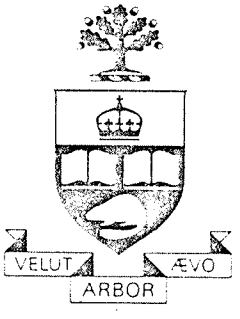
COMPUTER ENGINEERING

VASTOR 1978

W.M. Loucks and W.M. Snelgrove
University of Toronto

Computer Engineering Report 13

UNIVERSITY OF TORONTO



DEPARTMENT OF
ELECTRICAL ENGINEERING

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter I

INTRODUCTION TO THE MACHINE

1.1 TYPE OF COMPUTATION

VASTOR is a word-parallel, bit-serial computer. It can, for instance, logically AND bits 37 and 4A (all numbers herein are hexadecimal wherever an excuse may be found) of all words of memory and place the result in, say, bit 52.

Adjacent bits may be organised into fields within the rather large words VASTOR uses (1K bits at the moment), and the machine made to operate serially on the bit positions of the fields.

On the present machine, for instance, one could logically OR two fields bit-by-bit by using a FORTRAN call: e.g.

```
CALL FOR( WIDTH, SRC1, SRC2, DST)
```

1.2 STRUCTURE OF THE MACHINE

The idea with this machine is that overhead for control circuitry is amortized over a large number of words, so that the dominant cost is that of memory. It is therefore important to keep the complexity of backplane wiring and control information down, and we have therefore gone to some trouble

to limit the class of communication between system components to two types: 'bussing' and 'daisy-chaining'. All memory address lines, for instance, are common and driven from a central controller, as are opcodes for the 'Industrial Control Units' (MC14500B 'ICU') which perform logical operations on each word.

The repeating unit - the word - in this machine takes about 2 1/2 chips, 16 of these fit on a board, with space left over for a CCD backing store of 16K bits.

Provision is made for expansion by extending the 1-bit data wires for all 16 words on a board to an adjacent connector in the card cage. We have several ideas in progress for such expansion boards, which provide the means of tailoring the machine to unusual applications.

1.3 DATATYPES

We can treat fields somewhat differently and perform arithmetic or character-oriented operations on them: e.g.

```
CALL VPV( WIDTH, SRC0, SRC1, C IN, DST)
```

adds the two SRC fields and a carry-in bit into DST.

This machine is naturally vector-oriented, and we therefore use APL notation to describe its operation. The examples so far have been element-by-element extensions of simple scalar operations to vectors.

1.4 THE RESULT REGISTER

The ICU in each word contains a flip-flop called the result register (RR) which is used as a 1-bit accumulator. The boolean vector formed by these can be summed ('+/RR') to a finite resolution, this information allowing the central controller to branch on such interesting conditions as '0=+/RR'.

CALL MATCH(WIDTH, SCALAR, SRC)

uses this sum to associatively search vector SRC for SCALAR.

1.5 IEN AND OEN

The ICU contains two more bits of storage from which come the input enabling vector (IEN) and the output enabling vector (OEN). OEN allows selective writeback: stores to memory will only succeed in those words in which OEN is set. IEN is AND'ed with all input data to the logic apparatus. These vectors are therefore usually set up to mask away words that should not be affected by the current operation. This is the principal way to operate differently on different words.

1.6 INTERWORD COMMUNICATION

A truly associative machine imposes no order on the words contained in it. This can be inconvenient (as when two words come to have the same contents and can nevermore

be distinguished), so VASTOR orders them with a shift register (SH). This mechanism allows indexing, and therefore sorting and so forth, and incidentally makes I/O possible.

Some useful APL expressions involving SH are: '+/A', '+ A' and '1|A'.

1.7 A TYPICAL WORD

The elements described above are interconnected as shown in figure 1.

1.8 INPUT/OUTPUT

Data may enter SH a bit at a time from the controller, but this is not an adequate channel for large amounts. In a rare concession to reality, a data path was designed for 8-bit bytes from a scalar computer. Figure 2 depicts the details of the connection of SH to an outside world.

The SHifter has two ports: B, connected to the 1-bit data path in the word and A, which is the means of external access. Eight data lines serve the A port, so that words are grouped in eights: thus if we can load successive 'phrases' (8 word groups) with bytes of data, and then load a phrase of data into 8-bit fields of a word in each phrase, we will have done 1/8 of the work involved in transferring data by bytes. The shifters chosen are (coincidentally) 8 bits long, so that a shifter corresponds to a phrase.

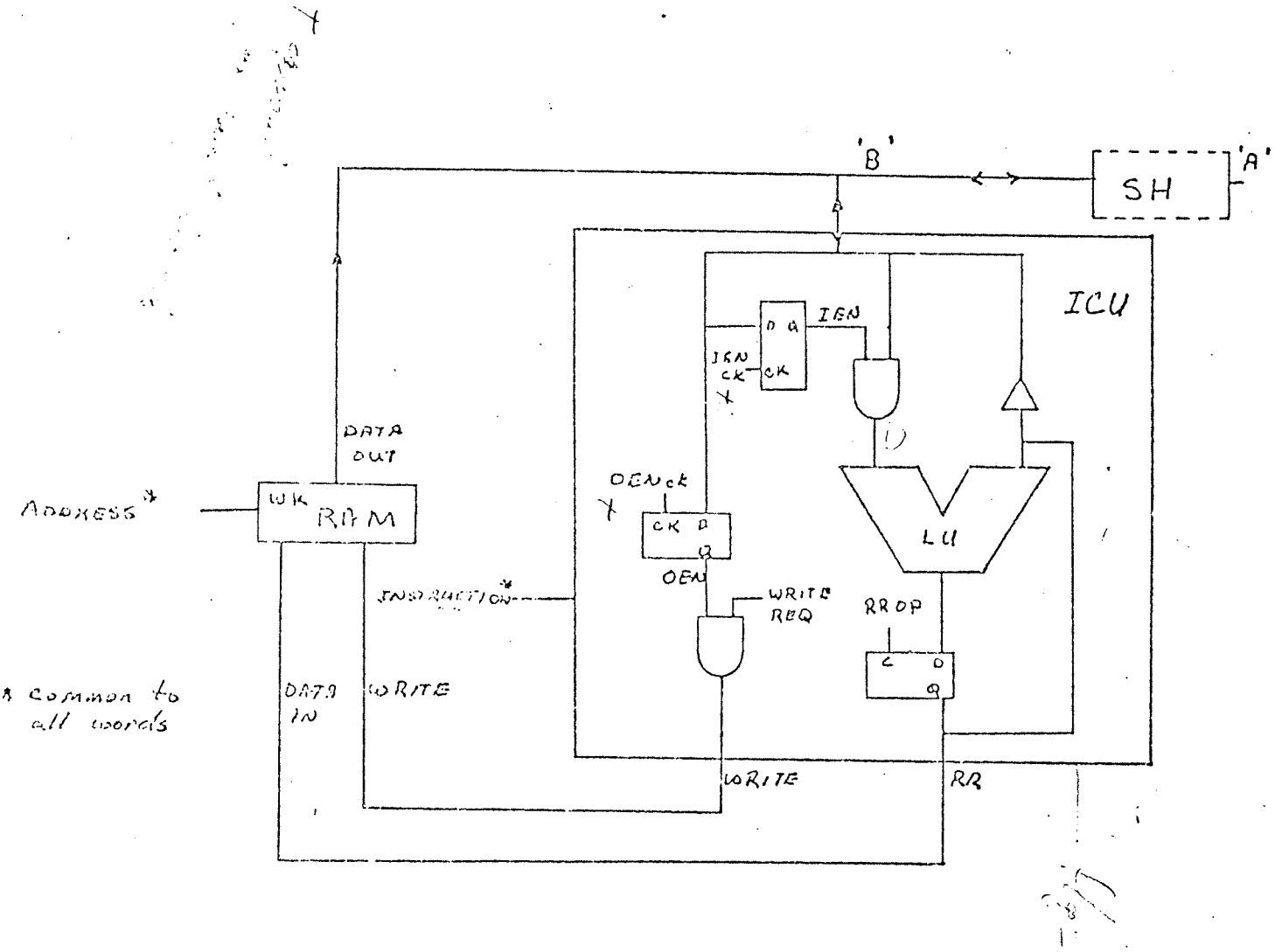


Figure 1: The word of VASTOR

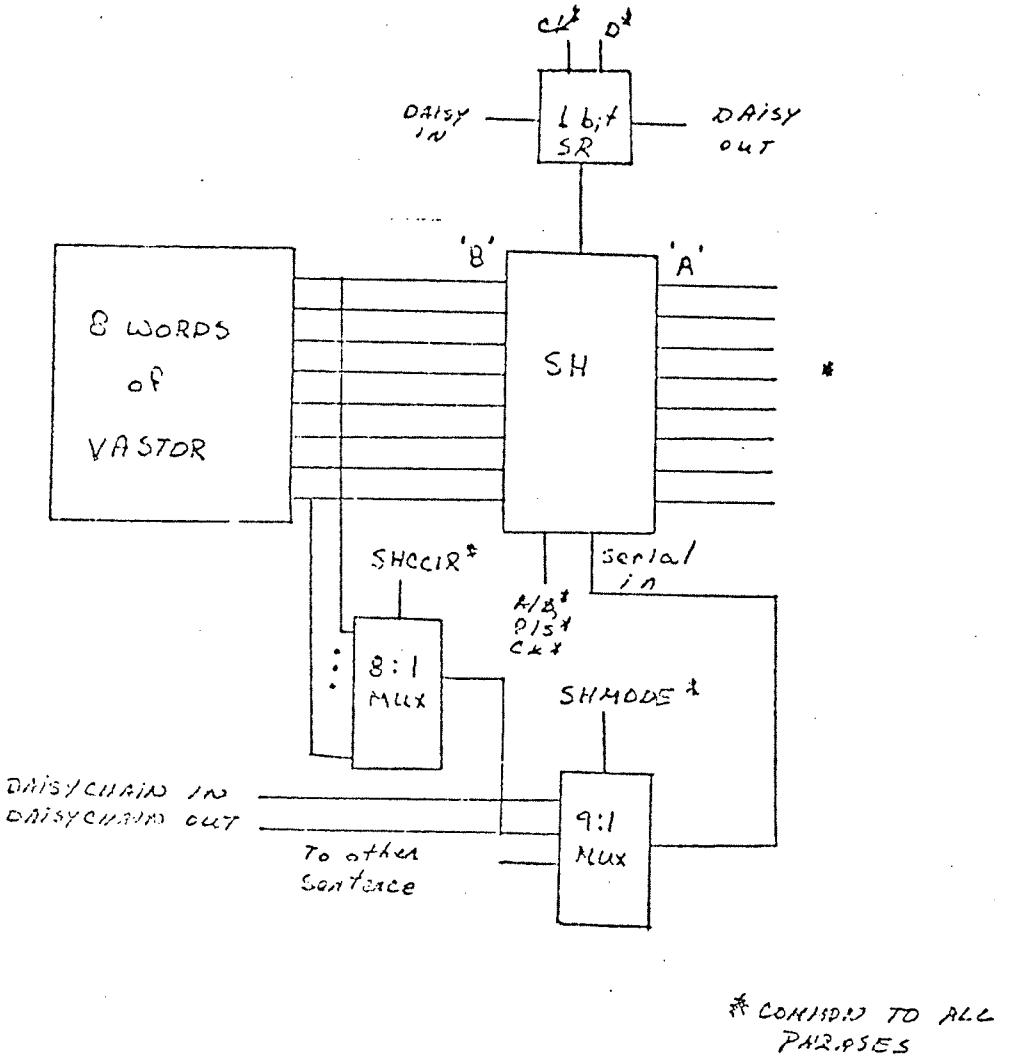


Figure 2: 8-Bit I/O Port to Vastor

In order to load but a single phrase (i.e. a single shift register) at a time, another - faster- shift register is used to distribute control. This shifter (hereinafter LSR, after the famous physicist,Low Power Schottky) controls a line on SH that enables its 'A' side. If only 1 bit is set in LSR, then only the corresponding phrase of SH may be loaded.

In order to read data by bytes, we must reverse the above process: we first read an 8-bit field from one word out of each phrase into the top of its SHifter, and then read bytes out of successive phrases to the scalar machine via the 'A' bus (again by shifting a single '1' through LSR).

Two multiplexers are used to achieve this - those labelled SHCCIRC (SHifter Column CIRCulate) and SHMODE (SHifter MODE) in figure 2. SHCCIRC makes any chosen word per phrase available to SHMODE, which then may pass it into the top of SH.

SHMODE selects whether SH behaves (on being clocked in serial mode) as a boolean vector, an eight column boolean matrix, or a sixteen column boolean matrix. SHCCIRC can confuse the issue, but the modes mentioned are the ones known to be useful.

1.9 FLAMES

Due to the extensive use of 3-state devices it is quite possible for several drivers to attempt to assert themselves simultaneously on the same wire. Hardware is installed to protect the system from this situation, but the side effects of an attempt to induce it are unpredictable and may destroy data, especially that in SH.

The first bus on which this can occur is 'A': if more than one shifter is enabled to drive that bus a conflict could occur. A 2-bit counter decides whether more than one '1' has been clocked into LSR since it was last cleared. If so, it uses a common line to prevent any shifter from writing to 'A'.

The second place for a conflict is on the B side of the shifter: memory, ICU and SHifter are all capable of driving this node. An 'enable' line exists for the memory, so that it is relatively easy to turn off; the ICU may be driving this node if the last opcode was some kind of 'STORE'; and the shifter must drive either 'A' or 'B', except that LSR can force the 'A' outputs high-impedance. If more than one '1' is in (or thought to be in) LSR, the HCF (halt and catch-fire) detector will force SH to drive 'B', and therefore forbid 'STORE' opcodes and memory references.

1.10 CONTROLLER STRUCTURE

Figure 3 shows the hierarchic position of the simple-minded microcontroller that drives VASTOR. The controller that actually exists is yet simpler, and relies heavily on the PDP-11/34 to emulate the one we describe.

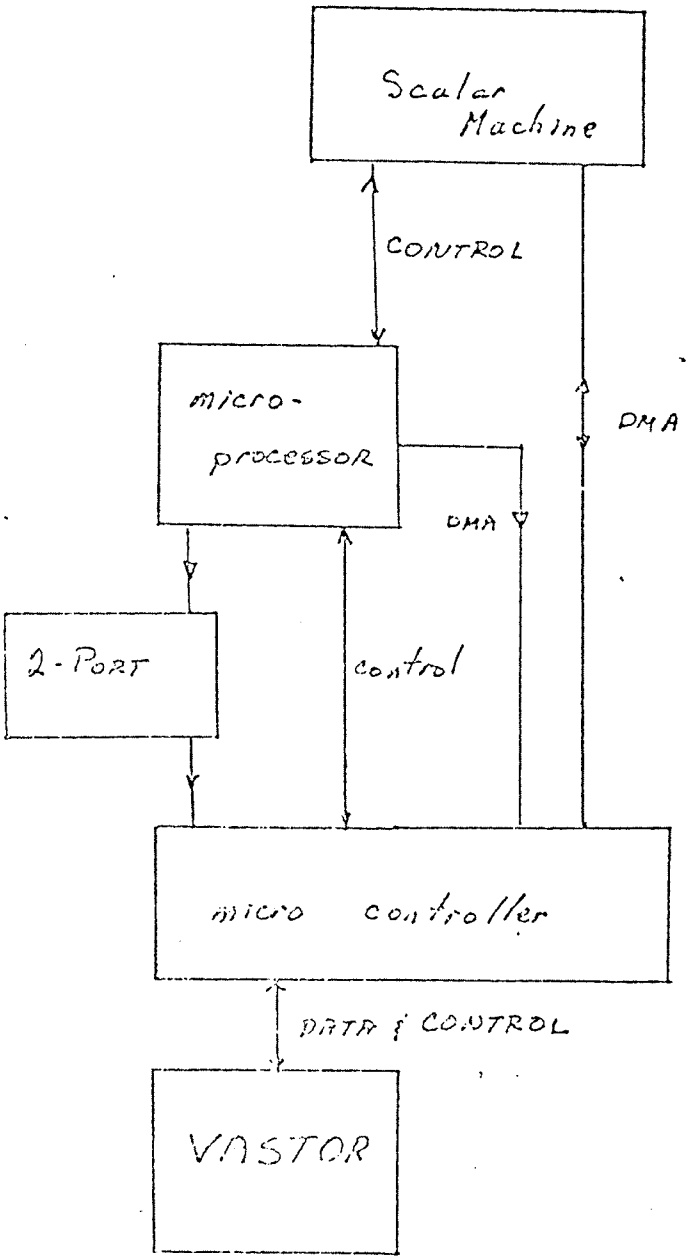


Figure 3: Control hierarchy

1.10.1 ROM

There exists a read-only memory (ROM) whose bits correspond approximately to wires on the VASTOR backplane, and in which are stored the signal sequences required to perform

desired operations. Note that the 'ROM' is at present really a table in the 11/34.

Before hardware came APL: we wrote a simulator for the word structure planned, keeping in mind available IC's, and developed a control structure (entirely in APL) to allow us to test early algorithms. This has since turned into a microcode development system described more fully in chapter 4.

1.10.2 ROM start addresses

The 'microprocessor' shown in figure 3 makes up starting addresses (and a lot of other things) for the controller and puts them in an area of two-port memory (this byte is called ROMADD[1]) from which the controller retrieves them when ready. This scheme allows the microprocessor to work on preparing the next start address (and those other things) while VASTOR beats bits.

1.10.3 Microcode loop control

The microcontroller has elementary looping ability: a counter (UCTR, pronounced microcounter) loaded by the microprocessor is decremented by a special microword field, at which point a new ROM address is obtained from ROMADD[2] - another byte of 2-port memory - or, if the counter reaches -1, a new operation is started.

The main advantage of UCTR is that fields of bits may be dealt with by a single microprocessor command: UCTR can be made to modify the address used for WK (the VASTOR word memory).

UCTR is 8 bits wide, and loops may therefore be executed between 1 and 100 (hex, remember) times.

The microcontroller also has a type of data-dependent branch according to the approximate number of responders (1's in RR). Table 1 shows the relationship between condition code and number of reponders: each microword may test this code against a reference, and branch to either of two places according as the condition is or is not less than the reference.

Table 1.

Branch conditions

code	meaning
7	1/2 <= #RR
6	1/4 <= #RR < 1/2
5	1/8 <= #RR < 1/4
4	1/16 <= #RR < 1/8
0	0 <= #RR < 1/16

where #RR is the fraction of RR's responding.

ROM actually contains pointers into ROMADD (part of the 2-port) which in turn gives new ROM addresses. ROMADD[0] is special, and just continues with the next line of microcode; ROMADD[1] is special in that execution starts there; and reference to ROMADD[2] has the side-effect of decrementing UCTR, which in turn may cause an exit if UCTR goes negative. ROMADD has a few normal addresses: 3 to 15.

1.10.4 WK addressing

Figure 4 shows how we generate addresses for working store (WK): each ROM word has a 2-bit pointer into two more areas of 2-port, which we call BASES and DIR. The address produced at any iteration is BASES[I]+DIR[I]*UCTR, where DIR may be -1,0, or 1. This allows an algorithm to work across fields either MSB (most significant bit) or LSB (least significant bit) first or to repeatedly refer to the same bit.

1.10.5 I/O

The A-bus may take data either from the main scalar machine via a direct memory access channel or from the microprocessor. Data on the A-bus may either go to the scalar machine (again by DMA) or be fed one bit at a time into the 'constant' line.

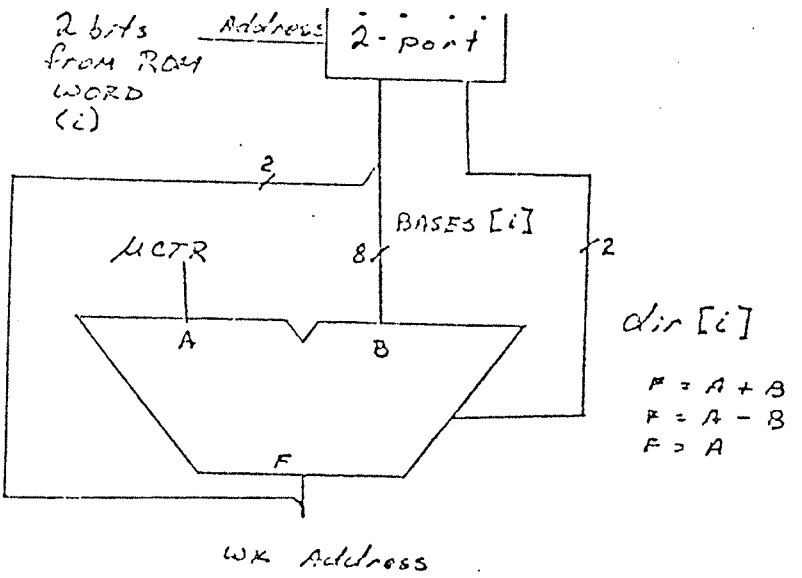


Figure 4: Working store address generation

1.11 PROPOSED MICROPROCESSOR

The microprocessor of figure 3 serves to translate the commands of the scalar machine into sequences of starting addresses and the like. This function may also be performed in software on the scalar machine, leaving the distinction between machines metaphysical, if the extra transactions to the 2-port memory may be grafted onto it.

Very little of this software is written, (that little documented in chapter 2) but we expect that such functions as allocating fields to variables and dealing with datatype properties (like proper sign extension for integers) will appear here.

Chapter II

FORTRAN INTERFACE ROUTINES

These routines are provided to allow FORTRAN programmers to use VASTOR on the 11/34. They are a mixture of microprocessor and scalar machine level routines.

The following sections describe the calling protocols for these routines. Some types of argument recur and so are explained here.

'RHO' is the length of a DMA transfer, and therefore of a vector. Routines that transfer bytes of data use it to count the number thereof.

'WIDTH' is often used as the starting value for UCTR, and should be set to one less than the width of the data fields for such operations as adding vectors.

Arguments like 'SRC' and 'DST' refer to LSB's or MSB's (according to the algorithm in use) of fields of data. Since words are 1K bits, SRC and DST may conceivably range over 0-3FF.

2.1 CALL IEN G SH()

This call results in the contents of SH being copied into the input enabling vector. Remember that IEN will affect almost all other operations, and so should be set appropriately before using them. Most routines do not have the brains to set or change IEN.

2.2 CALL OEN G SH()

This routine is to the output enabling vector as its predecessor to IEN. OEN must be set for any operation involving 'STORE's.

2.3 CALL XFRS(RHO, DATA)

RHO/8 elements of vector DATA are moved to SH from the 11/34. Each byte of DATA goes to fill a phrase, with its LSB in the first word (recall that SH defines the order).

2.4 CALL XFRV(RHO, DATA)

This operation is the converse of XFRS: RHO/8 bytes of data from SH appear in FORTRAN vector DATA.

2.5 CALL BOO R(SRC)

This reads a boolean vector (i.e. a single bit/word) from WK address SRC into SH.

2.6 CALL BOO W(DST)

This is inverse to BOO R: a boolean is written from SH to WK[;DST]. IEN and OEN must be set for this to work.

2.7 CALL F AND(WIDTH, SRC1, SRC2, DST)

This routine performs bit-by-bit logical 'AND' and places the result in DST. IEN and OEN must be set. Algorithm is LSB-last, so give the LSB location.

2.8 CALL F OR(WIDTH, SRC1, SRC2, DST)

This is the 'OR' analogue of F AND.

2.9 CALL F EQU(WIDTH, SRC1, SRC2, DST)

This tests for equivalence (inverted exclusive or) after the fashion of F AND.

2.10 CALL F NOT(WIDTH, SRC, DST)

This logically inverts field 'SRC' and places the result in DST. IEN and OEN must be set. Algorithm is LSB-last.

2.11 CALL XFR B S(RHO, DATA, SRC)

This performs the rather convoluted byte-oriented I/O touched on in chapter 1. RHO bytes of DATA are moved from the scalar machine to field 'SRC', LSB last. IEN must be set. If you wish to do non-associative things, be warned

that DATA is permuted a little on the way in: because bytes are written to successive phrases, a 10 long vector would go into words 0,8,1,9,2,3,4,5,6,7 of a 2-phrase machine.

2.12 CALL XFR BV(RHO, DATA, SRC)

This is the inverse of XFR BS. DATA is again permuted, such that they are inverse in this too.

2.13 CALL VPV(WIDTH, SRCA, SRCB,SRC C IN, DST)

This performs element-by-element addition of SRCA and SRCB into DST, using SRC C IN as carry-in and as a temporary for carry between stages. IEN and OEN must be set. Algorithm is MSB-last (backwards from XFR BS etc., so remember to add WIDTH-1 to SRC A, SRC B and DST). SRC C IN ought to be a zero vector if you don't want a carry-in, of course.

2.14 CALL SH G LRG(WIDTH, SRC, TEMP)

This routine puts the largest number found in the vector SRC into SH so that it can be read by operations such as XFRV. This is an LSB-last operation.

Chapter III

APL

Three APL workspaces evolved during the design of VA-STOR: SIMULA, EMU and ROMVAS. The first of these is now outdated, and was a simulator for hardware that now exists. It was used for early system architecture development, and eventually able to load and add 10-length vectors of 8-bit integers. The latter feat took about 10 seconds of CPU time, so development since then has been done on real hardware.

EMU was the next to be written, and drove the PDP-11/34 as a smart modem to allow control of the hardware down to the phasing of individual clocks. A simple-minded controller (described in chapter 5) allowed a DR11-C on the 11/34 to drive backplane wires.

ROMVAS has much in common with EMU, but uses its low-level functions to transfer microcode developed with EMU's debugging facilities to a FORTRAN programme running on the 11/34, from whence it could eventually be copied to real PROMs.

Both EMU and ROMVAS can be induced to call on FORTRAN in the 11/34 to execute a given piece of its version of microcode, rather than APL's.

Most functions and variables in the APL workspaces have underscores somewhere in their names, so that one may work with medium impunity by avoiding them.

EMU and ROMVAS contain several groups (APL sense): READONLY, SIMULATOR and TRANSFER. These names are more historical than reasonable. The functions of the various groups are described below.

3.1 READONLY

The functions in this group are identical in the two workspaces: the workspaces are differentiated by the behaviour of a small number of functions called by them.

3.1.1 UP

This function is a degenerate simulation of the microprocessor. It has sections corresponding to most of the FORTRAN routines described in chapter 2.

Table 2 gives the opcodes UP expects for the various functions to be performed. UP takes a single right argument consisting of a vector of opcodes followed by the arguments

appropriate to them. RHO is a global variable used by UP for the same functions as RHO in the FORTRAN routines. UP is not particularly bright about WIDTH, and in fact specialised thoroughly to bytes.

Table 2.

UP codes

opcode	FORTRAN subroutines	order of arguments
1	VPV	SRC A, SRC B, SRC C IN, DST
2	XFRS	DATA
3	XFRBS	DST, DATA
4	XFRV	-
5	XFRBV	SRC
6	IEN G SH	-
7	OEN G SH	-
8	BOOW	DST
9	BOOR	SRC
10	SHG LRG	SRC, TEMP, 6
10	MRK LRG	SRC, TEMP, 0 (not yet in FORTRAN)
11	-	set RHO to argument

Figure 5 is an APL listing of UP. Our apologies.

```
U OF TORONTO SHARP APL 07/JUN/1978 - PAGE 11

)SIO * 24193 ROUVAS * SAVED  14.36.30 07/JUN/1978

    ∇ UP STRING;CTRS
[1]  ROWS←1↑⍴(⍴POLY)↑10⍴1
[2]  PARSE:→(0,VRV,XCRS,XFRBS,XFRV,XFRHV,IFN,OBN,HOOH,HOOB,LARGEST,ROF,RESE,NEXTL)[13⌊1↑STRING]
[3]    →(FOR,FAND,FEQU,FNOT)[¯13+1↑STRING]
[4]  VRV:PAGE←0 ⋄ ROWADD← 0 0 ⋄ 'VRV'
[5]    BASES←4↑STRING,1↑STRING ⋄ UCTR←7 ⋄ DIR← ¯1 ¯1 0 ¯1
[6]  ⋄ IFN←OBN←(⍴IFN)↑10⍴1 ⋄ THAT SHOULD HAVE BEEN DONE BY ALLOC.
[7]    ROW ⋄ STRING←4↓STRING ⋄ →PARSE
[8]  XFRS:DMATRS←2↑STRING←1↓STRING ⋄ STRING←2↓STRING ⋄ 'XFRS'
[9]    PAGE←0 ⋄ ROWADD← 10 12
[10]   UCTR←1 ⋄ ROW ⋄ →PARSE
[11] XFRD:CTRS←⍎3↓S,17 ⋄ DIR←,1 ⋄ BASES←1↑STRING←1↓STRING ⋄ STRING←1↓STRING
[12]   'XFRBS'
[13]   PLACE←1 ⋄ DMATRS←BHO↑STRING ⋄ STRING←BHO↓STRING ⋄ SUCCINC←7
[14] XFRBSLP:UCTR←11↑CTRS ⋄ →(0=⍴CTRS←11↓CTRS)⍴PARSE
[15]   'XFRBSLP ';PLACE
[16]   PAGE←0 ⋄ ROWADD← 14 22 ⋄ ROW
[17]   →(PLACE↑12 8)⍴NEXTL ⋄ ROWADD← 24 25 ⋄ UCTR←8-⍴CTRS ⋄ ' ROTATE XFRD' ⋄ ROW
[18]   ROWADD←((26 27)[(24=1↑ROWADD)],27 ⋄ UCTR←7 ⋄ 'DO STORES' ⋄ ROW
[19]   PLACE←PLACE×2 ⋄ →XFRBSLP
[20] XFRV:STRING←1↓STRING ⋄ ' XFRV'
[21]   PAGE←0 ⋄ ROWADD← 30 32
[22]   UCTR←1 ⋄ ROW ⋄ DMATOS
[23]   DMATOS←,0 ⋄ →PARSE
[24] XFRHV:CTRS←VARX ⋄ SUCCINC←¯1 ⋄ BASES←1↑STRING←1↓STRING
[25]   STRING←1↓STRING ⋄ DIR←,1
[26] XFRHVLP:→(8<SUCCINC←SUCCINC+1)⍴DUMPIT
[27]   PAGE←0 ⋄ ROWADD← 34 35 ⋄ UCTR←7 ⋄ ' SHIFT ⍷' ⋄ ROW
[28]   'READ SH'
[29]   ROWADD← 36 36 ⋄ UCTR←CTRS[SUCCINC] ⋄ ROW ⋄ →XFRHVLP
[30] DUMPIT:DMATOS ⋄ DMATOS←,0 ⋄ →PARSE
[31] IFN:ROWADD← 40 40 ⋄ UCTR←0 ⋄ PAGE←0 ⋄ 'IFN' ⋄ ROW ⋄ STRING←1↓STRING
[32]   →PARSE
[33] OBN:ROWADD← 41 41 ⋄ UCTR←0 ⋄ PAGE←0 ⋄ 'OBN' ⋄ ROW ⋄ STRING←1↓STRING
[34]   →PARSE
[35] HOOH:BASES←1↑STRING←1↓STRING ⋄ STRING←1↓STRING ⋄ ROWADD← 42 42 ⋄ PAGE←0
[36]   UCTR←0 ⋄ 'HOOH' ⋄ ROW ⋄ →PARSE
[37] HOOB:BASES←1↑STRING←1↓STRING ⋄ STRING←1↓STRING ⋄ ROWADD← 44 44 ⋄ PAGE←0
[38]   UCTR←0 ⋄ 'HOOB' ⋄ ROW ⋄ →PARSE
[39] LARGEST:BASES←2↑STRING←1↓STRING ⋄ STRING←2↓STRING ⋄ PAGE←1 ⋄ SUCCINC←7
[40]   ROWADD← 0 1 4 ,1↑STRING ⋄ STRING←1↓STRING
[41]   DIR← 1 0 ⋄ UCTR←7 ⋄ 'LARGEST' ⋄ ROW ⋄ →PARSE
[42] ROF:RHO←STRING[1] ⋄ WIDTH←STRING[2] ⋄ STRING←3↓STRING ⋄ →PARSE
[43] RESE:RESET ⋄ STRING←1↓STRING ⋄ →PARSE
[44] FOR:ROWADD← 7 7 ⋄ PAGE←1 ⋄ UCTR←WIDTH ⋄ BASES←3↑STRING←1↓STRING ⋄ 'FIELD OR'
[45]   DIR← 1 1 1 ⋄ ROW ⋄ STRING←3↓STRING ⋄ →PARSE
[46] FAND:ROWADD← 10 10 ⋄ PAGE←1 ⋄ UCTR←WIDTH ⋄ 'FIELD AND'
[47]   BASES←STRING[1 2 3] ⋄ STRING←4↓STRING ⋄ DIR← 1 1 1 ⋄ ROW ⋄ →PARSE
[48] FEQU:ROWADD← 13 13 ⋄ PAGE←1 ⋄ UCTR←WIDTH ⋄ 'FIELD EQU'
[49]   BASES←STRING[1 2 3] ⋄ STRING←4↓STRING ⋄ DIR← 1 1 1 ⋄ ROW ⋄ →PARSE
[50] FNOT:ROWADD← 16 16 ⋄ PAGE←1 ⋄ BASES←STRING[1 2] ⋄ DIR← 1 1 ⋄ UCTR←WIDTH
```

Figure 5: Listing of UP

3.1.2 ROM

Each line of ROM (except the first) is in one-to-one correspondence with a word of read-only memory in the putative controller. The first line just transfers control to some piece of ROM, or to a function called GOROM described later. Figure 6 is a listing of ROM.

APL functions are used as mnemonics for ROM fields: descriptions follow.

1. -> <loc t> <loc f> BLT <cond> is the conditional branch instruction. A branch to ROMADD[<loc t>] will occur if the code for the approximate number of responders is less than that specified by <cond>. Otherwise control passes to ROMADD[<loc f>].

2. CLOCKS <ARG> is responsible for doing all the work requested by operations appearing before it on the line, and for producing clocking signals. It also reads a word back from VASTOR and stops everything if an HCF condition has been raised (see section 1.9). Its argument tells it what to clock: there are three things that you may clock, and the phasing is also adjustable. CLOCKS 1 clocks the ICU; 2 clocks SH; 3 (2+1) clocks both simultaneously; 4 clocks LSR for LSRT; '1 7003' clocks the ICU, then the SH while ICU clock stays high; 5 is 4+1; and so forth.

```
U OP TO ONTO SHARP APL 07/JUN/1978 - PAGE 6

WSID = 24193 ROMVAS * SAVED  14.36.30 07/JUN/1978

    ∇ ROM
[1]   ROMADD ← ARBOUT 27 86 ,ON ← SHCCIRCC SHCCIRC ← -PGFLG[PAGE]/PGLOC[PAGE]+ROMADD[0] ← GOROM ← -0
[2]   BRG WECONT 0 ← CLOCKS 1
[3]   BRFOU WECONT 1 ← CLOCKS 1
[4]   SHG BRCONT ← CLOCKS '1 7003'
[5]   BRFOU WECONT 2 ← CLOCKS 1
[6]   WEGEN 3 ← CLOCKS 1
[7]   BRG WECONT 1 ← CLOCKS 1
[8]   BRFOU WECONT 2 ← CLOCKS 1
[9]   BRAND SHCONT ← CLOCKS 1
[10]  BRFOU WECONT 3 ← CLOCKS 1
[11]  WEGRR 2 ← CLOCKS 1 ← -LOOPER
[12]  LSHT 1 0 ← CLOCKS 4 ⍝ LOCTR←¯2 FROM XERBS2
[13]  1 LSHH 1 1 ← CLOCKS 0
[14]  BRNOP 1 ← SHRG BVAERS ← CLOCKS 3 ⍝XERBS2:
[15]  1 LSHH 1 0 ← CLOCKS 0 ← -LOOPER
[16]  ONEG CONST 1 ← CLOCKS 1
[17]  BRNOP SHCONT ← LSHT 1 1 ← CLOCKS 5 ⍝LOCTR←¯7 FROM XERBS2
[18]  SHRG BIAERO ← CLOCKS 2
[19]  BRG SHCONT ← LSHT 1 0 ← SHVEC ← CLOCKS 5
[20]  SHG BRCONT ← CLOCKS '1 7003'
[21]  ONEG SHCONT ← CLOCKS 1
[22]  LSHT 1 0 ← SHMAT ← CLOCKS 4
[23]  1 LSHH 1 1 ← CLOCKS 0
[24]  SHRG BVAERS ← CLOCKS 2 ⍝XERBS2:
[25]  1 LSHH 1 0 ← CLOCKS 0 ← -LOOPER
[26]  SUFIX ← LSHT 0 0 ← CLOCKS 4 ⍝LOCTR←¯1 FROM  XERBS4
[27]  SHIFT SHCONT ← CLOCKS 2 ← -LOOPER ⍝XERBS4:
[28]  LSHT 0 0 ← CLOCKS 4
[29]  BRG SHCONT ← CLOCKS 1 ⍝XERBS5:
[30]  WEGRR 0 ← CLOCKS 1
[31]  SHIFT SHCONT ← CLOCKS 2 ← -LOOPER
[32]  LSHT 1 0 ← CLOCKS 4 ⍝ ¯2 FROM XERBV2
[33]  1 LSHH 1 1 ← CLOCKS 0
[34]  BVATOS SHRCONT ← CLOCKS 0 ⍝ XERBV2:
[35]  1 LSHH 1 0 ← CLOCKS 0 ← -LOOPER
[36]  BRNOP 1 ← SHFIX ← LSHT 0 0 ← SHMAT ← CLOCKS 5 ⍝ ¯1 FROM XERBV2
[37]  SHIFT WECONT 0 ← CLOCKS 2 ← -LOOPER ⍝XERBV2:
[38]  LSHT 1 0 ← CLOCKS 4 ⍝ ¯2 FROM XERBV4
[39]  1 LSHH 1 1 ← CLOCKS 0
[40]  BVATOS SHRCONT ← CLOCKS 0 ⍝ XERBV4:
[41]  1 LSHH 1 0 ← CLOCKS 0 ← -LOOPER
[42]  IENG SHCONT ← CLOCKS 1 ← -LOOPER ⍝ IEN:
[43]  OENG SHCONT ← CLOCKS 1 ← -LOOPER ⍝ OEN:
[44]  BRG SHCONT ← CLOCKS 1 ⍝ BOOLEAN WRITE
[45]  WEGRR 0 ← CLOCKS 1 ← -LOOPER
[46]  SHG WECONT 0 ← CLOCKS 2 ← -LOOPER ⍝ BOOLEAN READ
[47]  ⍝
[48]  ⍝ THIS SPACE LEFT FOR EASY EXPANSION OF PAGE 0 OF ROM
[49]  ⍝
[50]  SUFIX ← SHVEC ← BRG CONST 1 ← CLOCKS 1 ⍝ ¯1 FROM "LARGEST"



[51]  WEGRR 1 ← CLOCKS 1 ⍝ "LARGEST":
[52]  SUFIX ← BRAND WECONT 0 ← CLOCKS 1 ← - 3 0 BLT 4
[53]  SHIFT 1,0⍴SHCONT ← CLOCKS 2 ← - 4 2 BLT 5
[54]  SHIFT 0,0⍴SHCONT ← CLOCKS 2 ⍝ZERO RESPONDERS
[55]  SUFIX ← BRG WECONT 1 ← CLOCKS 1 ← -LOOPER
[56]  -LOOPER ⍝ EXACTLY ONE RESPONDER
[57]  SUFIX ← BRG WECONT 0 ← CLOCKS 1 ⍝FIELD OR
[58]  BROR WECONT 1 ⍝ CLOCKS 1
[59]  WEGRR 2 ← CLOCKS 1 ← -LOOPER
[60]  SUFIX ← BRG WECONT 0 ← CLOCKS 1 ⍝FIELD AND
[61]  BRAND WECONT 1 ← CLOCKS 1
[62]  WEGRR 2 ← CLOCKS 1 ← -LOOPER
[63]  SUFIX ← BRG WECONT 0 ← CLOCKS 1 ⍝FIELD EQUIV
[64]  BRFOU WECONT 1 ← CLOCKS 1
[65]  WEGRR 2 ← CLOCKS 1 ← -LOOPER
[66]  SUFIX ← BEGC WECONT 0 ← CLOCKS 1 ⍝FIELD INVERT
[67]  WEGRR 1 ← CLOCKS 1 ← -LOOPER
    ∇ (3040 BYTES)
```

Figure 6: ROM listing

3. DMATOS <ARG> performs a DMA transfer from ARG to the scalar machine. The only known valid argument is SHRCONT.

4. CONST <1 | 0 | ABUSPOP> drives the constant line via the resistors to 'B' lines shown in figure 1. Any other 'B' driver will override this input: take care.

5. DMAFRS may be used to pass DMA data to SHRG.

6. IENG <SHCONT | WKCONT n | CONST n> loads IEN (the input enabling vector) from the named source.

7. ->LOOPER decrements UCTR: on becoming negative it causes exit, and otherwise control passes to ROMADD[2].

8. count LSRH 0 <0 | 1> shifts 'count' copies of the second element of its right argument into LSR. It does its own clocking, since LSR is faster than anyone else.

9. LSRT 0 <0 | 1> sets all elements of LSR to the second element of its right argument. A '1' enables the 'A' bus side of all SHifters, so be sure SH is pointed to 'B' if you do that. LSRt 0 0 is a good way to start a function, since the last one may have left LSR in a strange state. Setting LSR

to zero gives you a better chance of getting along well with the HCF logic, which recognizes this but does not understand about 'one's being shifted off the end.

10. OENG ARG is similar to IENG ARG, and sets the output enabling vector.

11. PIAFRD is like DMAFRS, but gets data from the microprocessor.

12. RR??? <SHCONT | WKCONT n | CONST> is the general form for a set of routines that operate on RR. The form is RR <- RR <op> (DATA ^ IEN), though some <op>'s are one and zero-dimensional degenerate.

    a. RRAND: <op> is AND

    b. RRANDC: <op> is .AND..NOT.

    c. RREQU: <op> is equivalence (XNOR)

    d. RRG: degenerate, RR <- DATA^IEN

    e. RRGC: RR <- ¬DATA^IEN

    f. RRNOP: no-op. Used to ensure that the ICU stops driving the 'B' lines in case previous operation was a store of some kind.

    g. RROR: <op> is .OR.

h. RRORC: <op> is .OR..NOT.

13. RRCONT does a STORE. It may only be used to store into SH, since there is a special function to store into WK.

14. RRCONTC may be used to store the complement of RR into SH.

15. SHCONT points SH towards 'B'. It may be an argument to RRG etcetera. RRNOP SHCONT is a popular way to allow serial shift.

16. SHFIX points SH towards 'A' so as to free up 'B'. May fail on HCF.

17. SHG <WKCONT n | RRCONT | RRCONTC | CONST n> loads SH from the 'B' lines.

18. SHIFT <0 | 1 | SH255 | SHCONT | WKCONT n | RRCONT | RRCONTC | CONST n> puts SH in serial mode. See SHMODE and SHCCIRC to find out what happens on clocking SH. The first three options for arguments make sense when SHMODE is set by SHVEC, and the rest when by SHMAT.

19. SHVEC sets SHMODE multiplexer to make SH appear as a 256-length vector.

20. SHMAT sets SHMODE mux so that SH is SHCCIRC rows by 32 columns.

21. SHRCONT points the shifter to 'A' and parallel mode. Can have HCF problems.

22. SHRG DMAFRS points the SHifter from 'A' to 'B', puts it in parallel mode, and requests a byte of DMA data from the scalar machine.

23. SKZ causes all ICUs with RR=0 to skip their next instructions.

24. WKCONT i: selects BASES[i] and DIR[i] to generate an address for working store, points shifter to 'A', and enables WK to drive 'B' lines.

25. WKGPR i: selects BASES[i] and DIR[i] to generate a WK address, points SH to 'A' to avoid destroying its data, enables WK, and generates a 'STORE' op-code for the ICU. Only those words with OEN=1 will be altered.

3.1.3 GOROM

This routine gets called instead of the appropriate section of ROM if PGFLG[PAGE]=0 and starts the FORTRAN ROM. It uses some escape sequences described in chapter 4 to pass startup information to the PDP-11 and expects DMA data from

VASTOR to be sent back together with prompts for DMA data to VASTOR. After APL has sent an '$VG<cr>' (GO command, see ch. 4) it queues all hex numbers returned as data from VASTOR, and sends back 'microprocessor' or 'scalar machine' data according as a 'P' or an 'M' occurs in the string. If a 'quad' (upper case L) appears in the input stream GOROM assumes that the FORTRAN controller has halted and awaits the next command. Since we use APL on a half-duplex line, it is necessary for FORTRAN on the PDP-11 to delay for a little while after recieving a carriage return before sending back any data. Control over this delay is afforded by one of the escape sequences described in chapter 4. Refer to programme listings in the appendices for more detail.

3.2 SIMULATOR

This group contains the functions used by routine ROM described above. Most of these are the same for both workspaces, and set and clear bits in APL's image of the current word of read-only memory (array OUT). Chapter 4 lists the assignment of bits in this version of ROM.

Some functions behave differently in the two workspaces, though they are used in exactly the same way. The following subsections describe the two versions.

3.2.1 EMU

EMU actually emits microcode when UP is given instructions (unless PGFLG says that the instruction can be handled from FORTRAN). The 300-baud serial transmission of microwords involved obviously makes VASTOR a little slow.

1. Function CLOCKS actually emits the microcode built up by the current line of ROM. It sends an '$VN' sequence to FORTRAN followed by any microcode new for this line and checks for HCF before sending an '$VF' to turn VASTOR off and returning.

2. DMAFRS removes one item from the queue of DMA data waiting to leave the scalar machine and returns it to the calling function.

3. LOOPER decrements UCTR (which APL keeps in this workspace) and returns either the line number in ROM corresponding to ROMADD[1] (which corresponds to ROMADD[2] in hardware) or zero according as UCTR stays positive or goes negative.

4. LSRH in EMU generates its own clock, rather than leaving it for CLOCKS, because the original design provided for multiple LSR shifts in one microcycle. It also emits the '6000'-series opcode (see chapter 4) corresponding to its argument.

5. PIAFRU is rather like DMAFRS, though it uses a scalar data source rather than a queue.

6. A function called PLACE is used by functions like WKCONT to generate working store addresses from BASES and DIR, which are kept in EMU.

3.2.2 ROMVAS

ROMVAS will either start the FORTRAN ROM executing any instruction requested by a call to UP or transfer that instruction from APL ROM to FORTRAN, according to the value of PGFLG. Transfer is accomplished by letting functions like RRAND build up microwords, and then having a special version of CLOCKS use an escape sequence to transfer them.

1. CLOCKS in ROMVAS transfers APL's array OUT to the appropriate line of the currently loaded page of ROM in FORTRAN by using an '$VD' sequence.

2. DMAFRS simply returns the ASCII for an 'm', with which FORTRAN should prompt APL (and the console) for any DMA data needed from the scalar machine.

3. LOOPER always returns a zero in ROMVAS, since there is no point in actually looping when the only objective is to transfer microcode. It also deposits an unconditional branch to ROMADD[2] in the appropriate microword.

4. LSRH just puts its arguments into OUT and returns.

5. PIAFRU returns the ASCII for a 'p', again for FORTRAN's use as a prompt.

6. PLACE returns its argument without any ado, because BASES and DIR are not available to ROMVAS.

Chapter IV

ROM

This chapter contains: information about known microcode down to the bit level; details on how 'microprocessor' programmes gain access to known microcode; timing for existing routines; and the FORTRAN implementation of ROM, which is currently kept on disk.

4.1 CURRENT CONTROLLER

The current developmental controller uses a DR11-C 16-bit parallel interface, whose output is interpreted as a 4-bit address and 12 bits of data. Table 3 shows the functions corresponding to various addresses on the DR11. Schematics and the like for the simple controller may be found in chapter 5.

VASTOR can return a 16-bit word containing output data and such: table 4 shows what the subfields of this word signify.

Two FORTRAN-callable subroutines exist to get and put words to and from VASTOR: PUTVAS and GETVAS. See supplied routines for examples of their use.

Table 3.

Current controller opcodes

ADDRESS	FIELD	REMARKS
0	0	Reset ICU and controller when set
1	9-0	WK address
2	-	reserved for CCD backing store
3	8	CONST input
	6-4	'B' data enable: 0 (default) -> SH 1 -> WK 2 -> backing store (future) 3 -> ICU
	3-0	ICU opcode
4	8	data for top of SH.
	7-0	parallel data for 'A' of SH.
5	9-7	SHMODE: 0 -> SHVEC; 1 -> SHMAT; 2 -> SHSENT
	6-3	SHCCIRC
	2	SH parallel loading when set
	1	SH points from 'A' to 'B' when set
	0	SH circulates when set, gets new data otherwise.
6	4	LSR loops data when set
	3	serial data for top of LSR.
	2	load/shift (load when 1)
	0	parallel data for LSR.
7	2	LSR clock
	1	SH clock
	0	ICU clock

4.2 MICROPROCESSOR VIEW OF ROM

Table 5 lists entry points to ROM and timing information for the various algorithms there, together with the number of addresses worth of BASES and DIRection they expect. The FORTRAN routines of chapter 2 use this information to set up common blocks which are then used by a routine called VARUN to run VASTOR programmes. Some of the pieces of ROM are hidden as components of larger chapter 2 operations, and so do not appear there.

Table 4.

VASTOR output through the DR11

FIELD	REMARKS
F	HCF condition encountered
B-9	condition code out: #RR
8	last element of SH ('B' for last word)
7-0	value on 'A' bus.

Table 5.

ROM addressing and timing

FORTRAN name	UPcode	ROMADD 1	2	3	4	# of BASES	TIMING setup	in cycles per loop
page 0								
VPV	1	0	0	-	-	4	-	A
XFRS	2	A	C	-	-	-	2	2
XFRBS	3							
LDAPT		E	16	-	-	-	8	2
MATROT		16	17	-	-	-	1	1
MATSTR		1A	1B	-	-	1	1	3
XFRV	4	1D	20	-	-	-	2	2
XFRBV	5							
MATRD		22	23	-	-	1	1	1
XFRV	(4)	1D	20	-	-	-	2	2
IENGSH	6	28	28	-	-	-	-	1
OENGSH	7	29	29	-	-	-	-	1
BOOW	8	2A	2A	-	-	1	-	2
BOOR	9	2C	2C	-	-	1	-	1
page 1								
SHGLRG	A	0	1	4	6	2	1	3 or 4
FOR	D	7	7	-	-	3	-	3
FAND	E	A	A	-	-	3	-	3
FEQU	F	D	D	-	-	3	-	3
FNOT	10	10	10	-	-	2	-	3

4.3 FORTRAN DEVELOPMENT SYSTEM

A FORTRAN programme called 'UPVAS' (microprocessor for VASTOR) existing on CGPACK 3 (property of WML) is designed for use with the APL development system, VASTOR and its rudimentary controller, and the Hazeltine 1500 terminal. The 11/34 usually runs as an overpriced modem cable, but may be ordered to simulate the proposed controller, perform some microprocessor functions, edit microcode, and do various housekeeping tasks described below.

A disk file called VASTOR.ROM on CGPACK3 contains 10 pages of FORTRAN ROM with 50 microwords in each page. Of these only one is loaded into core at any time, and most commands refer to that one. Columns of UPVAS array ROM correspond to the opcodes of table 3 above, except that some functions must be done by FORTRAN: thus UPVAS must get DMA data from APL or the console in order to produce data for opcode 4, and must compute addresses for working store and for branches. Table 6 shows differences between ROM for UPVAS and the opcodes of table 3, while figure 7 is a complete listing of ROM as of June 1978.

This programme may also be used without benefit of APL to run existing microcode or debug at the bit level by hand. The main thing lost thereby is symbolic microcode.

Since the routine is intended for use with an ASCII character set terminal and APL, it performs some rather odd

Table 6.

UPVAS ROM opcodes

ADDRESS	FIELD	REMARKS
1	1-0	index into BASES and DIRection
4	A	request DMA data from scalar machine or microprocessor.
	7-0	An ASCII character wherewith to prompt APL if bit 'A' is set.
5	A	Transfer DMA data to scalar machine.
6	9-5	Count for clocking LSR.
7	6-3	Second phase of clocks.
8	B-8	test condition for BLT.
	7-4	index into ROMADD if success.
	3-0	" " " " failure.

character translations to make the output read like APL. Most of this is fairly straightforward, but the user should note that the terminal cannot print overstruck characters: we therefore show these by printing the characters to be overprinted one above another. Experiment with it for more detail.

We list below the escape sequences understood by UPVAS at the moment. All of these start with an ASCII 'escape' (or 'altmode') character, which is echoed as '$'. The letters following the escape must be upper case and will not be echoed. Either APL or the terminal may enter any of these sequences. The two or three characters of the sequence must all come from the same source.

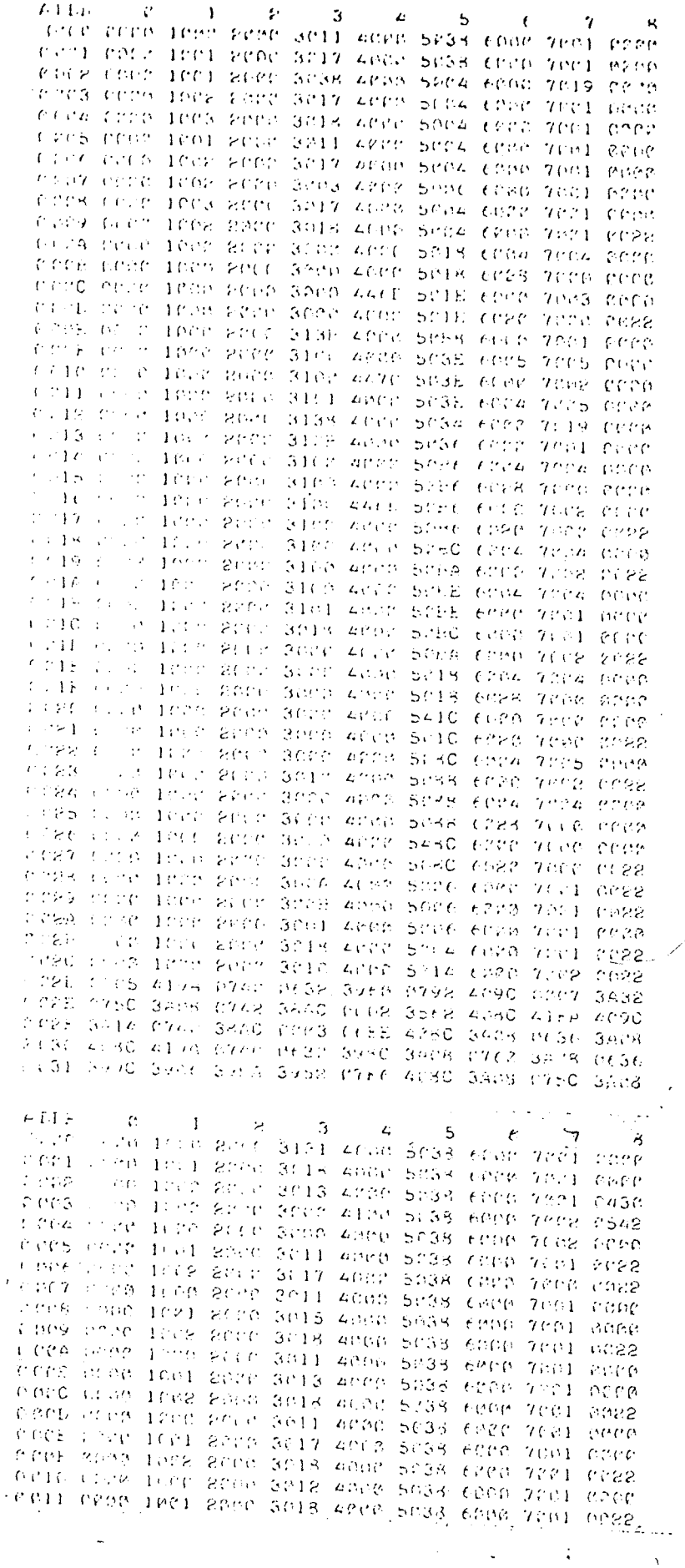


Figure 7: Hex listing of FORTRAN ROM

1. '$$' (escape escape) transmits an escape character to APL.

2. '$E' (END) terminates programme and returns to RT-11. This is the only software way to finish. 'Control-C' will not work. This was done to allow transparency for use with the PDP-10.

3. '$M' (maintenance) puts/removes the Teletype logically in/from parallel with the glass terminal so that hardcopy may be obtained/suppressed. This is a toggle function. This feature is also useful when debugging VASTOR hardware.

4. '$P' (parity) may be followed by 'E', 'O', '1' or '0' to force transmitted data to have even, odd, mark or space parity respectively. Default is 'even' and should be fine.

5. '$S' (set escape character) may be followed by a new escape character, although one of the two parity versions of 'escape' will still work. This feature interacts oddly with parity.

6. '$Vx' is a series of VASTOR escapes. The options for character 'x' are detailed below.

Several of these sequences take lists of numeric arguments: in all cases numbers are in

2's-complement 16-bit (4 digit) hexadecimal and delimited by blanks or other non-numeric characters. Where a list is required it may be terminated by an '$VF' or any other valid '$Vx' sequence.

Everything after the 'x' is transmitted to APL, which may sometimes produce confusing phenomena.

a. '$VB' (bases) tells the simulated controller what values of 'bases' and 'directions' to use for the next time ROM is used. Alternate 'base' values with their corresponding 'directions'.

b. '$VC' (counters) is to be followed by values for UCTR, SHCCIRC and a list of ROMADD's.

c. '$VD' (deposit) deposits hex data into the current page of simulated ROM. Give row number, column number and then value. Arbitrarily many triples may be entered.

d. '$VE' (examine) examines the contents of the current page of simulated ROM. Type row and column number and it will respond with the contents. Arbitrarily many locations may be examined.

e. '$VF' (off) will return from the current '$Vx' state to the modem state.

f. '$VG' (go) starts the simulated microcontroller executing according to the most recent values for '$VB' '$VC' and '$VR'. It will start at the next carriage return after the 'G', and an 'I' or an 'L' may be typed first, in which case the controller single-steps by instruction or by loop (resp.).

g. '$VL' (list) lists the current page of ROM ten rows at a time either on the glass terminal or TTY according to the state of '$M'. The next ten lines may be made to appear by typing any character.

h. '$VM' (VASTOR maintenance) is to be followed by two numbers. The first may be 0, 1, 2 or 3 to get different levels of maintenance output: 0 for compatibility with APL workspaces EMU and ROMVAS with minimum noise; 1 for listing of issued microcode; 2 for microcode and extra GETVASTOR's to allow inspection of device state; and 3 for no noise at all (which is incompatible with the workspaces). The second number is a delay (approximately in milliseconds) necessary to communicate

with half-duplex APL without losing data. Good values are 300 (hex, default) if APL is running things and 0 (hex) if FORTRAN.

i. '$VN' (on) allows immediate input of microcode and ignores ROM completely. This is the mode used by APL workspace EMU for microcode development and is also useful to force resets by hand and the like.

j. '$VP' (proceed) continues processing from a single-step halt in '$VG' or earlier '$VP'. Like '$VG' it may be followed by 'I' or 'L' and should be followed by a carriage return.

k. '$VR' (read) reads the page of ROM specified by the following number into core.

l. '$VU' (microprocessing) does for the 11/34 what ⎕P does in APL: executes higher-level instructions specified by the following vector of numbers. Be warned that these are hexadecimal and those in APL usually base ten. Refer to table 5 for the 'upcodes'.

m. '$VW' (write) is the inverse of '$VR' and writes the core page of ROM out to disk at a page number to be given after the 'W'. Remember to get out quickly with '$VF'.

Chapter V

HARDWARE

This chapter gives details on the layout of the VASTOR board, the developmental controller, the proposed controller, and a section on miscellany.

5.1 VASTOR BOARD

Figure 8 describes the layout of a single word of VASTOR of which there are 16 on each board. Of interest in this diagram is the presence of BK (i.e. the backing store) which although as yet undefined has sockets placed on the prototype board for future development. A second point of interest in this figure is that the ICU clock accepts instructions on the rising edge and produces results on the falling edge of the clock.

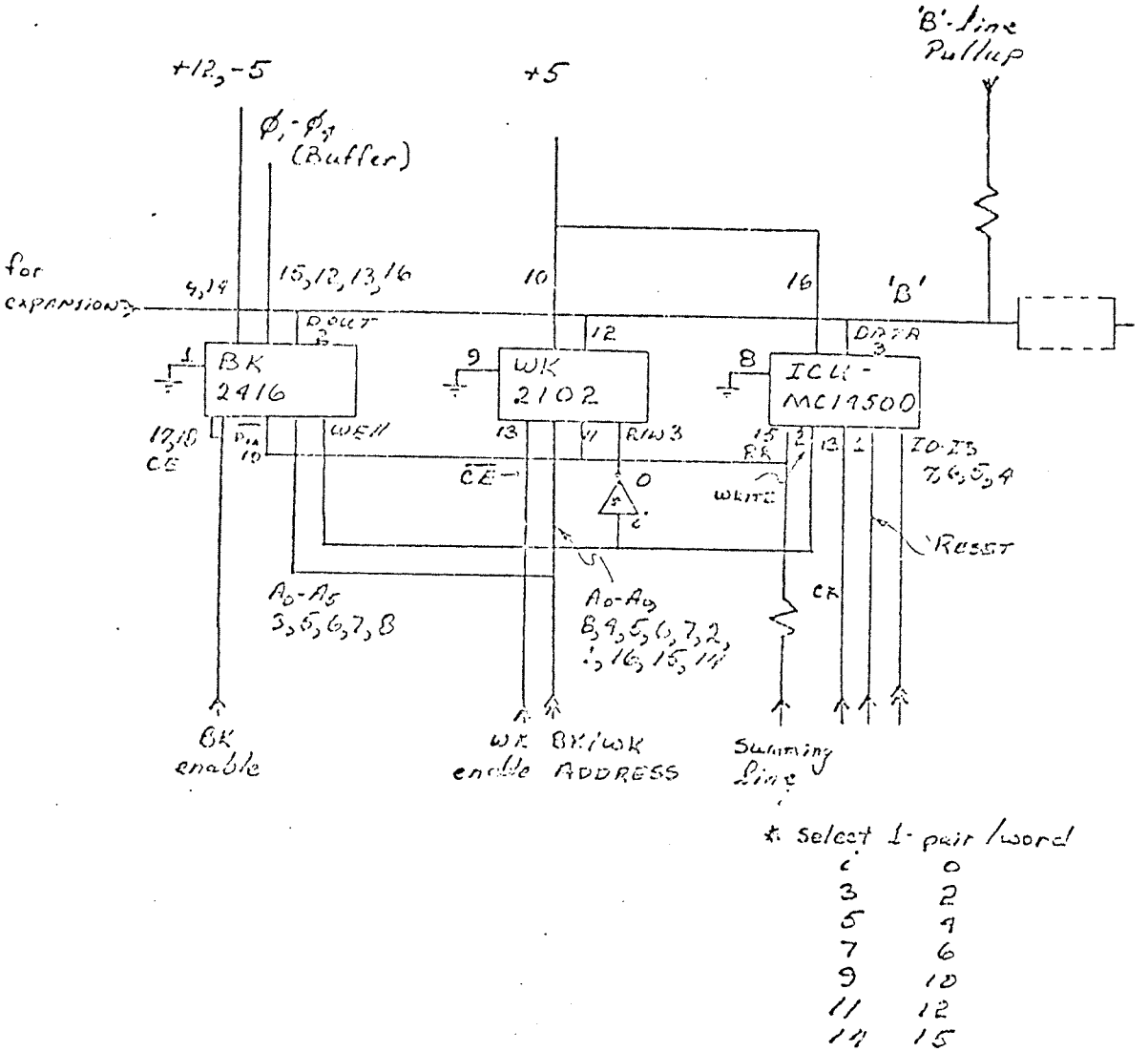


Figure 8: Schematic for One Word of VASTOR

Figure 9 illustrates the layout of the collection of words called a phrase. Of interest on this diagram is the layout of the multiplexers driven by SHMODE. Note that the shifter performs the shift/load operation in a level sensitive mode.

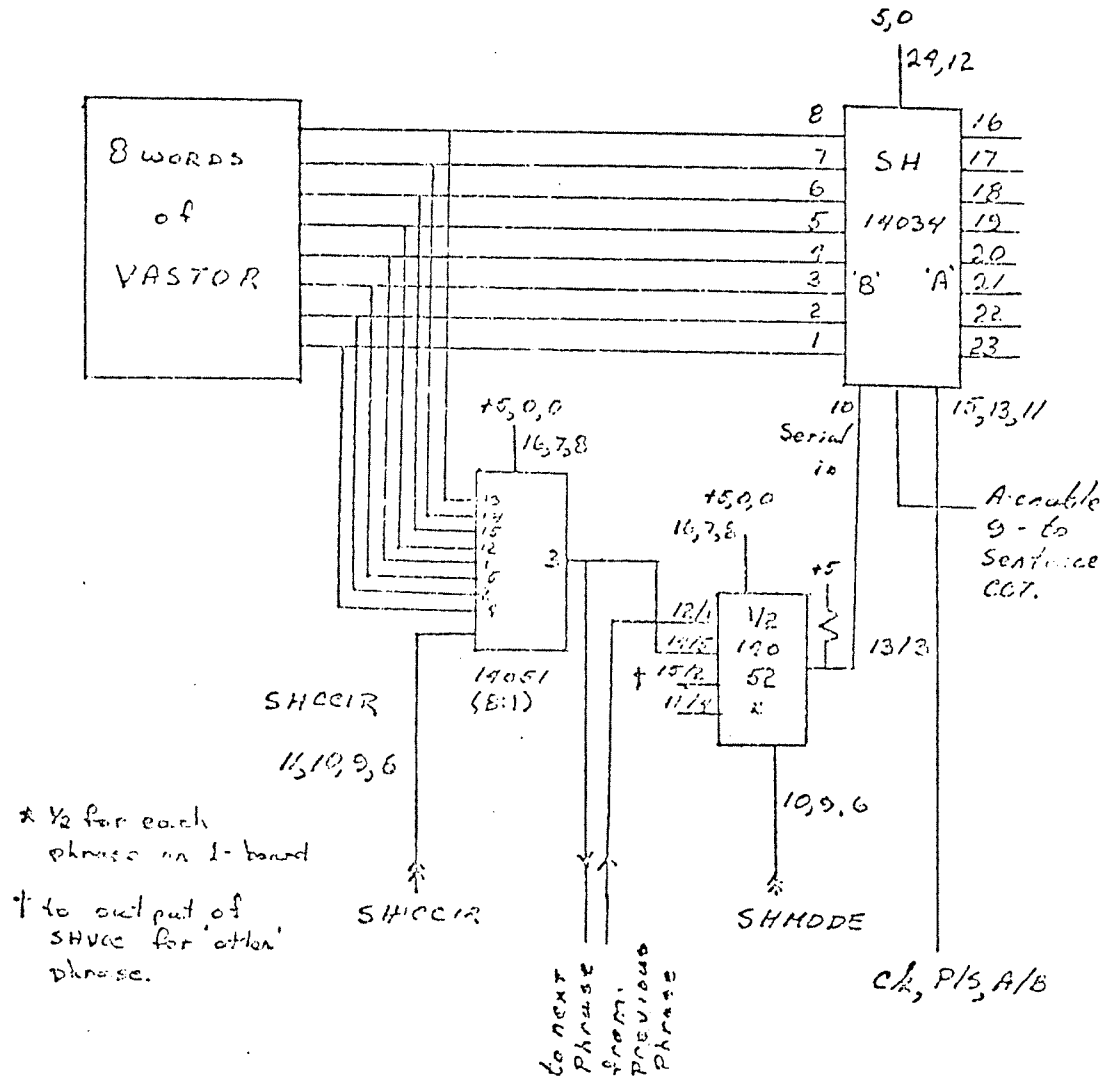


Figure 9: Schematic for One Phrase of VASTOR

Figure 10 illustrates how the phrases are interconnected to form a sentence. Note that the LSR is actually 4 bits long while the description and use only needs 2 bits. The reasons for this are historical as at one time 2 bits of information were believed useful per phrase. This facility has been left in a disabled form so that future changes are possible.

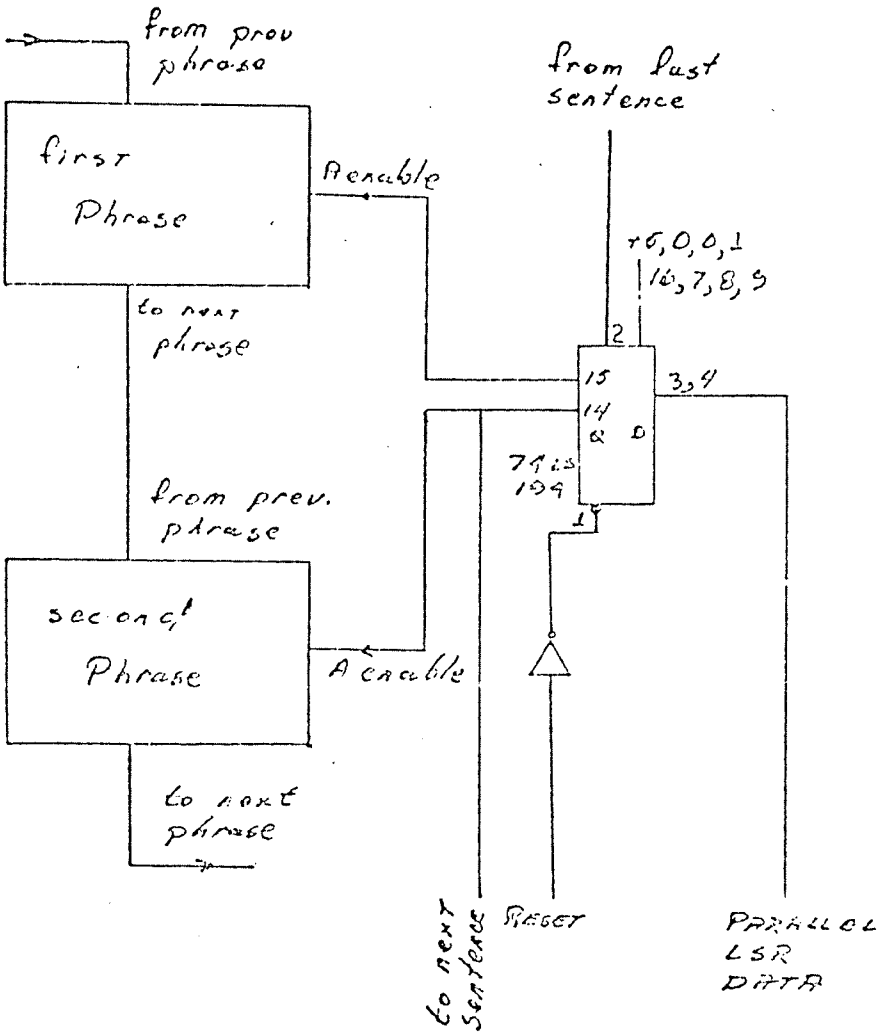


Figure 10: Schematic for One Sentence of VASTOR

Figure 11 gives the actual board layout of the ICs on the prototype unit.

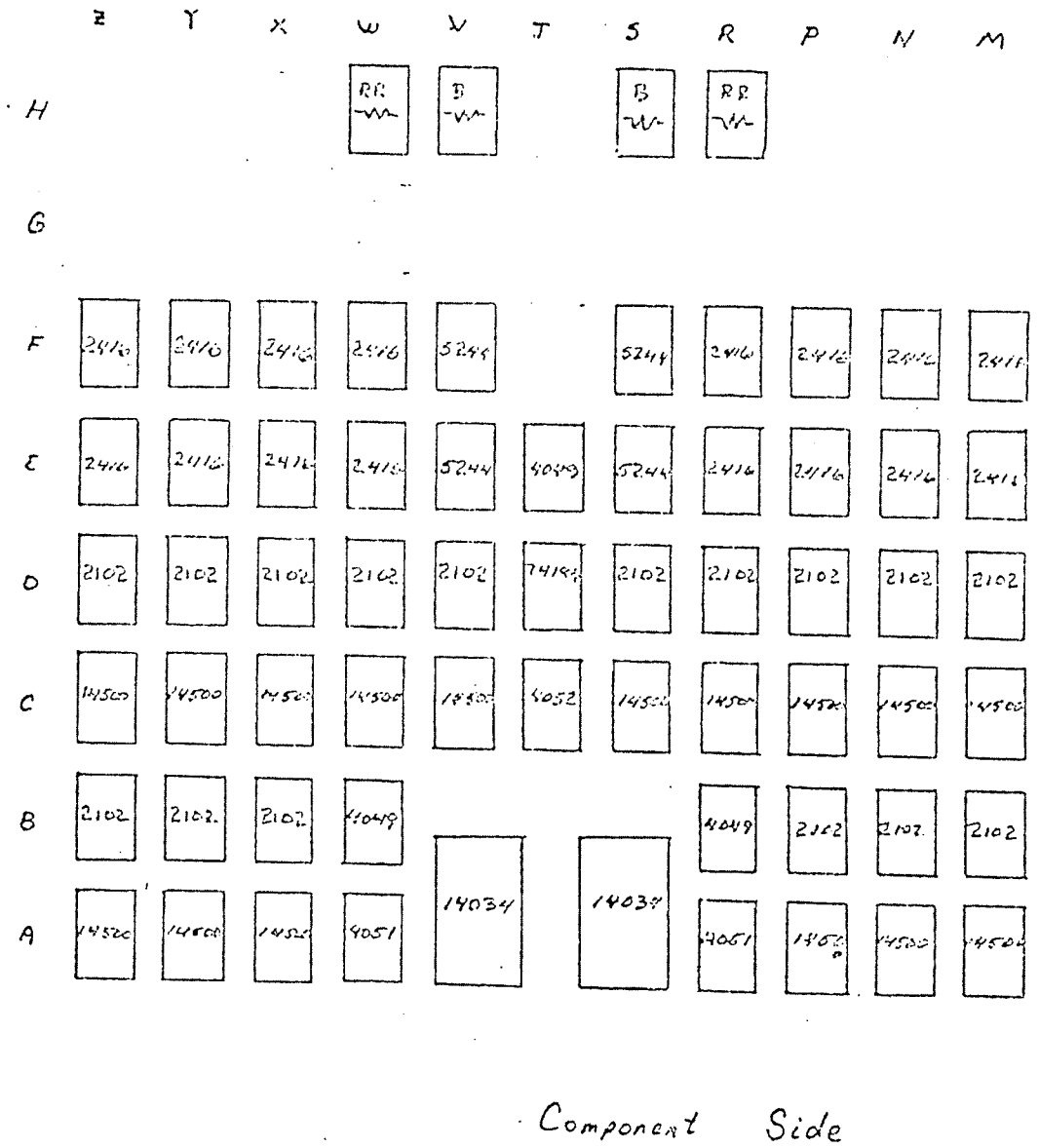


Figure 11: Board Layout for Prototype Sentence of VASTOR

5.2 DEVELOPMENTAL CONTROLLER

The developmental controller described in this section is based on the prototype and assume familiarity with the previous discussions with respect to PDP-11/34 interface.

Figure 12 shows the base for the controller: it decodes 4 bits from the DR11 into an opcode and produces a clock pulse for data latches. This diagram also shows the reset circuitry, which corresponds to opcode '0'. A UNIBUS 'INIT' forces the controller into the reset state. This requires that the first instruction from the PDP-11 be a 'clear reset' op-code. RESET clears most data registers as well as resetting all VASTOR ICU's.

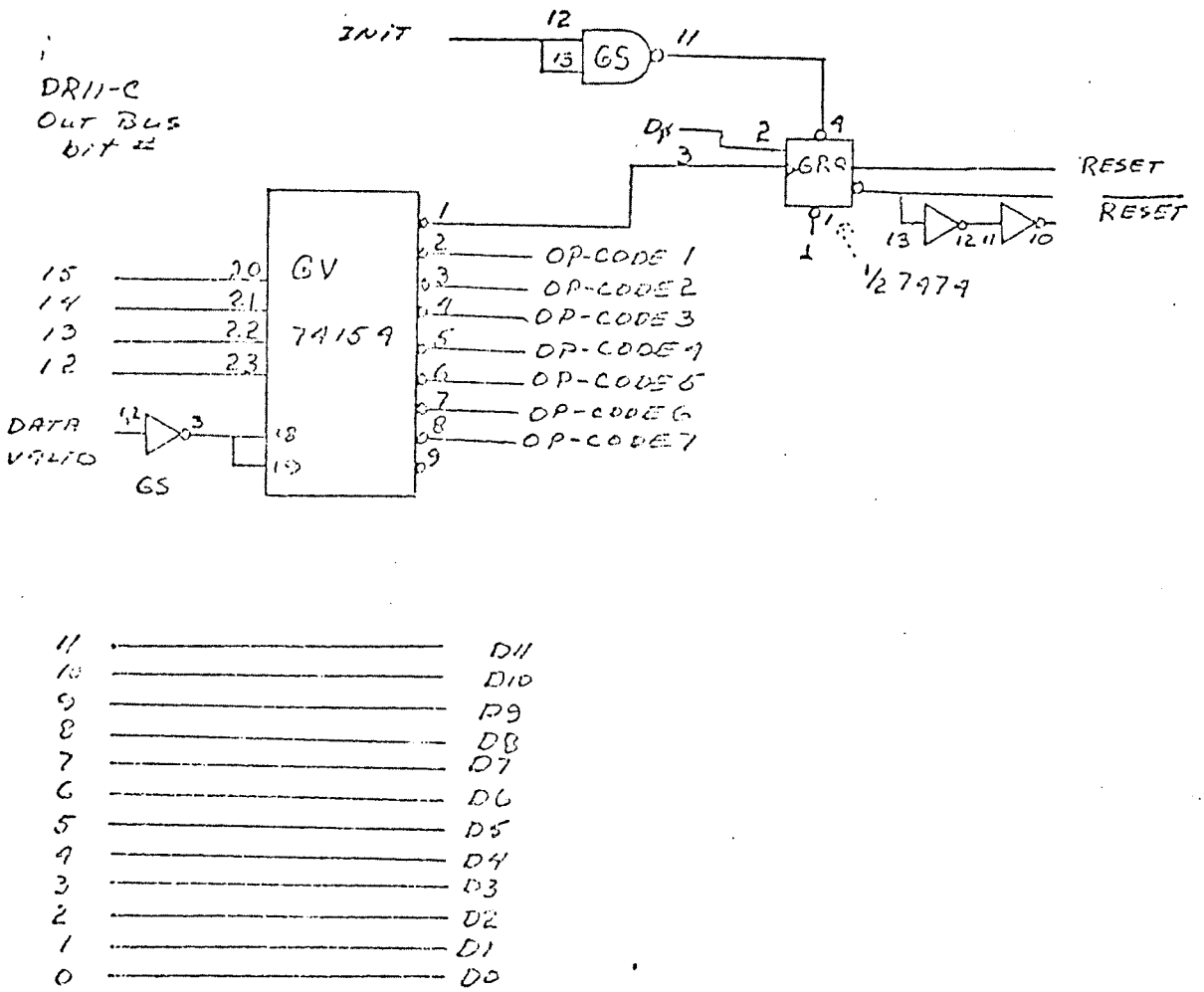


Figure 12: Controller Instruction Decoding and Reset

Figures 13 - 18 illustrate the circuit required to produce the necessary actions for the op-codes described elsewhere. Of particular note is the chain of actions produced by the Halt and Catch Fire circuitry. In Figure 14 a request for the SH to drive the 'B' bus disables ICU OP-CODES 8 and 9 and prevents other sources from being given a Chip Select. In Figure 16 the A/B request line is altered for those cases where more than one SH may be attempting to

drive the 'A' bus. Figure 17 shows how the number of A-Enables currently present are counted.

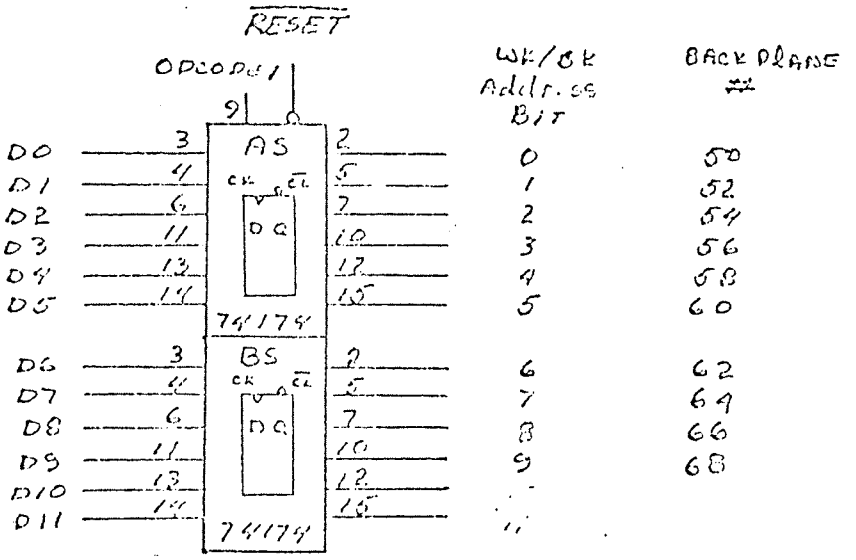


Figure 13: OP-CODE Type 1 Schematic

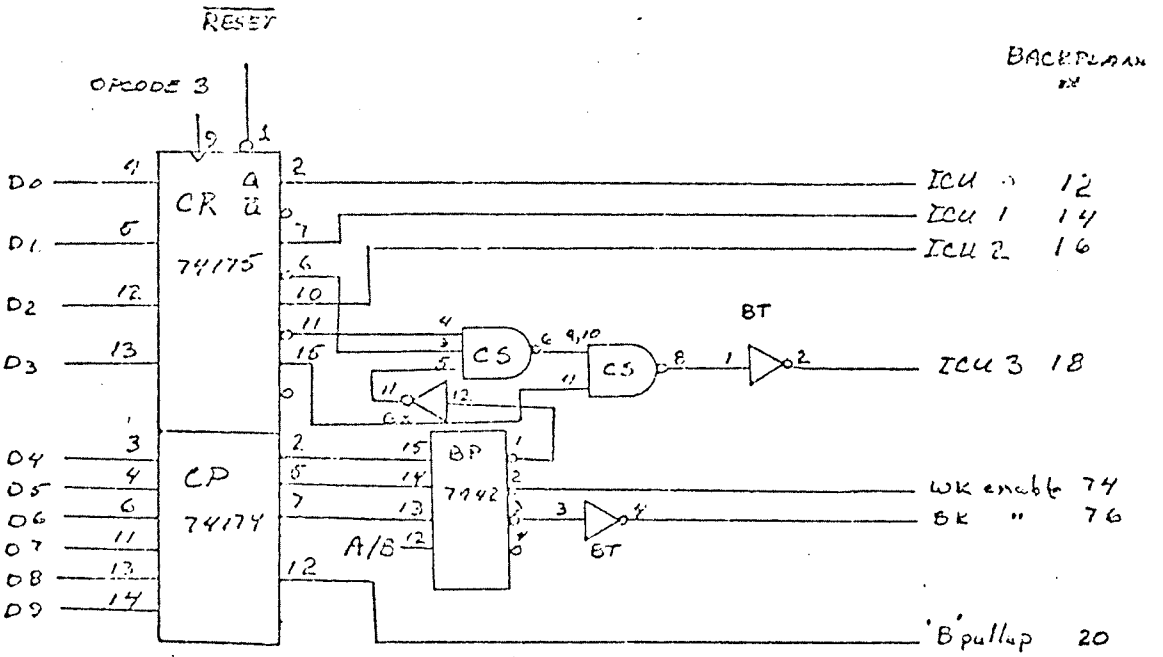


Figure 14: OP-CODE Type 3 Schematic

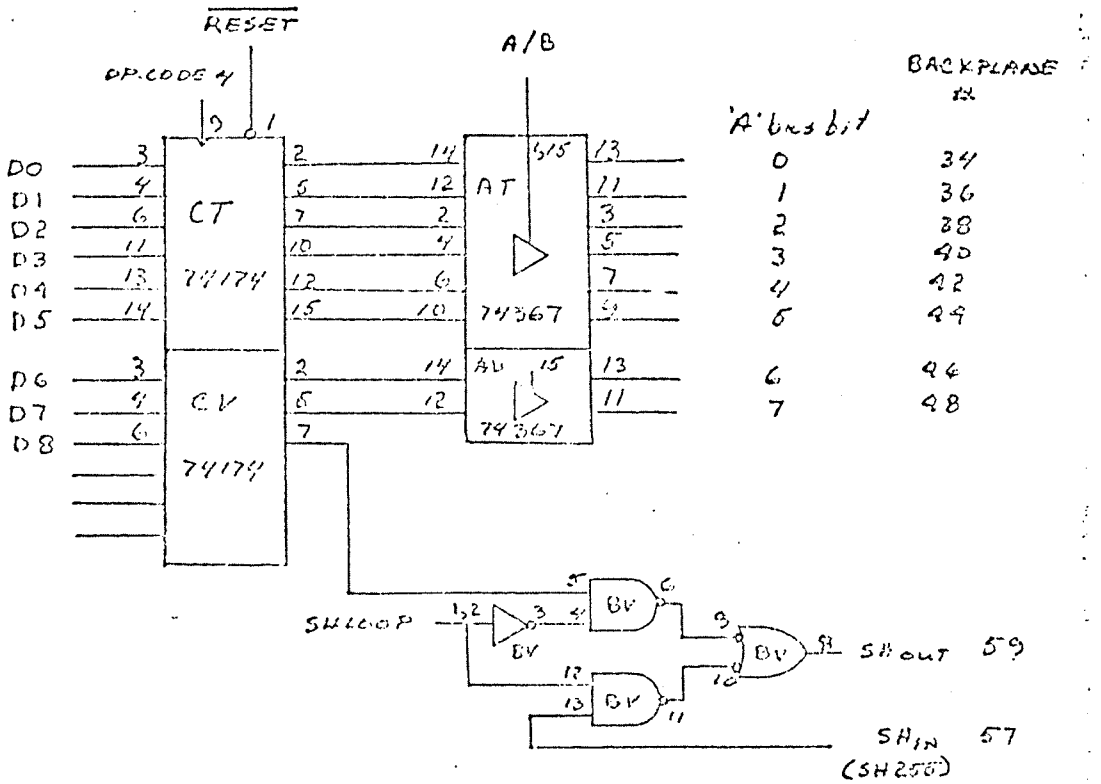


Figure 15: OP-CODE Type 4 Schematic

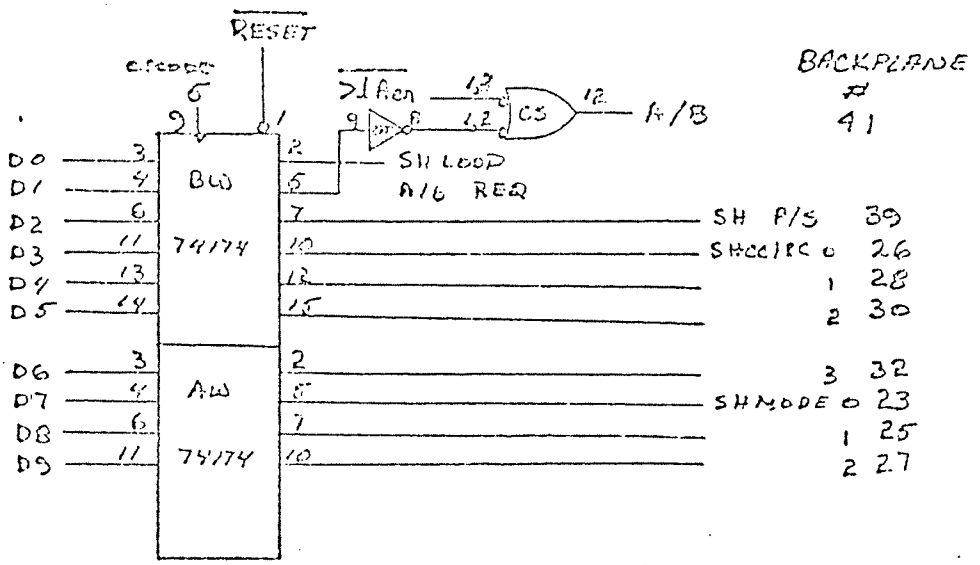


Figure 16: OP-CODE Type 5 Schematic

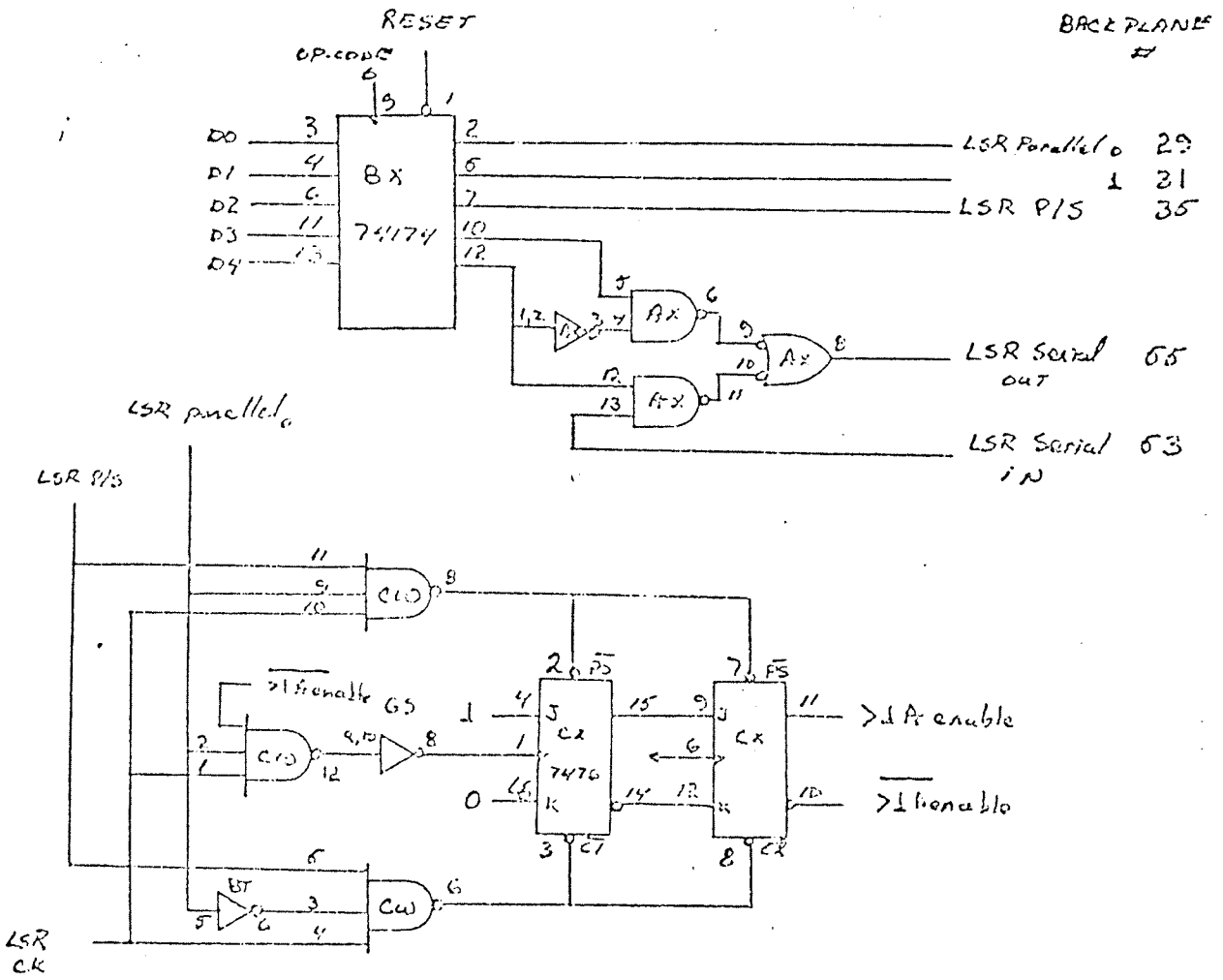


Figure 17: OP-CODE Type 6 Schematic

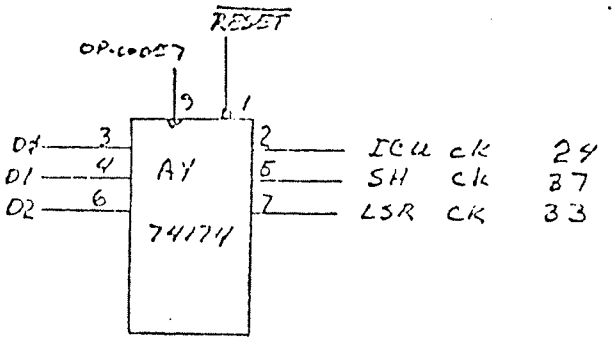


Figure 18: OP-CODE Type 7 Schematic

Figure 19 illustrates the Halt and Catch-Fire detection circuitry. An HCF condition may be caused by either an attempt to write to the 'A' bus which fails or a request for a 'store' ICU op-code (8 or 9) in some cases.

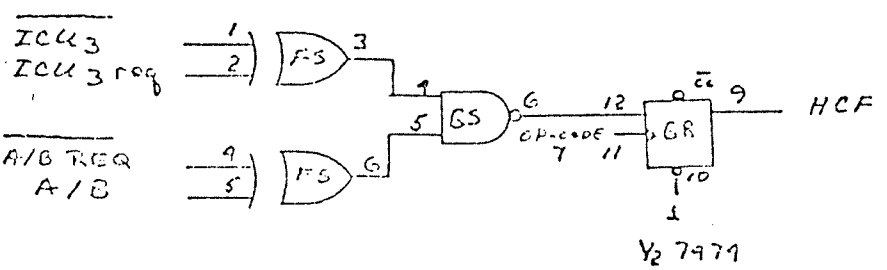


Figure 19: Schematic for Halt and Catch Fire Registering

Figure 20 shows the circuitry used to derive the value which is to be read into the PDP-11/34.

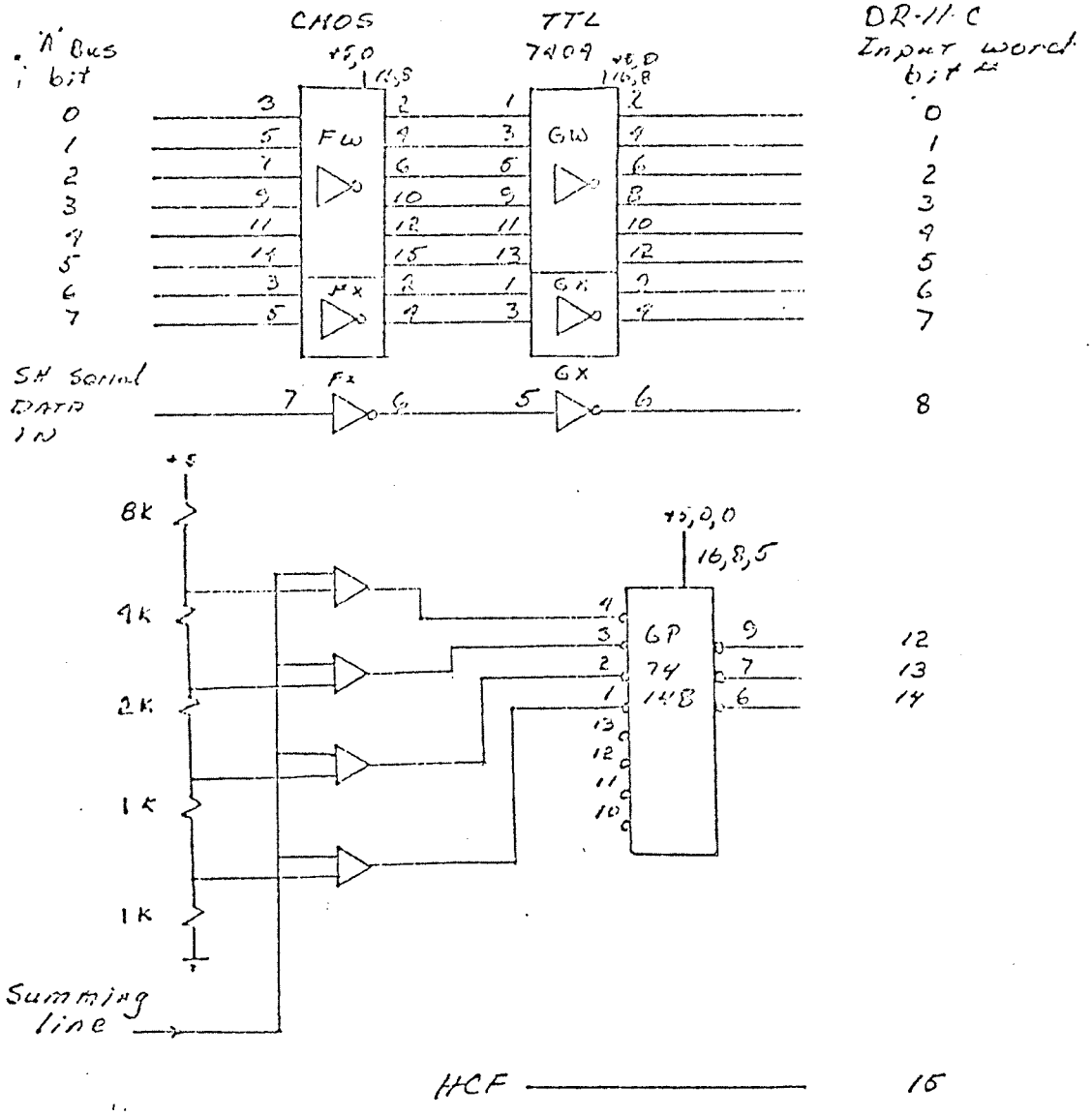


Figure 20: DR11 Input Driver Schematic

Figure 21 shows the IC layout for the developmental controller.

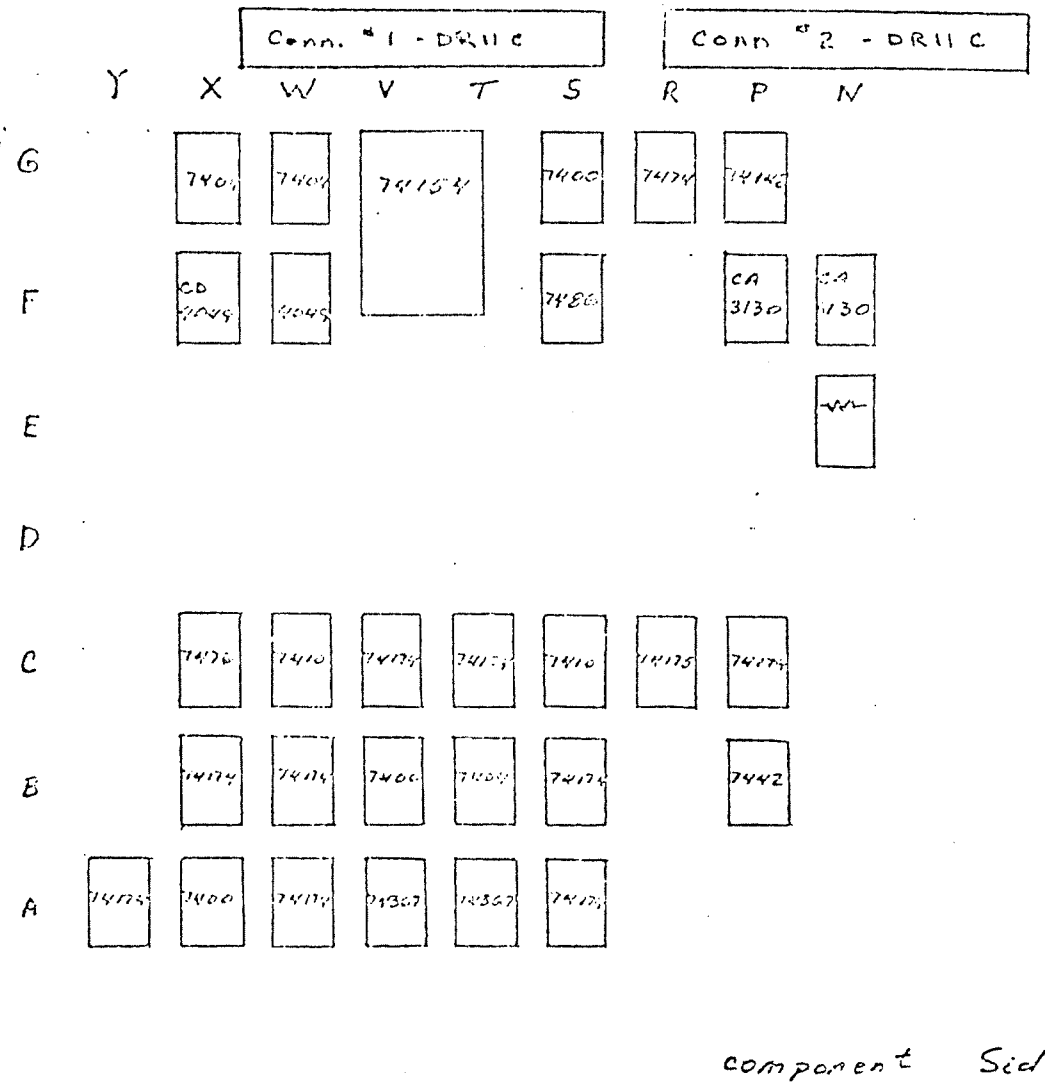


Figure 21: Board Layout for the Developmental Controller

5.3 PROPOSED CONTROLLER

The proposed controller structure is diagrammed in figure 22. It uses the DR11-C output lines as a simulation of a microprocessor bus: 8 bits of data and 8 bits of address plus two address-like bits from the DR11 control word. The control word bits specify whether VASTOR is to run, control store be loaded, or high-order address bits for control store loaded (address bits being scarce).

Control store (referred to as ROM in simulation) has at least one writeable page, and is expected to reside on a board separate from the rest of the controller. Table 7 shows the assignment of functions to bit fields in control store.

Two-port memory is implemented as 512 8-bit words from the DR11's point of view and as 256 16-bit words to the controller. The actual memories are conventional (e.g. 9111 or 9112) 200ns RAM's whose address lines are multiplexed between the two sources. 'Writes' all come from the DR11 and 'reads' all come from the controller, which simplifies multiplexing.

The DR11 and controller both see two-port memory divided into 16 pages, each a separate task. At any time the DR11 should be writing into one page while the controller reads from another: the choice of pages for both functions

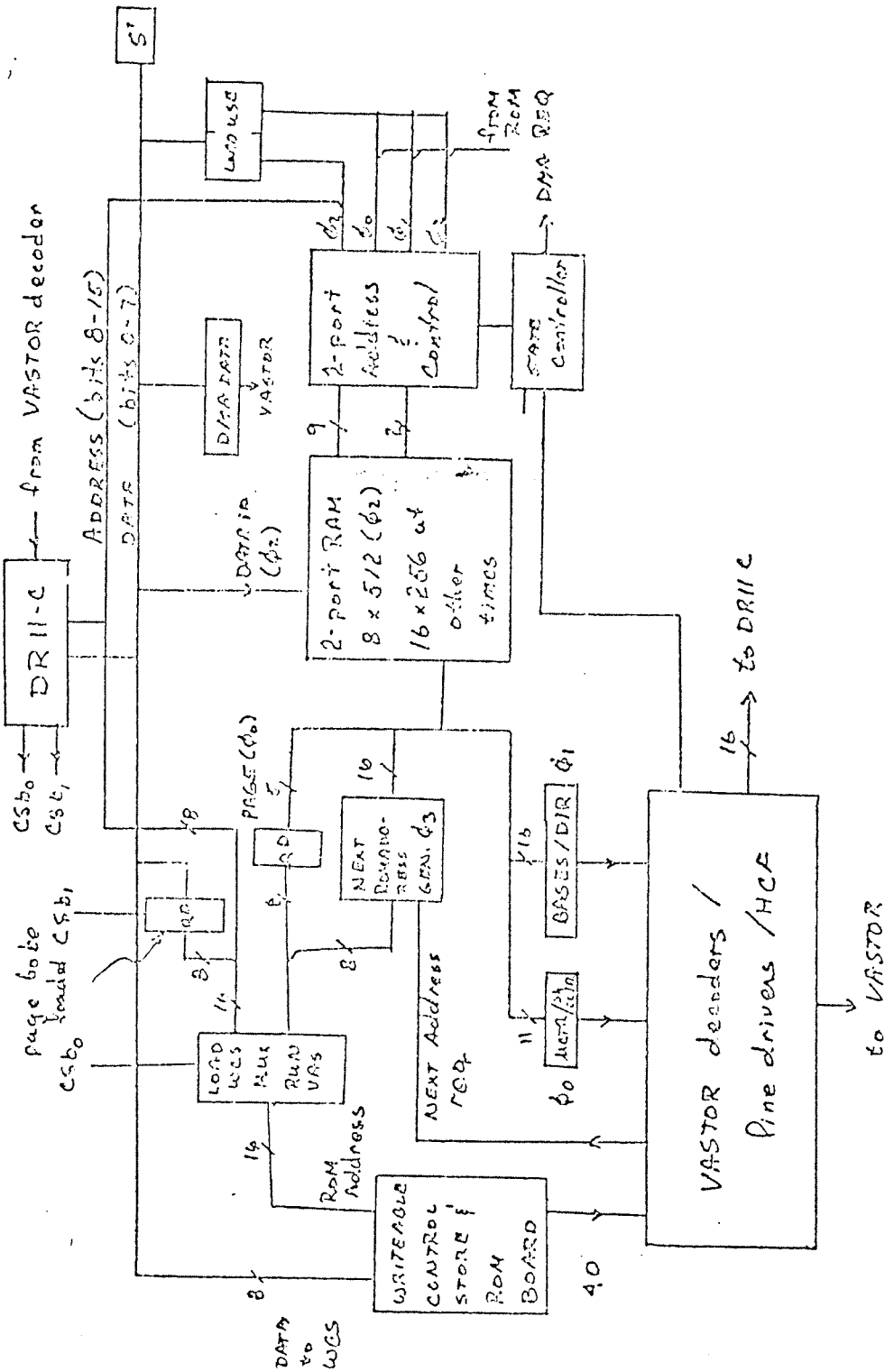


Figure 22: Proposed Controller Block Diagram

Table 7.

Control store bit assignments

Field		Function
0		first phase ICU clock.
1		first phase SH clock.
3-2		second phase ICU and SH clocks.
6-4	1xx:	'xx' is parallel data for LSR.
	0xy:	'x' sets mux. to connect LSR as loop or shift register. 'y' is data in if LSR is a shift register. bit 6 itself goes to P/S of LSR.
8-7		number of clock pulses for LSR. bit 8 probably surplus.
B		request DMA to scalar machine.
D-C		SHMODE mux. drive.
E		A/B (on SH) drive.
F		P/S (on SH) drive.
11-10	0x:	no-op.
	1x:	request DMA from scalar
	10:	DMA for DMAFRS.
	11:	DMA for PIAFRU.
15-12		ICU opcode drive
16		CONST
17	0:	selects CONST to drive SHTOP and the 'B' line resistors.
	1:	selects SH255 for above functions.
19-18		'B' line enables
	00:	SH
	01:	WK
	10:	BK
	11:	RR
1C-1A		index for BASES and DIR top bit redundant so far.
1F-1D		test condition for ROMADD generation.
23-20		new ROMADD index on .true.
27-24		" " " " .false.

is made by the 11/34, which is also responsible for synchronisation. Five of the 8 available address bits (on the DR11) are used to specify an 8 bit word in two-port memory, while the top three bits select whether two-port, control information, or DMA data are to be written. Table 8 shows

the assignment of functions to the two control bits and three high-order address bits.

Table 8.

Address space utilization

Control bits	Controller Function
00	Run: normal controller activity
01	Write to control store
10	reserved
11	load high-order bits of control store address for a following write

address bits 7-5	controller function addressed
000	two-port
001	8-bit register containing two two-port page numbers: one used for reads, the other for writes.
010	register containing DMA data from the 11/34. Both DMAFRS and PIAFRU go here.
011	start VASTOR command.
1xx	reserved.

The controller cycles through four major states (cf. figure 23) in running VASTOR. Three of these are executed for every microword and one to get a new task started.

The following subsections describe the two-port in more detail and each of the four controller states.

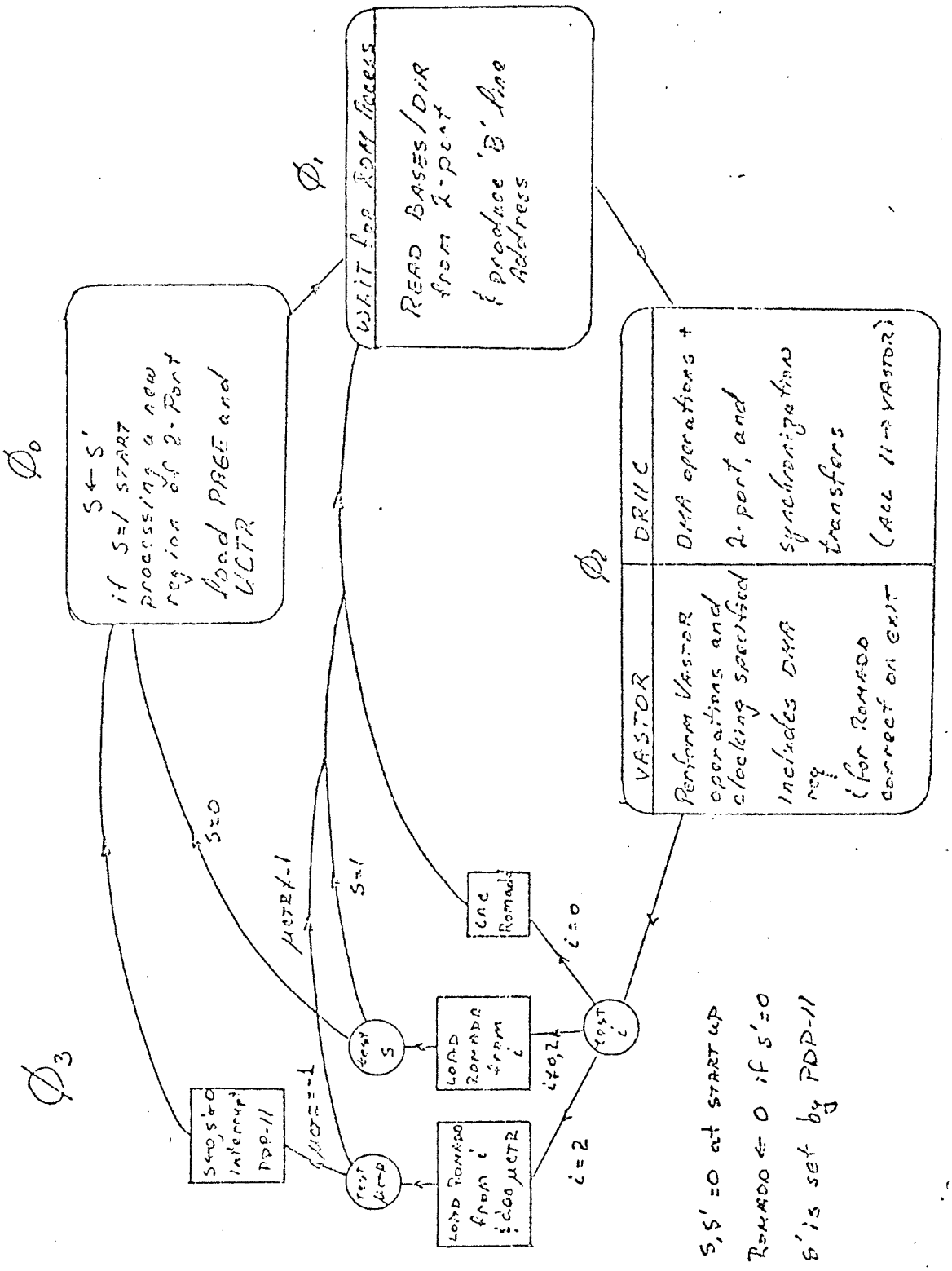


Figure 23: Major controller states

5.3.1 Two-port access

Address generation for two-port memory is detailed in figure 24.

5.3.2 State ø0

This state is needed only once per task, and on exit should leave UCTR, SHCCIRC and the ROM page number correct. Figure 25 shows the latch into which these are loaded from the current (read) page of two-port, and figure 24 above described the necessary address generation.

5.3.3 State ø1

This state should leave the WK address correct on exit, and therefore fetches BASES and DIRections. Figure 26 gives details.

5.3.4 State ø2

This state allows the DR11 access to two-port and VASTOR control while a VASTOR instruction runs. Figure 27 shows how DR11 access to two-port and VASTOR control functions is synchronized to state ø2. Several 'writes' could theoretically happen in a single instance of this state, but the controller does no 'read's.

The measured condition code is assumed to have settled on exit.

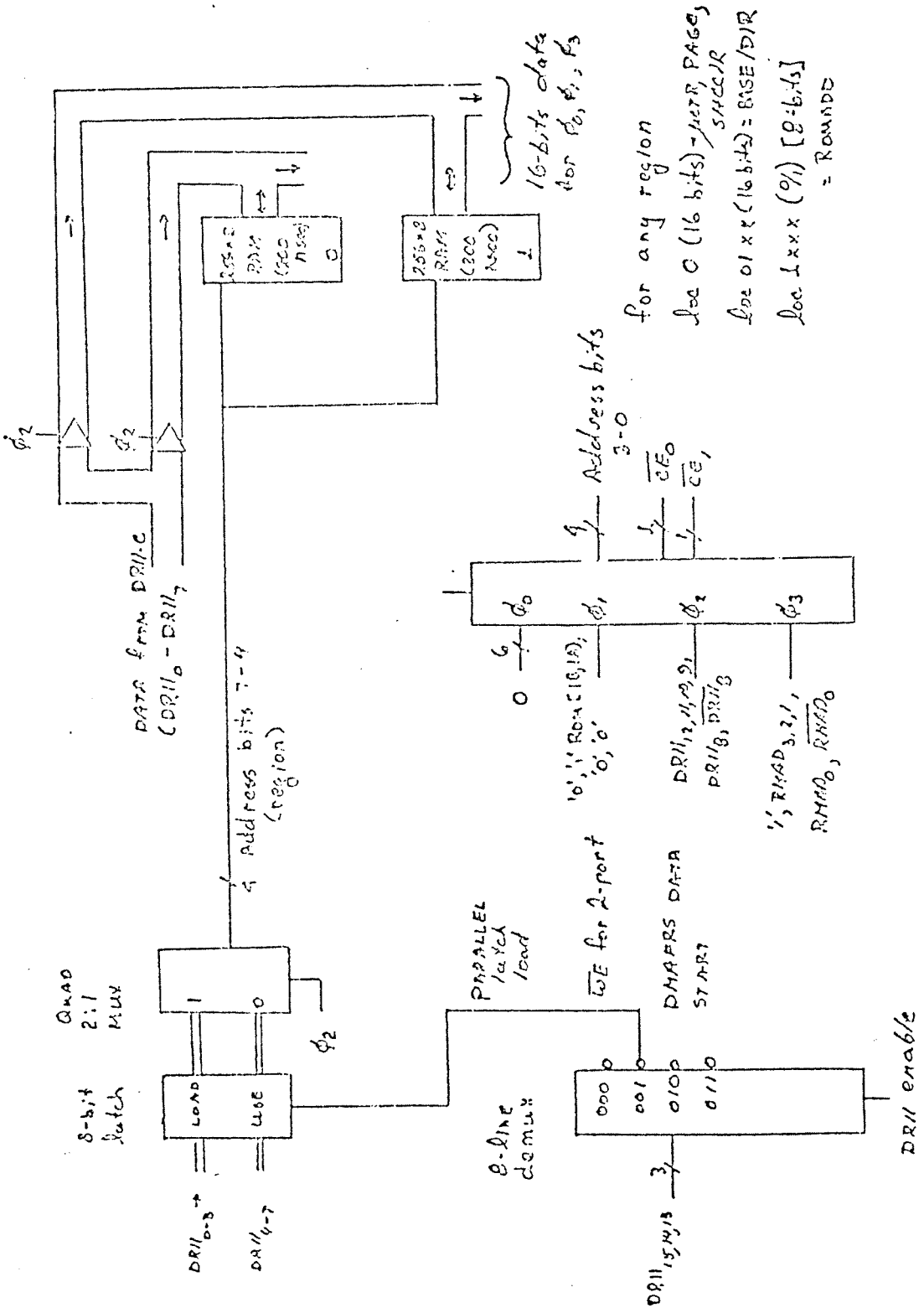


Figure 24: Two-port address generation

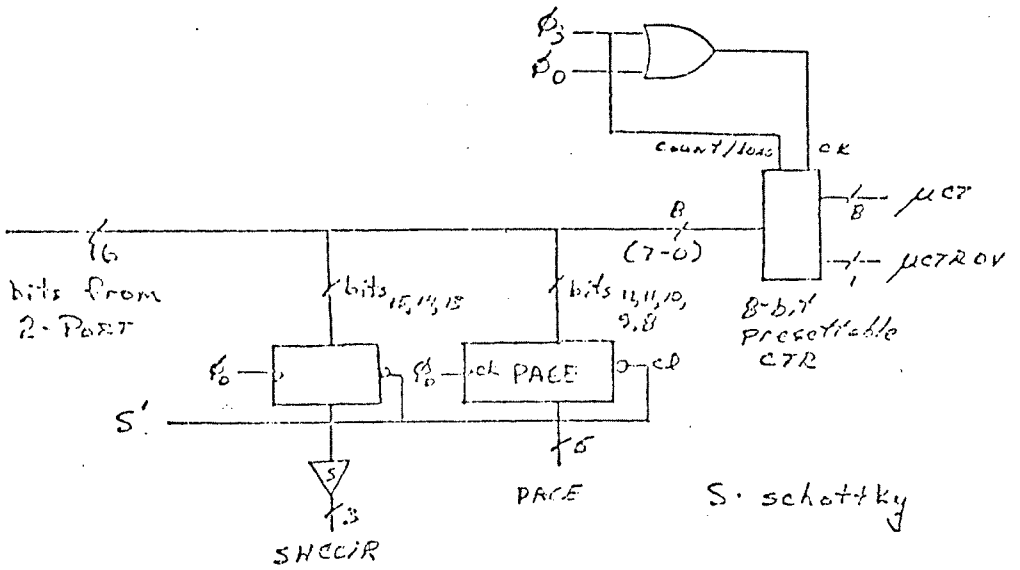


Figure 25: State ø0 latch

Figure 28 shows five substates of this major state. Two-port writes are independent of these, which perform the tasks stated in the figure.

Figure 29 shows how two phases of ICU and SH clocks are generated in ø2, and how LSR's clock is produced. This version cannot produce multiple LSR clocks.

Figure 30 shows DMA details and SH control. Note that it defines the layout of the DR11 input word, which gives the 11/34 enough information to do DMA. This word is similar to that in the existing controller except for the addition of two bits which specify the type of interrupt request.

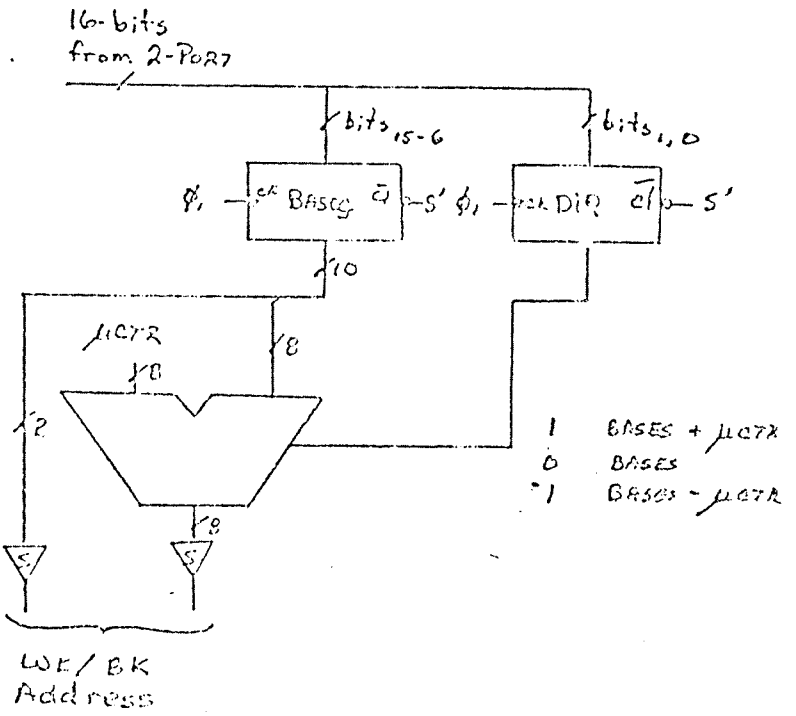


Figure 26: State ø1 operation

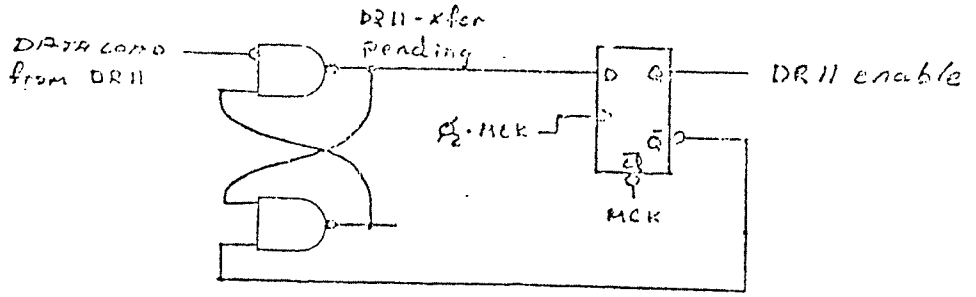
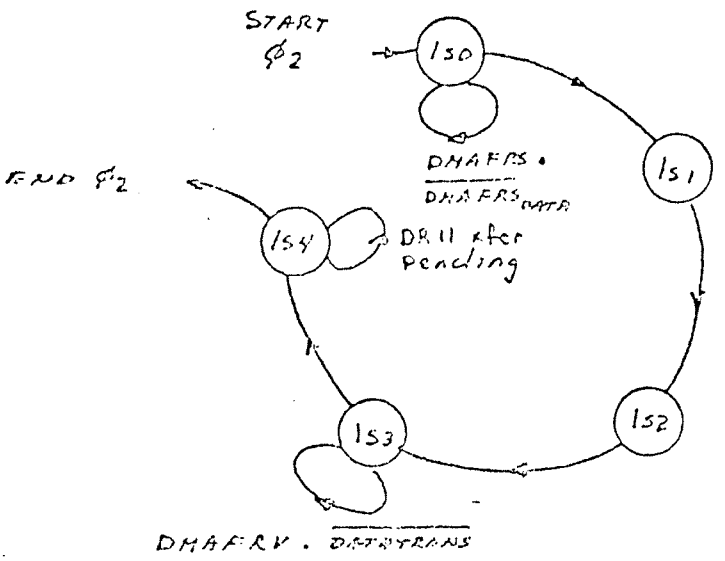


Figure 27: ø2: Enable DR11 writes

Figure 31 shows drivers for the ICU, 'B' line control and SHifter input data generation.



lsϕ produce lsr clocks, DMAFRS

ls1 Phase 1 of ICU/SM CAS

ls2 Phase 2 of ICU/SM CAS

ls3 DMAFRV

ls4 wait until DRH is Complete.

Figure 28: ϕ2 Substates

5.3.5 State ϕ3

Figure 32 shows logic associated with state ϕ3, which is responsible for stepping to the next microinstruction. This may involve two-port access (for ROMADD), decrementing UCTR, or causing the current task to halt.

The bit labelled s' in figure 32 is used to halt a task. When it is clear, all ROM accesses are to location 0 because the value produced for ROMADD is cleared. Location

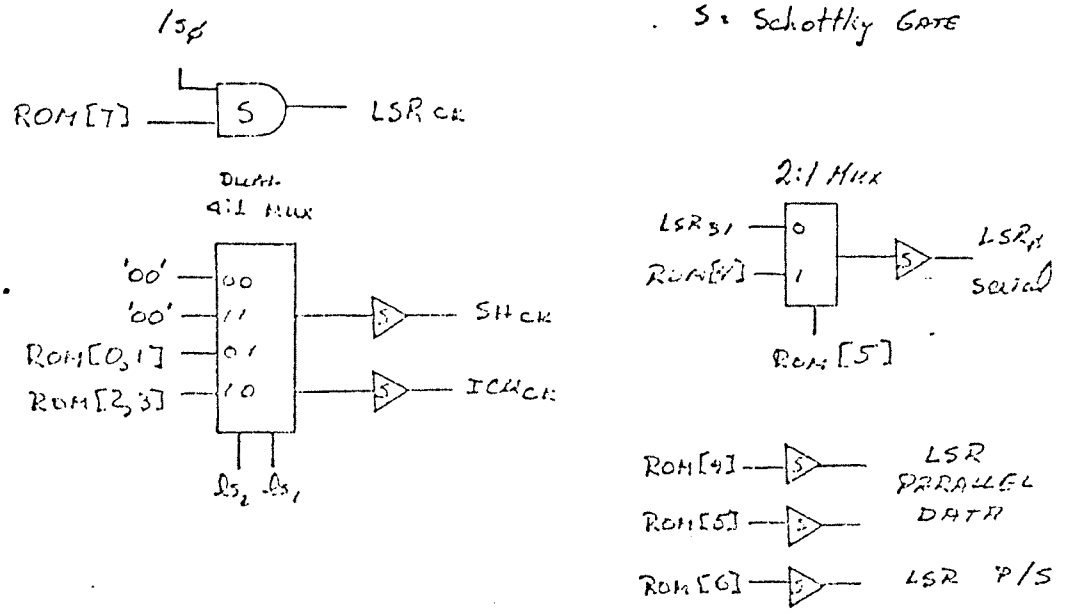


Figure 29: Clock generation and LSR control

0 of each page of ROM should therefore contain a no-op with an unconditional branch to ROMADD[1]. The DR11 may set this bit only in state ø2, and another bit is copied from this one on entry to state ø1 to allow that state to be skipped when an instruction is running.

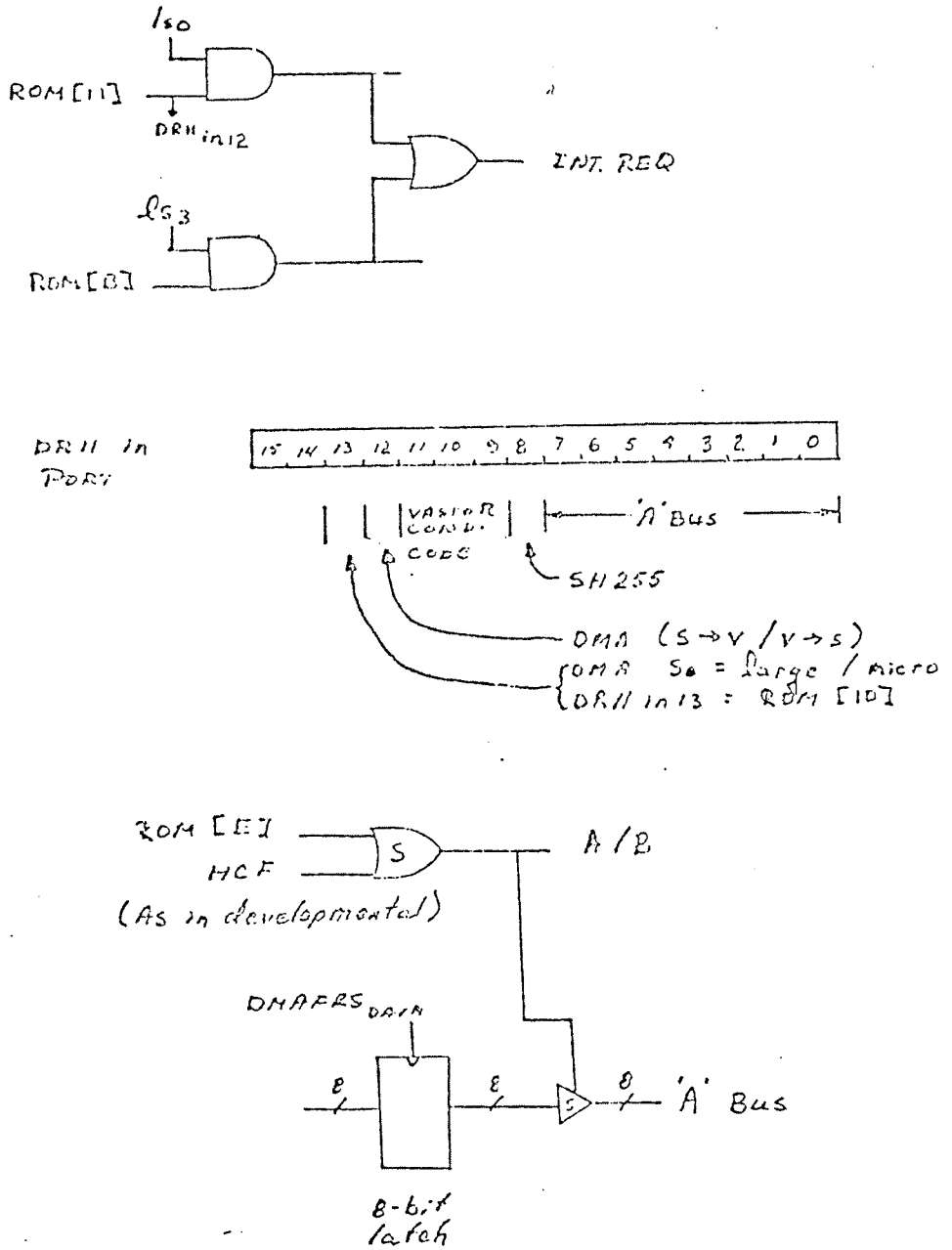


Figure 30: DMA and SH control

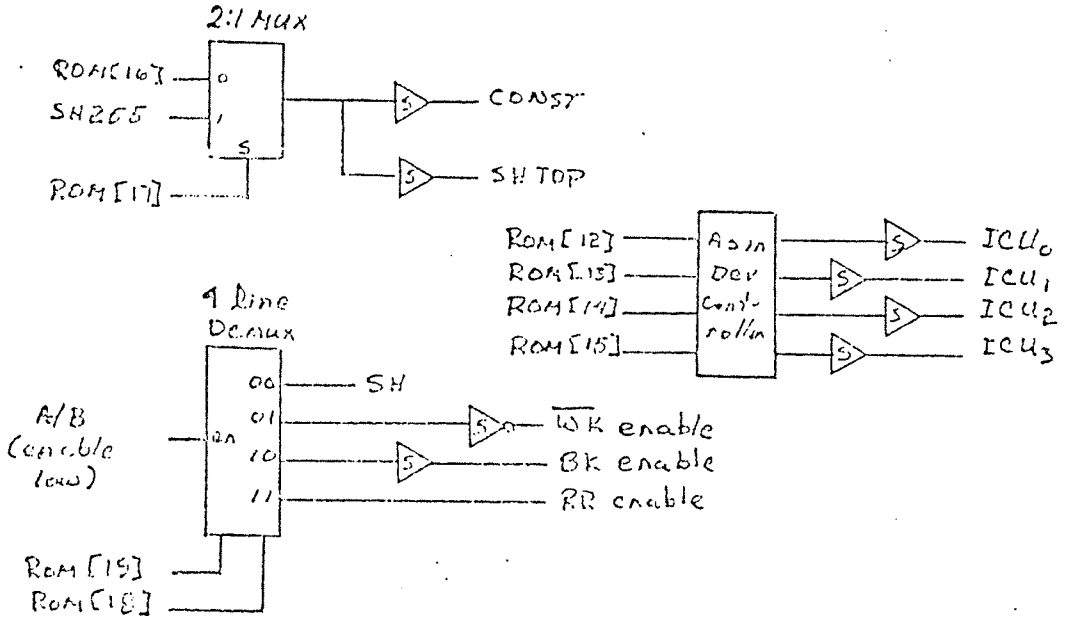


Figure 31: ICU and B line control

5.4 MISCELLANY

Tables 9 and 10 give pin assignments for the VASTOR backplane. The boards used are VECTOR number 4350 with an 80-pin connector to fit R680 sockets. Numbers are as shown on the board: a variety of sockets exists with another numbering scheme.

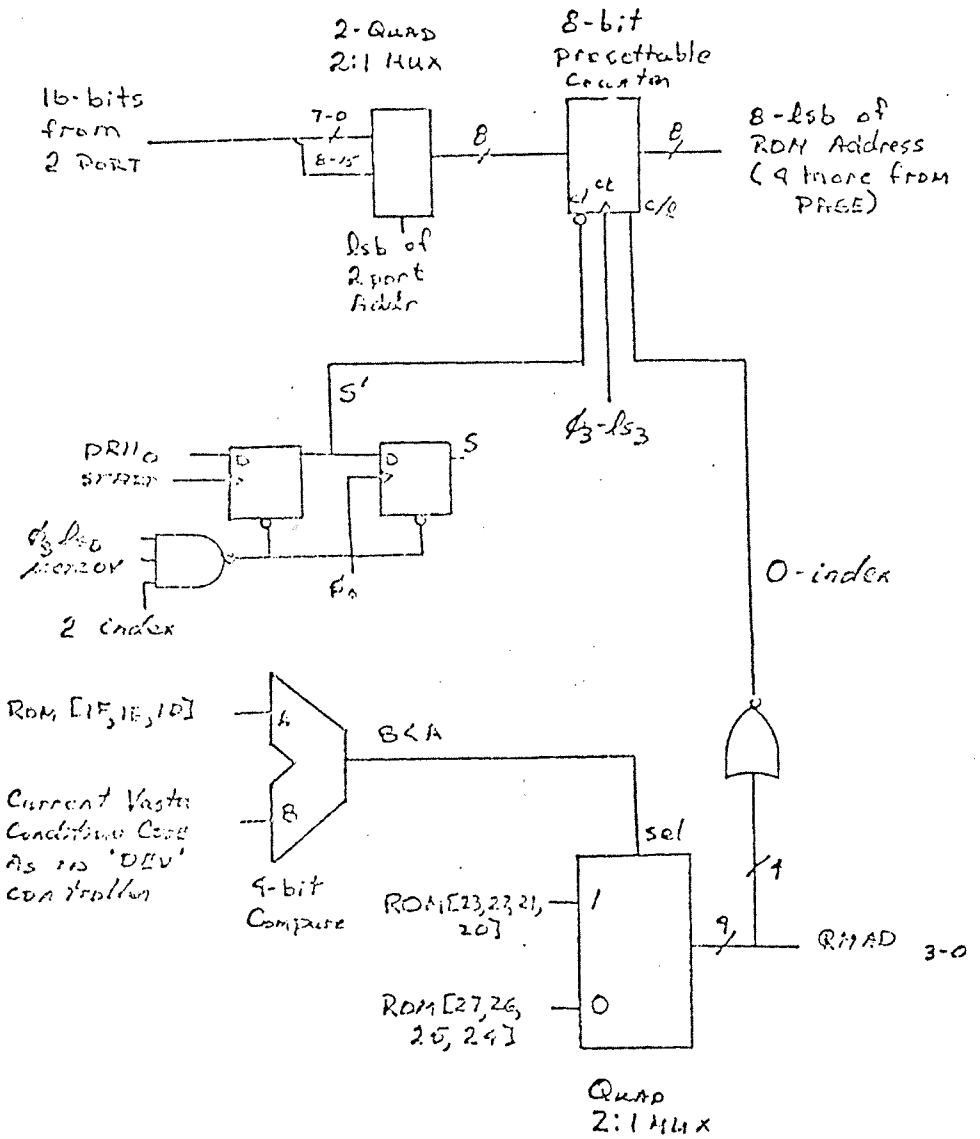


Figure 32: State ø3

Table 9.

Backplane pin assignments

pin #	signal
1	ground
3	CCD clock phase 1
5	" " " 3
7	extender board word 0
9	1
11	2
13	3
15	4
17	5
19	6
21	7
23	SHMODE bit 0
25	1
27	2
29	LSR parallel data, bit 0
31	1 (reserved)
33	LSR clock
35	LSR parallel/serial
37	SHifter clock
39	SHifter P/S
41	SHifter A/B
43	-
45	-
47	-
49	-
51	-
53	LSR serial data in
55	LSR serial data out
57	SHifter serial data in
59	SHifter serial data out
61	extender board word 8 (word 0 of phrase 1)
63	9
65	10
67	11
69	12
71	13
73	14
75	15
77	-5 volt supply
79	ground

Table 10.

Backplane continued

pin #	signal
2	+5 volt supply
4	CCD clock phase 2
6	CCD clock phase 4
8	reset
10	-
12	ICU opcode bit 0
14	1
16	2
18	3
20	'B' line pullup resistor reference
22	'fuzzy plus' summing line.
24	ICU clock
26	SHCCIRC bit 0
28	1
30	2
32	3
34	'A' bus, bit 0 (LSB!)
36	1
38	2
40	3
42	4
44	5
46	6
48	7
50	WK/BK address bit 0
52	1
54	2
56	3
58	4
60	5
62	6
64	7
66	8
68	9
70	-
72	-
74	WK enable low
76	BK enable high
78	+12 volt supply
80	+5 volt supply

Chapter VI

CONCLUSIONS

This chapter discusses possible changes to VASTOR, the things it is known to do badly, and some things we hope it will do well.

6.1 WHERE TO GO

The 'proposed controller' and several more boards of VASTOR words are needed before the device is 'economic' for anything at all on the PDP-11. The 'microprocessor' of chapter 1 should remain a software fiction until more is known about it.

'Extender boards' have been considered for attaching VASTOR to special hardware and for accelerating important functions that overpower ICU's.

1. VASTOR could be seen as a front end for mass storage devices (e.g. disk) with each word corresponding to a different area (or unit). This results in the best efficiency when many units are accessible at one time. Data could either be processed as it came off the (e.g.) disk or moved

into backing or working stores for more cogitation.

2. Telephone exchanges have been suggested as an example of an application for a little special control hardware per VASTOR word.

3. One function that might be worth accelerating is an exact '+/' (summing a bit field, perhaps of length 1, over all words).

4. A way of finding the 'first' marked word may be useful in associative sorts, searches, matches etcetera. This is fairly cheap to do.

Alternatives to the ICU as processing element have been considered, of which one is faithful to VASTOR's bit orientation and one not.

1. A more conventional 4-bit slice would offer faster processing for two reasons: firstly because bipolar speeds are available and secondly because of the inherent parallelism of the slice size. This greatly increases the processing cost per word.

2. An interesting way to stay at the bit level is to use RAM for processing as well as for storage. This processing RAM would be a separate chip whose address lines are derived from the data lines of

storage RAM. This implements arbitrary truth tables directly. One function that stands to profit is '+', because this is tedious with the ICU's instruction set. The main cost is the loss of 'OEN', whose selective write-back capacity is useful enough that it would have to be duplicated externally. ROM could be used instead of RAM for processing, but a developmental machine should use RAM.

6.2 LIMITATIONS OF VASTOR

Algorithms which involve random shuffles of data among words are slow because of the limited backplane bandwidth available. They may be implemented in roughly the same way as sorting.

Difficulty with I/O transfers seem to be inherent in the machines associative structure, though VASTOR makes reasonable use of available bandwidth for this.

The ICU's microsecond cycle time and limited instruction set make VASTOR do arithmetic slowly relative to a more conventional machine, for which 'add' at least is somewhat optimized. Arithmetic looks better on VASTOR when unusual field sizes (e.g. 33 bits) are desired, and when a good number of VASTOR boards exist.

6.3 APPLICATIONS

VASTOR was originally thought of as an inexpensive machine with some architectural relation to APL by virtue of its vector orientation. This means that its suitability for a particular application can be understood in terms of the operators required for that application and those that VASTOR handles well.

The machine could probably be used advantageously as a text store when associative lookup is required, as for editing or symbol table uses. Some database management functions fall under this heading.

One application discussed for extender boards for VASTOR is telephone line monitoring. Each word could have one or a few telephone line interfaces (tone decoders etc.) to monitor and accumulate data on for later attention by a conventional processor.