR<1>	UVCTR<0>
BUS/PRB.RD BUS/PRB.RD.MODE BUS/PRB.WR BUS/PRB.WR.MODE	***	*	V.OR.PA	(AC.AND.V).OR.PA
BUS/PRB.RD.PTE BUS/PRB.RD.PTE.K BUS/PRB.WR.PTE	0	**	V.AND.AC	AC
WCTRL/UVCTR_CM,IS	UNDEF	PSL<IS>	PSL<CUR>	
WCTRL/REICHK AND WBUS = SAVED PSL	UNDEF	0	0 = REI CHECK OK 1 = REI CHECK OK & AST 2 = REI CHECK IS NOT OK 3 = REI CHECK IS NOT OK	
WCTRL IS NOT UVCTR_COM.IS OR REICHK(SEE ABOVE) BUS IS NOT ONE THE PROBE MICRO ORDERS(SEE ABOVE)	UNDEF	0 = SOFT INTERRUPT 1 = CONSOLE INTERRUPT 2 = UNIBUS INTERRUPT 3 = INTERVAL TIMER INTERRUPT 4 = CORRECTED MEMORY INTERRUPT 6 = WRITE BUS ERROR INTERRUPT 7 = POWER FAIL INTERRUPT		

LEGEND : M = PTE MODIFY BIT
V = 1 IF VALID PTE
AC = 1 IF ACCESS ALLOWED
PBOK = 1 IF NOT CROSSING A PAGE BOUNDRY
PA = 1 IF MEMORY MAPPING IS NOT ENABLED
* = M.AND.((V.AND.AC).OR.PA)
** = M.AND.V.AND.AC
*** = (PBOK.AND.V.AND.AC).OR.PA

Table 5.4 Memory Management Microtraps

CSA	CONDITION	PRIORITY
0022	MSRC XB TB MISS	highest
0023	MSRC XB ACV	
002A	TB MISS, READ	
002B	TB MISS, WRITE	
002E	ACV, READ	
002F	ACV, WRITE	
0027	WRITE, Crossing PAGE BOUNDARY	
0026	WRITE UNLOCK, Crossing PAGE BOUNDARY	
0021	UNALIGNED DATA READ	
0025	UNALIGNED DATA WRITE	
0024	UNALIGNED DATA WRITE UNLOCK	
0029	BUT XB TB MISS	
002D	BUT XB ACV	lowest

CHAPTER 6. MICROPROGRAMMING EXAMPLES

In this chapter, we show two examples of COMET microcode. The first was taken directly from the VAX emulation; it is the execution flow for the INDEX instruction. The second is an example of what a user might come up with to handle a special task.

6.1 Example 1. The INDEX Instruction.

The VAX INDEX instruction has the following format:

```
INDEX  subscript.rl, low.rl, high.rl, size.rl, indexin.rl,
       indexout.wl
```

Figure 6.1 shows the execution flow for the INDEX instruction after the first two operands have been fetched. The IRD1 ROM causes a branch (cf., Section 2.3) to the OS.RED microcode to evaluate the first operand. The IRDX ROM, with IRDCNT = 0, causes a branch again to the OS.RED microcode to evaluate the second operand. Finally, the IRDX ROM, with IRDCNT = 1, cuases a branch to MS.INDEX to begin the execution flows. At this point, (0 on Figure 6.1), MDR contains the second operand (the lower limit) and Q contains the first operand (the subscript).

Microinstructions (1) - (4) control the evaluation of the remaining four operands. Note that in each case, the branch to OS.RED or OS.WRT1 is specified by the NEXT field as augmented by the addressing mode (BUT/LOD.INC.BRA). The address of the current microinstruction is pushed on the stack (JSR/PUSH), providing for the return mechanism at the end of each operand specifier routine (i.e., BUT/IRDX causes the microstack to be popped when IRDCNT is greater than one). The OS.RED subroutine stores in MDR the operand which is fetched and stores in Q the previous contents of MDR. Ergo, in (1) - (3) the contents of Q are saved before the branch to OS.RED is taken. The OS.WRT1 subroutine does not affect the Q register. It stores in VA the operand address of the destination operand.

```
                                ;20667  .TOC "  Misc. and Queue                  : INDEX INSTRUCTION"
                                ;20668
                                ;20669  ;*************************************************************************
                                ;20670  ;       INDEX   subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl
                                ;20671  ;
                                ;20672  ;       Input           Q               Subscript
                                ;20673  ;                       MDR             Low limit
                                ;20674  ;
                                ;20675  ;       Resources       TEMP4           Pass multiplier to MULSUB.MDR_0
                                ;20676  ;                       TEMP6           Pass multiplicand to MULSUB.MDR_0
                                ;20677  ;                       TEMP7           Save subscript
                                ;20678  ;                       RTEMP9          Save low limit
                                ;20679  ;                       RTEMP10         Save high limit
                                ;20680  ;                       D               Save subscript + indexin
                                ;20681  ;                       Q
                                ;20682  ;                       FLAG3           Set if range trap occurs.
                                ;20683  ;
                                ;20684  ;       Subroutines     OS.RED
                                ;20685  ;                       OS.WRT1
                                ;20686  ;                       MULSUB.MDR_0
                                ;20687  ;                       IE.INDEX.RANGE
                                ;20688  ;
                                ;20689  ;*************************************************************************
                                ;20690
                                ;20691  .REGION/IRDX.R1L,IRDX.R1H/IRDX.R2L,IRDX.R2H
                                ;20692  =000
                                ;20693  MS.INDEX:      (0)
                                ;20694          ;000-------------------------;
                                ;20695          M[TEMP7]_Q,                 ; SAVE SUBSCRIPT.
                                ;20696          LOD INC BRA?,               ;                    (1)
U 0318, 0867,03A4,1BD8,0470,4100 ;20697         PUSH,NEXT/OS.RED            ; FETCH HIGH LIMIT.
                                ;20698
                                ;20699          ;001-------------------------;
                                ;20700          R[TEMP9]_Q Q_D,             ; SAVE LOW LIMIT.
                                ;20701          LOD INC BRA?,               ;                    (2)
U 0319, 4840,02C7,1B24,0470,4100 ;20702         PUSH,NEXT/OS.RED            ; FETCH SIZE.
                                ;20703
                                ;20704          ;010-------------------------;
                                ;20705          R[TEMP10]_Q Q_D,            ; SAVE HIGH LIMIT.
                                ;20706          LOD INC BRA?,               ;                    (3)
U 031A, 4840,02C7,1B28,0470,4100 ;20707         PUSH,NEXT/OS.RED            ; FETCH INDEXIN.
                                ;20708
                                ;20709          ;011-------------------------;
                                ;20710          D_M[MDR]+R[TEMP7],          ; SAVE SUBSCRIPT + INDEXIN.
                                ;20711          LOD INC BRA?,               ;                    (4)
U 031B, C812,0012,1B1C,0470,4140 ;20712         PUSH,NEXT/OS.WRT1           ; FETCH INDEXOUT.
                                ;20713
                                ;20714          ;100-------------------------;
                                ;20715          R[TEMP6]_D,                 ; SETUP MULIPLICAND FOR MULTIPLY.(5)
U 031C, C840,5B24,0219,8C70,0CA0 ;20716         SIZE[LONG],ALUS_SIGND       ;
                                ;20717  =
                                ;20718  .REGION/MISQUE.R1L,MISQUE.R1H/MISQUE.R2L,MISQUE.R2H/MISQUE.R3L,MISQUE.R3H
```


FIGURE 6.1 Execution Flow, INDEX Instruction

```
                                ;20719    =C00     ;C00----------------------------;
                                ;20720             M[TEMP4]_Q,                     ; SETUP MULTIPLIER FOR MULTIPLY.  (6)
                                ;20721             SIGND CMP?,SIZE[LONG],          ;
U 0CA0, C864,03A4,B6D8,0470,4030 ;20722            PUSH,NEXT/IE.MULSUB.MDR_0       ; COMPUTE (INDEXIN + SUBSCRIPT) * SIZE
                                ;20723
                                ;20724    =011     ;011----------------------------;
                                ;20725             R[DST.R]_M[MDR].OR.Q,           ; RESULT IS POSITIVE, USE Q.
                                ;20726             WRITE NOTREG,SIZE[LONG],CCOP2,  ;                                 (7)
U 0CA3, 4852,00A4,02C5,5DA0,0CA1 ;20727            NEXT/MS.INDEX.50                ;
                                ;20728
                                ;20729             ;100----------------------------;
                                ;20730             R[DST.R]_M[MDR]-Q,              ; RESULT IS NEGATIVE, USE -Q.
U 0CA4, 4852,0080,02C5,5DA0,0CA1 ;20731            WRITE NOTREG,SIZE[LONG],CCOP2   ;                                 (8)
                                ;20732    =
                                ;20733    MS.INDEX.50:
                                ;20734             ;-----------------------------;
                                ;20735             WB_M[TEMP7]-R[TEMP9],           ;
U 0CA1, 8807,C000,8624,0470,8C99 374* ;20736       SIZE[LONG],SIGND CMP?           ; IS SUBSCRIPT LESS THAN LOW LIMIT? (9)
                                ;20737
                                ;20738    =01      ;01-----------------------------; SUBSCRIPT IS GREATER OR EQUAL THAN LOW
                                ;20739             WB_R[TEMP10]-M[TEMP7],          ;
                                ;20740             SIGND CMP?,SIZE[LONG],          ;                                 (10)
U 0C99, 8807,0030,8628,0470,8CA5 374* ;20741       NEXT/MS.INDEX.60                ; IS SUBSCRIPT GREATER THAN HIGH LIMIT?
                                ;20742
                                ;20743             ;11-----------------------------; SUBSCRIPT IS LESS THAN LOW LIMIT
                                ;20744             SET FLAG3,                      ; PUSH PC ON TRAPS.
U 0C98, C580,0364,0300,0470,0FBA ;20745            NEXT/IE.INDEX.RANGE             ;                                 (11)
                                ;20746
                                ;20747    =01
                                ;20748    MS.INDEX.60:
                                ;20749             ;01-----------------------------; SUBSCRIPT IS LESS OR EQUAL THAN HIGH
U 0CA5, 0800,0364,1300,0470,03F9 ,20750            IRD1                            ;
                                ;20751                                             ;                                 (12)
                                ;20752             ;11-----------------------------; SUBSCRIPT IS GREATER THAN HIGH LIMIT
                                ;20753             SET FLAG3,                      ; PUSH PC ON TRAPS.
U 0CA7, C580,0364,0300,0470,0FBA ;20754            NEXT/IE.INDEX.RANGE             ;
                                                                                   (13)
```

Figure 6.1 (continued) Execution Flow, INDEX Instruction

A word about the assembler notation in Figure 6.1 is in order. In (4), the statement D_M[MDR]+R[TEMP7] means the following: The ALU is to perform an addition; its sources are the MBUS and the RBUS. The contents of MDR are gated on the MBUS; the contents of Scratch Pad register TEMP7 are gated on the RBUS. The output of the ALU is loaded into the D register.

In (5) and (6) the arguments are set up for the multiply subroutine. The multiplicand is loaded into TEMP6, the multiplier is loaded into TEMP4, and a branch is taken to IL.MULSUB.MDR_0 where the multiplication is performed. Return is to CSA 11F3 if the result of the multiplication is positive, or to CSA 11F4 if the result is negative. The absolute value of the product is stored in Q. Note that in both CSA 11F3 and CSA 11F4, (7) and (8) in Figure 6.1, the product is written to its destination. Since MDR = 0, either 0.OR.Q or 0 - Q is computed in the ALU. The output of the ALU is stored either in DST.R if register mode, or in memory otherwise (cf. Section 4.2.3).

Finally, in (9) - (13), the subscript is checked against the lower and upper bounds. In (9), for example, the subscript (stored in TEMP9) is subtracted from the lower limit (stored in TEMP7), the result is placed on the WBUS, and a branch (BUT-OR, cf. Section 2.1.1) is taken to CSA 1399 or CSA 139B, depending on whether the output of the ALU is positive or negative. Note that the BUT code uses WBUS <31> and WBUS <30> for performing the multiway branch; however, since NEXT<0>=1, the state of WBUS<30> has no affect on the branching.

If both bounds checks pass, (12) is executed. BUT/IRD1 causes the machine to start emulating the next VAX instruction. If either bounds check fails, FLAG3 is set and a branch to IE.INDEX.RANGE is taken to initiate the trap.

6.2 Example 2. A User-Defined Instruction.

We conclude this Introduction to the COMET Microarchitecture with the development of a new instruction

MATCHP pattern.rw, streamaddr.ab, streamlen.rw, count.wl*

---

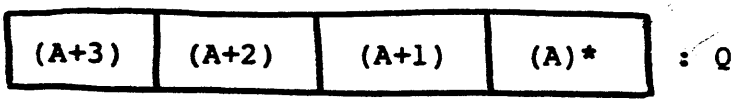
* The operand notation is that used in the VAX System Reference Manual, i.e. "name.ad", where name is a descriptive identifier for the operand, a represents the access type (r=read, w=write, a=address) and d represents the data type (b=byte, w=word, l=longword).

The instruction is to search the bit stream specified by streamaddr (its starting address) and streamlen (its length in bytes), counting occurrences of a 16 bit pattern. To simplify the bookkeeping, we will assume that the length of the bit stream is an exact multiple of 32 bits (i.e., it is contained in an integer number of longwords). The pattern is the first operand. At completion of execution, the number of occurrences is to be stored in count. The bit stream is literally a bit stream; i.e., occurrences of patterns are to be counted if the pattern occurs across a byte boundary.

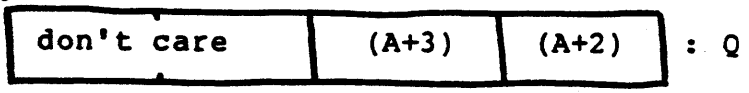
We will assume that common VAX microcode has been used to obtain the four operands, and that our job starts with M[TEMP1] containing pattern (with 16 leading 0's), M[TEMP2] containing streamaddr, Q containing streamlen, and D containing the address of count.

First we study the problem to see if there is some way to exploit the parallelism of COMET microcode in developing our algorithm. Resumably, speed is important, or we would have elected to use VAX's bit string instructions. We note that the crux of the solution involves the following (in this discussion we will refer to stream address as A):

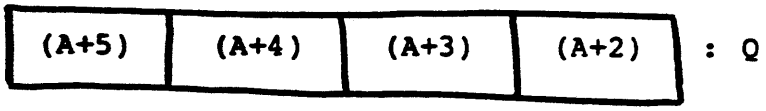
(1.) Load four bytes into Q. Then iterate 16 times the compound operation consisting of comparing the right-mode 16 bits of Q with the pattern, incrementing a counter if they match, and shifting Q right one bit. For example, initially Q contains



Sixteen iterations later, Q contains



(2.) If we next load Q with the longword starting at A+2, the microarchitecture will be ready for another sequence of 16 iterations, this time starting with



---

* (A) = contents of A.

(3.) Sixteen iterations later, we are ready to load Q with the longword starting at A+4.

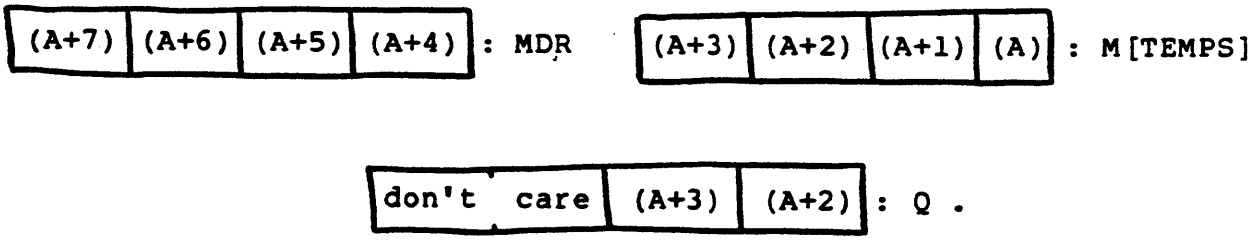
Thus, the Q register can be continually loaded correctly by the following scheme (assume we have already performed the first READ and that MDR contains the result). Note that the scheme we are going to describe allows READs to be scheduled such that, after initialization, we never have to wait for memory accesses; that is, memory access time in 0, independent of the length of the bit stream. The scheme is as follows:

(1) M[TEMP5]<---MDR

(2) Q<---M[TEMP5],READ

This results in the first 32 bits of the bitstream being loaded into the Q register and the second read initiated. Sometime later, MDR will contain the contents of the longword starting at address A+4.

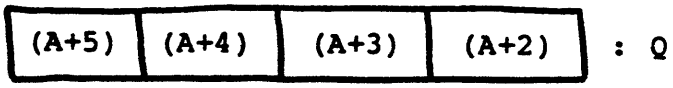
(3) Sixteen shifts and compares later, we have



Q is ready to be updated with the longword starting at A+2. This done by

Q <--- MDR'M[TEMP5], rotated right 16 bits.

There is a ROT micro-order which will do this (cf. Section 4.1), but we don't attend to that level of detail yet. The contents of Q at this time is



(4) Sixteen shifts and compares later, Q is again ready to be updated, this time with the longword starting at A+4. Since that longword is contained in MDR, we return to step 1 above, and the crux of the algorithm has been developed.

Second, we incorporate the above four steps into a flowchart, keeping track of the bookkeeping, in particular the initialization and graceful exit. Figure 6.2 shows the flowchart.

Third, we refine the flowchart, identifying end microinstruction, assigning specific registers to symbolic names, and keeping in mind the BUT OR-ing nature of conditional branches.

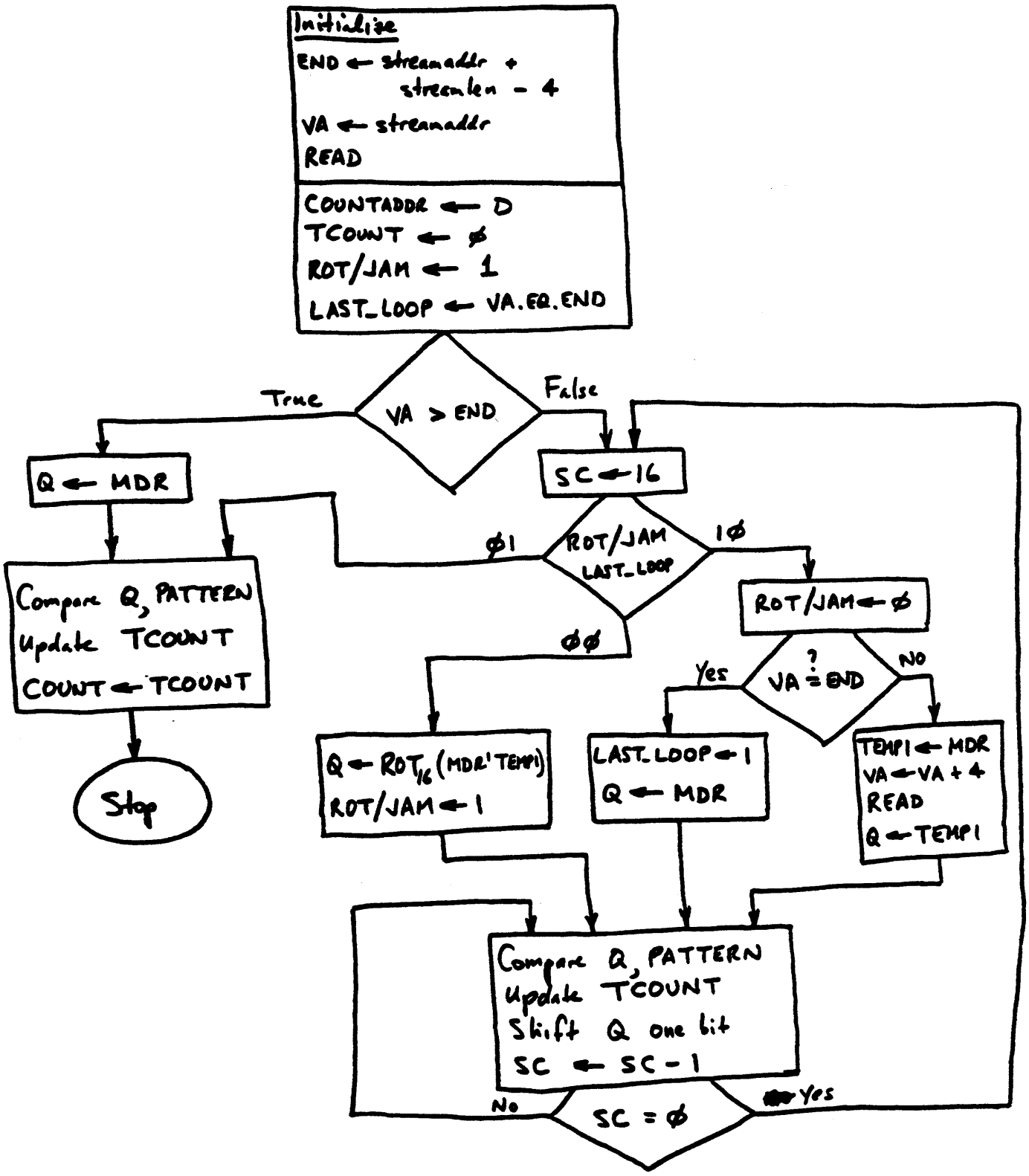


Figure 6.2 Initial Flowchart for the MATCHP Example

Finally, we write the microprogram. The method is to write macros without coding them at the same time, then to go back and define each macro in terms of the microinstruction which implements it. The initial microprogram is shown in Figure 6.3, the macro definitions in Figure 6.4.

We conclude this example with a statement about its performance. After the initialization procedure, memory accesses were done in parallel with the "comparison and testing" microcode. Assuming a 320 nsec microcycle, execution can be accomplished (using Figure 6.5) in

$$2\ 5/32\ n - 23 + (\text{no. of matches})\quad \text{microcycles}$$

An equivalent VAX machine language program to solve the same problem is described below.

```
        ASHL    #3, SIZE, R4
        SUBL2   #16, R4
        CLRL    R1
        CLRL    R3
NEXT:   CMPZV   R1, #16, STRING, R2
        BNEQ    A
        INCL    R3
A:      CMPL    R1, R4
        BEQL    DONE
        INCL    R1
        BRB     NEXT
DONE:
```

In the above program, R1 contains the location of the current 16 bits being compared, as an offset from STRING, R2 contains the zero-extended pattern, R3 contains the COUNT, and R4 contains the number of comparisons which must be made (n - 15). A conservative estimate for the execution of this VAX program for large n is at least 23n microcycles, an order of magnitude slower than the microcoded version.

```
U 00, C042,0090,0320,0470,0002  ;1324          R[TEMP8]_M[TEMP2]+Q              ;STREAM ADDR + STRE   EN
                                ;1325
                                ;1326          ;-------------------------------;
U 02, 0040,73C0,0220,0470,0007  ;1327          R[TEMP8]_RB-CONX(4)              ;END = STREAM ADDR + STREAM LEN - 4
                                ;1328
                                ;1329          ;-------------------------------;
U 07, C002,0024,03D8,4A70,0008  ;1330          VA_M[TEMP2]                      ;LOAD STREAM ADDR
                                ;1331
                                ;1332          ;-------------------------------;
U 08, 9000,EF64,0200,0500,000A  ;1333          READ,SIZE[LONG],                 ;FETCH FIRST 32 BITS OF STREAM
                                ;1334          PL_[16.]                         ;USED LATER FOR ROTATING
                                ;1335
                                ;1336          ;-------------------------------;
U 0A, 0040,5B24,0328,0470,0010  ;1337          R[TEMP10]_D                      ;SAVE ADDRESS OF RESULT
                                ;1338
                                ;1339          ;-------------------------------;
                                ;1340          WB_R[TEMP8]-M[VA],SET FLAG1,     ;CHECK FOR END, SET ROT/JAM FLAG
U 10, 0498,0030,B620,0470,0001  ;1341          SIGND CMP?,SIZE[LONG]            ;
                                ;1342
                                ;1343  =01     ;01-------------------------------;
                                ;1344          R[TEMP11]_0,                     ;ZERO COUNT
U 01, 0040,5B70,032C,0470,000D  ;1345          CLEAR FLAG0,NEXT/OUTER_LOOP      ;NOT LAST LOOP
                                ;1346
                                ;1347          ;11-------------------------------;
                                ;1348          R[TEMP11]_0,                     ;ZERO COUNT
U 03, 0440,5B70,032C,0470,0011  ;1349          SET FLAG0,NEXT/DO_ONLY_ONE       ;ONLY NEED TO DO ONE COMPARE
                                ;1350
                                ;1351  DO_ONLY_ONE:
                                ;1352          ;-------------------------------;
U 11, 4012,5925,0300,0470,0005  ;1353          Q_M[MDR],NEXT/LAST_COMPARE       ;MOVE SOURCE INTO Q
                                ;1354  =0
                                ;1355  INNER_LOOP:
                                ;1356          ;00-------------------------------;
                                ;1357          WB_M[TEMP1]-Q SQR,               ;COMPARE STREAM AND PATTERN 16 BITS
U 0C, C001,00C1,B500,0470,000E  ;1358          SIGND CMP?,SIZE[WORD],           ;
                                ;1359          NEXT/CONT_INNER_LP               ;
                                ;1360
                                ;1361  OUTER_LOOP:
                                ;1362          ;11-------------------------------;
                                ;1363          STEPC_ZLIT0[16.],                ;
U 0D, 1000,C370,5300,1070,0004  ;1364          FLAG<1-0>?                       ;IS IT TIME TO FETCH A NEW LONGWORD?
                                ;1365
                                ;1366  =00     ;00-------------------------------;
                                ;1367          Q_(M[MDR] R[TEMP5]),RR,P,         ;SHIFT IN NEXT 16 BITS OF STRING
U 04, C492,1371,0314,0470,000C  ;1368          SET FLAG1,NEXT/INNER_LOOP        ;SET ROT/JAM FLAG
                                ;1369
                                ;1370  LAST_COMPARE:
                                ;1371          ;01-------------------------------;DO ONE LAST COMPARE
                                ;1372          WB_M[TEMP1]-Q SQR,               ;COMPARE STREAM AND PATTERN 16 BITS
U 05, 4001,00C1,B500,0470,0012  ;1373          SIGND CMP?,SIZE[WORD],           ;
                                ;1374          NEXT/EXIT                        ;
                                ;1375
                                ;1376          ;10-------------------------------;
                                ;1377          WB_M[VA]-R[TEMP8],CLEAR FLAG1,   ;CHECK FOR END, CLEAR ROT/JAM FLAG
```

Figure 6.3 Microprogram for the MATCHP example

```
U 06, 009B,0000,0620,0470,0009    ;1378          SIGND CMP?,SIZE[LONG]              ;
                                  ;1379  =
                                  ;1380  =01     ;01-------------------------------;
                                  ;1381          Q_M[MDR],SET FLAG0,                ;SET LAST LOOP FLAG
U 09, 4412,5925,0300,0470,000C    ;1382          NEXT/INNER_LOOP                    ;
                                  ;1383
                                  ;1384          ;11-------------------------------;
                                  ;1385          R[TEMP5]_Q_M[MDR],VA_VA+4,         ;
                                  ;1386          READ,SIZE[LONG],                   ;
U 0B, C052,5925,0214,4500,000C    ;1387          NEXT/INNER_LOOP                    ;
                                  ;1388
                                  ;1389  =10
                                  ;1390  CONT_INNER_LP:
                                  ;1391          ;10-------------------------------;NO MATCH
U 0E, 4000,0364,3300,0470,000C    ;1392          DBZ STEPC?,NEXT/INNER_LOOP         ; REPEAT INNER LOOP 16 TIMES.
                                  ;1393
                                  ;1394          ;11-------------------------------;MATCH
                                  ;1395          R[TEMP11]_R8+1,                    ;INCREMENT COUNT
U 0F, 0040,E7C0,032C,0470,000E    ;1396          NEXT/CONT_INNER_LP                 ;
                                  ;1397  =10
                                  ;1398  EXIT:   ;10-------------------------------;
U 12, 000A,0024,03D8,4A70,0014    ;1399          VA_M[TEMP10],NEXT/WRITE_RESULT     ;LOAD DESTINATION ADDRESS
                                  ;1400
                                  ;1401          ;11-------------------------------;
                                  ;1402          R[TEMP11]_R8+1,                    ;INCREMENT COUNT
U 13, 0040,E7C0,032C,0470,0012    ;1403          NEXT/EXIT                          ;
                                  ;1404
                                  ;1405  WRITE_RESULT:
                                  ;1406          ;-------------------------------;
U 14, 4000,5BE4,022C,5D80,0000    ;1407          WRITE R[TEMP11],SIZE[LONG]         ;
                                  ;1408
                                  ;1409
```


Figure 6.3 Microprogram for the MATCHP example (continued)

```
CLEAR FLAG0                    "MISC/CLR.FLAG0"
CLEAR FLAG1                    "MISC/CLR.FLAG1"
PL_R[]_RB+Q                    "ROT/SL.PL_WB,RSRC/@1,SPW/RLONG,ALU/A+B+CI,MUX/R.Q"
Q_(M[] R[]).RR.P               "ALPCTL/WX_Q_S,MSRC/@1,RSRC/@2,ROT/RR.MR.P"
Q_M[]                          "DQ1/Q_WX,MSRC/@1,ROT/ZERO,MUX/M.S,ALU/OR"
PL_[]                          "ROT/OLIT0.PL_LIT,LIT/LITRL,LITRL/@1"
P[]_-ZEXT(M[])                 "RSRC/@1,SPW/RLONG,MUX/XM.S,ROT/ZERO,ALU/B-A-CI,ALUXM/ZERO,M

R[]_0                          "RSRC/@1,SPW/RLONG,ROT/ZERO,ALPCTL/WX_S"
R[]_D                          "RSRC/@1,SPW/RLONG,ROT/ZERO,MUX/D.S,ALU/OR"
R[]_M[]+Q                      "RSRC/@1,SPW/RLONG,MSRC/@2,MUX/M.Q1,ALU/A+B+CI"
R[]_Q_M[]                      "RSRC/@1,SPW/RLONG,MSRC/@2,ROT/ZERO,MUX/M.S,ALU/OR,DQ1/Q_WX"
R[]_RB+1                       "RSRC/@1,SPW/RLONG,ROT/MINUS1,MUX/R.S,ALU/A-B-CI"
P[]_RB-CONX(4)                 "RSRC/@1,SPW/RLONG,ALU/A-B-CI,MUX/R.S,ROT/CONX.SIZ,VSIZE/1,D

STEPC_ZLIT@[]                  "WCTRL/STEPC_WB,ALPCTL/WX_S,ROT/ZLIT0,LIT/LITRL,LITRL/@1"
VA_M[]                         "WCTRL/VA_WB,MSRC/@1,RSRC/ZERO,MUX/M.R1,ALU/OR"
VA_VA+4                        "WCTRL/VA_VA+4"
WB_M[]=Q SQR                   "MUX/M.Q2,MSRC/@1,ALU/A-B-CI,DQ2/SQR"
WB_M[]=R[]                     "MUX/M.R1,MSRC/@1,RSRC/@2,ALU/A-B-CI"
WB_R[]=M[]                     "RSRC/@1,MSRC/@2,MUX/M.R1,ALU/B-A-CI,ALUCI/ZERO"
DBZ STEPC?                     "BUT/DBZ.SC"
FLAG<1-0>?                     "BUT/FLAG1TO0"
SIGND CMP?                     "BUT/CCBR,CC/NOP.CCBR_SIGND"
SET FLAG0                      "MISC/SET.FLAG0"
SET FLAG1                      "MISC/SET.FLAG1"
SIZE[]                         "VSIZE/1,DTYPE/@1"
READ                           "BUS/READ"
WRITE R[]                      "BUS/WRITE,WCTRL/WDR_WB,RSRC/@1,ALU/OR,MUX/R.S,ROT/ZERO"
```

Figure 6.4 Macrodefinitions for the MATCHP example

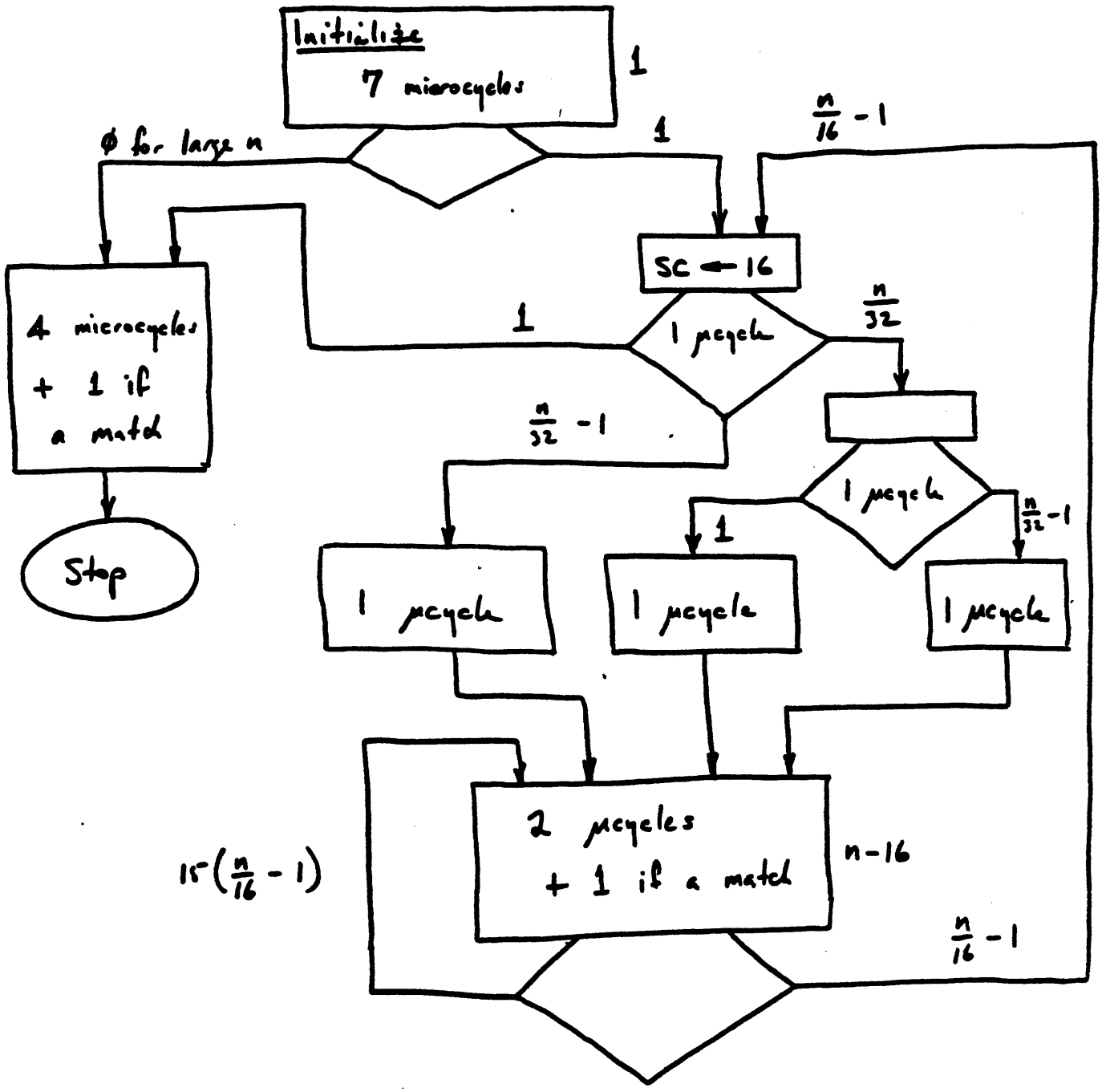


Figure 6.5 Analysis of the Execution Time of MATCHP

APPENDIX

ACRONYMS AND THEIR MEANINGS

ACV	Access Control Violation; specified by the VAX Memory management system.
ALPCTL	A 10 bit field of the microinstruction which specifies one of 50 special functions to be performed by the ALU.
ALU	4 bit microinstruction field, controls the function of the ALU.
ALUCI	2 bit microinstruction field, specifies the carry input to the ALU.
ALUSHF	3 bit microinstruction field, controls (with the DQ field) the shifting of the ALU output.
AMUX	Multiplexer which controls the A port of the ALU.
AST	Asynchronous Systems Trap (VAX architecture).
ASTLVL	A two bit internal register; specified by the VAX architecture, designates the most privileged access mode for which an AST is pending.
ATCR	Arithmetic Trap Control Register; specified by the VAX architecture, contains a code indentifying the nature of the condition which caused the Arithmetic Trap.
BMUX	Multiplexer which controls the B port of the ALU.
BUT	6 bit microinstruction field, specifies the method for obtaining the next microinstruction.
CMI	The 32 bit COMET memory bus.
CSA	Control Store Address, the address of a microinstruction.
D REG	32 bit register, destination for ALU output.
DOSERVICE	hardware routine which checks for traps and interrupts.
DSIZE ROM	A 2k by 2 bit ROM containing the data type of each operand of each VAX instruction.

FC	VAX opcode reserved for customer use.
FPD	First part done. A bit in the PSL; set when a unrecoverable destination has been written into.
ICR	Internal Count Register (VAX architecture).
IICR	Internal ICR; the low order 16 bits of ICR.
INIR	Internal NIR; the low order 16 bits of NIR.
IPL	Interrupt Priority Level (VAX architecture).
IR	8 bit COMET register, used to store the opcode of the VAX instruction being emulated.
IRD1 ROM	1k by 8 bit ROM used for obtaining the CSA of the first microinstruction in the emulation of a VAX instruction.
IRDCNT	A 3 bit COMET register; contains the number of the operand being evaluated in the emulation of a VAX instruction. Part of the index into the DSIZE ROM.
IRDX ROM	2k by 14 bit ROM used for obtaining the CSA of the first microinstruction in an operand specifier routine.
JSR	One bit microinstruction field; when set it pushes the current CSA on the microstack.
LIT	2 bit microinstruction field, used for bit steering the immediate LITRL (9bits) and LONLIT (32 bits) fields.
LONLIT	32 bit literal field in the microinstruction; enabled (bit steering) by LIT field.
MBUS	32 bit COMET bus; memory data and M Scratch Pad registers are major sources.
MDR	COMET register, destination of a memory read.
MSP[i]	M Scratch Pad Register i.
MSRC	4 bit field of the microinstruction, used to specify the source gated onto the MBUS.
MUX	4 bit microinstruction field, controls the sources to the ALU.
NIR	Next Interval Register (VAX architecture).

OSR	8 bit COMET register, used to store the operand specifier being evaluated in the emulation of the current VAX instruction.
PSL	32 bit processor status longword; defined by VAX architecture.
PTE	Page Table Entry; defined by VAX memory management system.
Q REG	32 bit register, associated with ALU.
RBS	Register Back-up Stack.
RBSP	Register Back-up Stack Pointer.
RBUS	32 bit COMET bus; R Scratch Pad registers are major sources.
RNUM	4 bit COMET register, used to store the VAX general purpose register which is being addressed.
ROT	Six bit field of the microinstruction, used to control the function of the Super Rotator.
RSP [i]	R Scratch Pad register i.
RSRC	6 bit field of the microinstruction, used to specify the source gated onto the RBUS.
SCB	System Control Block. A set of longwords, one for each exception and interrupt; each contains the starting address of the corresponding service routine and whether to service it on the kernel stack, interrupt stack or in WCS.
SPASTA	Scratch Pad Address Status; 2 bits, used in microsequencer control.
SPW	Scratch Pad Write; Two bit field of the microinstruction, used to control writing to the Scratch Pad registers.
SRKSTA	Super Rotator Control Status; 2 bits, output of the Super Rotator, used in microsequencer control.
TB	Translation Buffer; a cache for PTE's; part of VAX architecture.
TCSR	Timer Control and Status Register (VAX architecture).

TNV	Translation Not Valid; specified by the VAX memory management system.
USTK	The 16 deep COMET microstack, part of the microsequencer.
USTKP	Microstack pointer, used to address the COMET microstack.
VA	32 bit virtual address; specified by the VAX architecture.
WBUS	Main COMET Internal bus, 32 bits.
WDR	COMET register, source of a memory write.
XB	Execution Buffer. Provides storage for prefetching eight bytes of the Instruction Stream.