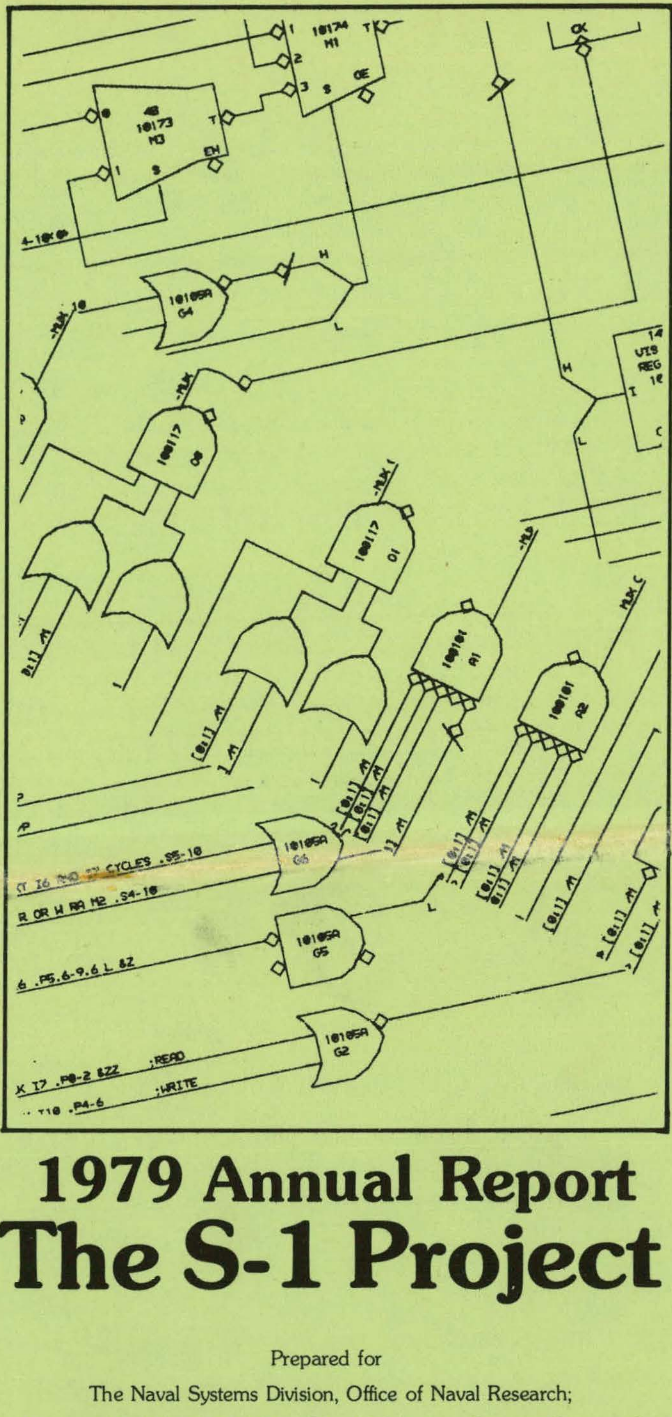
UCID-18619

Volume II: Hardware



1979 Annual Report The S-1 Project

Prepared for
The Naval Systems Division, Office of Naval Research;
The Command and Control Division, Naval Electronics Systems Command; and
The Command, Control, Communication, and Intelligence Program Office,
Naval Material Command. Work in part performed under the auspices of
the U.S. Department of Energy under Contract No. W-7405-ENG-48.

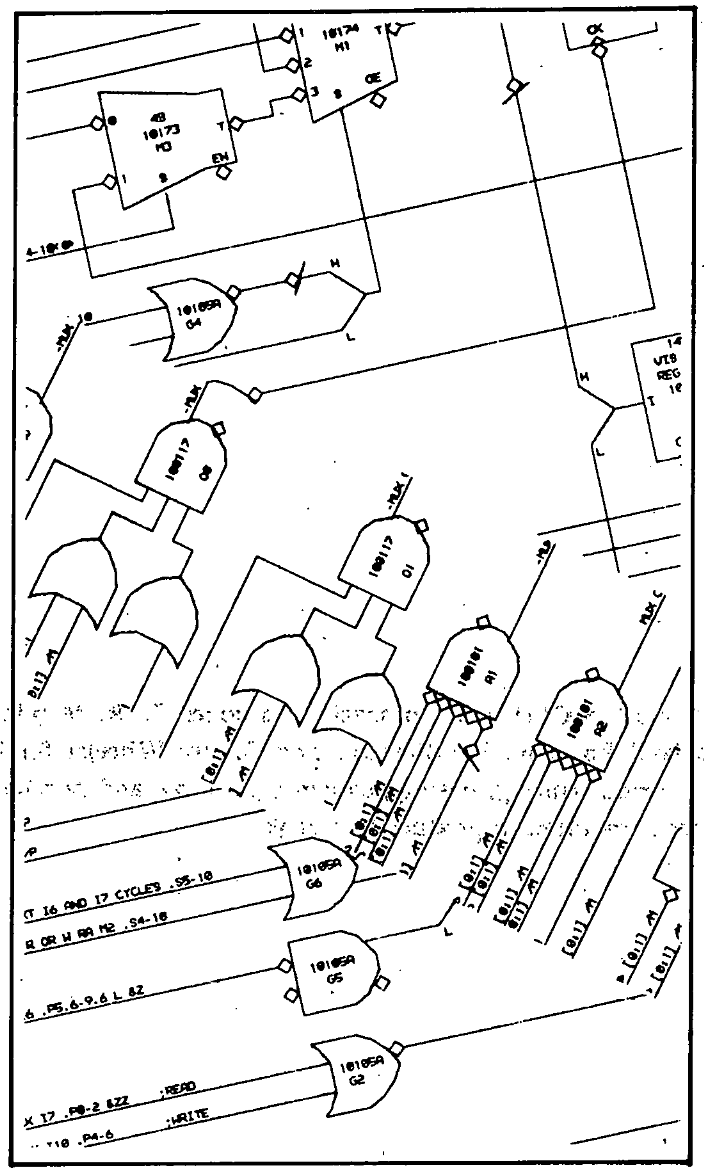
LAWRENCE LIVERMORE LABORATORY

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Price: Printed Copy $; Microfiche $3.50

Page Range	Domestic Price	Page Range	Domestic Price
001-025	$ 5.00	326-250	$18.00
026-050	6.00	351-375	19.00
051-075	7.00	376-400	20.00
076-100	8.00	401-425	21.00
101-125	9.00	426-450	22.00
126-150	10.00	451-475	23.00
151-175	11.00	476-500	24.00
176-200	12.00	501-525	25.00
201-225	13.00	526-550	26.00
226-250	14.00	551-575	27.00
251-275	15.00	576-600	28.00
276-300	16.00	601-up[1]	
301-325	17.00		

[1] **Add 2.00 for each additional 25 page increment from 601 pages up.**

Volume II: Hardware



1979 Annual Report
The S-1 Project

Prepared for
The Naval Systems Division, Office of Naval Research
The Command and Control Division,
Naval Electronics Systems Command
The Command, Control, Communication, and Intelligence
Program Office, Naval Material Command

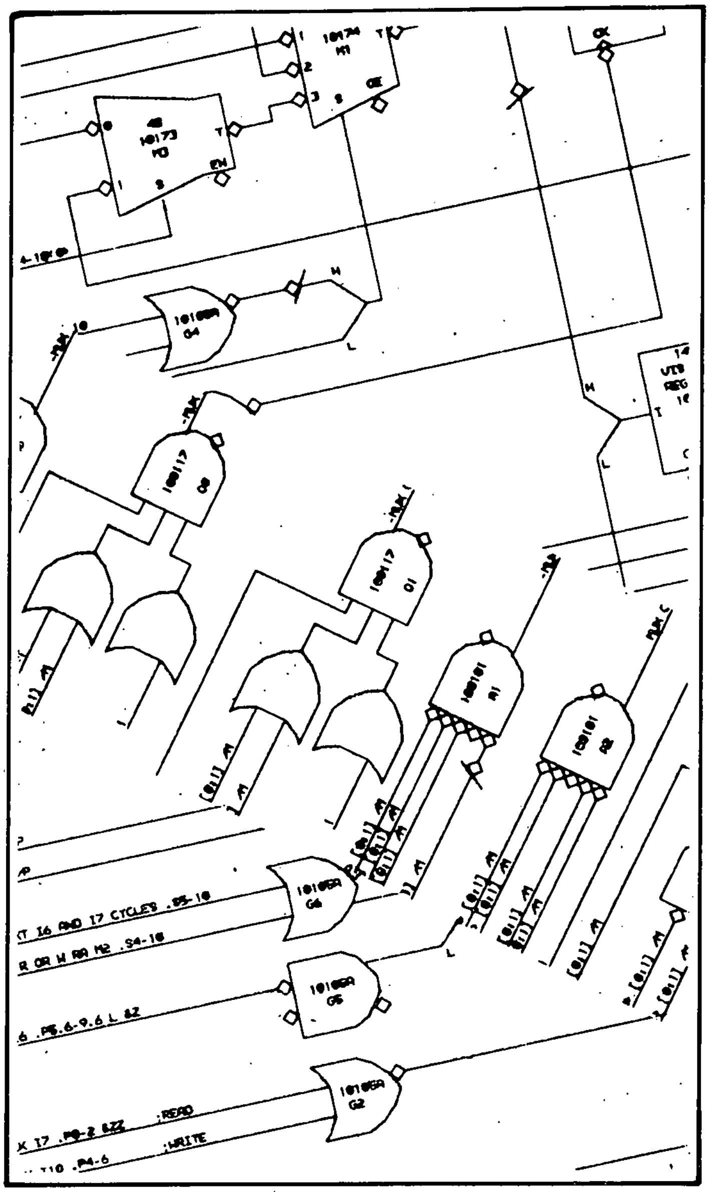
LAWRENCE LIVERMORE LABORATORY

[illegible]

Thanks to Christine Ghinazzi, Lois Jones, L. R. Mendonca, Roland Portman, Joe Simpson, and Cindra Wheeler for help in various cutting, pasting, copying, proofing, and purchasing chores required to produce this book.

CONTENTS

1



Highlights of the Design of the Mark IIA Uniprocessor

(SMI-2)

William R. Bryson, P. Michael Farmwald, Thomas M. McWilliams, and Jeffrey B. Rubin

1 Introduction

These drawings and the accompanying text provide a preliminary look at a sampling of the hardware in the S-1 Mark IIA uniprocessor; the 1980 annual report will provide a complete set, corresponding to the system as built and debugged. The drawings, created with the D graphics editor, are used as input to the SCALD computer-aided design system, and they use the notation described in the *SCALD II User's Manual* elsewhere in this annual report.

Extensive use of pipelined parallelism contributes greatly to the high performance of the Mark IIA processor. Unlike most modern computers, which have used pipelining primarily in the execution of instructions (that is, the streaming of vectors of operands through pipelined arithmetic or logical operation functional units), the Mark IIA pipelines the fetching of instructions and the preparation of operands as well, and it applies pipelining to the processing of every instruction, whether scalar or vector.

Some stages of the pipeline, particularly those dealing with operand address arithmetic and instruction execution, necessarily have a wide variety of functions, since the pipeline must process a wide range of instructions. This variability in operation is effected through the unusually extensive use of microcode.

The processor consists of five microengines (extremely fast, relatively special-purpose programmable controllers) operating in parallel to provide high performance (Figure 1-1). Four of the microengines form the instruction pipeline, which consists of the instruction-fetch, instruction-decode, operand-preparation, and arithmetic segments. (Some of the segments are themselves internally pipelined, a level of detail not shown in the figure.) A single microengine handles memory traffic in parallel with the operation of the instruction pipeline. A one-processor system can be configured by connecting an S-1 Mark IIA uniprocessor directly to a memory controller; this requires neither hardware nor microcode changes.

The designs of the S-1 Mark I and Mark IIA uniprocessor pipelines constituted significant advances in computer technology. The Mark I introduced a new, simple branch prediction strategy to forecast the outcome of each test-and-branch operation in an instruction stream before its execution, thereby allowing subsequent instructions to be prepared without disruption or time loss when the forecasts are correct. The Mark I also refined the use of dual cache memories (one for instructions, one for data) to increase total cache bandwidth. The Mark IIA allows advance computation of simple operations in early pipeline stages; this technique minimizes idling of pipeline stages when a computation (particularly, an operand-address computation) depends on some as-yet unavailable result. The Mark IIA includes refined control mechanisms to coordinate the operation of multiple pipeline stages controlled by the independent programmable microengines.

The S-1 Mark IIA also employs vector operations to achieve high arithmetic performance. Certain vector operations encountered frequently in signal processing use multiple functional units in the pipelined arithmetic module, achieving a peak computation rate on the S-1 Mark IIA Uniprocessor of 400 million floating-point operations per second and using the maximum cache bandwidth, which is unavailable in scalar mode.

The drawings are logically divided into two groups: the *IBOX*, which performs instruction fetching, instruction decoding, and operand preparation; and the *ABOX*, which performs arithmetic processing.

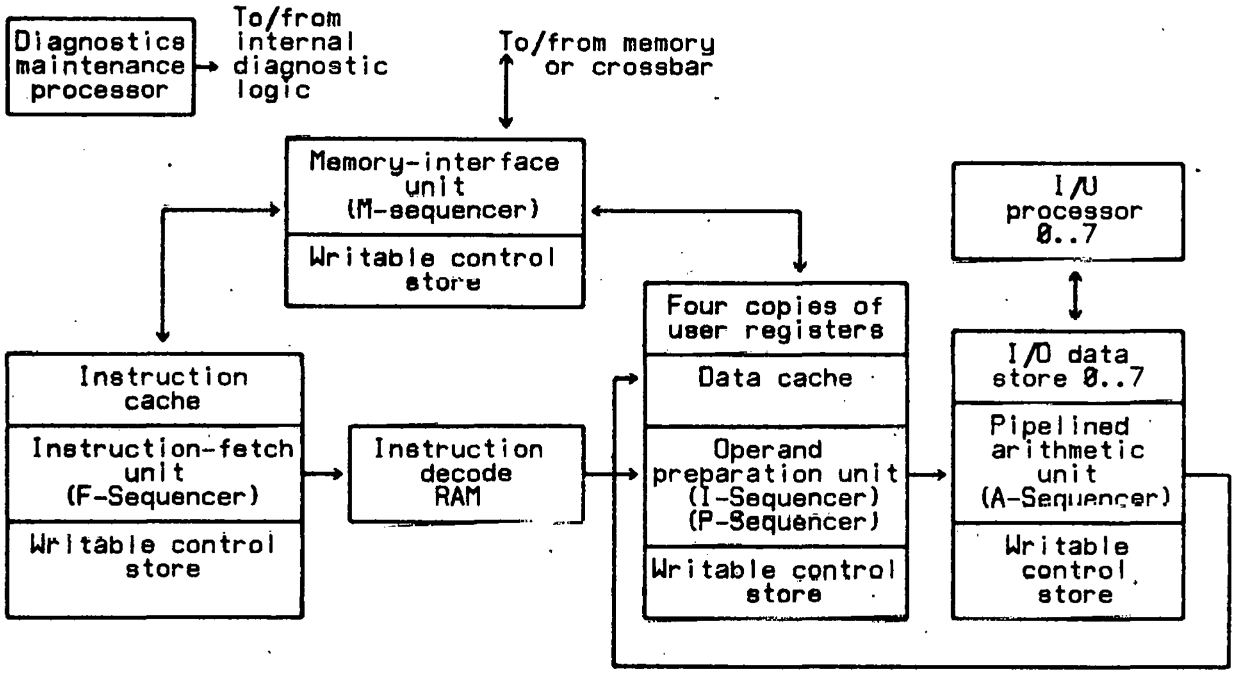

Diagnostics maintenance processor
To/from internal diagnostic logic
To/from memory or crossbar
Memory-interface unit (M-sequencer)
Writable control store
I/O processor 0..7
Four copies of user registers
Data cache
Operand preparation unit (I-Sequencer) (P-Sequencer)
Writable control store
I/O data store 0..7
Pipelined arithmetic unit (A-Sequencer)
Writable control store
Instruction cache
Instruction-fetch unit (F-Sequencer)
Writable control store
Instruction decode RAM

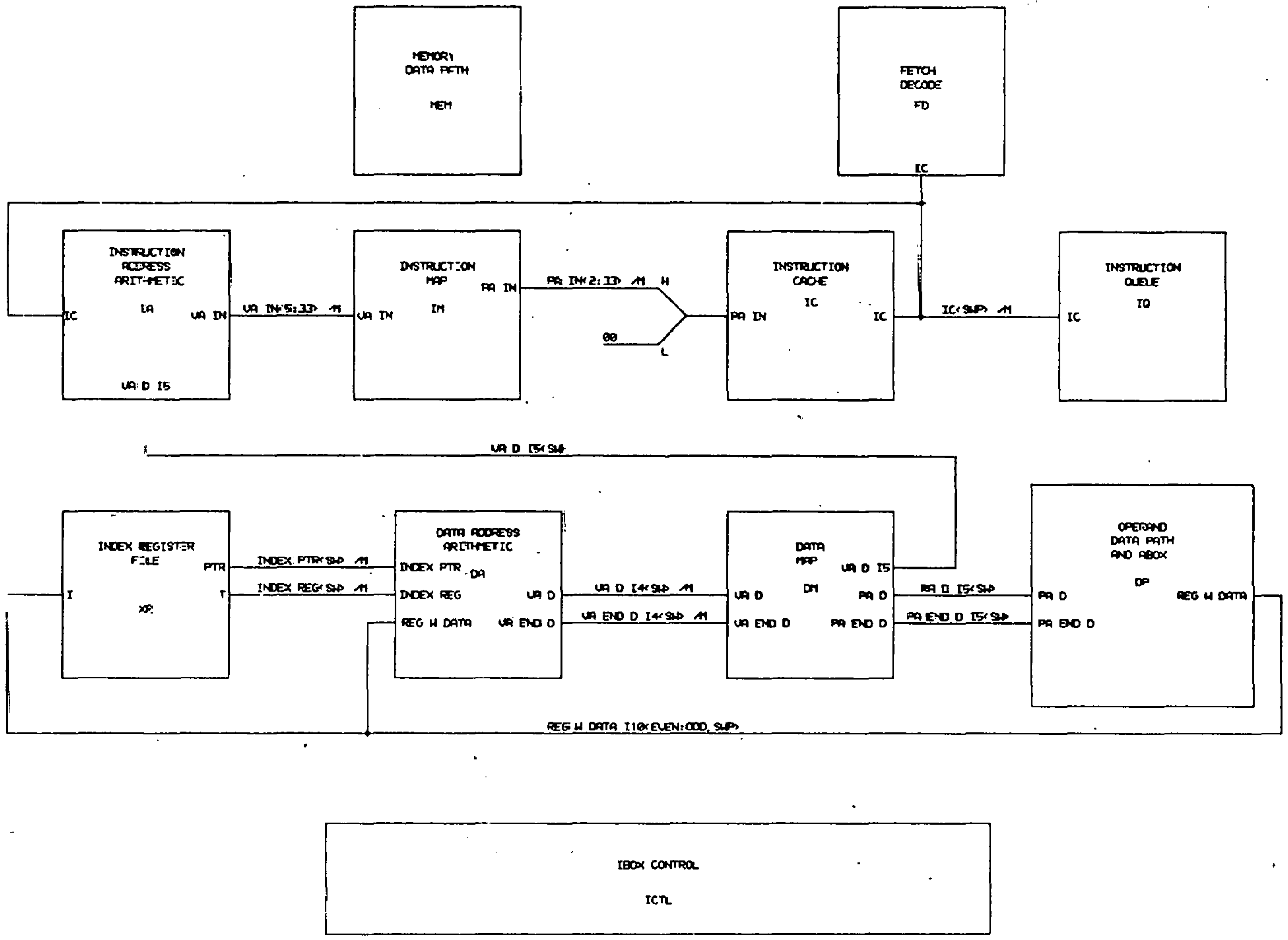
Figure 1-1
Internal structure of the S-1 Mark IIA processor

2 IBOX drawings

2.1 IBOX

Drawings: ITOP1, ITOP2, IBOXC

These drawings show the top level organization of the Mark IIA uniprocessor. They indicate the interconnections among the major submodules such as the instruction and data caches and maps, index register files, instruction and data address calculation units, and the operand data paths. They also show the major control units in the IBOX: the microcoded sequencers, pipeline control units, and the write queue control. Finally, they present the symbolic field definitions that are used throughout the remainder of the IBOX. These are in the form of text macro definitions, making it relatively easy to modify or add new fields.



IBOX TOP LEVEL

```
COMMENT

IR, IRX FIELDS

DEFINE

OPCODE = 0:11

OD = 0:11
OD.X = 0
OD.MODE= 1:5
OD.MODE.0.3 = 1:4
OD.MODE.4 = 5
OD.F = 6:11
OD.F.0 = 6
OD.F.1.5 = 7:11
OD1 = 12:23
OD2 = 24:35

OD1.X = 12
OD1.MODE= 13:17
OD1.MODE.4 = 17
OD1.F = 18:23
OD1.F.0 = 18
OD1.F.4 = 22
OD1.F.5 = 23
OD1.F.1.5 = 19:23

OD2.X = 24
OD2.MODE = 25:29
OD2.MODE.4 = 29
OD2.F = 30:35
OD2.F.0 = 31
OD2.F.4 = 34
OD2.F.5 = 35
OD2.F.1.5 = 31:35

EXT.TAG = 0:4
EXT.TAG.3.4 = 3:4
EXT.TAG.4 = 4
EXT.REG = 5:9
EXT.LA.LD = 5:35
EXT.SD = 10:35
```

```
COMMENT

GENERAL FIELD DEFINITIONS

DEFINE

QWP = 0:9
HWP = 0:19
SWP = 0:39
DWP = 0:79

QW = 0:8
HW = 0:17
SW = 0:35
DW = 0:71

EVEN = 0
ODD = 1

P = 0
G = 1
```

```
COMMENT

PROCESSOR STATUS FIELDS

DEFINE

REGISTER.FILE = 0:3
PRIORITY = 4:6
EMULATION = 7:8
RING.ALARM = 9:10
VIRTUAL.MACHINE.MODE = 11
FLOW.TABLE = 12:13
FLOW.ENABLE = 14
FLOW.TRAP.ENABLE = 15
TRACE.ENABLE = 16
TRACE.PENDING = 17
CALL.TRACE.ENABLE = 18
CALL.TRACE.PENDING = 19
UNMAPPED.MODE = 20
```

```
COMMENT

ACCESS MODES AND RING BRACKETS

DEFINE

WRITE.PERMIT = 0
EXECUTE.PERMIT = 1
READ.PERMIT = 2
IO.PAGE = 3
WB = 0:1
EB = 2:3
RB = 4:5
```

```
COMMENT

EMULATION MODE FIELDS

DEFINE

TEN.OP = 0:8
TEN.AC = 9:12
TEN.I = 13
TEN.XR = 14:17
TEN.Y = 19:35
TEN.Y.REG = 32:35
```

```
COMMENT

TIMING VERIFIER VALUES

DEFINE

CORR = 9
CORR4 = 4

CASE0 = 0
CASE0L = 0 L
CASE1 = 1
CASE1L = 1 L
```

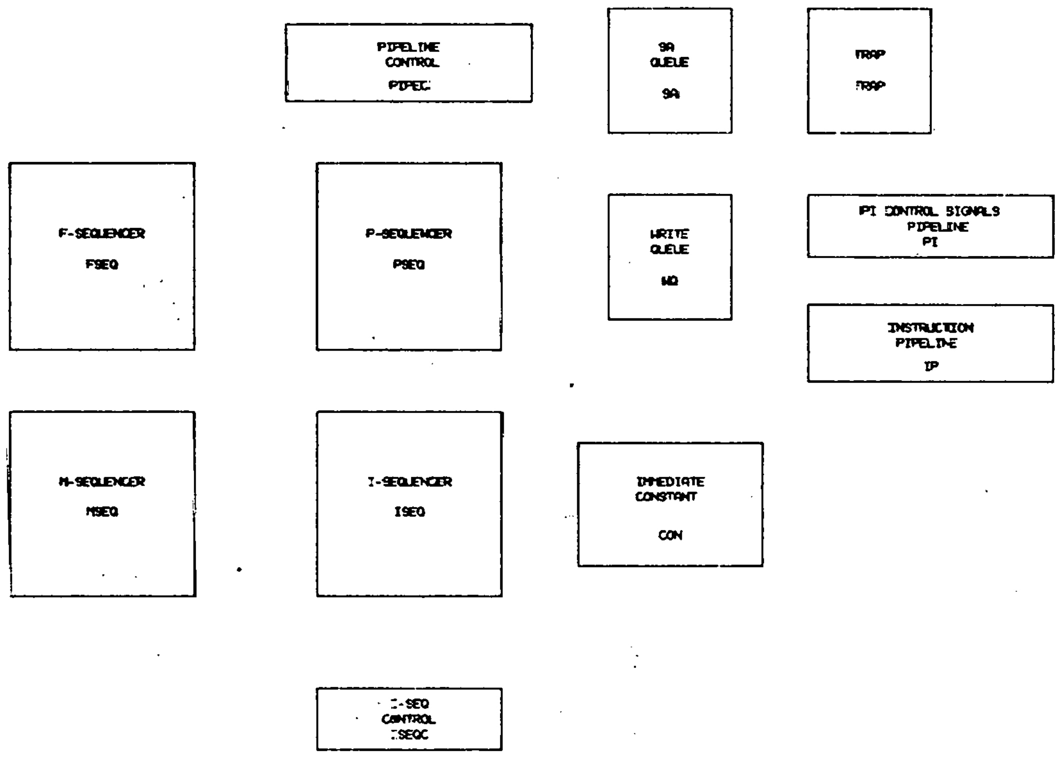
```
COMMENT

SPECIFIES WHETHER TO EXPAND FOR
TIMING VERIFIER OR LAYOUT PROGRAM

DEFINE

TIMVER = 1
```

IBOX TOP LEVEL



IBOX CONTROL

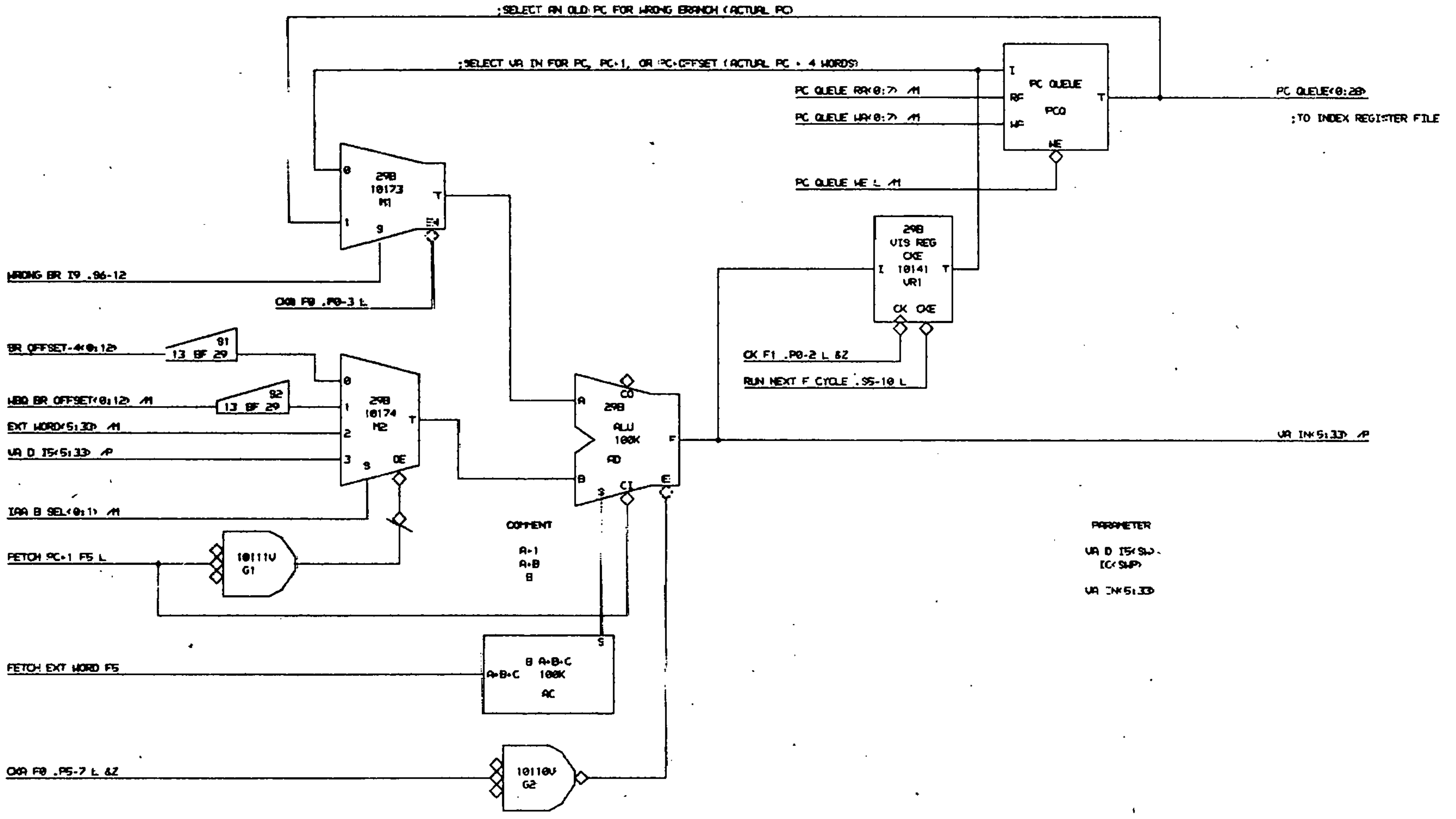
2.2 IADRA

Drawing: IADRA 1

The instruction address arithmetic (IADRA) presents the logic that computes the (word) address for the next instruction fetch. VA IN<5:33> is the primary output of this unit; it is the address of the next word to fetch from the instruction cache. VA IN is often referred to as the PC (program counter). The IADRA is controlled primarily by the F-Sequencer, which executes one microinstruction for each word fetched from the instruction cache. The IADRA is capable of computing the new PC in several ways. The simplest way is to fetch the sequentially next location (PC+1). Next, all skip instructions that are predicted to skip are computed by taking the current PC and adding the appropriate skip offset. Finally, jump instructions fall into several classes. Some of these are calculated by the IADRA and some are computed by the I-Sequencer and "force-fed" to the IADRA. The IADRA can calculate PR-type jumps, extended PC-relative jumps, and jumps to absolute addresses.

One final source for the new PC is the PC queue plus some offset. The PC queue maintains a history of the last 256 PCs. Every time the PC of a new instruction is calculated, it is added to the end of the PC queue. Whenever the ABOX finishes the execution of an instruction, it signals the IADRA to remove from the PC queue the entry corresponding to that instruction. Thus the PC queue contains the address of the first word of each instruction that has been started down the pipeline but has not completed. When the ABOX detects certain exception conditions, it is necessary to reset the PC to one of the old values in the queue, or, in the case of a branch that was predicted incorrectly, to set the PC to the correct value.

The PC queue is also used (in the absence of exception conditions) to read out the "current" value of the PC for later stages of the pipeline.



COMMENT

VA IN GETS LOADED FROM
PC, PC+1, PC+OFFSET, OR IC
EVERY CYCLE

COMMENT

A+1	PC+1	SEQUENTIAL EXECUTION
A+B	PC+SHORT OFFSET	SKIP AND PR BRANCH
A+B	PC+LONG OFFSET	EXTENDED BRANCH
B	LONG ADDRESS	EXTENDED BRANCH
A	PC	XCT
A+B	OLD PC+WB OFFSET	WRONG BRANCH

COMMENT

PC QUEUE(0:28) READS THE PC
CURRENTLY AVAILABLE TO THE IBOX
FOR INDEXING

INSTRUCTION ADDRESS ARITHMETIC

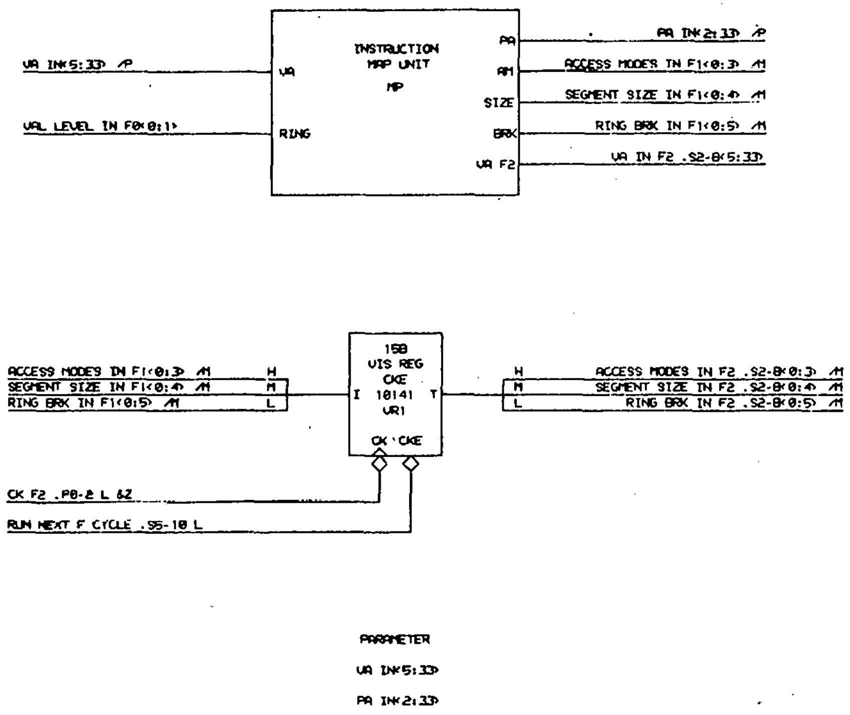
2.3 IMAP

Drawings: IMAP1, IMU

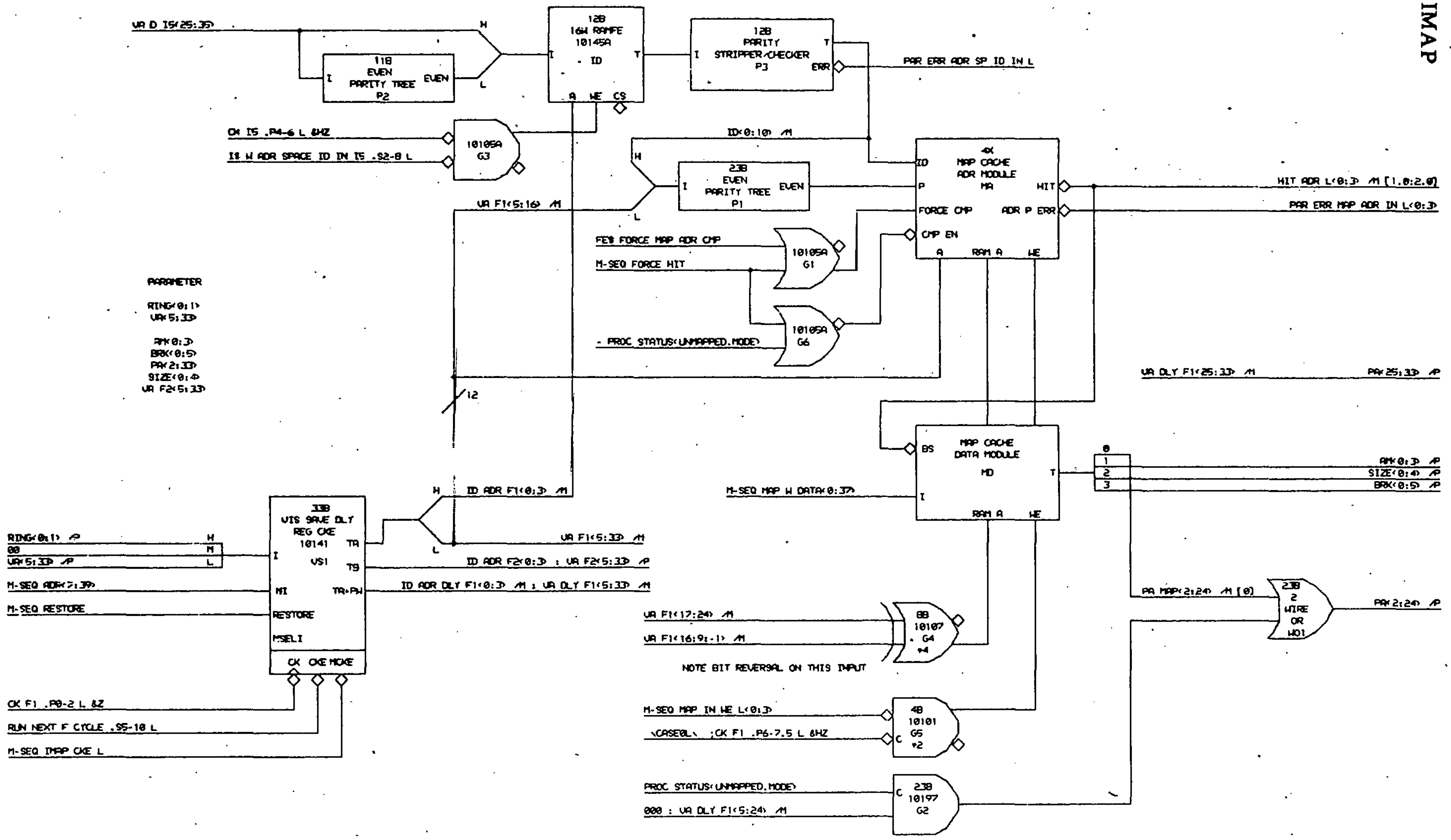
The instruction map unit consists of a 4-way set-associative cache of 1024 mapping entries. This cache has an address and a data part in series. The low 16 incoming virtual address bits address both the address and data parts in parallel (after passing through a simple hashing function) thereby selecting four candidate mapping entries. The address part decides which of the four, if any, contains the mapping information for the given virtual address. The correct entry is then selected and used to create the physical (word) address (PA<2:33>). The address part also indicates when there are no locations in the cache matching the virtual address (map cache miss). The selected entry contains additional information about the physical page being addressed, namely: the access modes, segment size of the containing segment, and the ring brackets, which indicate how much privilege any accessor must have.

The address part of the cache, in addition to comparing the input virtual address to the stored ones, compares the current address space ID against the stored ones. The address space ID is an 11 bit number associated with each ring.

The instruction map is accessed once per cycle during the F1 pipeline cycle. If there is an instruction map cache miss, then the address being translated is not allowed to proceed down the pipe. Instead the M-Sequencer is requested to go to memory (or cache) to compute the mapping information and to load it into the map cache. Once the correct information is loaded the cycle is allowed to repeat and is guaranteed not to miss on this second try.



INSTRUCTION MAP



INSTRUCTION MAP UNIT

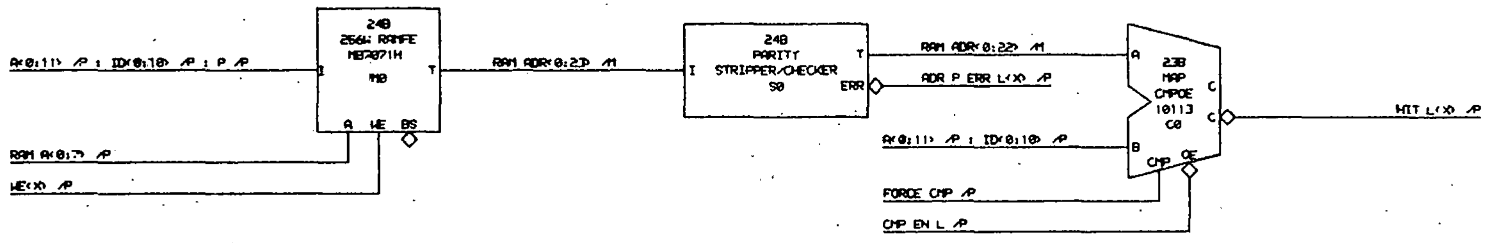
2.4 MAP

Drawings: MADR, MDATA, MAPCMP

These drawings indicate the construction of the map cache address and data modules. The address module consists of four 256 entry RAMs, which store the high order virtual address bits and the address space ID for the virtual address whose mapping information is stored in that location of the map cache. The addressed location is read out and compared against the current virtual address and address space ID to produce the unary "hit lines". These hit lines indicate which of the four elements matched against the virtual address, if any.

The comparator used for this matching is specially constructed to have a "force compare" and a "force no compare" input. These are used to implement "unmapped mode" and to enable the data part's chip select lines for the purpose of writing.

The data part of the map cache consists of four modules of 256 entries by 38 bits. The modules are all in parallel and only one is selected at a time for reading or writing.



PARAMETER

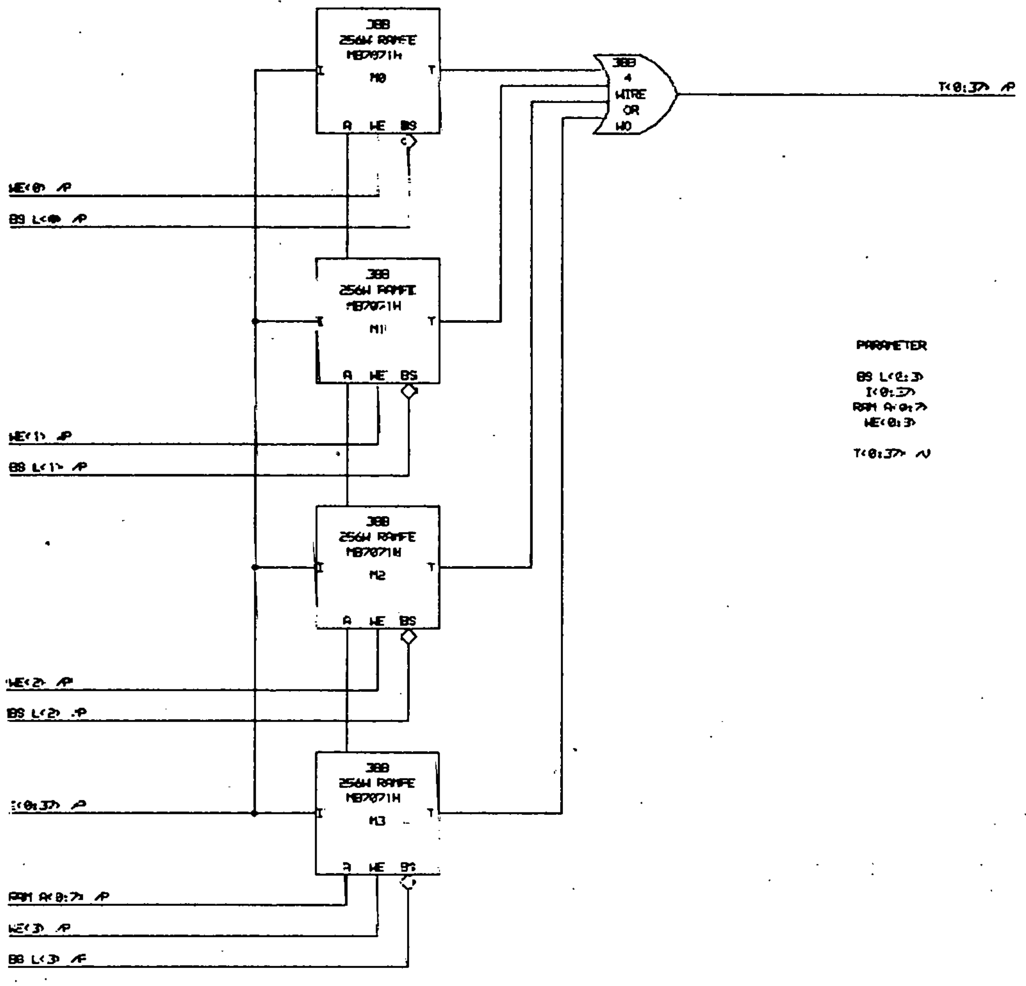
A<0:11>
CMP EN L
FORCE CMP
ID<0:10>
P
RAM A<0:7>
WE<0:3>

ADR P ERR L<0:3> /V
HIT L<0:3> /V

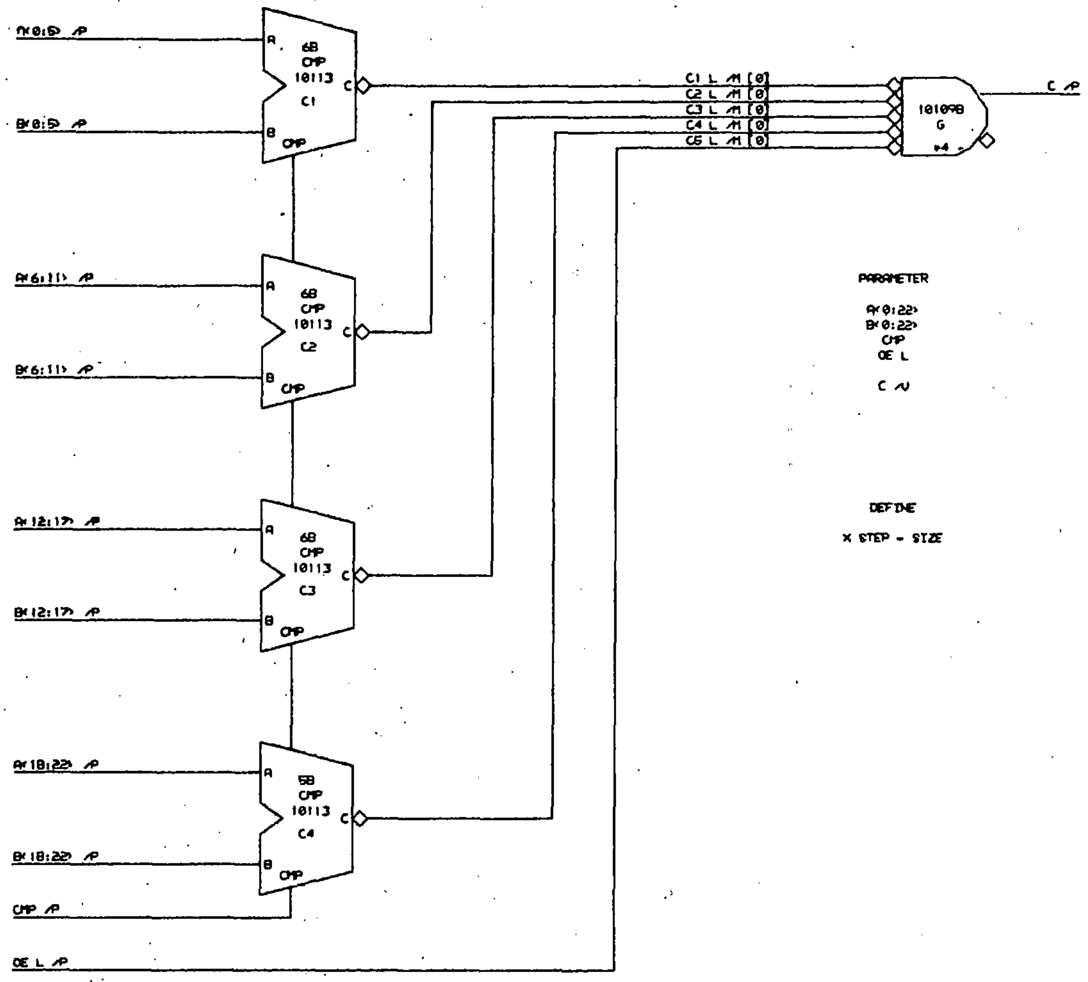
DEFINE

X STEP = 1

MAP CACHE ADR MODULE



MAP CACHE DATA MODULE



MAP CMPOE 10113
SIZE = 23

2.5 IC

Drawings: IC1, ICM ICAM

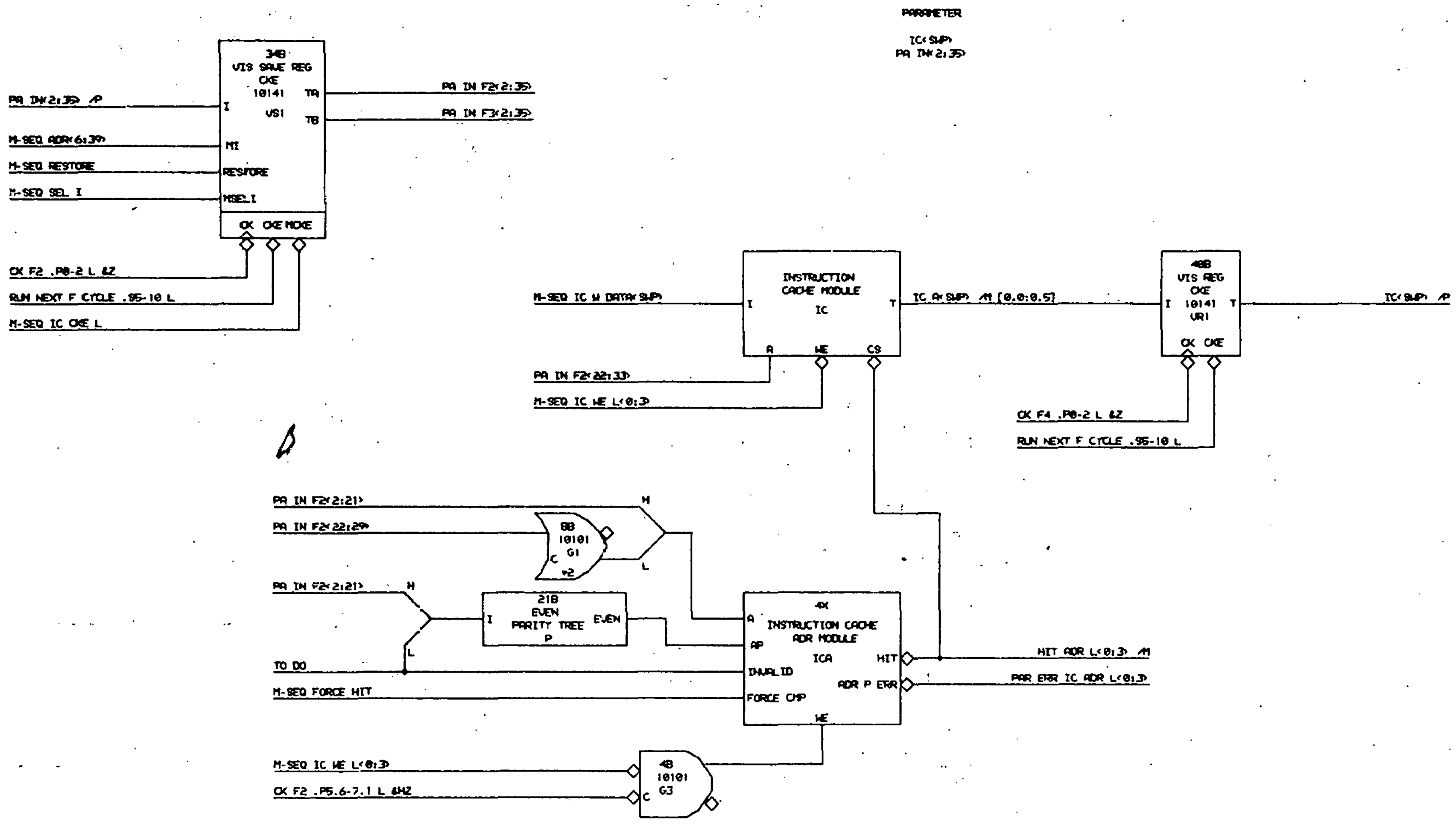
The instruction cache consists of a 16K word data part which is organized as a four-way set associative cache with 16 word lines. The address part of the instruction cache, therefore, consists of four RAMs of 256 entries each. Each RAM holds the address information for an element. Each entry in the address part contains the high order physical address bits corresponding to the line stored in the same element of the data part at the same address. The address part also stores an INVALID bit, indicating that there is not a valid line in the corresponding place in the data part.

The instruction cache is addressed during the F2 and F3 pipeline stages. During the F2 stage the address part is supplied by the physical address that came from the instruction map. The low 4 address bits of the physical word address are used to indicate which word within the 16 word line is being addressed, and so are not used to supply the address part. The next higher order 8 bits locate in the address part the four candidate entries. Each candidate RAM output is compared against the remaining high order physical address bits and the INVALID bit is checked. The output of the address part is a set of unary "hit lines" indicating which element, if any, matched the incoming physical address.

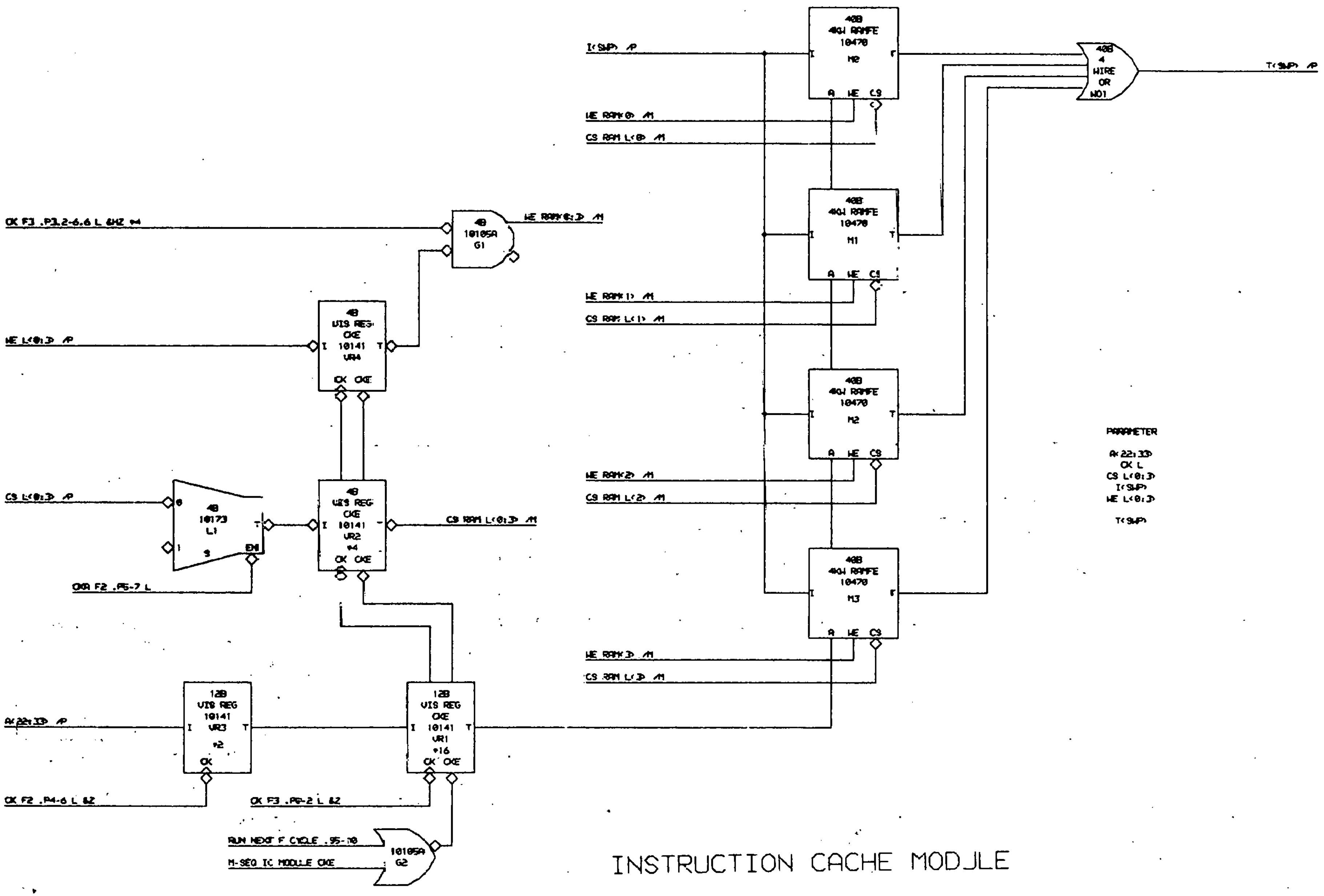
The hit lines are clocked into pipeline registers at the beginning of the F3 stage and are used to select the correct element of the data part. In this case, however, the low four address bits directly address the RAMs to select the one word out of the 16 word line.

The instruction cache is referenced once per cycle, giving an effective instruction cache bandwidth of 20 million instruction words per second.

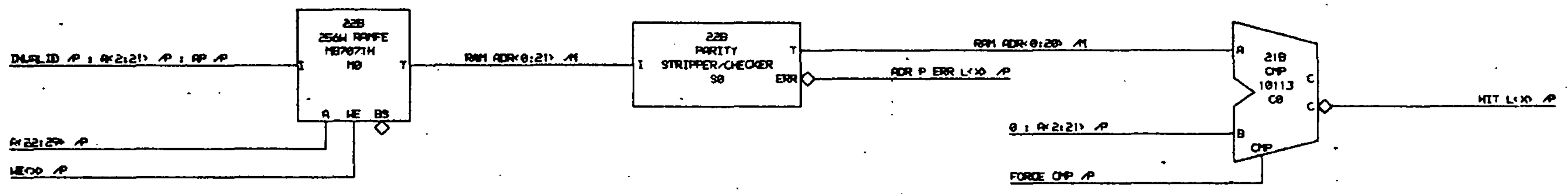
When none of the hit lines are set, an instruction cache miss occurs. This causes the M-Sequencer to fetch the requested word from main memory or secondary cache and load it into the instruction cache. The pipeline is held up at the beginning of the F3 stage until the cache is loaded with the correct memory data. The pipeline is then allowed to proceed with the data part reference.



INSTRUCTION CACHE



INSTRUCTION CACHE MODJLE



PARAMETER

A<2:29>
AP
FORCE CMP
INVALID
WE<0:3>

ADR P ERR L<0:3>
HIT L<0:3>

DEFINE

X STEP = 1

INSTRUCTION CACHE ADR MODULE

2.6 IXREG

Drawings:IXREG1, IXREG2, PIRF, IXRAM, IXRVPR, IRVVP

These pages contain copies of the user register files used for indexing in data address calculations. Some important address calculations require adding two different index registers; hence, there are two copies of the index register file available to be read out in parallel. One of these outputs is optionally shifted left by two bits to provide a word rather than byte offset. The two outputs are added in a fast 36 bit adder and the result is sent to the data address arithmetic.

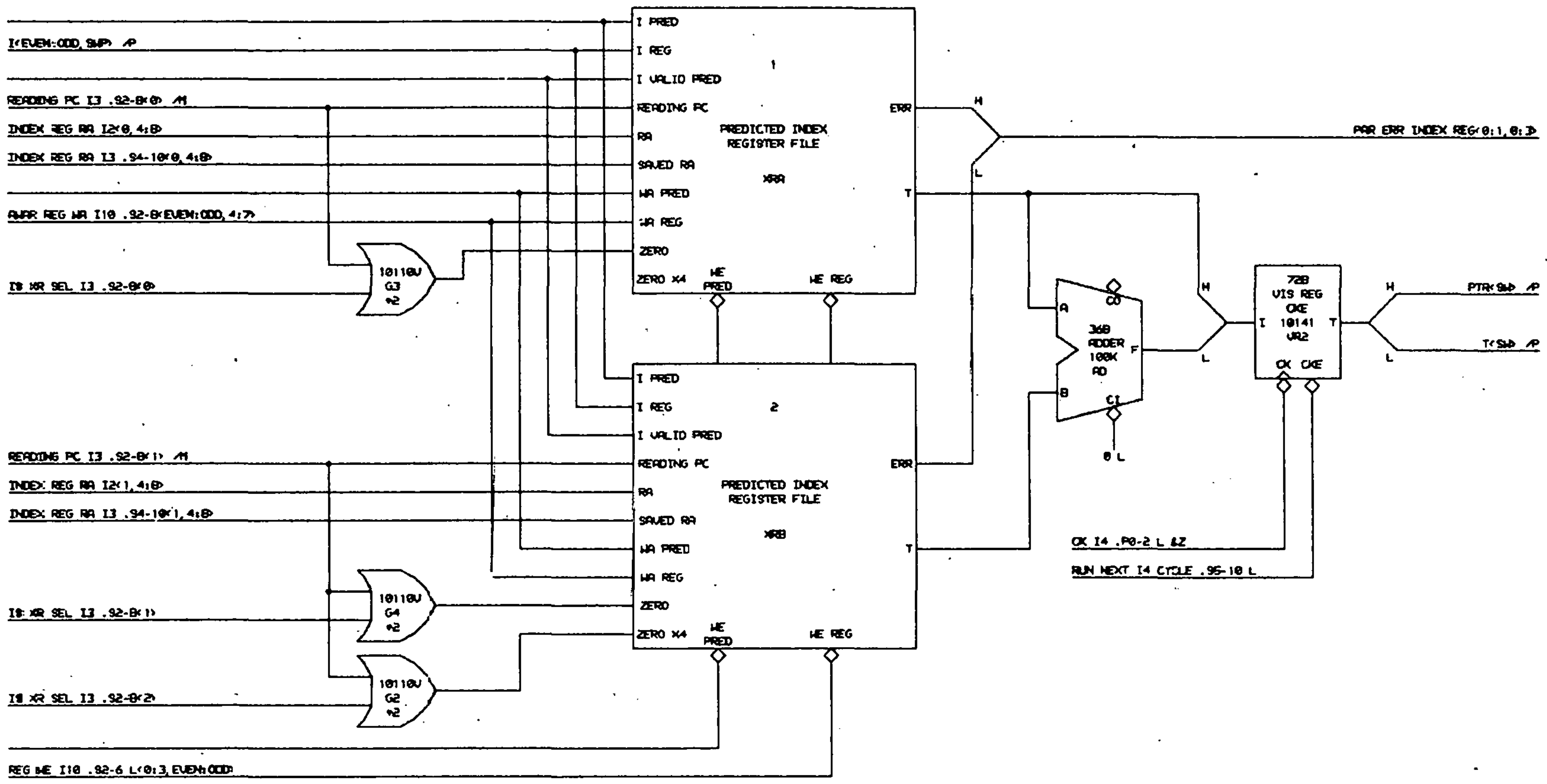
There are two copies of the control logic that select the address for the index registers so that two different addresses may be generated. The addresses may be selected from the OD field, the extended word, the PDP-10 AC or XR fields, or from microcode.

Within each copy of the index register file it is possible to select one of five outputs. The first two outputs are just the index register file and the index register file shifted left by two places. The third output is the current PC. This output is selected whenever the microcode indicates that an index register read operation should select the PC instead of register 3 and register 3 is selected by the index register address multiplexor. The fourth possible output comes from a separate register file which stores "predicted values". Unlike the index register files which have 16 sets, the value predict file has exactly one set. This file contains predicted values for the corresponding index registers in the current register file. As not all index registers have a predicted value, there is another parallel RAM, called the "valid value predict" RAM, which stores one bit for each register indicating that the value predicted version has the correct value. When an index register is read out, the value is taken from the value predict RAM if the valid value predict bit is set for that register. The fifth output is just a shifted version of the value predict output.

The index register file is implemented as an even/odd pair of RAMs, each of which is 256 by four quarter words long. Therefore, a total of 8 quarter words may be written into the index register file in one write cycle. This means that an entire double word result may be written to the index register file. There are 8 write enable lines corresponding to these quarter words. During a read cycle, all four quarter words from one or the other of the RAMs will be read out. It is possible both to read and to write the index register file every cycle. Reading happens during the first half of the cycle and writing during the second half.

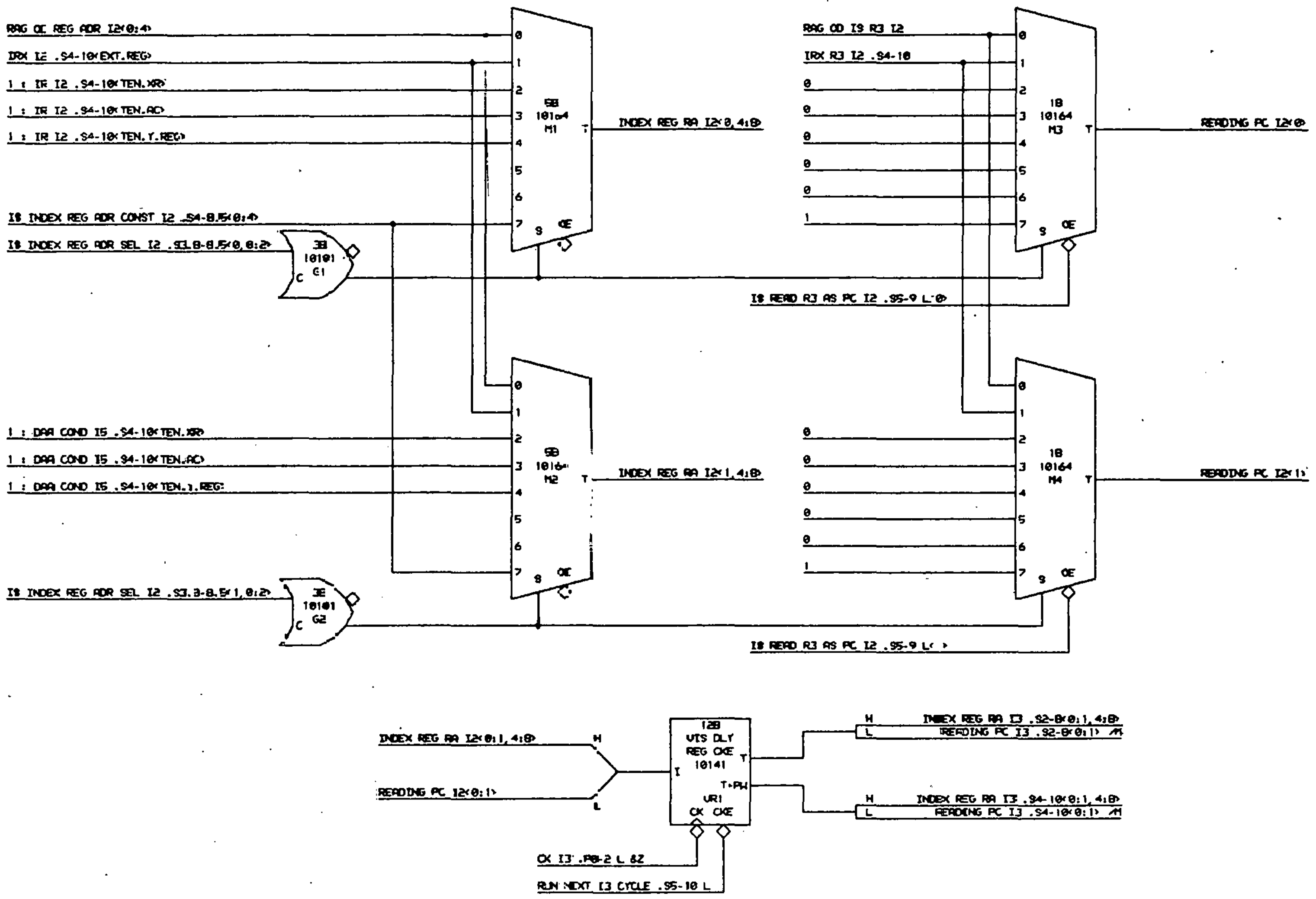
The value predict RAM is implemented as a single file of 32 singlewords. The timing of read operations and write operations is the same as for the index register file.

The valid value predict RAM is similar to the value predict RAM except that it must read out faster in order to select between the index register file and the value predict file. It is implemented with ECL 100K RAMs which also ease the addressing requirements (they have separate read and write addresses, as well as output latches).



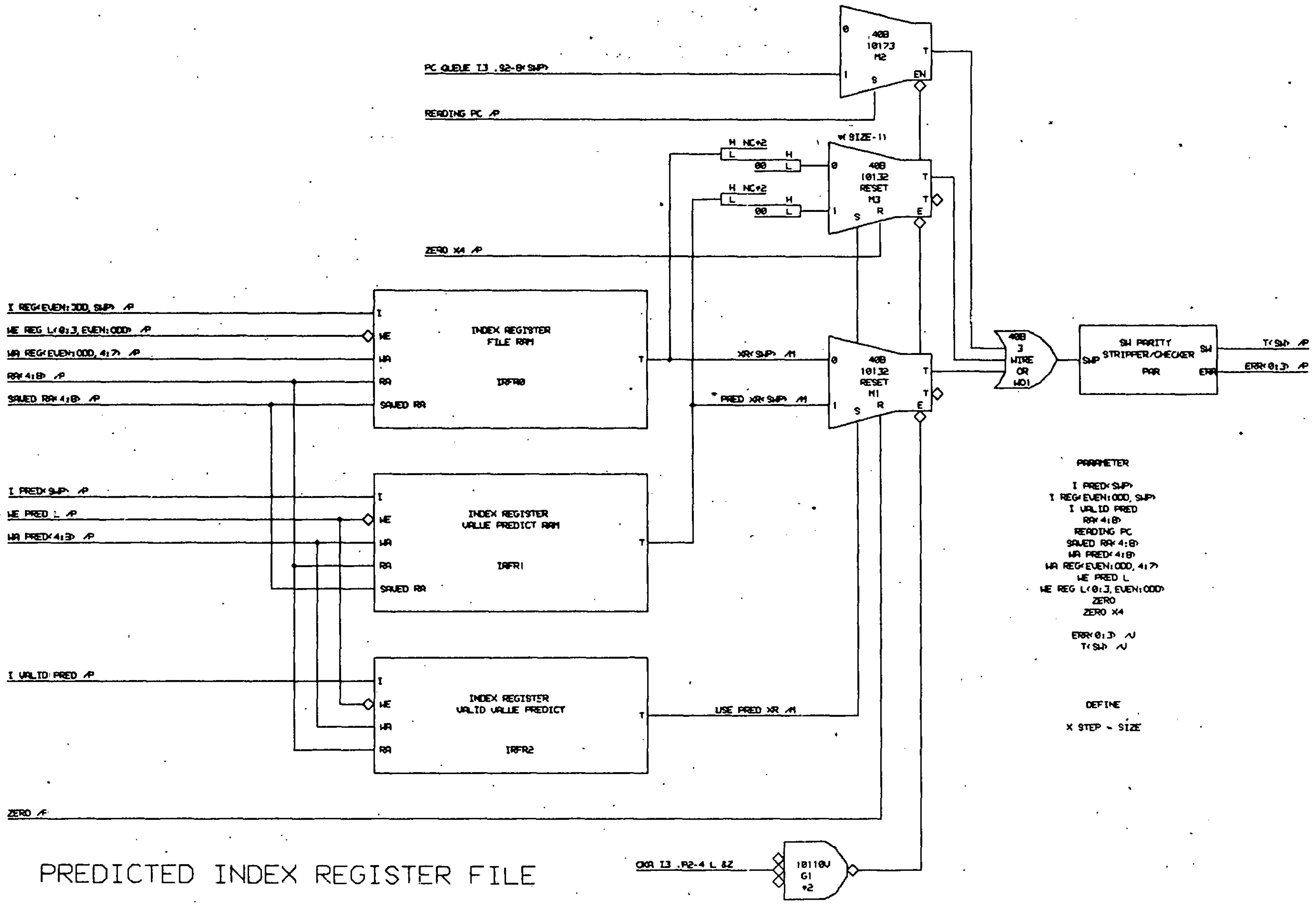
PARAMETER
I<EVEN:ODD, SWP>
PTR<SW> AU
T<SW> AU

INDEX REGISTER FILE



RAG OC REG ADR I2<0:4>
IRX I2 .S4-10<EXT.REG>
1 : IR I2 .S4-10<TEN.XR>
1 : IR I2 .S4-10<TEN.AC>
1 : IR I2 .S4-10<TEN.Y.REG>
I$ INDEX REG ADR CONST I2 .S4-8.5<0:4>
I$ INDEX REG ADR SEL I2 .S3.8-8.5<0,0:2>
3B 10101 G1
5B 10164 M1
INDEX REG RA I2<0,4:8>
1 : DAA COND I5 .S4-10<TEN.XR>
1 : DAA COND I5 .S4-10<TEN.AC>
1 : DAA COND I5 .S4-10<TEN.Y.REG>
5B 10164 M2
INDEX REG RA I2<1,4:8>
I$ INDEX REG ADR SEL I2 .S3.8-8.5<1,0:2>
3E 10101 G2
RAG OD I$ R3 I2
IRX R3 I2 .S4-10
1B 10164 M3
READING PC I2<0>
I$ READ R3 AS PC I2 .S5-9 L<0>
1B 10164 M4
READING PC I2<1>
I$ READ R3 AS PC I2 .S5-9 L< >
S
OE
T
C
INDEX REG RA I2<0:1,4:8>
READING PC I2<0:1>
H
L
12B
VTS DLY
REG CKE
10141
VR1
CK CKE
T+PW
CK I3 .P8-2 L 8Z
RUN NEXT I3 CYCLE .S5-10 L
INDEX REG RA I3 .S2-8<0:1,4:8>
READING PC I3 .S2-8<0:1> /M
INDEX REG RA I3 .S4-10<0:1,4:8>
READING PC I3 .S4-10<0:1> /M

INDEX REGISTER FILE



PARAMETER

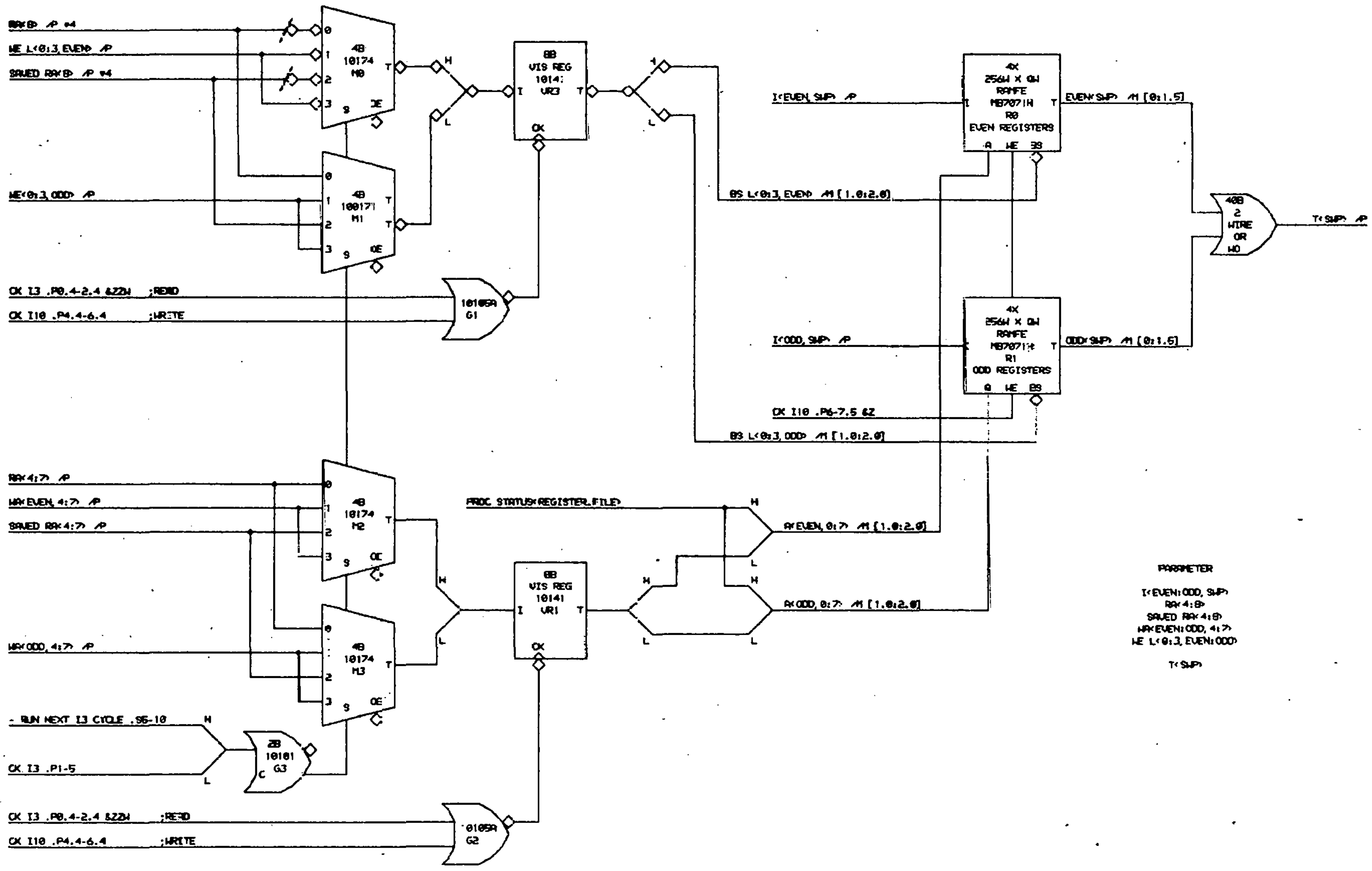
I PRED<SWP>
I REG<EVEN:ODD, SWP>
I VALID PRED
RA<4:8>
READING PC
SAVED RA<4:8>
WA PRED<4:8>
WA REG<EVEN:ODD, 4:7>
WE PRED L
WE REG L<0:3, EVEN:ODD>
ZERO
ZERO X4

ERR<0:3> AJ
T<SW> AJ

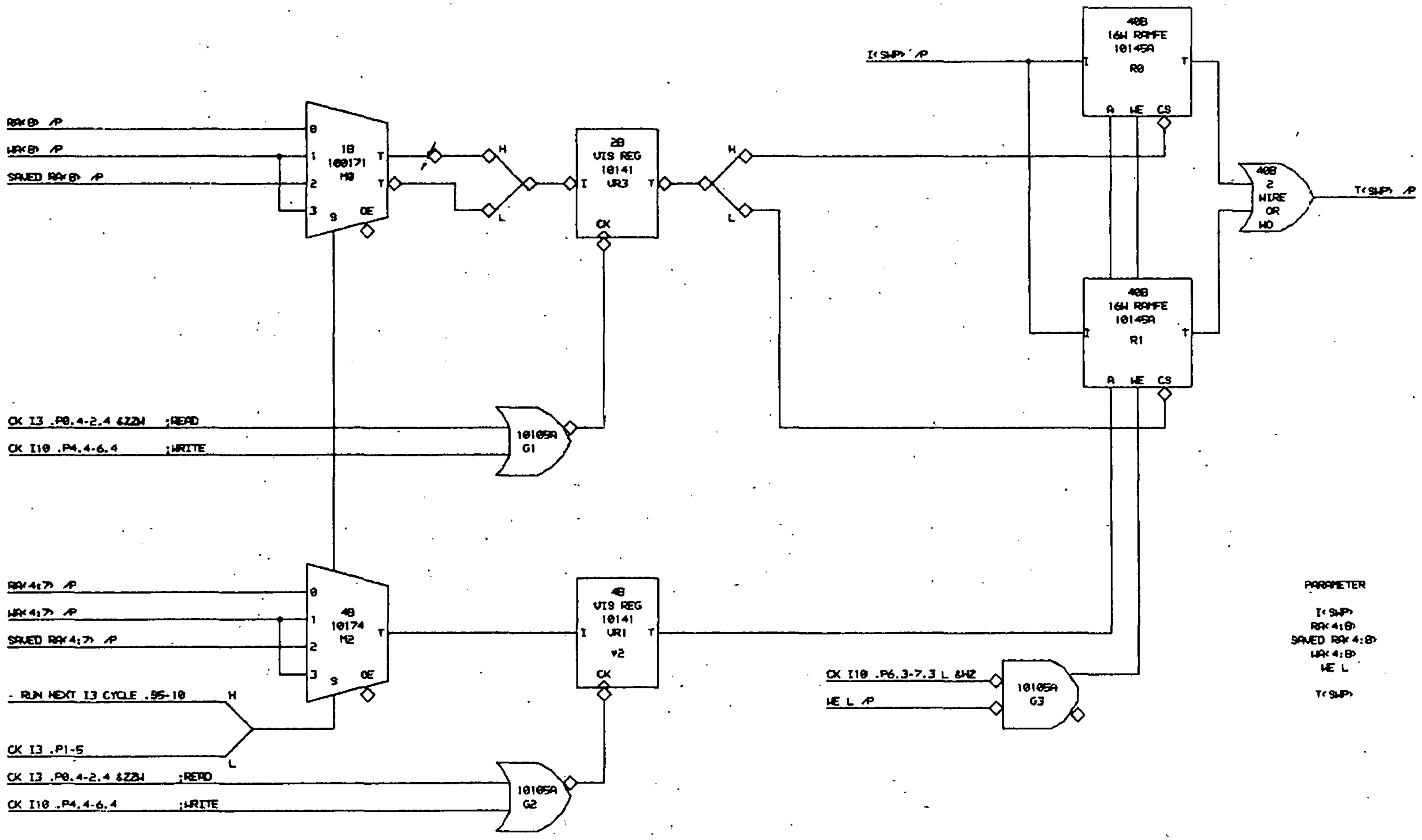
DEFINE

X STEP = SIZE

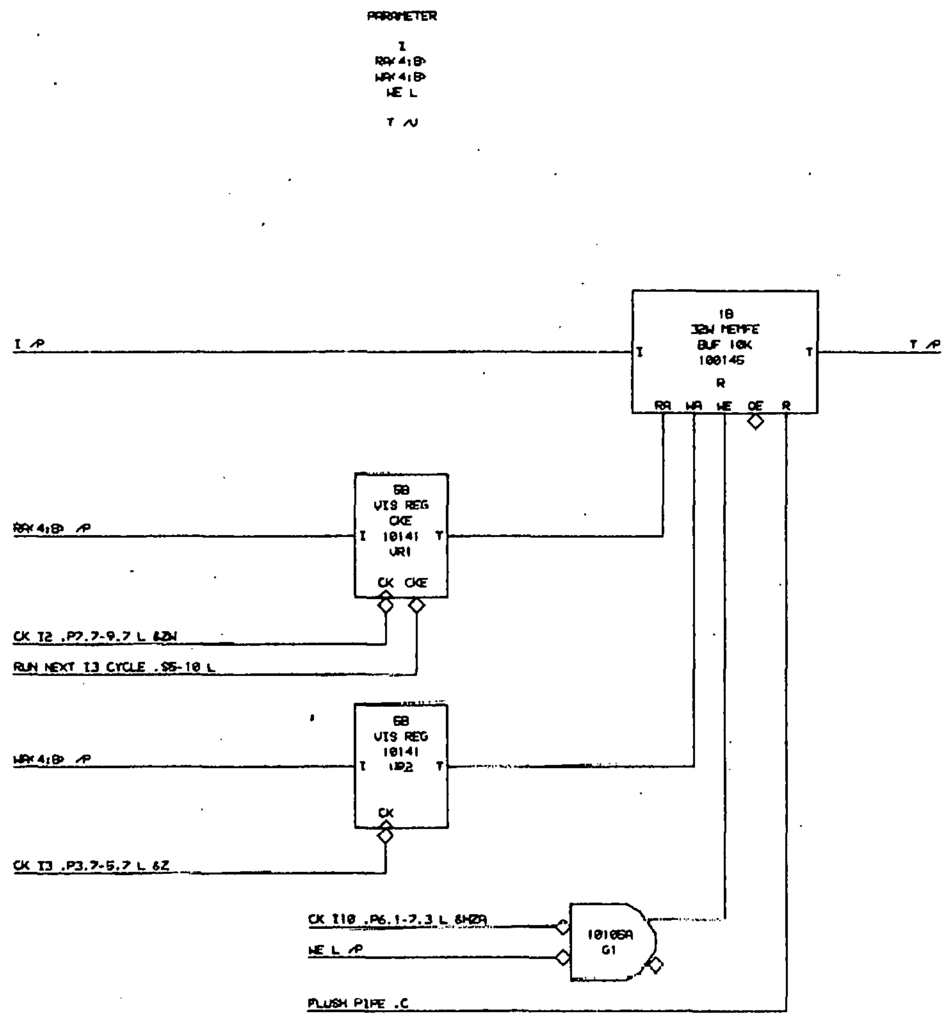
PREDICTED INDEX REGISTER FILE



INDEX REGISTER FILE RAM



INDEX REGISTER VALUE PREDICT RAM



INDEX REGISTER VALID VALUE PREDICT

2.7 DADRA

Drawings: DADRA1, DADRA2, DADRA3, DADRA4, DADRA5, DADRA6, DADRA7

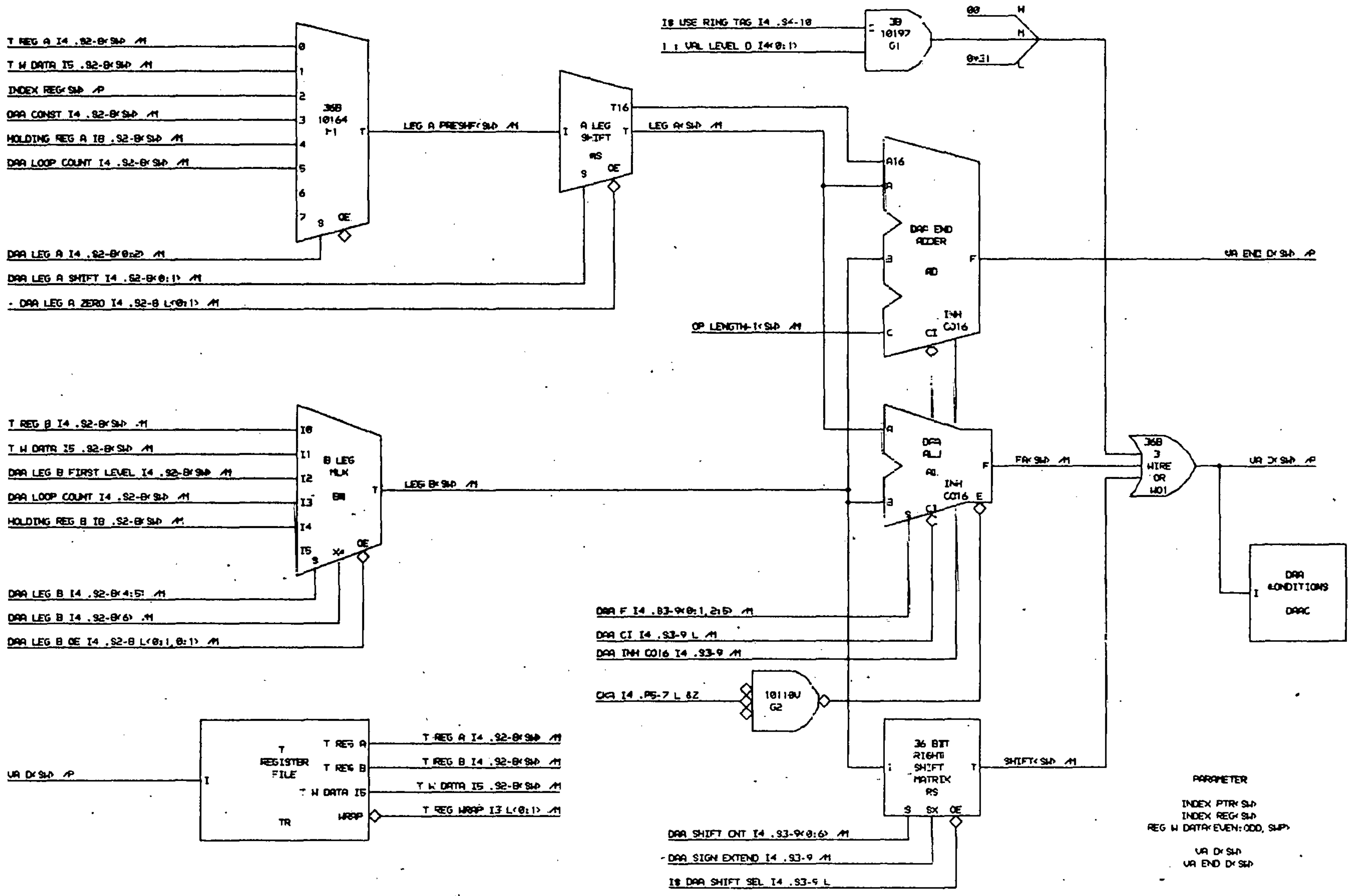
The data address arithmetic (DADRA) is the main computing engine of the IBOX. The computing elements are a 36 bit adder, ALU, and shifter. These elements operate on two operands, LEG A and LEG B which, in turn, are selected from a wide variety of signals. The two main outputs of the DADRA are the VA D and VA END D lines. VA D<SW> is a 36 bit quantity which may be a constant operand or the virtual address of an operand. VA END D<SW> is the virtual address of the end of the operand.

The ALU and shifter are microcode controlled as are the LEG A and LEG B multiplexors. The 36 bit DAA END ADDER adds LEG A, LEG B and OP LENGTH-1 to produce VAA END D. Both the adder and ALU have special connections to inhibit the carry out of bit 16 into bit 15 under microcode control. This is used for PDP-10 emulation. The LEG A value is generated by a shifter that can shift its input left by 0, 1, 2, or 3 places. This is used for those addressing modes that require an address to be shifted prior to being added in to the final address. The B LEG multiplexor selects either its input or its input shifted left by two as required for the addressing mode being evaluated. Both the A and B LEG multiplexors have the capability of zeroing out the top 16 and/or the low 20 bits. This is also used for PDP-10 emulation.

The T REGISTER FILE is a 256 word two port register file which is read and written every cycle. It is written, under microcode control, from VA D of the previous cycle. Two independent addresses are read from the file every cycle. One word is sent to each of the A and B LEG multiplexors. Special logic is provided to detect the case that one cycle is trying to read a T register which was scheduled to be written during the previous cycle. In this case, the write operation will not have happened by the time the data is required to be read. Instead, the previous cycle's write data is selected by the A or B LEG multiplexor instead of the stale T register output.

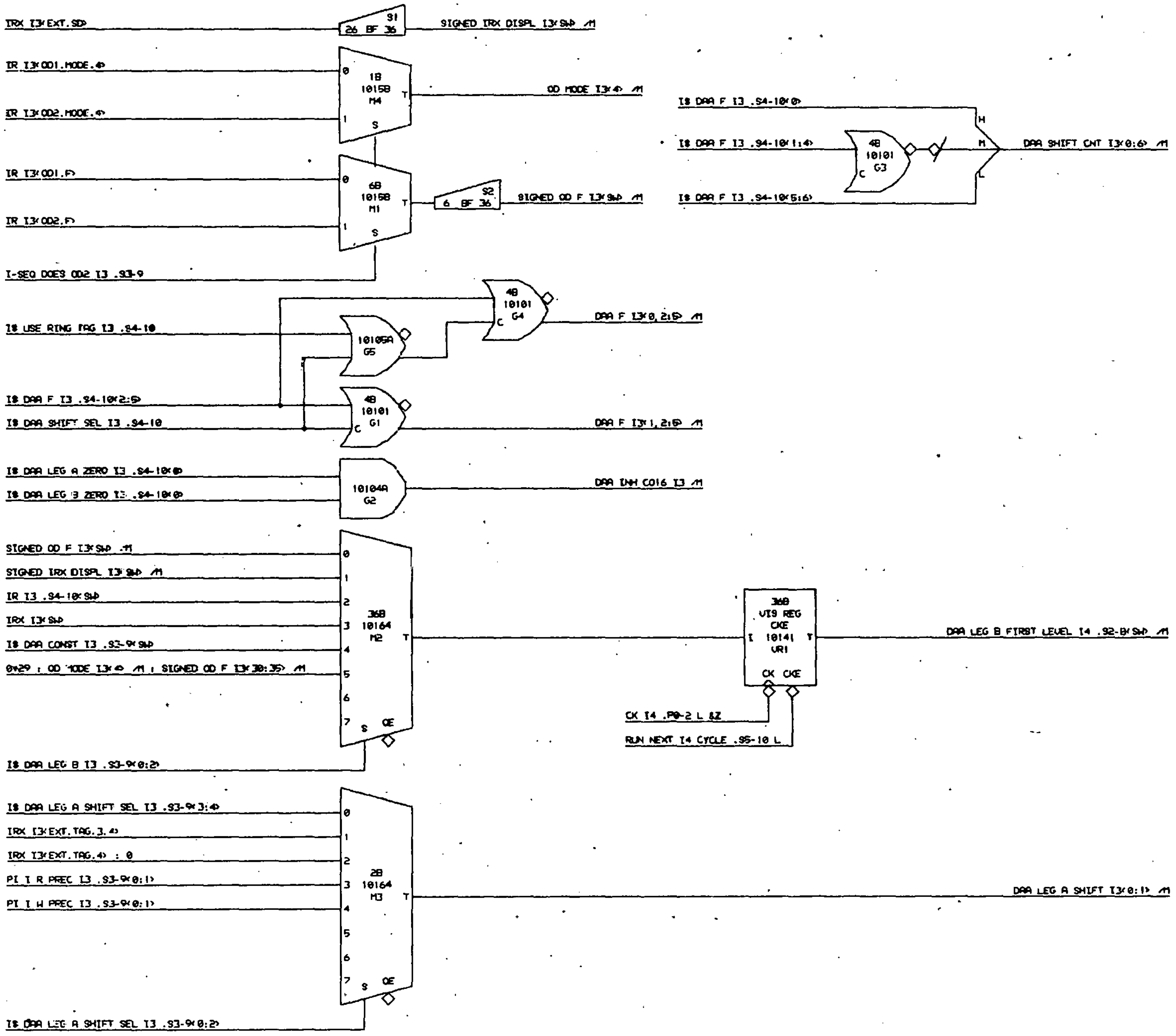
There are a pair of registers called HOLDING REG A and B which, under microcode control, can be loaded with the last value read out of the data cache. These registers can then be selected to LEG A and B, respectively. These are used in calculating addressing modes with indirect references or references to pseudo-registers.

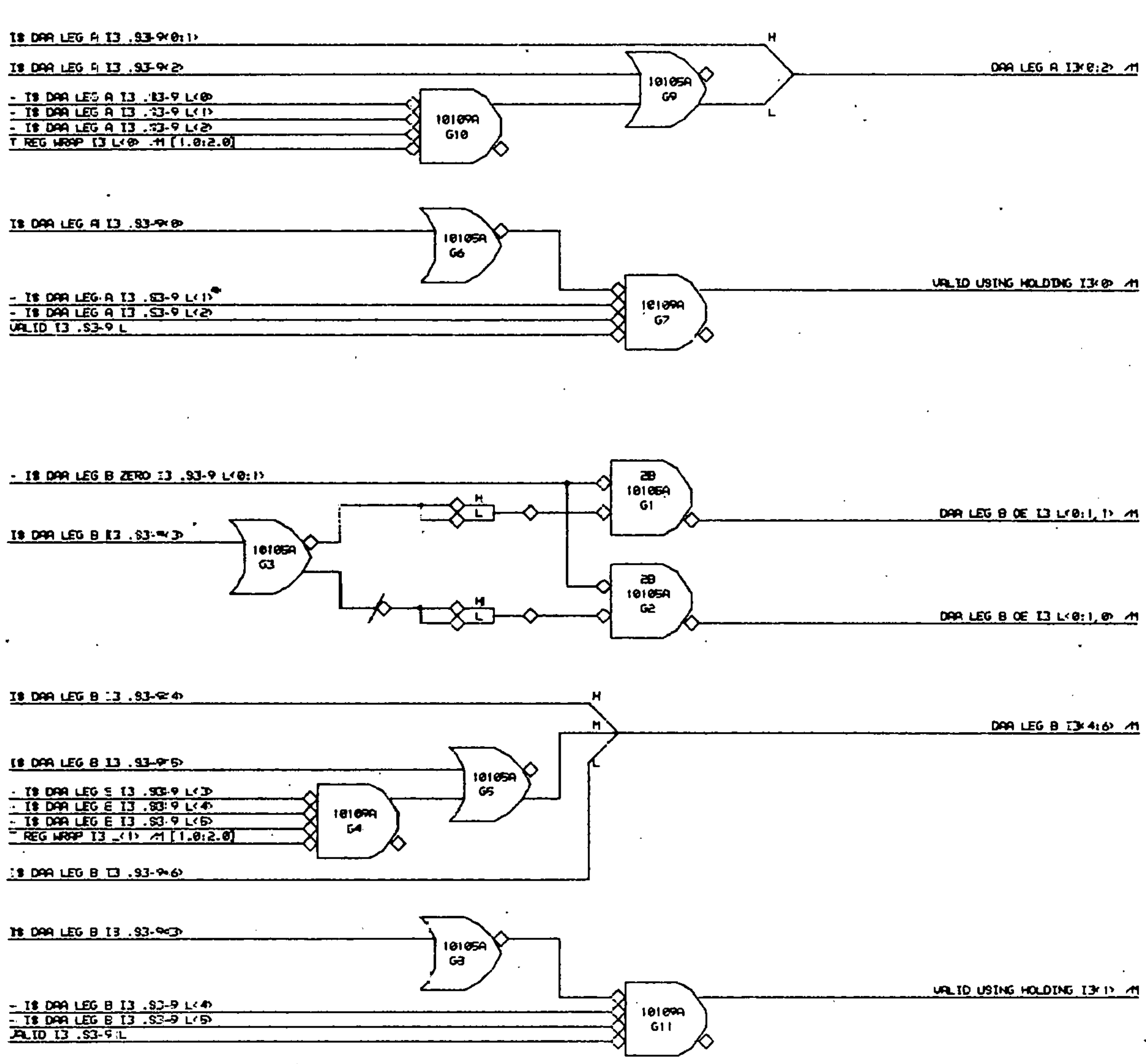
The data address arithmetic is performed during the I4 pipeline stage.



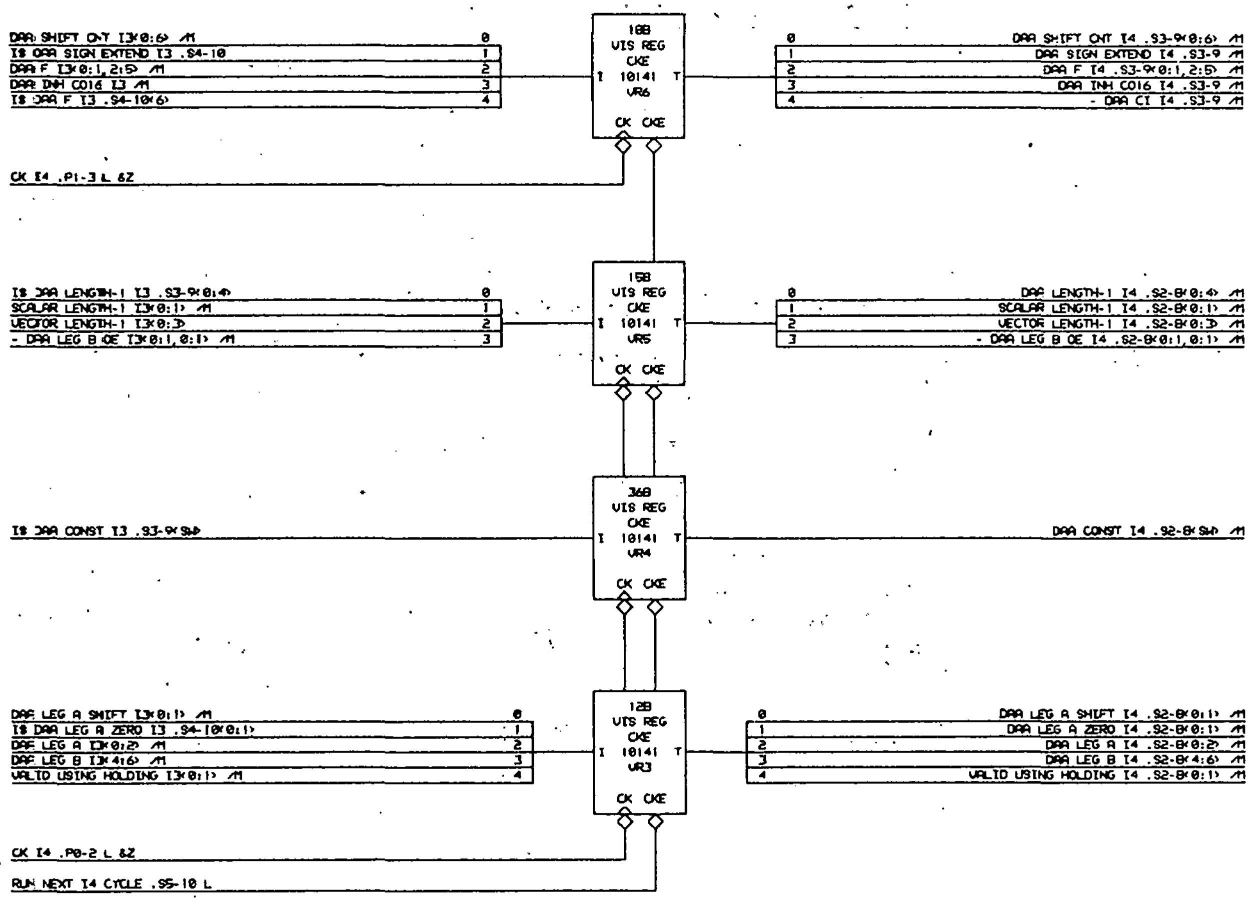
DATA ADDRESS ARITHMETIC

DATA ADDRESS ARITHMETIC

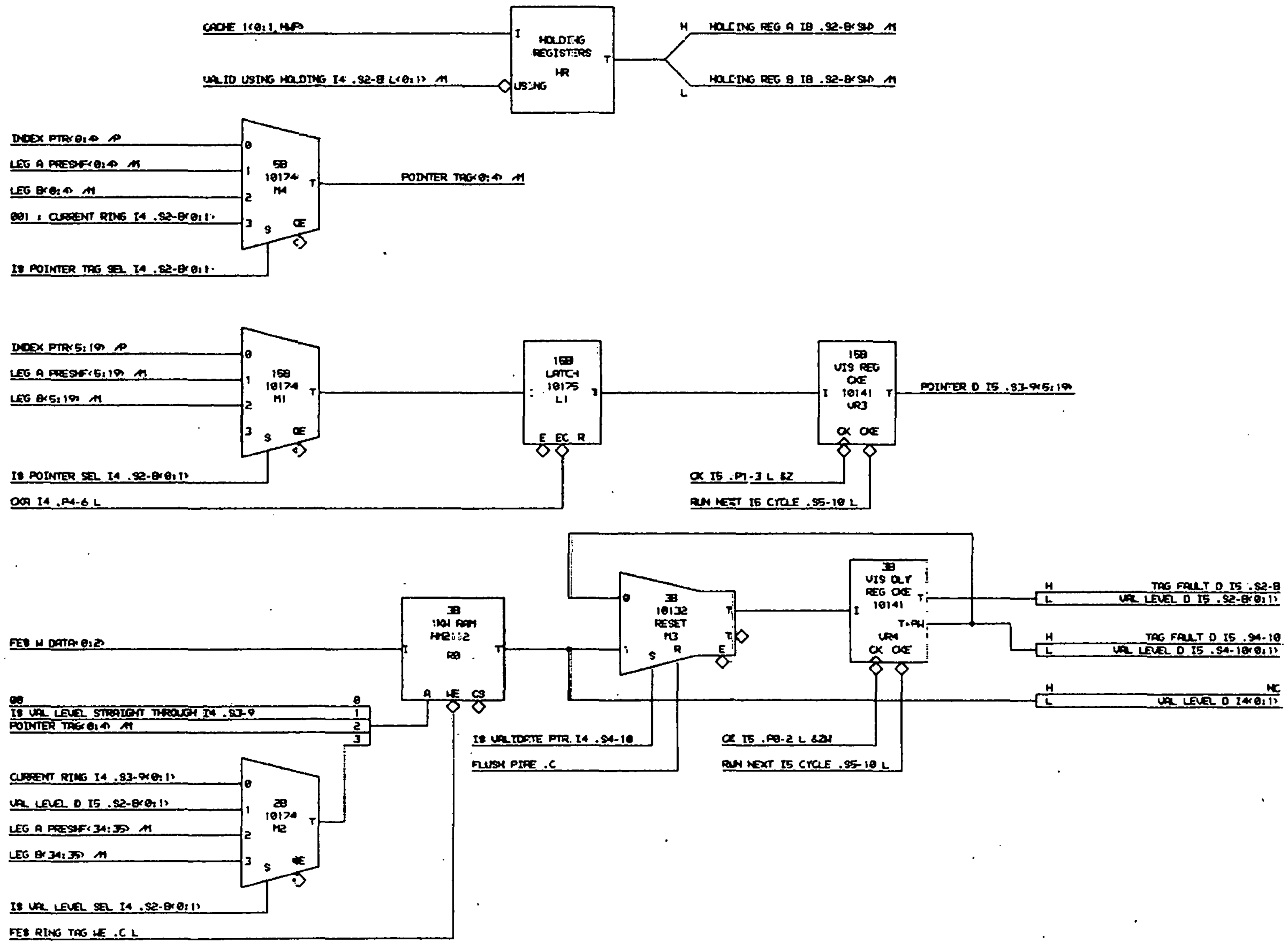



I$ DAA LEG A I3 .S3-9<0:1>
I$ DAA LEG A I3 .S3-9<2>
- I$ DAA LEG A I3 .S3-9 L<0>
- I$ DAA LEG A I3 .S3-9 L<1>
- I$ DAA LEG A I3 .S3-9 L<2>
T REG WRAP I3 L<0> .H [1.0:2.0]
10109A G10
10105A G9
H
L
DAA LEG A I3<0:2> .H
I$ DAA LEG A I3 .S3-9<0>
10105A G6
- I$ DAA LEG A I3 .S3-9 L<1>
- I$ DAA LEG A I3 .S3-9 L<2>
VALID I3 .S3-9 L
10109A G7
VALID USING HOLDING I3<0> .H
- I$ DAA LEG B ZERO I3 .S3-9 L<0:1>
I$ DAA LEG B I3 .S3-9<3>
10105A G3
H
L
2B 10105A G1
2B 10105A G2
DAA LEG B OE I3 L<0:1,1> .H
DAA LEG B OE I3 L<0:1,0> .H
I$ DAA LEG B I3 .S3-9<4>
H
M
L
DAA LEG B I3<4:6> .H
I$ DAA LEG B I3 .S3-9<5>
10105A G5
- I$ DAA LEG B I3 .S3-9 L<3>
- I$ DAA LEG B I3 .S3-9 L<4>
- I$ DAA LEG B I3 .S3-9 L<5>
T REG WRAP I3 L<1> .H [1.0:2.0]
10109A G4
I$ DAA LEG B I3 .S3-9<6>
I$ DAA LEG B I3 .S3-9<3>
10105A G8
- I$ DAA LEG B I3 .S3-9 L<4>
- I$ DAA LEG B I3 .S3-9 L<5>
VALID I3 .S3-9 L
10109A G11
VALID USING HOLDING I3<1> .H

DATA ADDRESS ARITHMETIC

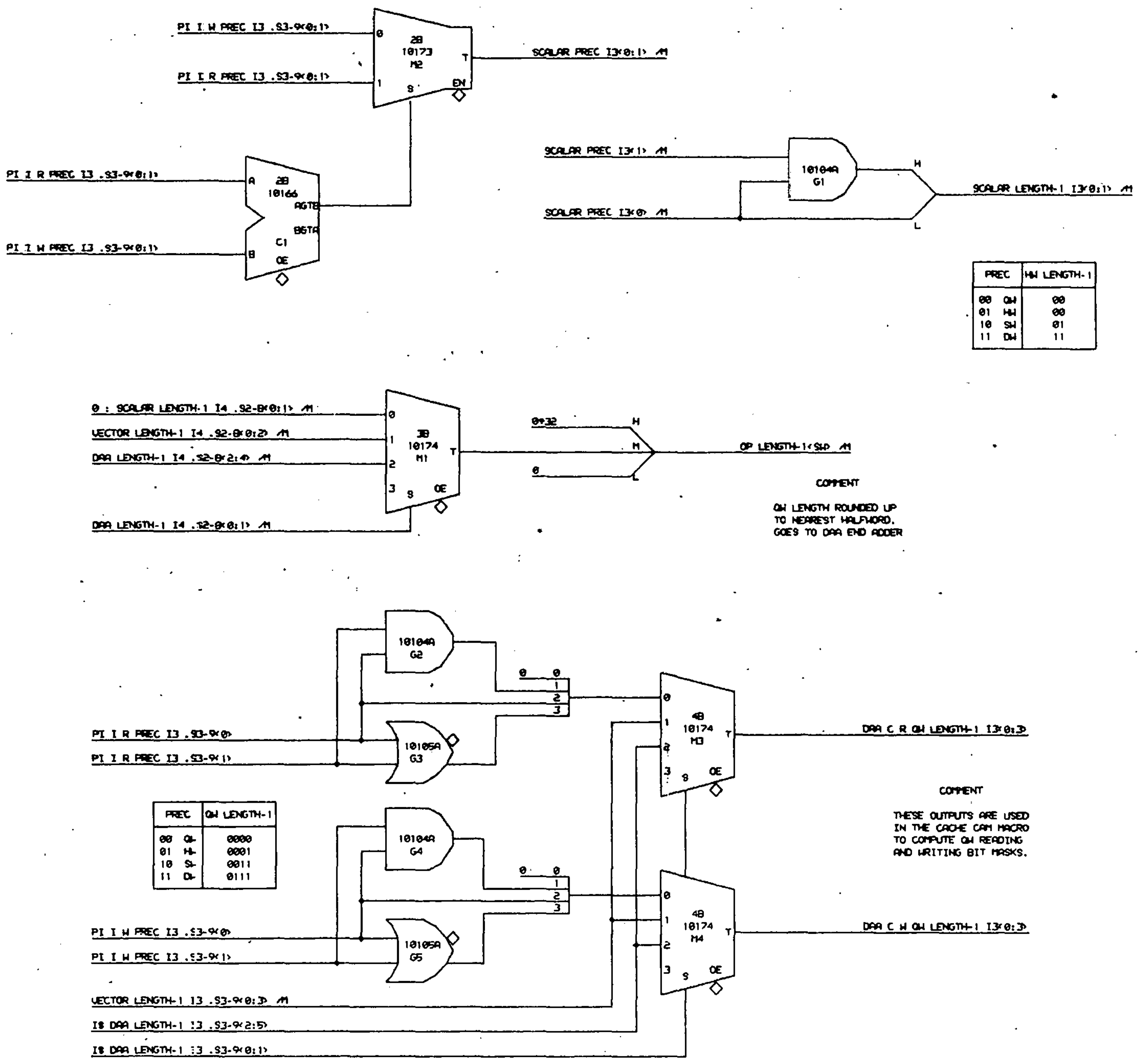

DAA SHIFT CNT I3<0:6> /M
I$ DAA SIGN EXTEND I3 .S4-10
DAA F I3<0:1,2:5> /M
DAA INH CO16 I3 /M
I$ DAA F I3 .S4-10<6>
0
1
2
3
4
18B
VIS REG
CKE
10141
VR6
I
T
CK CKE
DAA SHIFT CNT I4 .S3-9<0:6> /M
DAA SIGN EXTEND I4 .S3-9 /M
DAA F I4 .S3-9<0:1,2:5> /M
DAA INH CO16 I4 .S3-9 /M
- DAA CI I4 .S3-9 /M
CK I4 .P1-3 L &Z
I$ DAA LENGTH-1 I3 .S3-9<0:4>
SCALAR LENGTH-1 I3<0:1> /M
VECTOR LENGTH-1 I3<0:3>
- DAA LEG B OE I3<0:1,0:1> /M
15B
VIS REG
CKE
10141
VR5
DAA LENGTH-1 I4 .S2-8<0:4> /M
SCALAR LENGTH-1 I4 .S2-8<0:1> /M
VECTOR LENGTH-1 I4 .S2-8<0:3> /M
- DAA LEG B OE I4 .S2-8<0:1,0:1> /M
36B
VIS REG
CKE
10141
VR4
I$ DAA CONST I3 .S3-9<S4>
DAA CONST I4 .S2-8<S4> /M
DAA LEG A SHIFT I3<0:1> /M
I$ DAA LEG A ZERO I3 .S4-10<0:1>
DAA LEG A I3<0:2> /M
DAA LEG B I3<4:6> /M
VALID USING HOLDING I3<0:1> /M
12B
VIS REG
CKE
10141
VR3
DAA LEG A SHIFT I4 .S2-8<0:1> /M
DAA LEG A ZERO I4 .S2-8<0:1> /M
DAA LEG A I4 .S2-8<0:2> /M
DAA LEG B I4 .S2-8<4:6> /M
VALID USING HOLDING I4 .S2-8<0:1> /M
CK I4 .P0-2 L &Z
RUN NEXT I4 CYCLE .S5-10 L

DATA ADDRESS ARITHMETIC



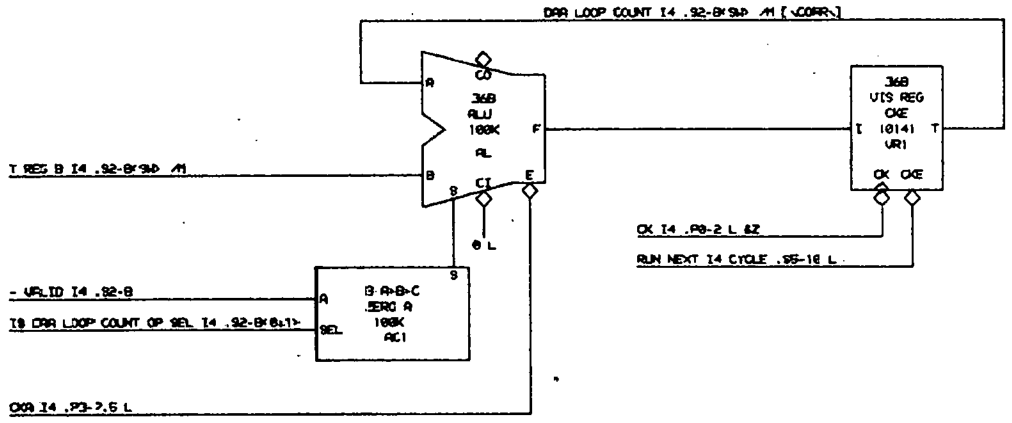
DATA ADDRESS ARITHMETIC

DATA ADDRESS ARITHMETIC



PREC	HW LENGTH-1
00 QW	00
01 HW	00
10 SW	01
11 DW	11

PREC	QW LENGTH-1
00 Q-	0000
01 H-	0001
10 S-	0011
11 D-	0111



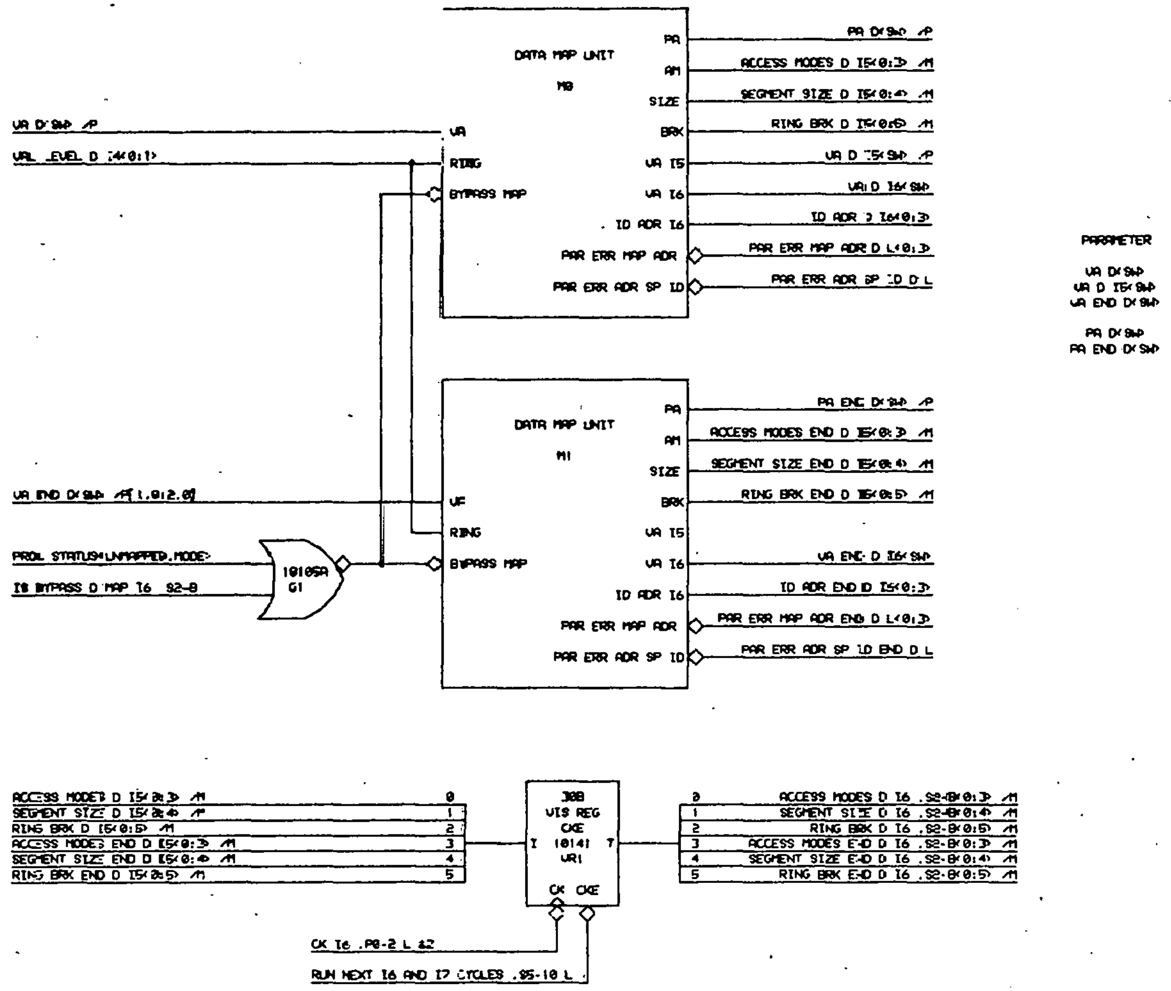
DATA ADDRESS ARITHMETIC

2.8 DMAP

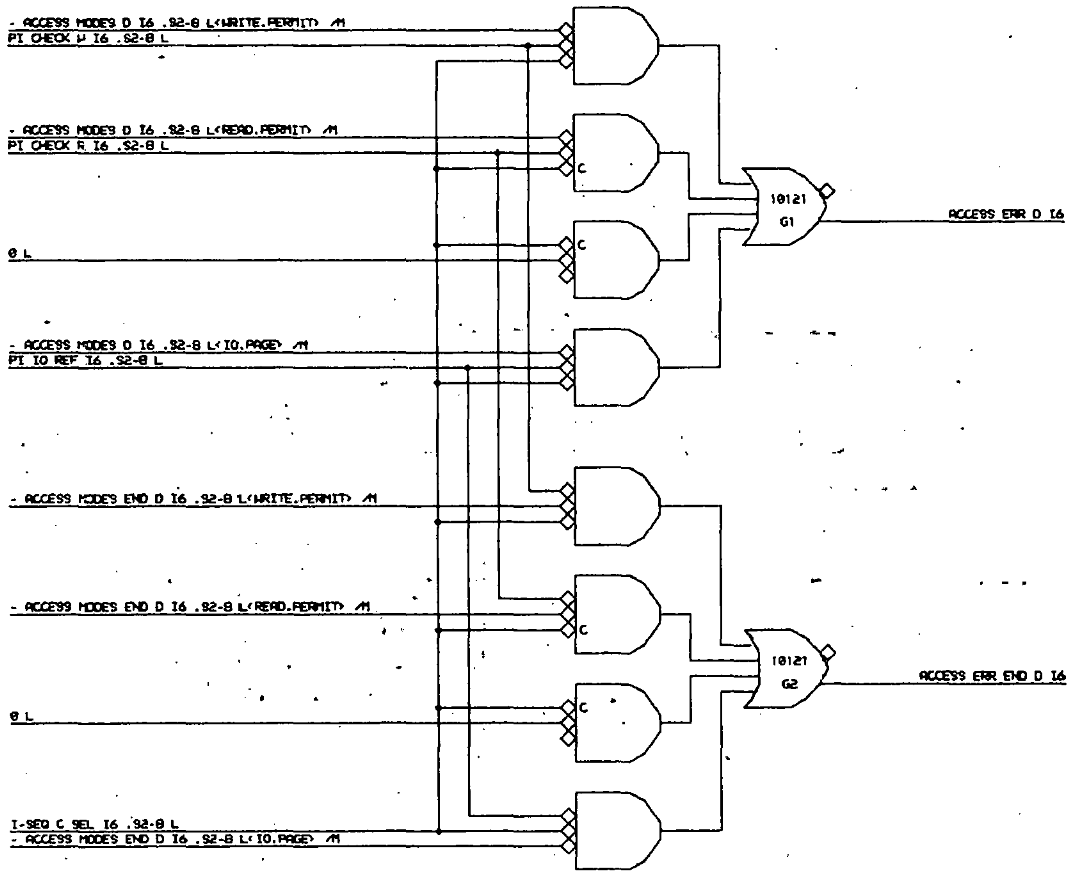
Drawings: DMAP1, DMAP2, DMAP3, DMAP4, DMU

The data map consists of two identical data map units. One unit maps VA D and the other maps VA END D. In this way, the beginning and ending addresses of an operand of any length are translated in parallel. Since an operand cannot be as long as a page, any page fault that can occur in an operand will be detected by this pair of map units. The outputs of the maps are PA D and PA END D respectively. The maps can be made transparent (all 36 input bits go unchanged to the output) under microcode control. The individual data map units are virtually identical to the instruction map unit described previously. The data maps perform their translations during the I5 pipeline stage.

Additional logic is also shown that provides three types of error checking: illegal access, segment bounds errors, and access protection violations.

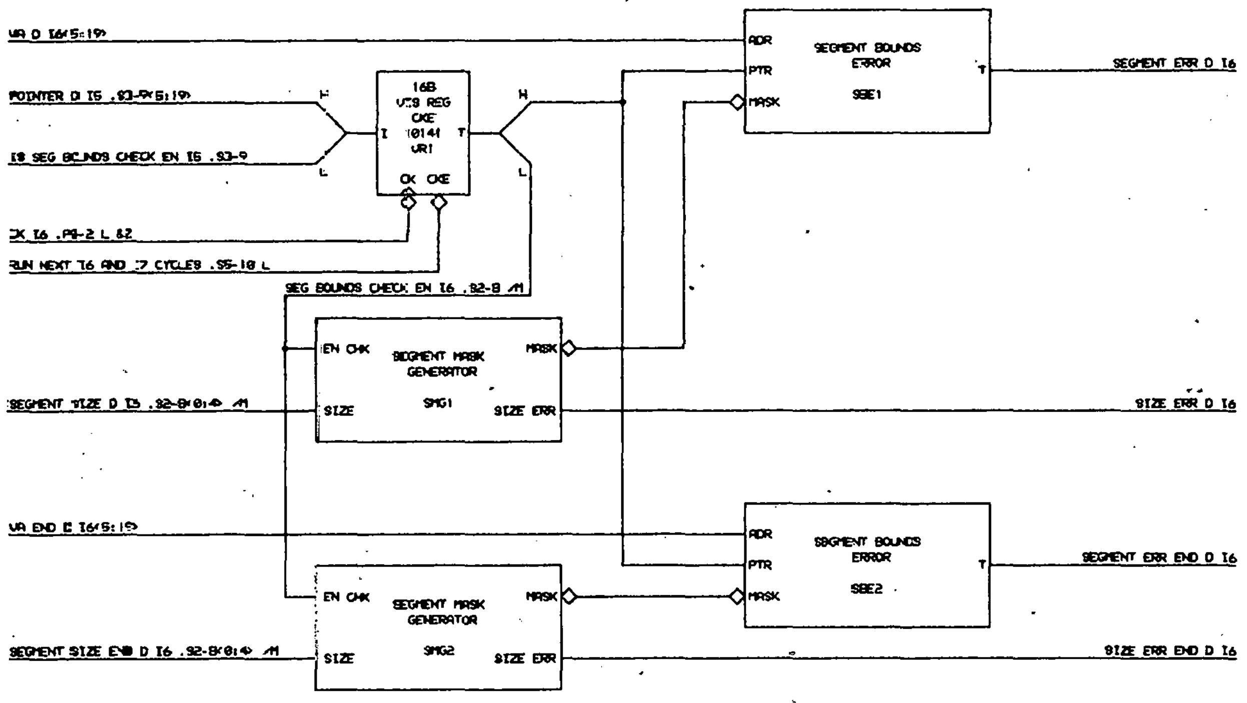

DATA MAP UNIT
M0
VA D<9:> /P
VA
VAL LEVEL D I4<0:1>
RING
BYPASS MAP
PA
AM
SIZE
BRK
VA I5
VA I6
ID ADR I6
PAR ERR MAP ADR
PAR ERR ADR SP ID
PA D<9:> /P
ACCESS MODES D I5<0:3> /M
SEGMENT SIZE D I5<0:4> /M
RING BRK D I5<0:5> /M
VA D I5<9:> /P
VA D I6<9:>
ID ADR D I6<0:3>
PAR ERR MAP ADR D L<0:3>
PAR ERR ADR SP ID D L
PARAMETER
VA D<9:>
VA D I5<9:>
VA END D<9:>
PA D<9:>
PA END D<9:>
DATA MAP UNIT
M1
VA END D<9:> /P[1.0:2.0]
VF
RING
BYPASS MAP
PROL STATUS<UNMAPPED.MODE>
I8 BYPASS D MAP I6 S2-8
10105A
G1
PA
AM
SIZE
BRK
VA I5
VA I6
ID ADR I6
PAR ERR MAP ADR
PAR ERR ADR SP ID
PA END D<9:> /P
ACCESS MODES END D I5<0:3> /M
SEGMENT SIZE END D I5<0:4> /M
RING BRK END D I5<0:5> /M
VA END D I6<9:>
ID ADR END D I5<0:3>
PAR ERR MAP ADR END D L<0:3>
PAR ERR ADR SP ID END D L
ACCESS MODES D I5<0:3> /M
SEGMENT SIZE D I5<0:4> /P
RING BRK D I5<0:5> /M
ACCESS MODES END D I5<0:3> /M
SEGMENT SIZE END D I5<0:4> /M
RING BRK END D I5<0:5> /M
0
1
2
3
4
5
30B
VIS REG
CKE
10141
VR1
I
T
CK CKE
CK I6 .P0-2 L &2
RUN NEXT I6 AND I7 CYCLES .95-10 L
0
1
2
3
4
5
ACCESS MODES D I6 .S2-8<0:3> /M
SEGMENT SIZE D I6 .S2-8<0:4> /M
RING BRK D I6 .S2-8<0:5> /M
ACCESS MODES END D I6 .S2-8<0:3> /M
SEGMENT SIZE END D I6 .S2-8<0:4> /M
RING BRK END D I6 .S2-8<0:5> /M

DATA MAP

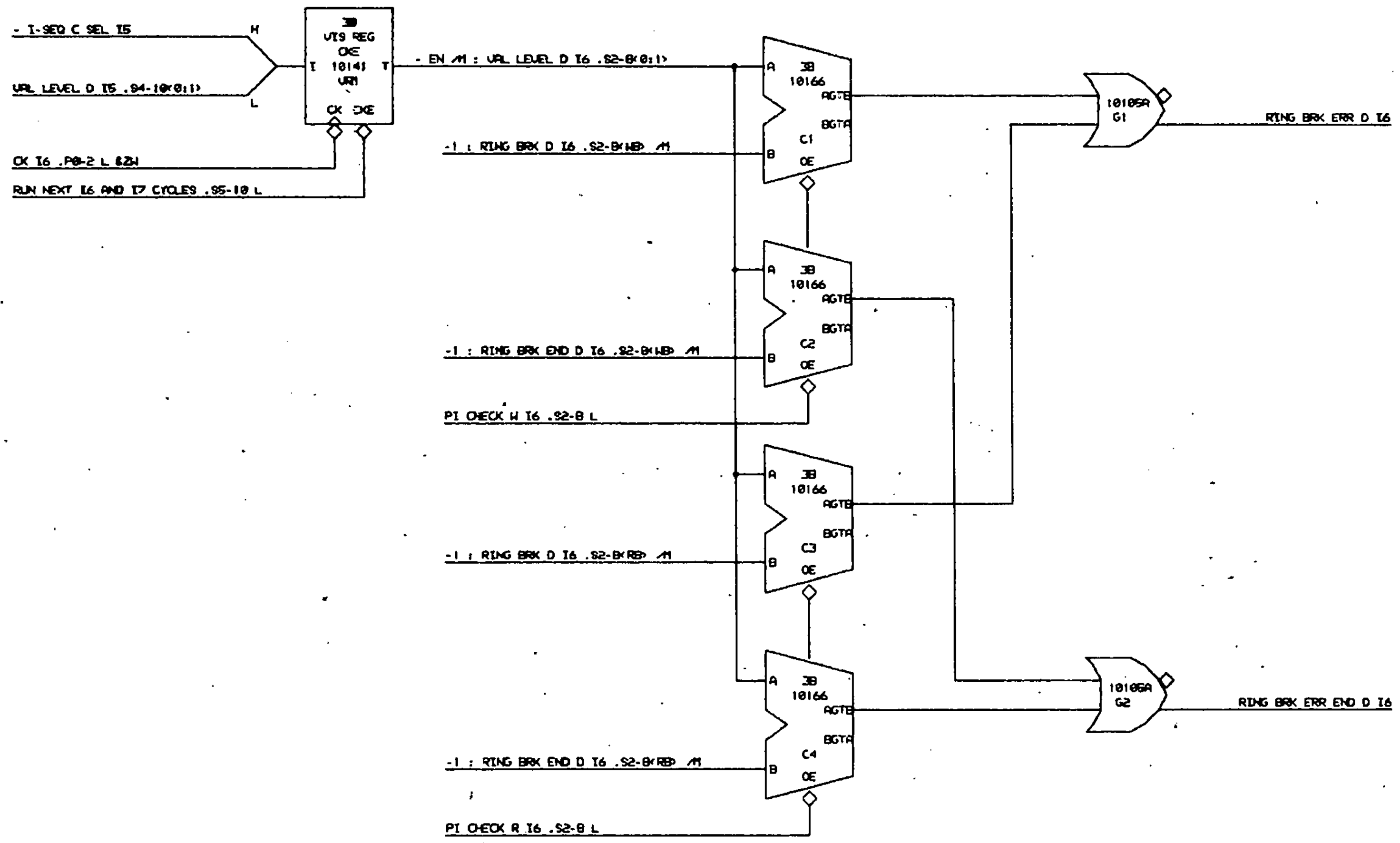


- ACCESS MODES D I6 .S2-8 L<WRITE.PERMIT> /M
PI CHECK W I6 .S2-8 L
- ACCESS MODES D I6 .S2-8 L<READ.PERMIT> /M
PI CHECK R I6 .S2-8 L
0 L
- ACCESS MODES D I6 .S2-8 L<IO.PAGE> /M
PI IO REF I6 .S2-8 L
- ACCESS MODES END D I6 .S2-8 L<WRITE.PERMIT> /M
- ACCESS MODES END D I6 .S2-8 L<READ.PERMIT> /M
0 L
I-SEQ C SEL I6 .S2-8 L
- ACCESS MODES END D I6 .S2-8 L<IO.PAGE> /M
C
C
C
C
10121
G1
10121
G2
ACCESS ERR D I6
ACCESS ERR END D I6

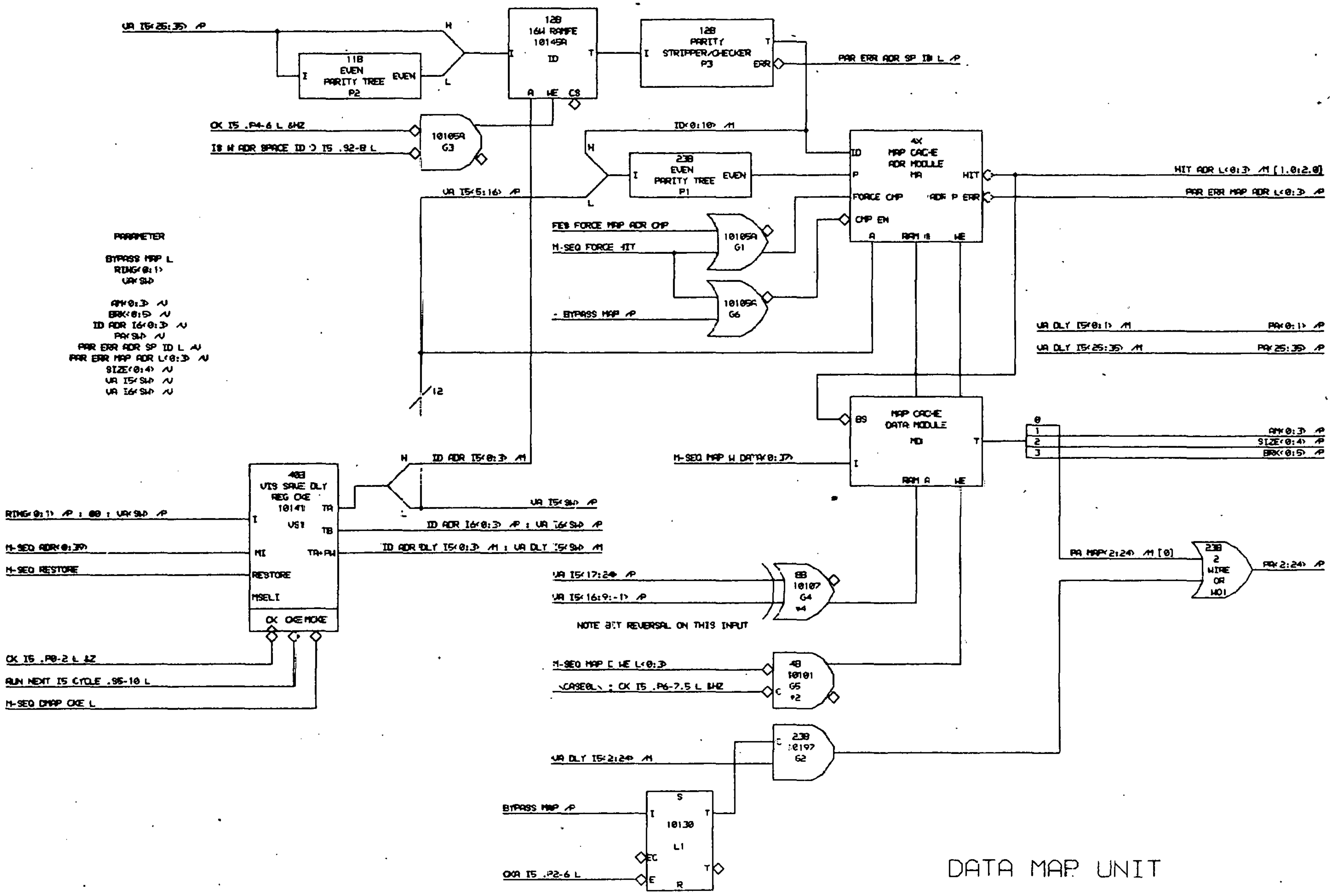
DATA MAP


VA D I6<5:19>
POINTER D I5 .S3-9<5:19>
IS SEG BOUNDS CHECK EN I5 .S3-9
CK I6 .P9-2 L &2
RUN NEXT I6 AND I7 CYCLES .S5-10 L
H
L
16B
VIS REG
CKE
10141
UR1
I
T
CK CKE
H
L
SEG BOUNDS CHECK EN I6 .S2-8 /H
EN CHK
SEGMENT MASK
GENERATOR
SMG1
MASK
SIZE
SIZE ERR
SEGMENT SIZE D I5 .S2-8<0:4> /H
VA END D I6<5:19>
EN CHK
SEGMENT MASK
GENERATOR
SMG2
MASK
SIZE
SIZE ERR
SEGMENT SIZE END D I6 .S2-8<0:4> /H
ADR
PTR
MASK
SEGMENT BOUNDS
ERROR
SBE1
T
SEGMENT ERR D I6
SIZE ERR D I6
ADR
PTR
MASK
SEGMENT BOUNDS
ERROR
SBE2
T
SEGMENT ERR END D I6
SIZE ERR END D I6

DATA MAP


- I-SEQ C SEL I5
H
VAL LEVEL D I5 .94-10<0:1>
L
3B
VIS REG
CKE
I 10141 T
VRM
CK CKE
CK I6 .P0-2 L &ZW
RUN NEXT I6 AND I7 CYCLES .95-10 L
- EN /M : VAL LEVEL D I6 .S2-B<0:1>
A 3B 10166 AGTB BGTA C1 OE B
-1 : RING BRK D I6 .S2-B<WB> /M
A 3B 10166 AGTB BGTA C2 OE B
-1 : RING BRK END D I6 .S2-B<WB> /M
PI CHECK W I6 .S2-B L
A 3B 10166 AGTB BGTA C3 OE B
-1 : RING BRK D I6 .S2-B<RB> /M
A 3B 10166 AGTB BGTA C4 OE B
-1 : RING BRK END D I6 .S2-B<RB> /M
PI CHECK R I6 .S2-B L
10105A G1
RING BRK ERR D I6
10105A G2
RING BRK ERR END D I6

DATA MAP

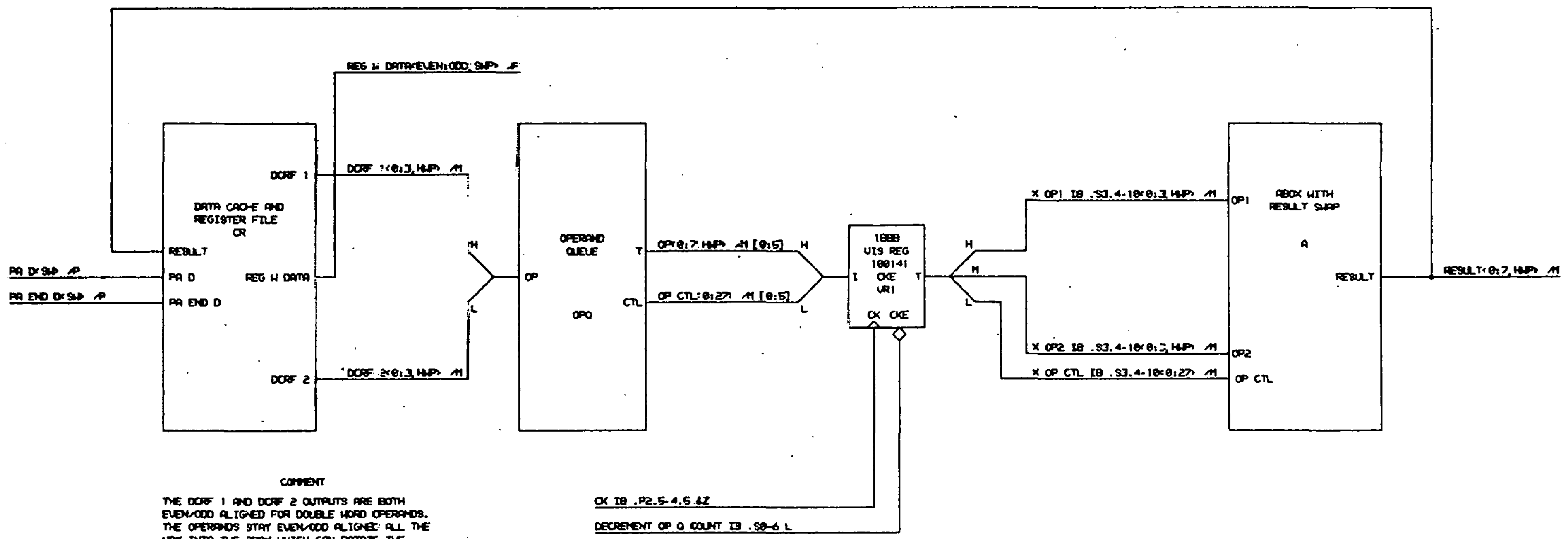


DATA MAP UNIT

2.9 OPABX

Drawings: OPABX

The operand data path and ABOX (OPABX) drawing contains the high-level description of the main operand data paths of the IBOX. Operands are read out of the data cache and register file on two legs: DCRF 1 and DCRF 2. Each operand contains up to a double word. The operands are addressed by PA D and PA END D. The DCRF 1 and 2 operands are delivered to the operand queue which acts as a buffer in the pipeline between the data cache and the ABOX. Operands are read out of the operand queue and loaded into a 4-word operand register along with a set of control bits. The output of this register is delivered to the ABOX. Quadword results from the ABOX are directed back to the data cache and register file for storing. Since the register file is capable of reading and writing at most a double word, vector operations are not permitted to use the registers. Only vector operations produce quadword outputs.



COMMENT

THE DCRF 1 AND DCRF 2 OUTPUTS ARE BOTH EVEN/ODD ALIGNED FOR DOUBLE WORD OPERANDS. THE OPERANDS STAY EVEN/ODD ALIGNED ALL THE WAY INTO THE ABOX WHICH CAN ROTATE THE OPERANDS ACCORDING TO THE LOW ORDER ADDRESS BITS IN ORDER TO HIGH/LOW ALIGN. ALL DATA PATHS FOR VECTOR OPERANDS ARE HIGH/LOW ALIGNED SINCE THERE IS NO WRAPAROUND.

PARAMETER

PA D<SWP>
PA END D<SWP>

REG W DATA<EVEN:ODD, SWP> /U

OPERAND DATA PATH AND ABOX

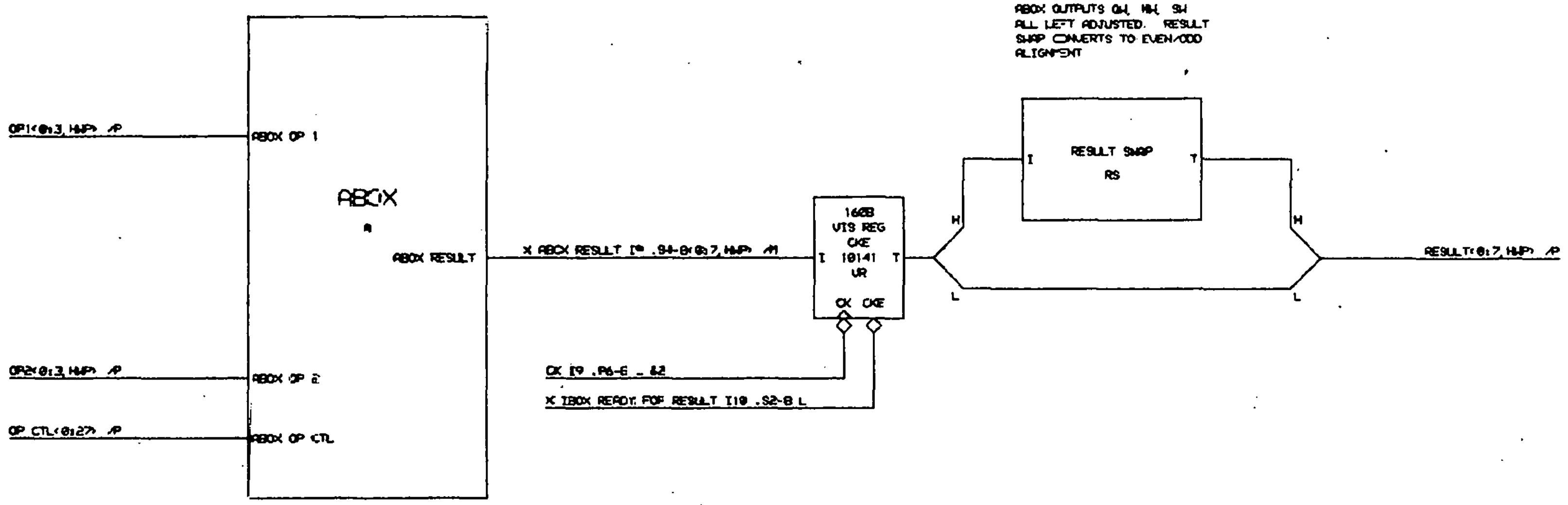
2.10 ABWRS

Drawing: ABWRS

ABOX with result swap (ABWRS) is a simple unit called from OPABX to take operands and put out results. It contains within it the entire ABOX. It clocks ABOX results into the result register which is enabled by the ABOX/IBOX result handshaking signal. The high double word of this register is passed through the result swap unit and then merged with the low double word. For vectors, the result swap unit is transparent, so that the final result is the same as the ABOX result. For scalar operations the ABOX result is high/low aligned within the upper double word. However, the result from the ABWRS macro is supposed to be even/odd aligned within that double word. It is the function of the result swap unit to make this alignment change.

COMMENT

ABOX OUTPUTS QW, MW, SW
ALL LEFT ADJUSTED. RESULT
SWAP CONVERTS TO EVEN/ODD
ALIGNMENT



PARAMETER

OP1<0:3, HWP>
OP2<0:3, HWP>
OP CTL<0:27>

RESULT<0:7, HWP>

COMMENT

OP 1, OP 2, AND RESULT
ARE ALL EVEN/ODD ALIGNED FOR DOUBLEWORD
OPERANDS.

ABOX RESULT IS ALWAYS HIGH/LOW ALIGNED.

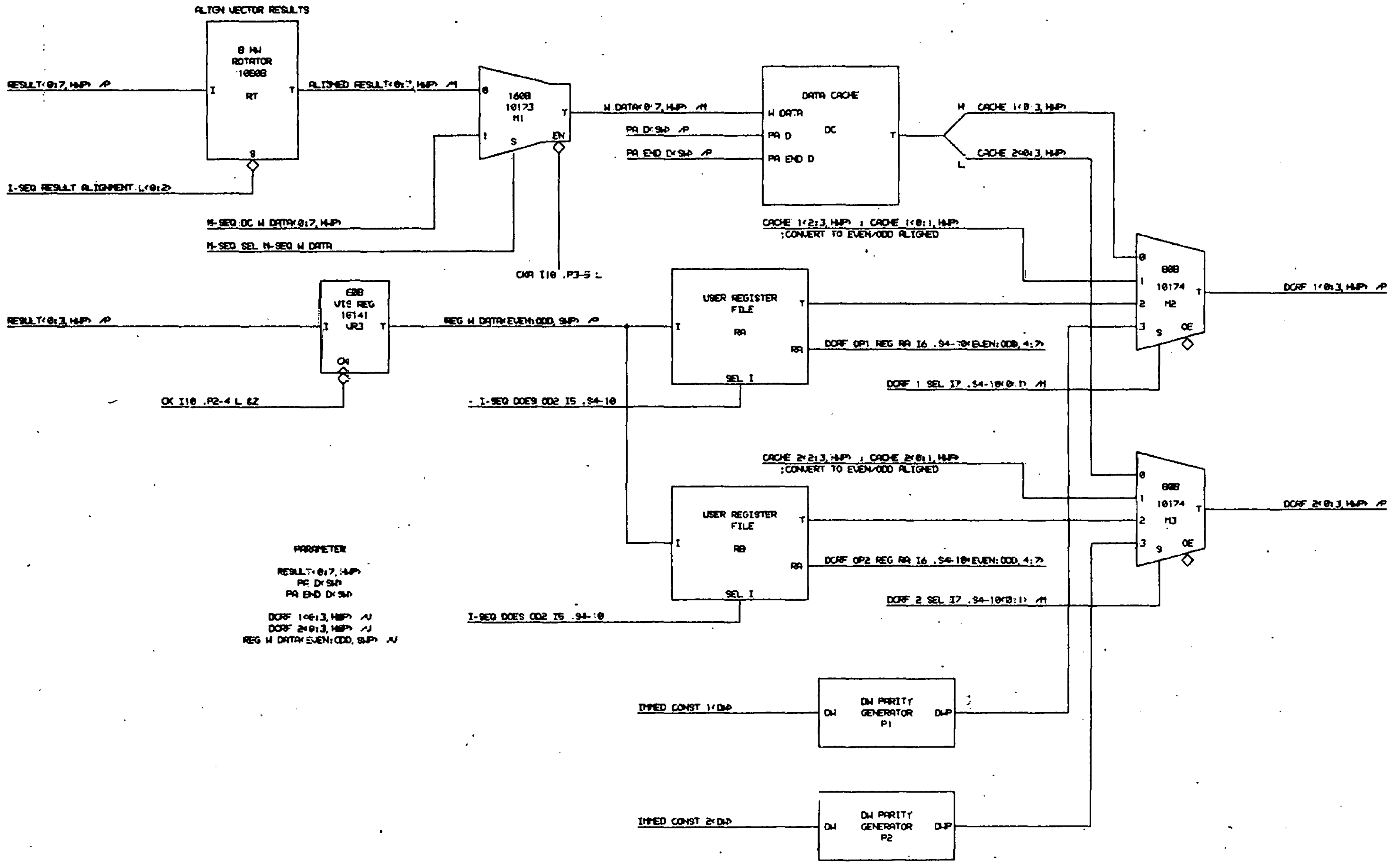
ABOX WITH RESULT SWAP

2.11 DCRF

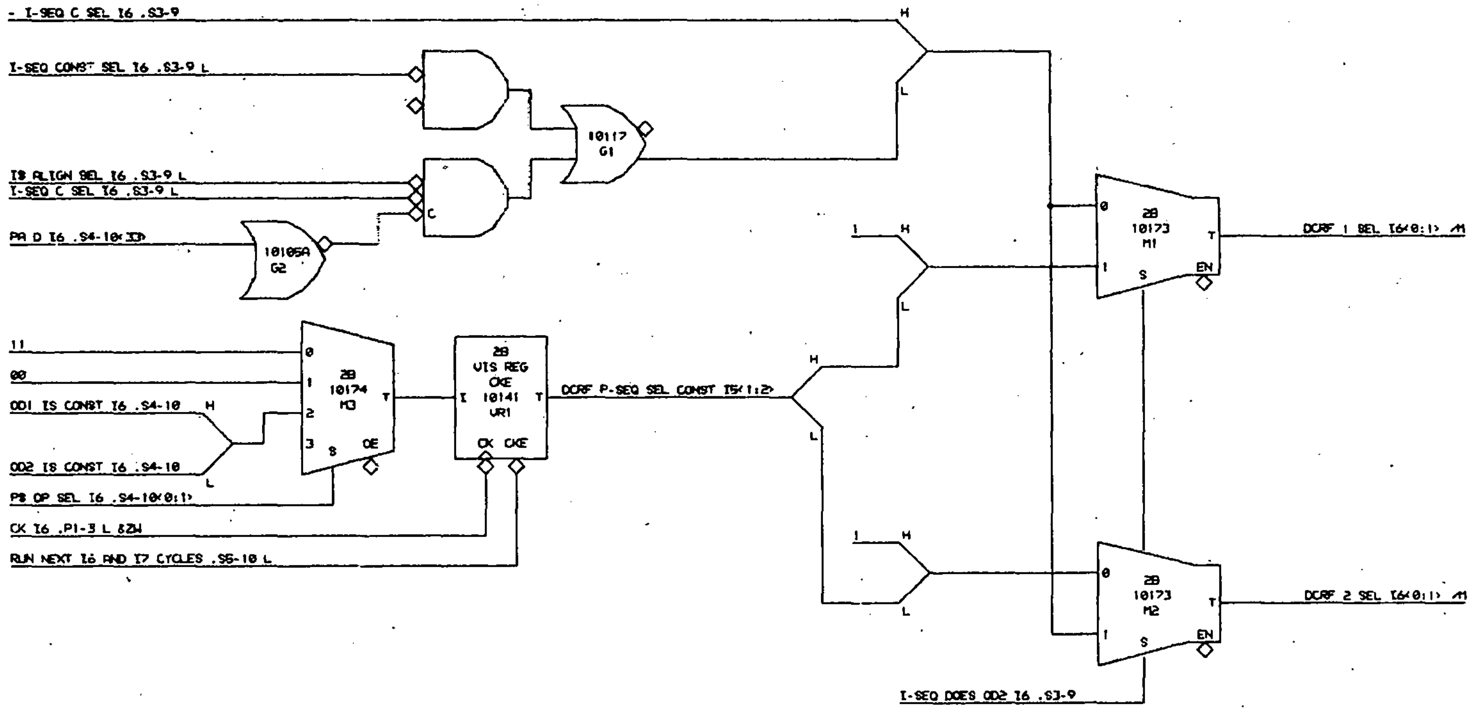
Drawing: DCRF1, DCRF2

The data cache and register file (DCRF) drawings show the data cache and the two copies of the user register file used for operand fetching. The data cache may be written from the even/odd result lines after they have been rotated to the correct position within the four word input or from the M-Sequencer write data lines. These lines are four words wide and provide a high bandwidth path over which the M-Sequencer can load the data cache from main memory or from a backing cache. The user register files are written from the high two words of the even/odd result lines.

The DCRF 1 and 2 outputs of this macro are the two operands for this microinstruction. They are independently selected from the cache, the user register files, and the immediate constant generator. For scalar operands, these outputs must be even/odd aligned. This is always true for the user register files and for the immediate constant, but is not necessarily the case for the cache. Therefore, another input is provided on each multiplexor to select the appropriate cache double word with the two words swapped.



DATA CACHE AND REGISTER FILE



DATA CACHE AND REGISTER FILE

2.12 DC

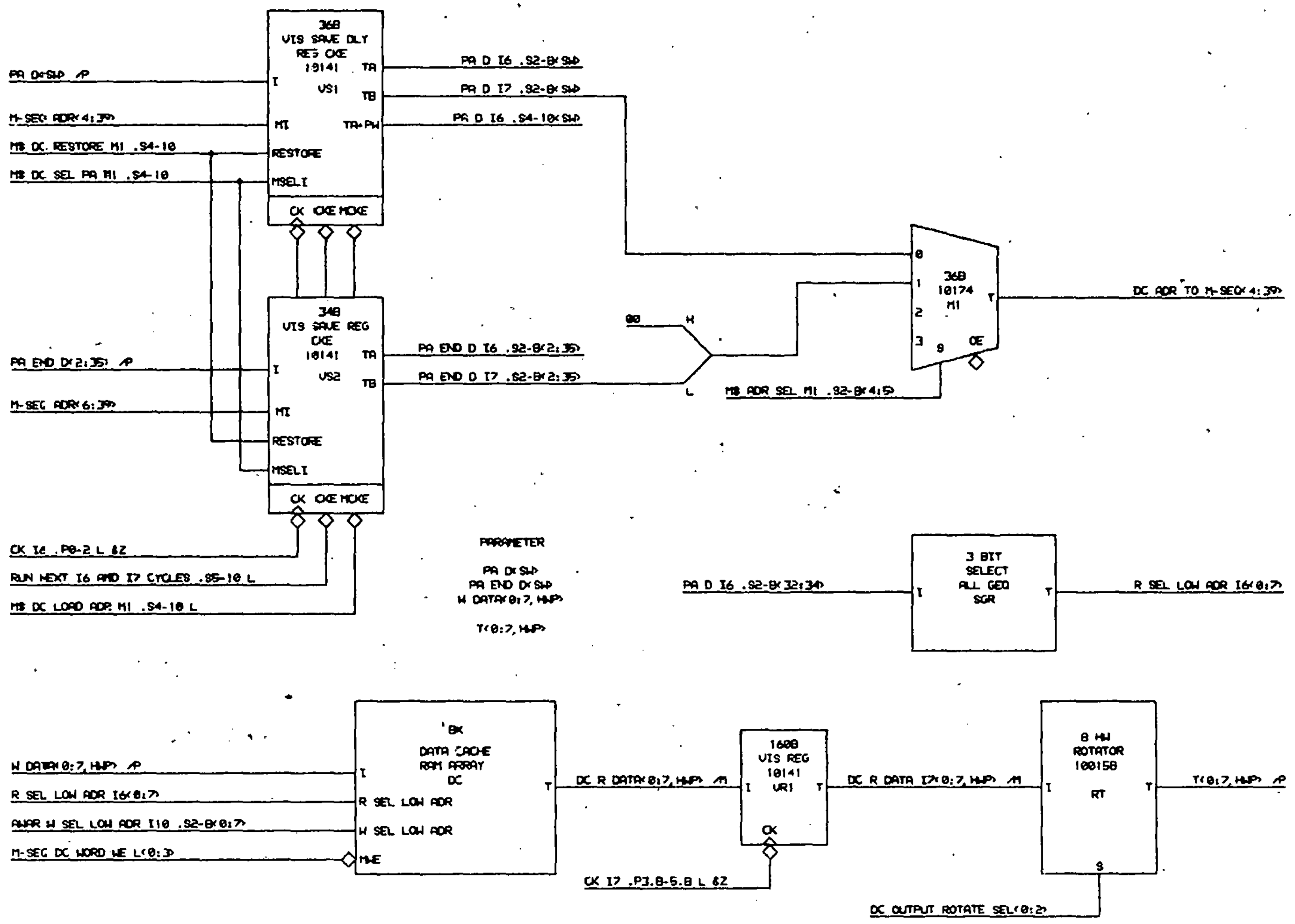
Drawings: DC1, DC2, DC3, DCRAM, DCDM, DCAM, 3BSAG

The data cache is a 16K word high speed memory that can read and write four words every cycle. The data cache is organized by half words. The eight half word output can begin on any halfword boundary. For writing, the input data can begin on any halfword boundary and, in addition, there are individual quarter word write enable lines. The data cache consists of an address part and a data part. The address part implements a four way set-associative cache organization with 16 word lines. Since cache addresses are not restricted to begin on four-word boundaries, it is possible to read a four word block from the cache that overlaps two cache lines. In order to do this read operation in one cycle, it is necessary to read from the address part of the cache the information for two consecutive lines. Therefore, there are two copies of the address part of the data cache. An eight halfword rotator is provided on the output of the data part in order to align operands. This does not effect the fact that the operands will be high/low aligned on output.

A data cache read operation occurs during two consecutive pipeline stages, I6 and I7. The address part is accessed during I6. It is given PA D and PA END D as addresses and produces a set of hit lines indicating which elements, if any, match the addresses. If either address part fails to hit on its respective address, then there is a cache miss. When this occurs, the M-Sequencer will load the appropriate cache line from memory and restart the cycle. This guarantees that when the cycle proceeds with no cache miss, BOTH lines that may be needed for the read operation are in the cache.

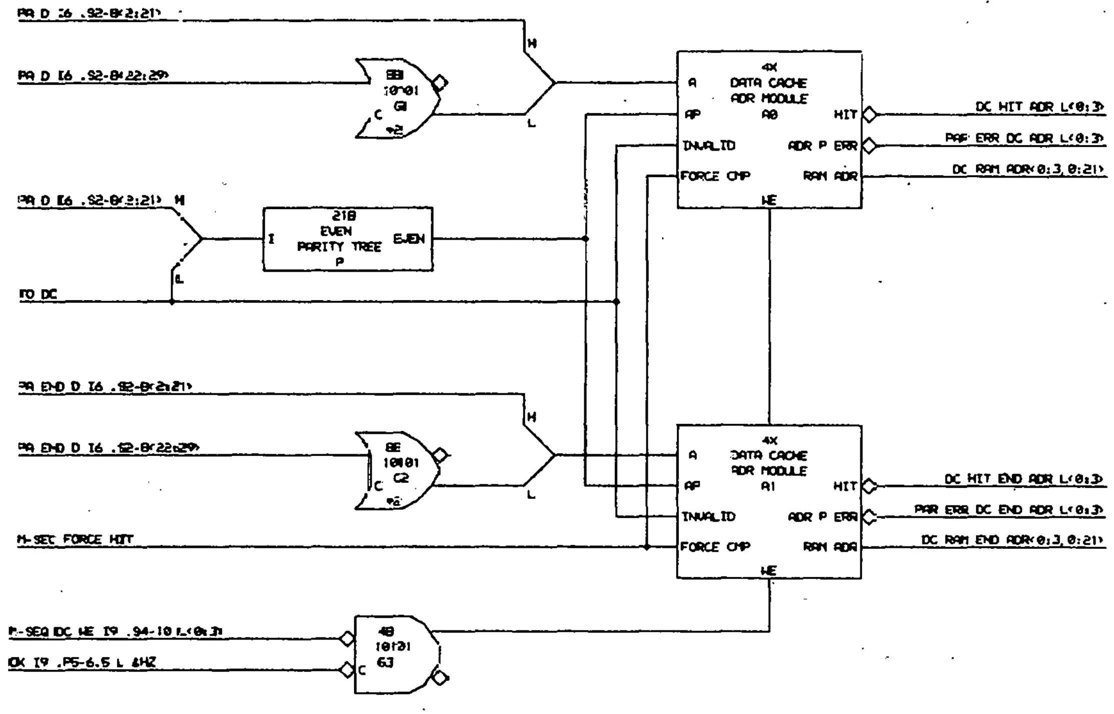
The data part read operation is performed during the I7 pipeline stage. The data part is constructed from 8 identical halfword modules each of which is a four way set-associative memory with 1024 half word elements. The elements selection is based on the two sets of four hit lines from the address part. The 3 BIT SELECT ALL GEQ macro looks at the low three half word address bits and produces an eight bit vector which is zeros up to some point and then all ones thereafter. The point at which the first one occurs corresponds to the input address. This eight bit vector, then, indicates (where there are ones) when to choose the hit vector from the address part that was addressed by PA D. It is also used to select the correct middle address bits for addressing the half word module.

For writing, the addresses and hit vectors are read from the write queue. The write occurs during the second half of the I10 pipeline stage. When a write operation is scheduled to happen, it is first attempted as a read operation during I6 and I7 (although the access checking checks for writing). This will detect page faults early in the execution of an instruction. It will also cause cache misses to be fixed before the write operation must actually happen. During the write operation at I10 it is guaranteed that the location will still be in the cache since the M-Sequencer will never kick out of the cache any location that is also present in the write queue.

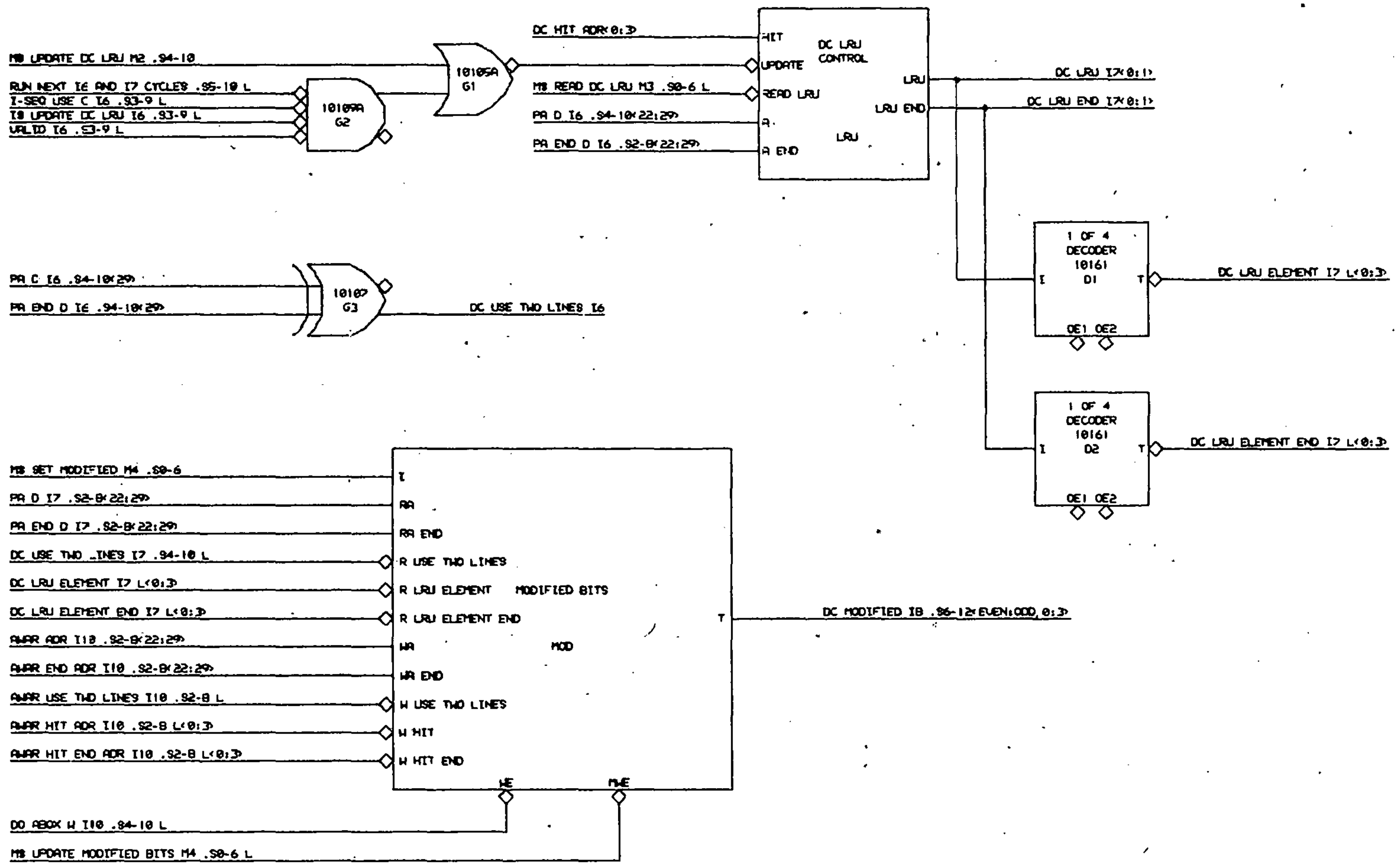


36B VIS SAVE DLY REG CKE 10141 VS1
PA D<SW> /P
M-SEQ ADR<4:39>
M$ DC RESTORE M1 .S4-10
M$ DC SEL PA M1 .S4-10
PA D I6 .S2-B<SW>
PA D I7 .S2-B<SW>
PA D I6 .S4-10<SW>
34B VIS SAVE REG CKE 10141 VS2
PA END D<2:35> /P
M-SEQ ADR<6:39>
PA END D I6 .S2-B<2:35>
PA END D I7 .S2-B<2:35>
CK I6 .P0-2 L &Z
RUN NEXT I6 AND I7 CYCLES .S5-10 L
M$ DC LOAD ADR M1 .S4-10 L
00
M$ ADR SEL M1 .S2-B<4:5>
36B 10174 M1
DC ADR TO M-SEQ<4:39>
PARAMETER
PA D<SW>
PA END D<SW>
W DATA<0:7, HWP>
T<0:7, HWP>
3 BIT SELECT ALL GEO SGR
PA D I6 .S2-B<32:34>
R SEL LOW ADR I6<0:7>
8K DATA CACHE RAM ARRAY DC
W DATA<0:7, HWP> /P
R SEL LOW ADR I6<0:7>
AWAR W SEL LOW ADR I10 .S2-B<0:7>
M-SEQ DC WORD WE L<0:3>
DC R DATA<0:7, HWP> /M
160B VIS REG 10141 VR1
CK I7 .P3.8-5.8 L &Z
DC R DATA I7<0:7, HWP> /M
8 HW ROTATOR 100158 RT
T<0:7, HWP> /P
DC OUTPUT ROTATE SEL<0:2>

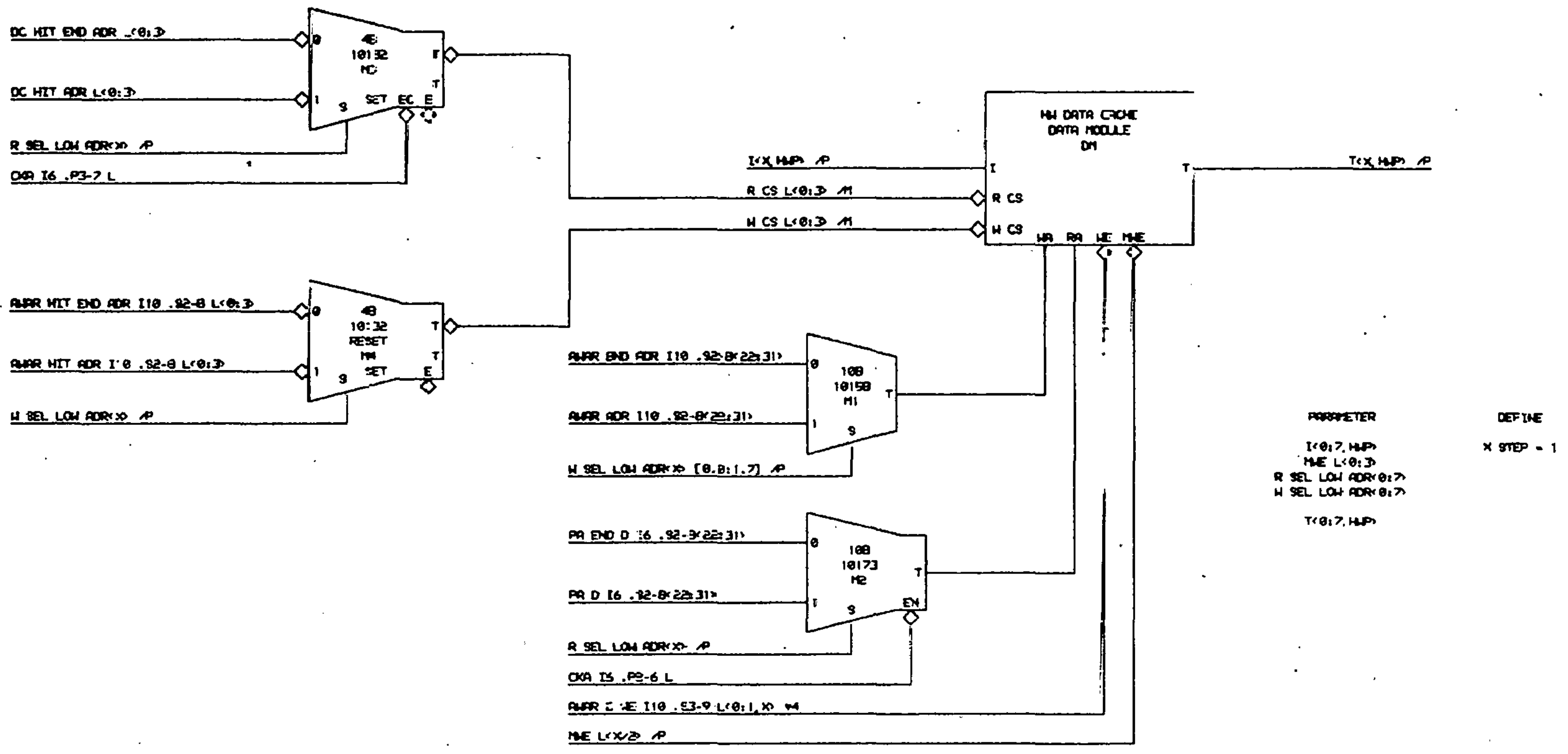
DATA CACHE


PA D I6 .S2-8<2:21>
PA D I6 .S2-8<22:29>
8B
10101
G1
C
H
L
PA D I6 .S2-8<2:21> H
L
21B
EVEN
PARITY TREE
P
I
EVEN
TO DC
PA END D I6 .S2-8<2:21>
PA END D I6 .S2-8<22:29>
8E
10101
G2
C
H
L
M-SEC FORCE HIT
M-SEQ DC WE I9 .94-10 L<0:3>
CK I9 .P5-6.5 L 4HZ
4B
10101
63
C
4X
DATA CACHE
ADR MODULE
A0
A
AP
INVALID
FORCE CMP
HIT
ADR P ERR
RAM ADR
WE
DC HIT ADR L<0:3>
PAR ERR DC ADR L<0:3>
DC RAM ADR<0:3,0:21>
4X
DATA CACHE
ADR MODULE
A1
A
AP
INVALID
FORCE CMP
HIT
ADR P ERR
RAM ADR
WE
DC HIT END ADR L<0:3>
PAR ERR DC END ADR L<0:3>
DC RAM END ADR<0:3,0:21>

DATA CACHE

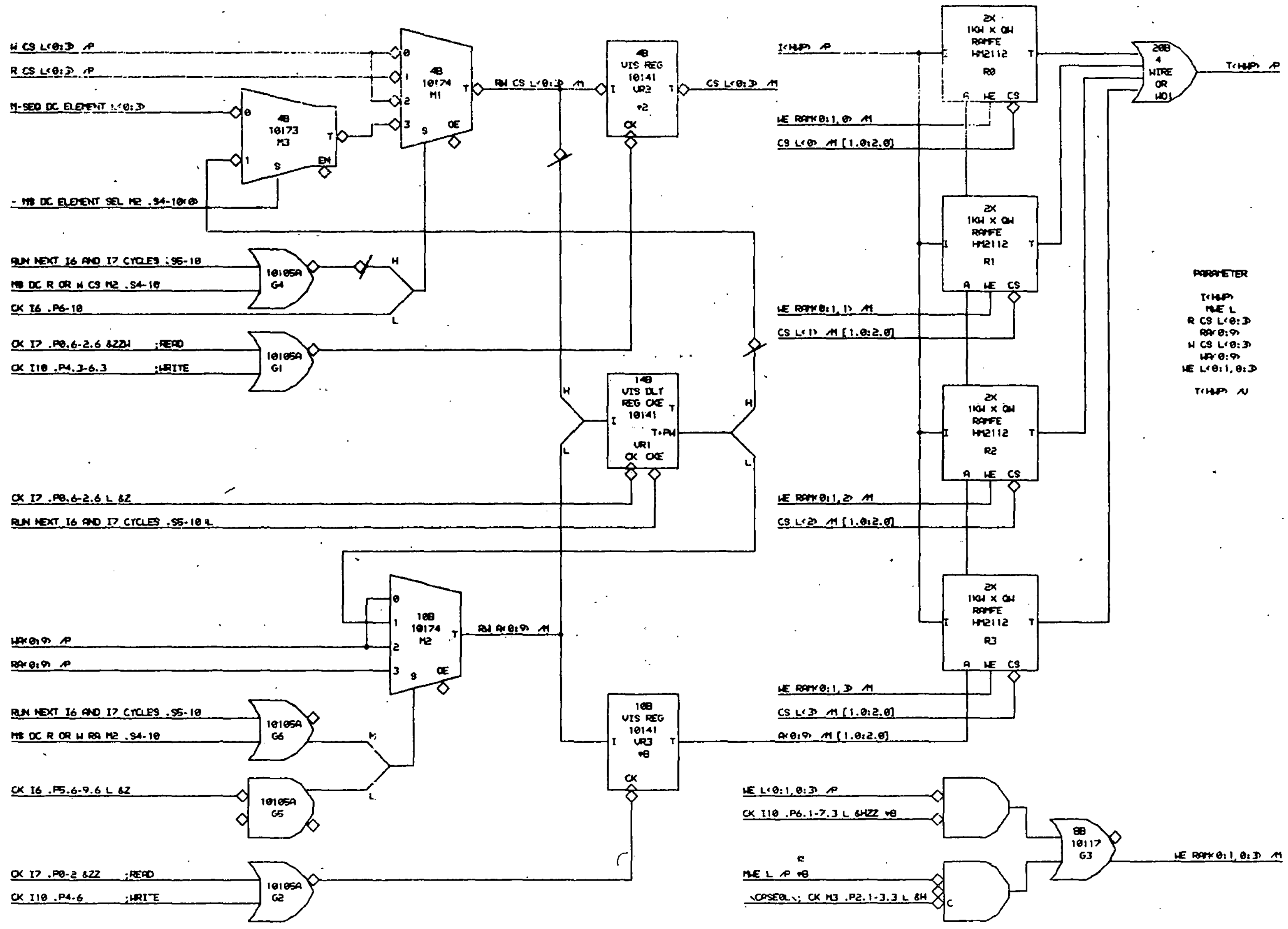

M0 UPDATE DC LRU M2 .S4-10
RUN NEXT I6 AND I7 CYCLES .S5-10 L
I-SEQ USE C I6 .S3-9 L
I0 UPDATE DC LRU I6 .S3-9 L
VALID I6 .S3-9 L
10109A
G2
10105A
G1
DC HIT ADR<0:3>
M0 READ DC LRU M3 .S0-6 L
PA D I6 .S4-10<22:29>
PA END D I6 .S2-8<22:29>
HIT
UPDATE
READ LRU
A
A END
DC LRU
CONTROL
LRU
LRU
LRU END
DC LRU I7<0:1>
DC LRU END I7<0:1>
1 OF 4
DECODER
10161
D1
I
T
OE1 OE2
DC LRU ELEMENT I7 L<0:3>
1 OF 4
DECODER
10161
D2
I
T
OE1 OE2
DC LRU ELEMENT END I7 L<0:3>
PA C I6 .S4-10<29>
PA END D I6 .S4-10<29>
10107
G3
DC USE TWO LINES I6
M0 SET MODIFIED M4 .S0-6
PA D I7 .S2-8<22:29>
PA END D I7 .S2-8<22:29>
DC USE TWO LINES I7 .S4-10 L
DC LRU ELEMENT I7 L<0:3>
DC LRU ELEMENT END I7 L<0:3>
AWAR ADR I10 .S2-8<22:29>
AWAR END ADR I10 .S2-8<22:29>
AWAR USE TWO LINES I10 .S2-8 L
AWAR HIT ADR I10 .S2-8 L<0:3>
AWAR HIT END ADR I10 .S2-8 L<0:3>
I
RA
RA END
R USE TWO LINES
R LRU ELEMENT
R LRU ELEMENT END
WA
WA END
W USE TWO LINES
W HIT
W HIT END
MODIFIED BITS
MOD
T
WE
MWE
DC MODIFIED I8 .S6-12<EVEN:ODD,0:3>
DO ABOX W I10 .S4-10 L
M0 UPDATE MODIFIED BITS M4 .S0-6 L

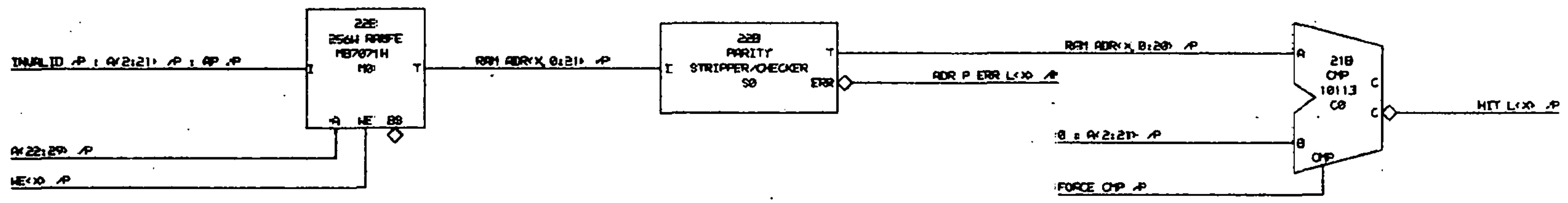
DATA CACHE



DATA CACHE RAM ARRAY

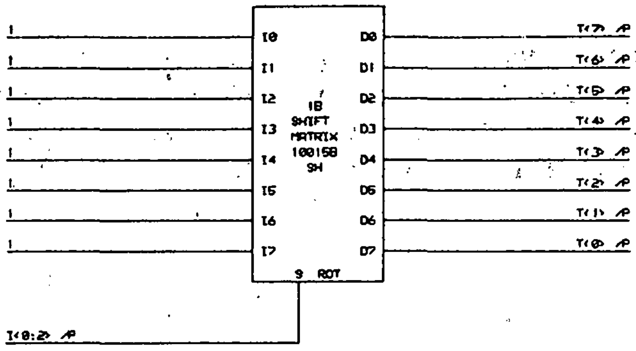
HW DATA CACHE DATA MODULE.





PARAMETER	DEFINE
A<2:29>	X STEP = 1
AP	
FORCE CMP	
INVALID	
WE<0:3>	
ADR P ERR L<0:3>	
HIT L<0:3>	
RAM ADR<0:3, 0:21>	

DATA CACHE ADR MODULE



PARAMETER

I<0:2>

T<0:7>

TRUTH TABLE

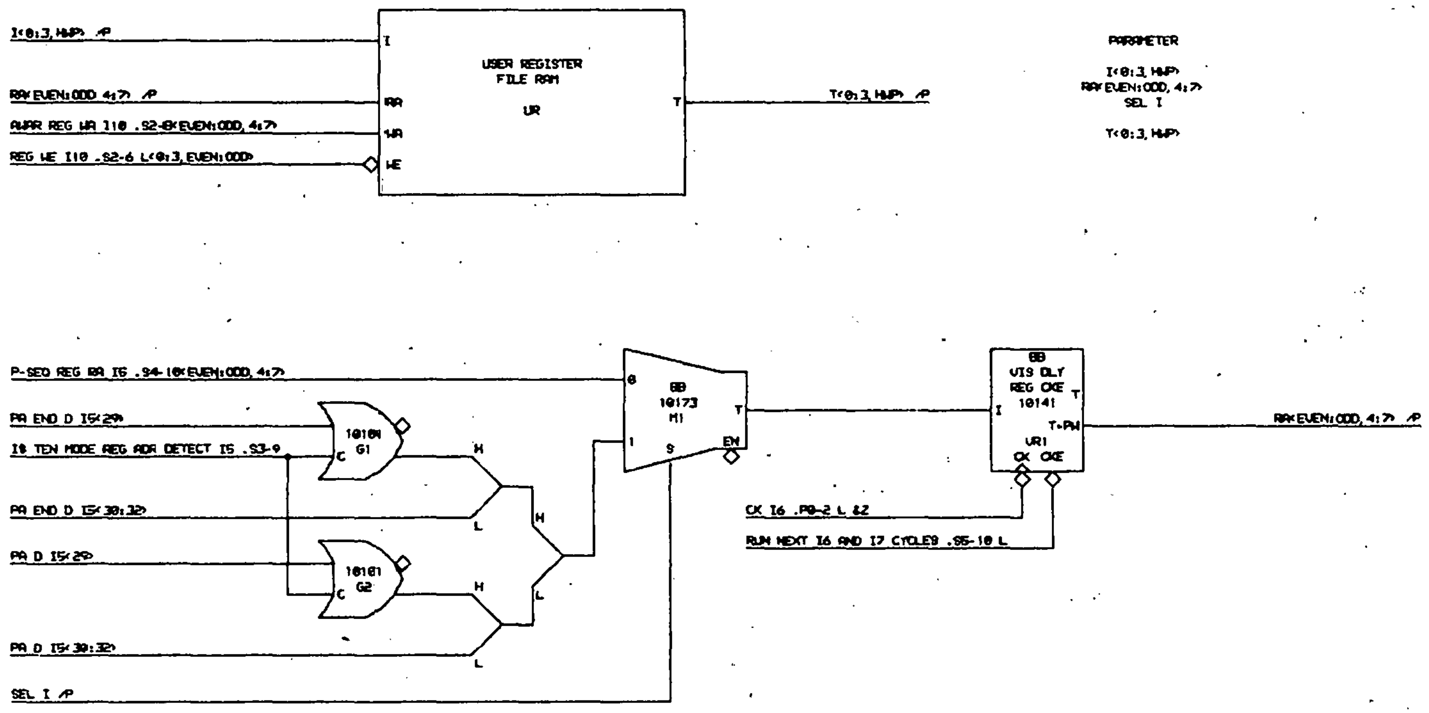
I0	I1	I2	T0	T1	T2	T3	T4	T5	T6	T7
0	0	0	1	1	1	1	1	1	1	1
0	0	1	0	1	1	1	1	1	1	1
0	1	0	0	0	1	1	1	1	1	1
0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1
1	0	1	0	0	0	0	0	1	1	1
1	1	0	0	0	0	0	0	0	1	1
1	1	1	0	0	0	0	0	0	0	1

3 BIT SELECT ALL GEQ

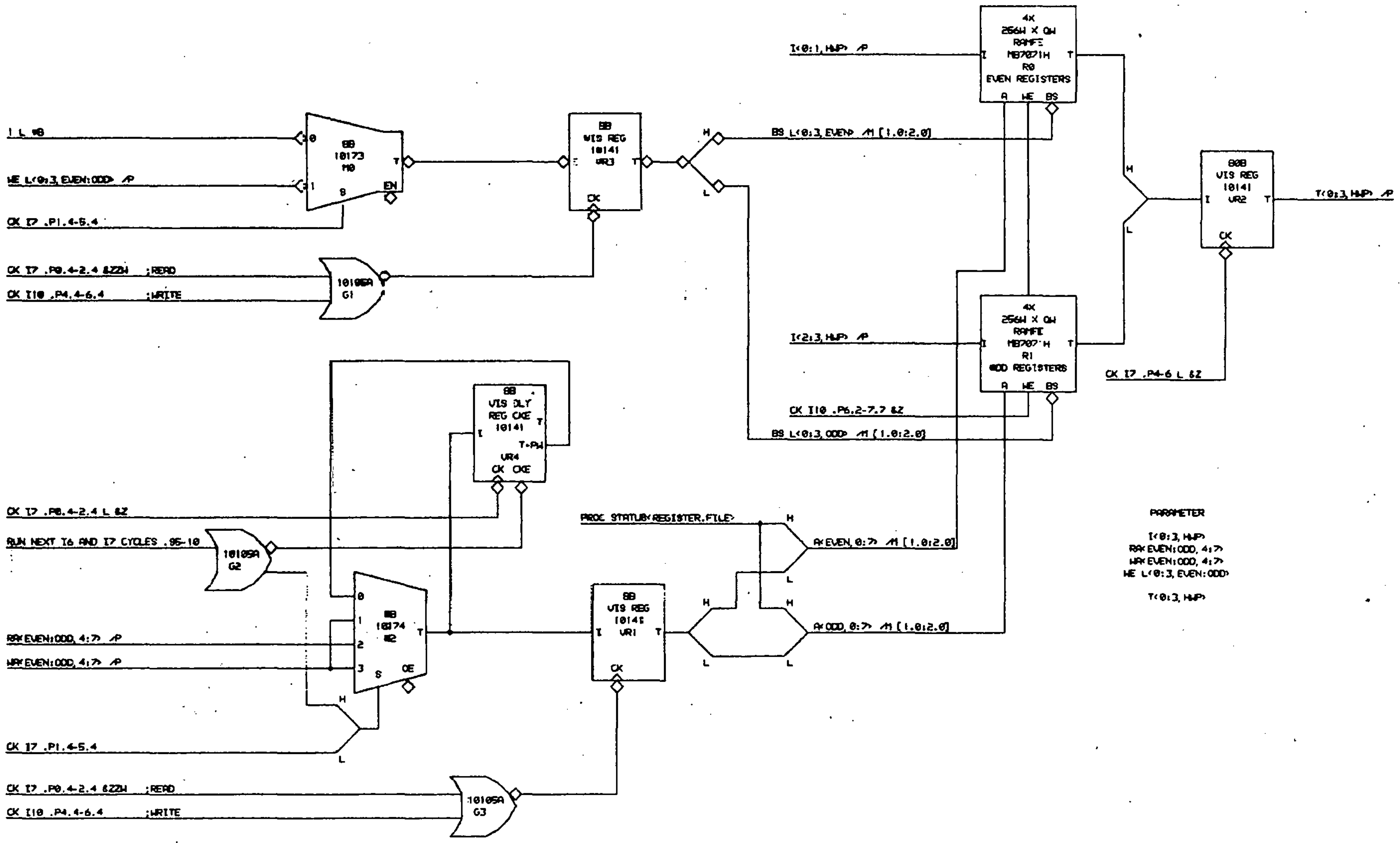
2.13 USREG

Drawings: USREG1, USRAM

The user register file is an even/odd pair of register files and is very similar to the index register file described previously. One difference is that both outputs of the user register file are used together. (In the index register file one or the other of the RAMs was used.) This produces a double word output. Both RAMs may be written simultaneously, so that a double word result may be written in one cycle.



USER REGISTER FILE



USER REGISTER FILE RAM

2.14 OPQ

Drawings: OPQ1, OPQ2, OPQ3, OPQ4, OPQ5

The operand queue stores data operands from the cache on their way to the ABOX. The size of the operand queue is 16 entries by 8 half words per entry. For scalar operations each entry may be considered to be a pair of double words, one for OP1 and one for OP2. For vector operations an entry may also be a pair of double words, or it may be a quad word from the cache. For all scalar and most vector operations the operand queue acts as a FIFO. However, there are vector operations that make use of the fact that the write addresses to the 8 individual halfwords can be different. In any case, the read addresses for the 8 halfwords are always the same.

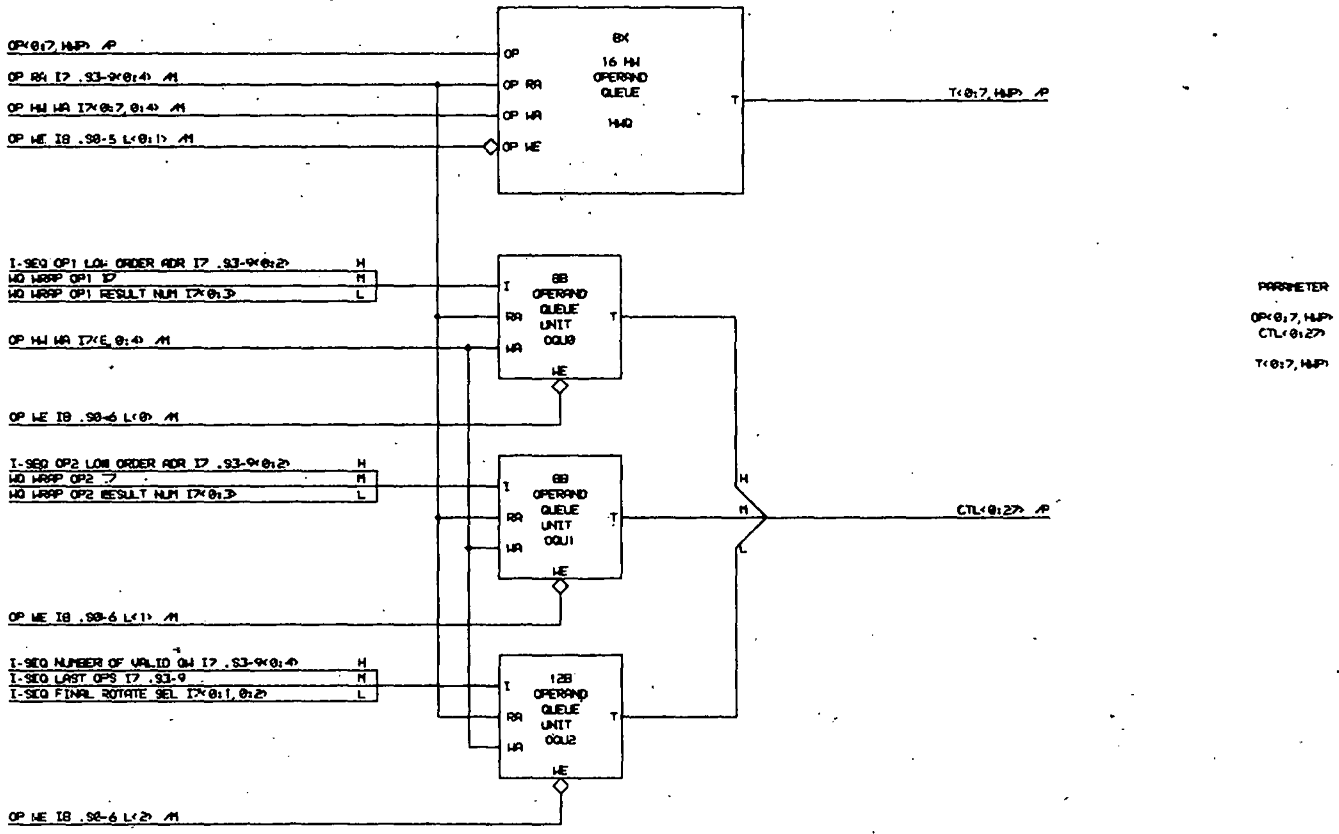
In addition to the main operand queue there are three auxiliary queues for passing control bits to the ABOX. These queues are also 16 entries long and are read out exactly in parallel with the operand queue. The control bits are all written together at a write address that is controlled independently from the operand queue write address. Two of these queues store control bits for OP1 and OP2 and the third stores control bits for both operands together.

The I-Sequencer microcode has a great deal of flexibility in how it controls the operand queue. LOAD OP1 and LOAD OP2 control the writing of the cache and register file outputs into the currently addressed operand queue locations. These signals also enable the writing of the OP1 and OP2 control bits. LOAD BOTH OP CTL enables the writing of the "both" control bits. After the operand queue loading is completed, the write address is incremented by an amount specified in microcode. If incrementing by this amount would cause the operand queue to overflow, then the actual writing of operands and updating of the write address is delayed until the ABOX empties enough entries from the queue to allow the write operation to proceed without overflowing. In order to allow writing operands at different addresses within the operand queue, the actual write addresses are computed by adding the main write address and 8 independent offsets. These offsets come from a RAM whose address is specified by I-Sequencer microcode. The RAM has one additional offset which is added to the main write address to form the write address for the three control bit queues.

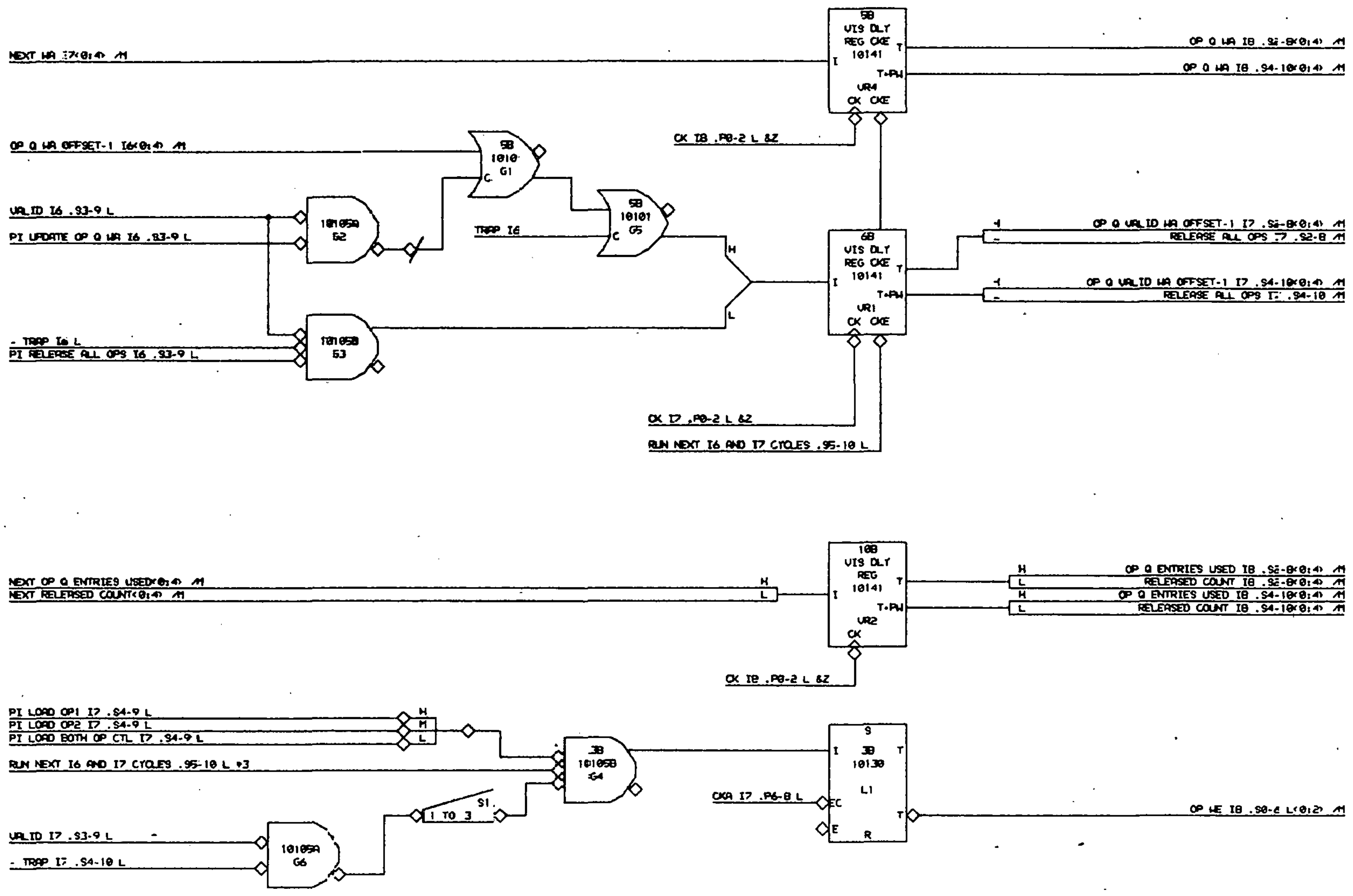
Operands that are written into the queue need not necessarily be immediately released. Multiple sets of operands may be scheduled and not released. However, once a set of operands is scheduled and released, all previously scheduled operands are released. One way to think of it is that there is a control line called RELEASE ALL OPS which may be asserted (with or without actually loading new operands) which releases all operands including any that are to be loaded in the same cycle. In this way, the operand queue may have some number of released operands that can be delivered to the ABOX followed by some number of additional operands that cannot. When these operands are released, they can then be delivered to the ABOX. This provides added flexibility to the microcode. For example, the I-Sequencer microcode can schedule a short block of operands for a vector instruction without releasing them until the last read operation of the block. When it schedules the last read operation, it also releases all the operands.

Several registers are used to store the state of the operand queue. One, called OP Q ENTRIES USED stores a count of how full the operand queue is. Another is called RELEASED COUNT and indicates how many entries, of those that are used, are released for reading by the ABOX. Finally the current operand queue write and read addresses are stored.

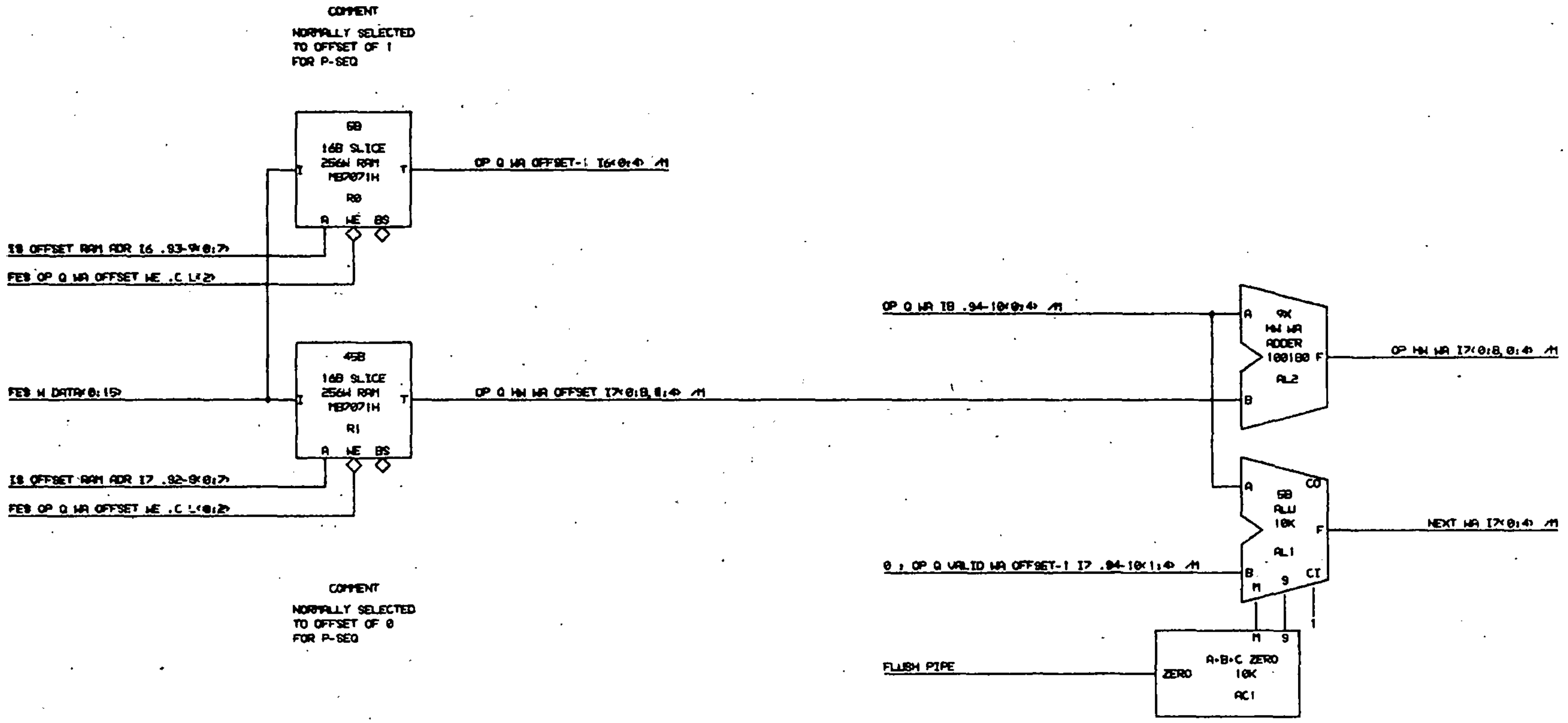
On the reading side, the RELEASED COUNT being non-zero indicates that there is at least one operand queue entry ready to be read by the ABOX. The output of the operand queue is sent to a four word wide register mentioned in the OPABX description, called the ABOX OP REG. The operand queue logic maintains a bit which indicates whether the ABOX OP REG is full or not. When the operand queue has an entry ready and the ABOX OP REG is not full, an operand is transferred from the queue to the register. Whenever the register is full the IBOX informs the ABOX that there is an operand ready (X ABOX OPS READY). When the ABOX is going to accept the operand, it informs the IBOX (X ABOX OPS TAKEN) which can then move another operand into the ABOX OP REG.



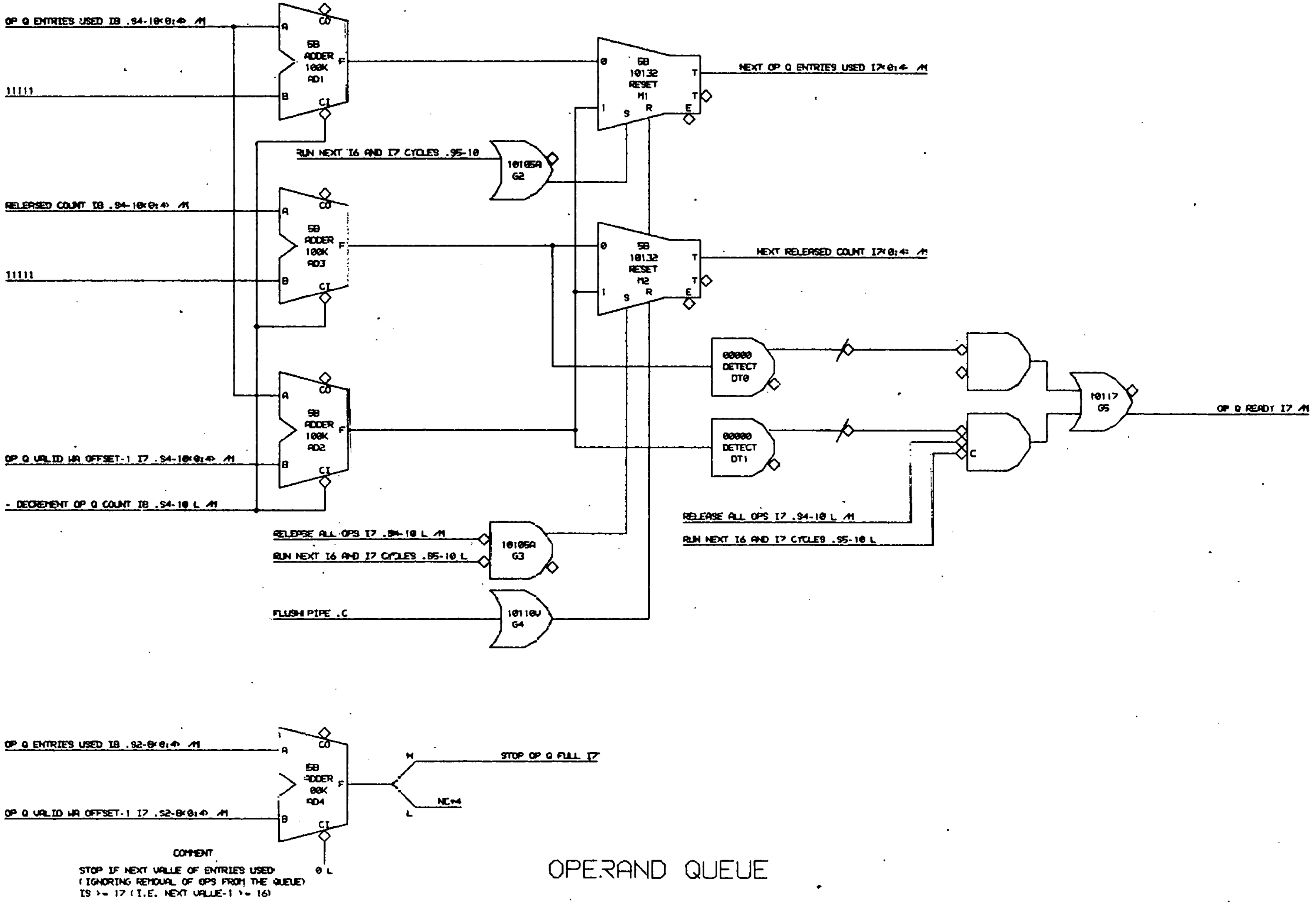
OPERAND QUEUE


NEXT WA I7<0:4> /M
5B VIS DLY REG CKE 10141 VR4
I
T
T+PW
CK CKE
OP Q WA I8 .S2-8<0:4> /M
OP Q WA I8 .S4-10<0:4> /M
CK I8 .P0-2 L &Z
OP Q WA OFFSET-1 I6<0:4> /M
5B 1010 G1
C
VALID I6 .S3-9 L
PI UPDATE OP Q WA I6 .S3-9 L
10105A G2
TRAP I6
5B 10101 G5
C
H
L
- TRAP I6 L
PI RELEASE ALL OPS I6 .S3-9 L
10105B G3
6B VIS DLY REG CKE 10141 VR1
OP Q VALID WA OFFSET-1 I7 .S2-8<0:4> /M
RELEASE ALL OPS I7 .S2-8 /M
OP Q VALID WA OFFSET-1 I7 .S4-10<0:4> /M
RELEASE ALL OPS I7 .S4-10 /M
CK I7 .P0-2 L &Z
RUN NEXT I6 AND I7 CYCLES .S5-10 L
NEXT OP Q ENTRIES USED<0:4> /M
NEXT RELEASED COUNT<0:4> /M
10B VIS DLY REG 10141 VR2
CK
OP Q ENTRIES USED I8 .S2-8<0:4> /M
RELEASED COUNT I8 .S2-8<0:4> /M
OP Q ENTRIES USED I8 .S4-10<0:4> /M
RELEASED COUNT I8 .S4-10<0:4> /M
CK I8 .P0-2 L &Z
PI LOAD OP1 I7 .S4-9 L
PI LOAD OP2 I7 .S4-9 L
PI LOAD BOTH OP CTL I7 .S4-9 L
H
M
L
RUN NEXT I6 AND I7 CYCLES .S5-10 L +3
3B 10105B G4
S1
1 TO 3
VALID I7 .S3-9 L
- TRAP I7 .S4-10 L
10105A G6
S
3B 10130 L1
EC
E
R
CKA I7 .P6-8 L
OP WE I8 .S0-6 L<0:2> /M

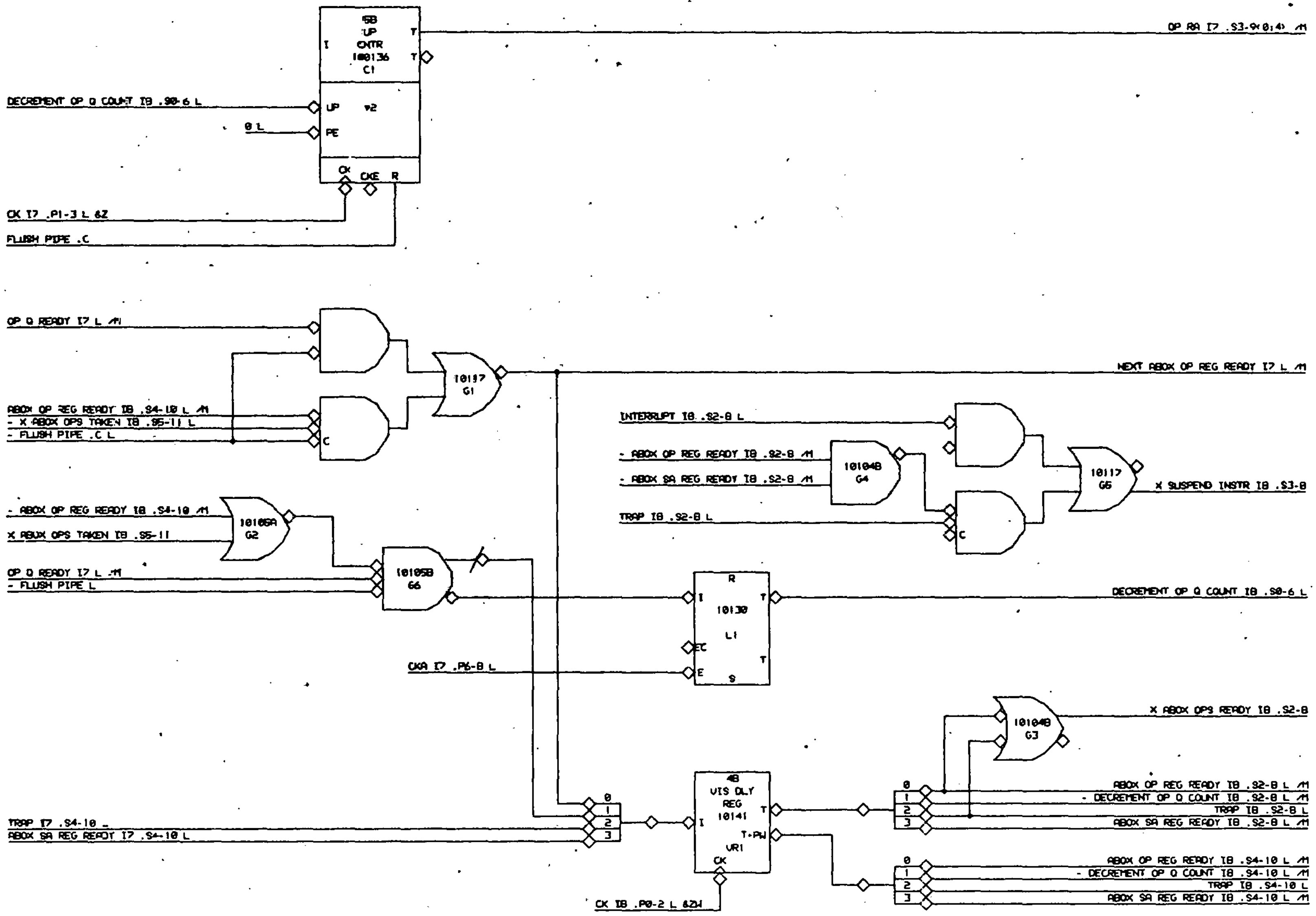
CPERAND QUEUE



OPERAND QUEUE


OP Q ENTRIES USED I8 .S4-10<0:4> /H
11111
5B ADDER 100K AD1
RUN NEXT I6 AND I7 CYCLES .S5-10
10105A G2
RELEASED COUNT I8 .S4-10<0:4> /H
11111
5B ADDER 100K AD3
5B ADDER 100K AD2
OP Q VALID WA OFFSET-1 I7 .S4-10<0:4> /H
- DECREMENT OP Q COUNT I8 .S4-10 L /H
RELEASE ALL OPS I7 .S4-10 L /H
RUN NEXT I6 AND I7 CYCLES .S5-10 L
10105A G3
FLUSH PIPE .C
10110V G4
5B 10132 RESET M1
NEXT OP Q ENTRIES USED I7<0:4> /H
5B 10132 RESET M2
NEXT RELEASED COUNT I7<0:4> /H
00000 DETECT DT0
00000 DETECT DT1
10117 G5
OP Q READY I7 /H
RELEASE ALL OPS I7 .S4-10 L /H
RUN NEXT I6 AND I7 CYCLES .S5-10 L
OP Q ENTRIES USED I8 .S2-8<0:4> /H
OP Q VALID WA OFFSET-1 I7 .S2-8<0:4> /H
5B ADDER 00K AD4
STOP OP Q FULL I7
NC=4
0 L
COMMENT
STOP IF NEXT VALUE OF ENTRIES USED
(IGNORING REMOVAL OF OPS FROM THE QUEUE)
IS >= 17 (I.E. NEXT VALUE-1 >= 16)

OPERAND QUEUE


5B
UP
CNTR
100136
C1
I
T
T
UP
PE
CK
CKE
R
÷2
OP RA I7 .S3-9<0:4> /M
DECREMENT OP Q COUNT I8 .S0-6 L
0 L
CK I7 .P1-3 L &Z
FLUSH PIPE .C
OP Q READY I7 L /M
ABOX OP REG READY I8 .S4-10 L /M
- X ABOX OPS TAKEN I8 .S5-11 L
- FLUSH PIPE .C L
C
10117
G1
NEXT ABOX OP REG READY I7 L /M
- ABOX OP REG READY I8 .S4-10 /M
X ABOX OPS TAKEN I8 .S5-11
10105A
G2
OP Q READY I7 L /M
- FLUSH PIPE L
10105B
G6
INTERRUPT I8 .S2-8 L
- ABOX OP REG READY I8 .S2-8 /M
- ABOX SA REG READY I8 .S2-8 /M
TRAP I8 .S2-8 L
10104B
G4
C
10117
G5
X SUSPEND INSTR I8 .S3-8
R
I
10130
L1
T
EC
E
T
S
DECREMENT OP Q COUNT I8 .S0-6 L
CKA I7 .P6-8 L
10104B
G3
X ABOX OPS READY I8 .S2-8
0
1
2
3
TRAP I7 .S4-10 L
ABOX SA REG READY I7 .S4-10 L
4B
VIS DLY
REG
10141
VR1
I
T
T-PW
CK
CK I8 .P0-2 L &2W
0
1
2
3
ABOX OP REG READY I8 .S2-8 L /M
- DECREMENT OP Q COUNT I8 .S2-8 L /M
TRAP I8 .S2-8 L
ABOX SA REG READY I8 .S2-8 L /M
0
1
2
3
ABOX OP REG READY I8 .S4-10 L /M
- DECREMENT OP Q COUNT I8 .S4-10 L /M
TRAP I8 .S4-10 L
ABOX SA REG READY I8 .S4-10 L /M

OPERAND QUEUE

2.15 PSEQ

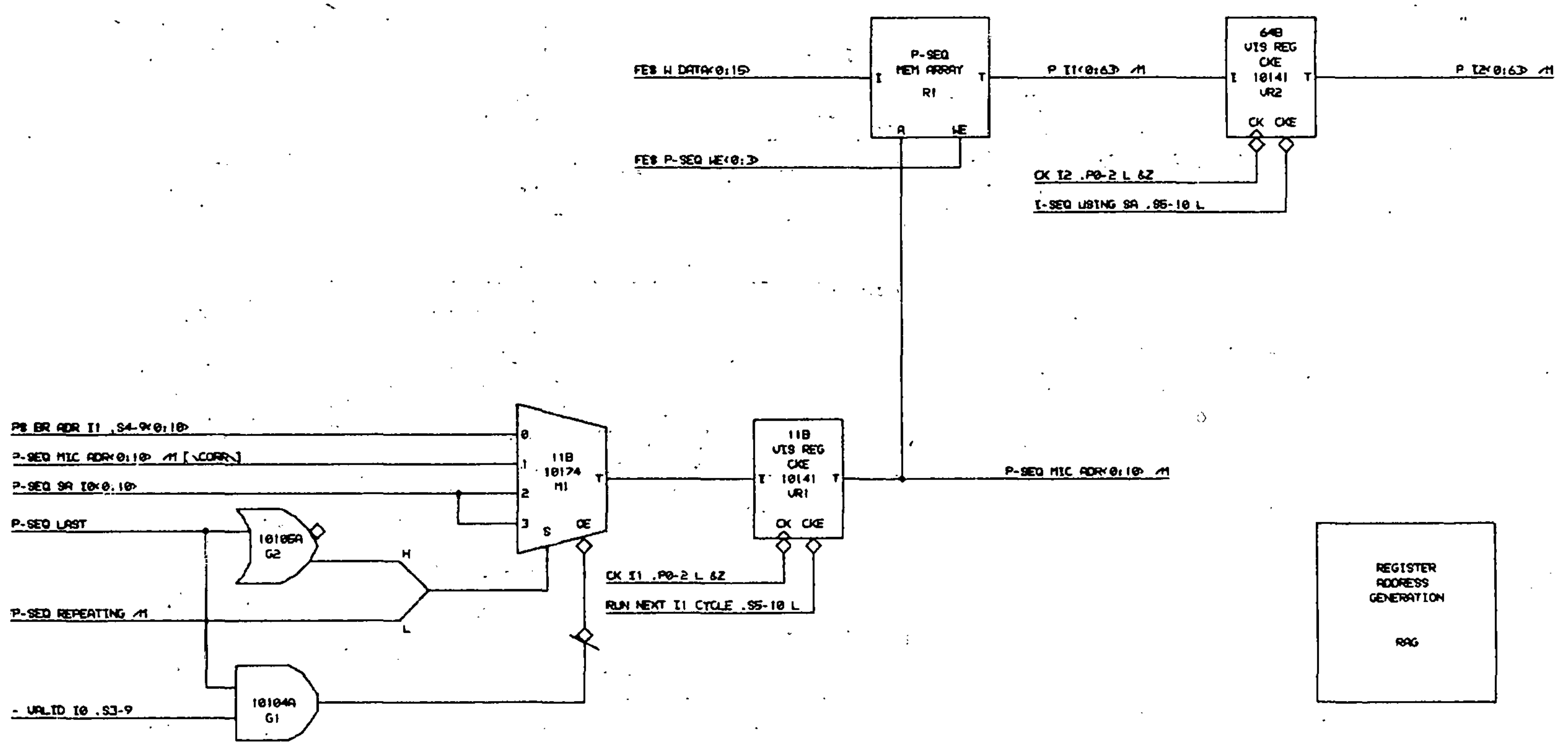
Drawings: PSEQ1, PSEQ2, PSEQ3, ODDEC, RAG1, RAG2, RAG3

The P-Sequencer is one of the major microcoded sequencers in the IBOX. This sequencer specifies register read and write operations and constant read operations (except for indexed constants). The P-Sequencer works in conjunction with the I-Sequencer. Every P-Sequencer microinstruction is associated with an entire sequence of I-Sequencer microinstructions. Each P-Sequencer microinstruction specifies a starting address for the I-Sequencer. In essence every P-Sequencer microinstruction calls an I-Sequencer subroutine. Usually the P-Sequencer works on one operand (say OP1) while the I-Sequencer works on the other (OP2). In this way, the P-Sequencer can specify a register read operation at the same time that the I-Sequencer specifies a cache or register read operation.

One of the more commonly called I-Sequencer subroutines is the operand calculation routine. This routine is actually a collection of routines to perform the calculations necessary to implement the various addressing modes of the S-1 native mode architecture. The P-Sequencer dispatches to the correct I-Sequencer routine according to the addressing mode used by the operand in question.

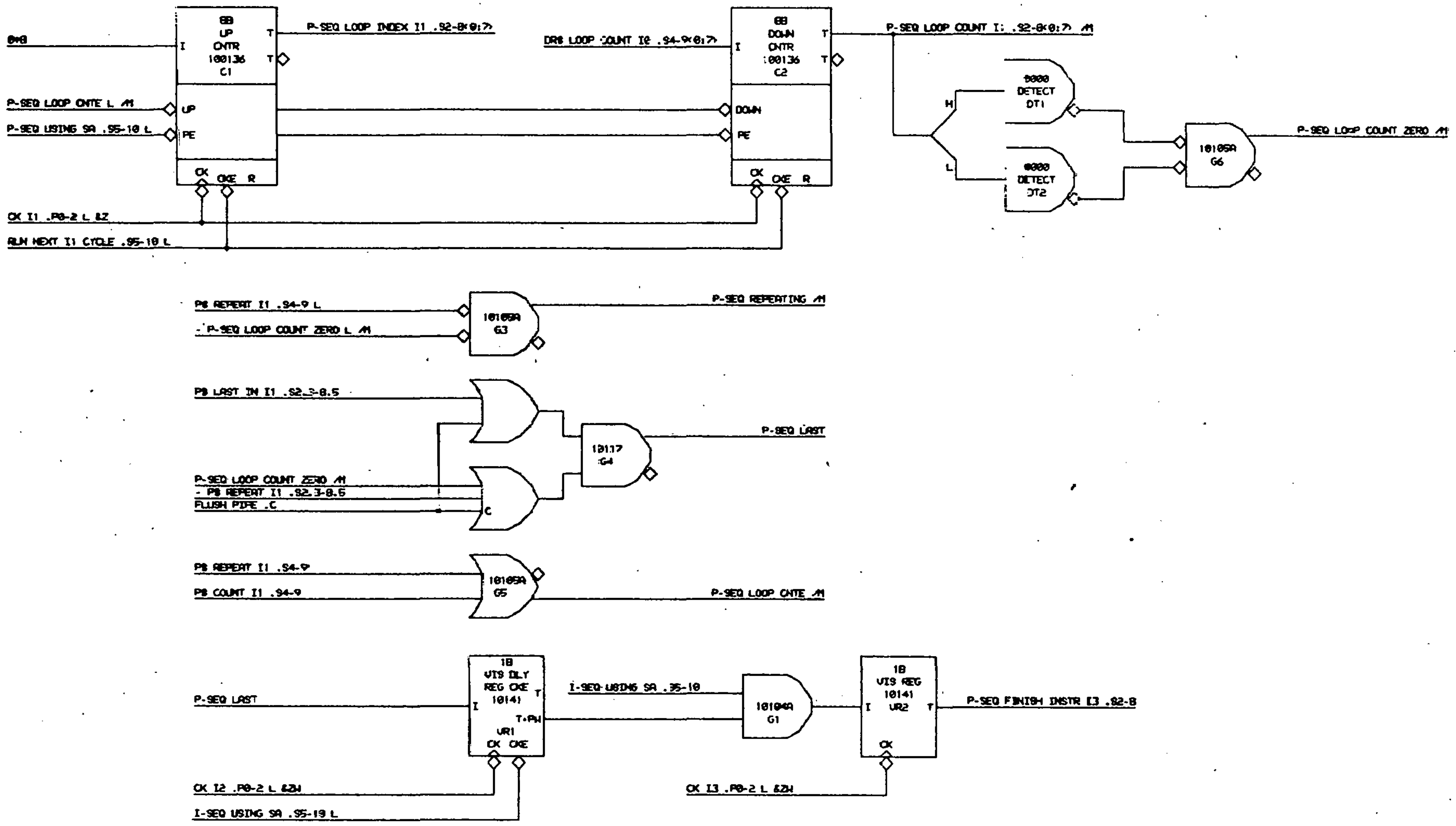
Multiple P-Sequencer instructions may be required for the execution of some macroinstructions. When the P-Sequencer has completed the execution of one macroinstruction, it proceeds to take a starting address for the next macroinstruction. This starting address is provided by the decode RAM which is being addressed by the opcode of the next instruction coming out of the instruction queue during the I0 pipeline stage. The fetch of the first and succeeding P-Sequencer instructions is performed during the I1 stage. The execution of the microinstruction fetched during I1 occurs during multiple succeeding cycles starting with the latter half of I1. In particular, the first I-Sequencer instruction specified by the P-Sequencer is fetched during the I2 cycle. When an I-Sequencer sequence finishes, the P-Sequencer may fetch another microinstruction. If only a single instruction sequence was needed then the P-Sequencer will fetch the starting address for the next macroinstruction. If more than one instruction was needed then it will fetch the next instruction in the sequence, which is specified as a branch address in the previous P-Sequencer microinstruction. One final alternative is that the P-Sequencer may repeat a microinstruction a number of times determined by a counter. This loop counter may be loaded from the decode RAM at the start of the sequence.

The PSEQ drawings contain the logic used to compute register read and write addresses for operands and for address calculations. This logic is on the RAG drawings. The P-Sequencer has a good deal of flexibility in these register address calculations. The register addresses may come from various fields of the macroinstruction or the P-Sequencer microinstruction and may be added to the loop count with an optional shift.

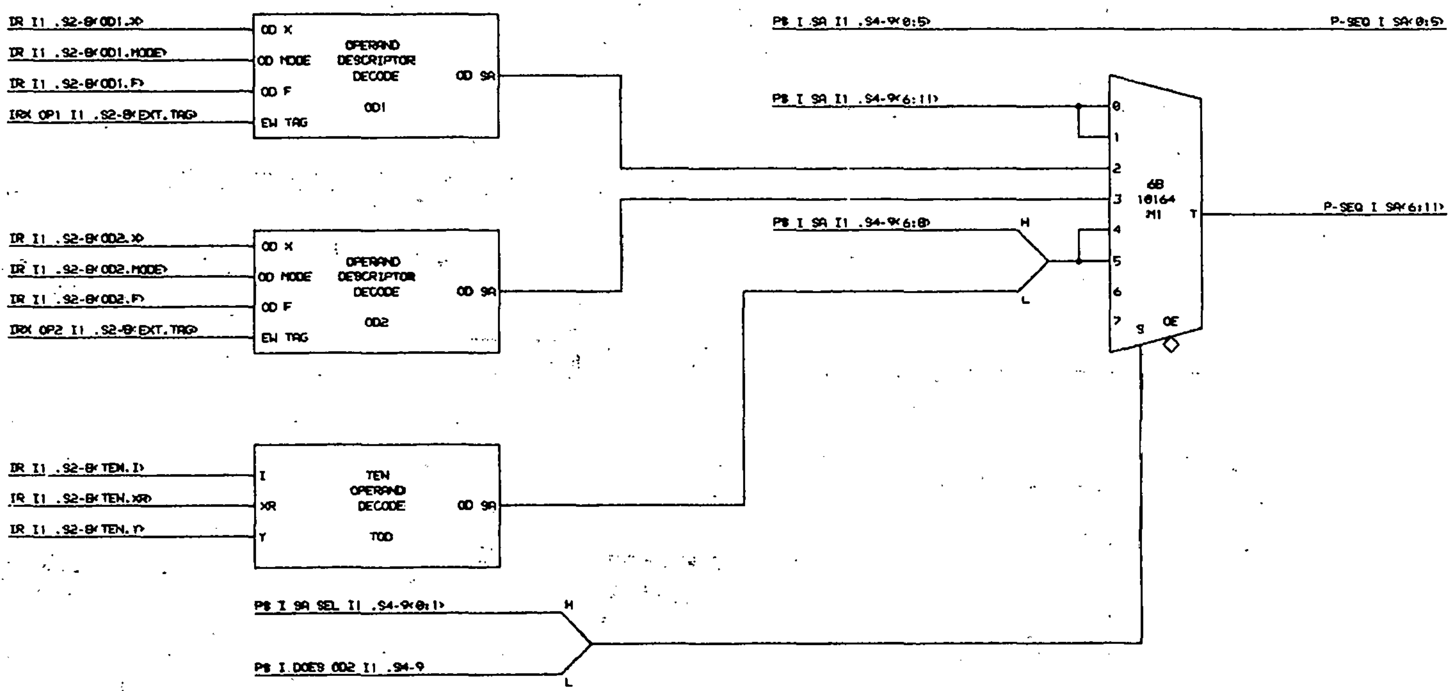


FE$ W DATA<0:15>
P-SEQ MEM ARRAY R1
I
T
A
WE
FE$ P-SEQ WE<0:3>
P I1<0:63> /M
64B VIS REG CKE 10141 VR2
CK CKE
P I2<0:63> /M
CK I2 .P0-2 L 6Z
I-SEQ USING SA .S5-10 L
P$ BR ADR I1 .S4-9<0:10>
P-SEQ MIC ADR<0:10> /M [\CORR\]
P-SEQ SA I0<0:10>
P-SEQ LAST
P-SEQ REPEATING /M
- VALID I0 .S3-9
11B 10174 M1
0
1
2
3
S
OE
10105A G2
10104A G1
H
L
11B VIS REG CKE 10141 VR1
CK CKE
CK I1 .P0-2 L 6Z
RUN NEXT I1 CYCLE .S5-10 L
P-SEQ MIC ADR<0:10> /M
REGISTER ADDRESS GENERATION
RAG

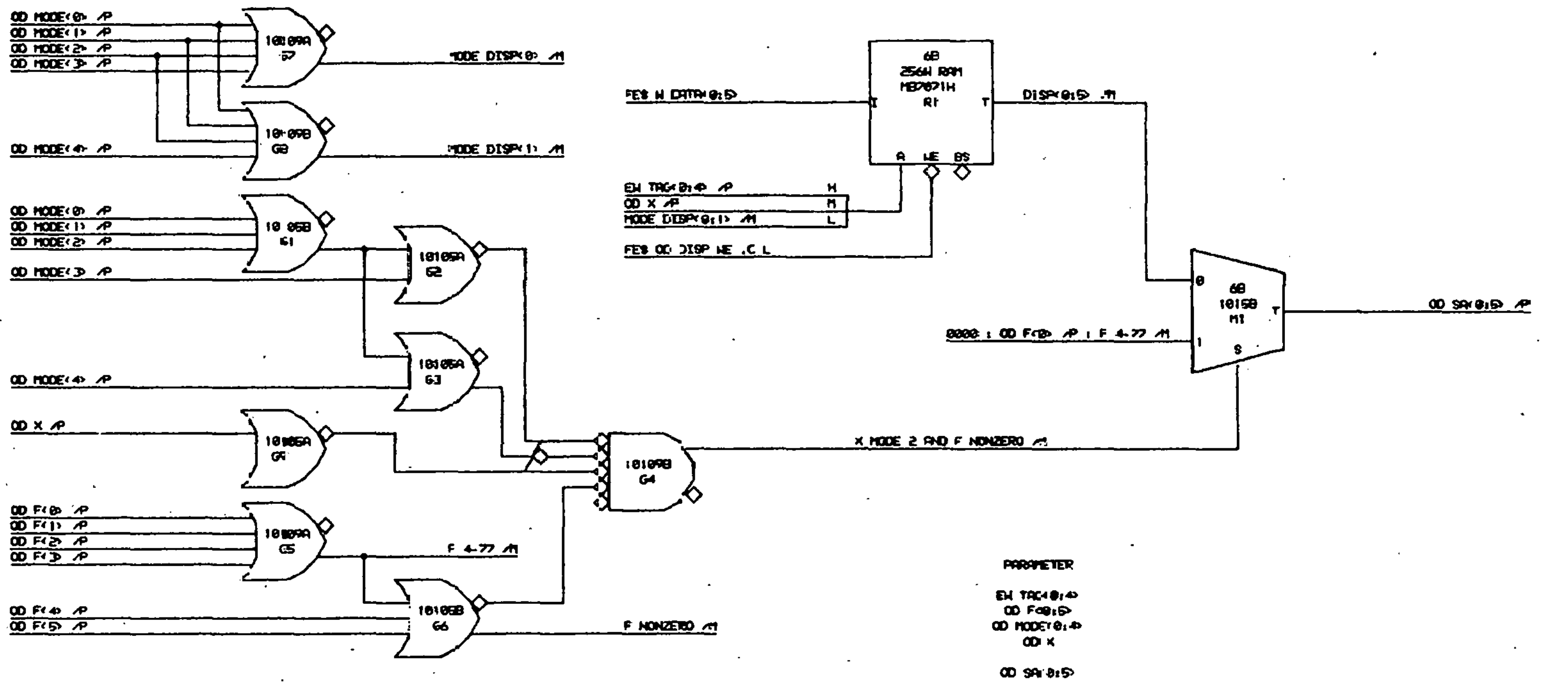
P-SEQUENCER


0+0
8B UP CNTR 100136 C1
I
T
T
P-SEQ LOOP INDEX I1 .92-8<0:7>
P-SEQ LOOP CNTE L /M
UP
P-SEQ USING SA .95-10 L
PE
CK CKE R
CK I1 .P0-2 L &Z
RUN NEXT I1 CYCLE .95-10 L
DR8 LOOP COUNT I0 .94-9<0:7>
8B DOWN CNTR 100136 C2
DOWN
PE
CK CKE R
P-SEQ LOOP COUNT I1 .92-8<0:7> /M
H
L
0000 DETECT DT1
0000 DETECT DT2
101050 G6
P-SEQ LOOP COUNT ZERO /M
P8 REPEAT I1 .94-9 L
- P-SEQ LOOP COUNT ZERO L /M
101050 G3
P-SEQ REPEATING /M
P8 LAST IN I1 .92_3-8.5
P-SEQ LOOP COUNT ZERO /M
- P8 REPEAT I1 .92.3-8.5
FLUSH PIPE .C
C
10117 G4
P-SEQ LAST
P8 REPEAT I1 .94-9
P8 COUNT I1 .94-9
101050 G5
P-SEQ LOOP CNTE /M
P-SEQ LAST
1B VIS DLY REG CKE 10141
I
T
T+PW
VR1
CK CKE
CK I2 .P0-2 L &2W
I-SEQ USING SA .95-19 L
I-SEQ USING SA .95-10
101040 G1
1B VIS REG 10141
I VR2 T
CK
CK I3 .P0-2 L &2W
P-SEQ FINISH INSTR I3 .92-8

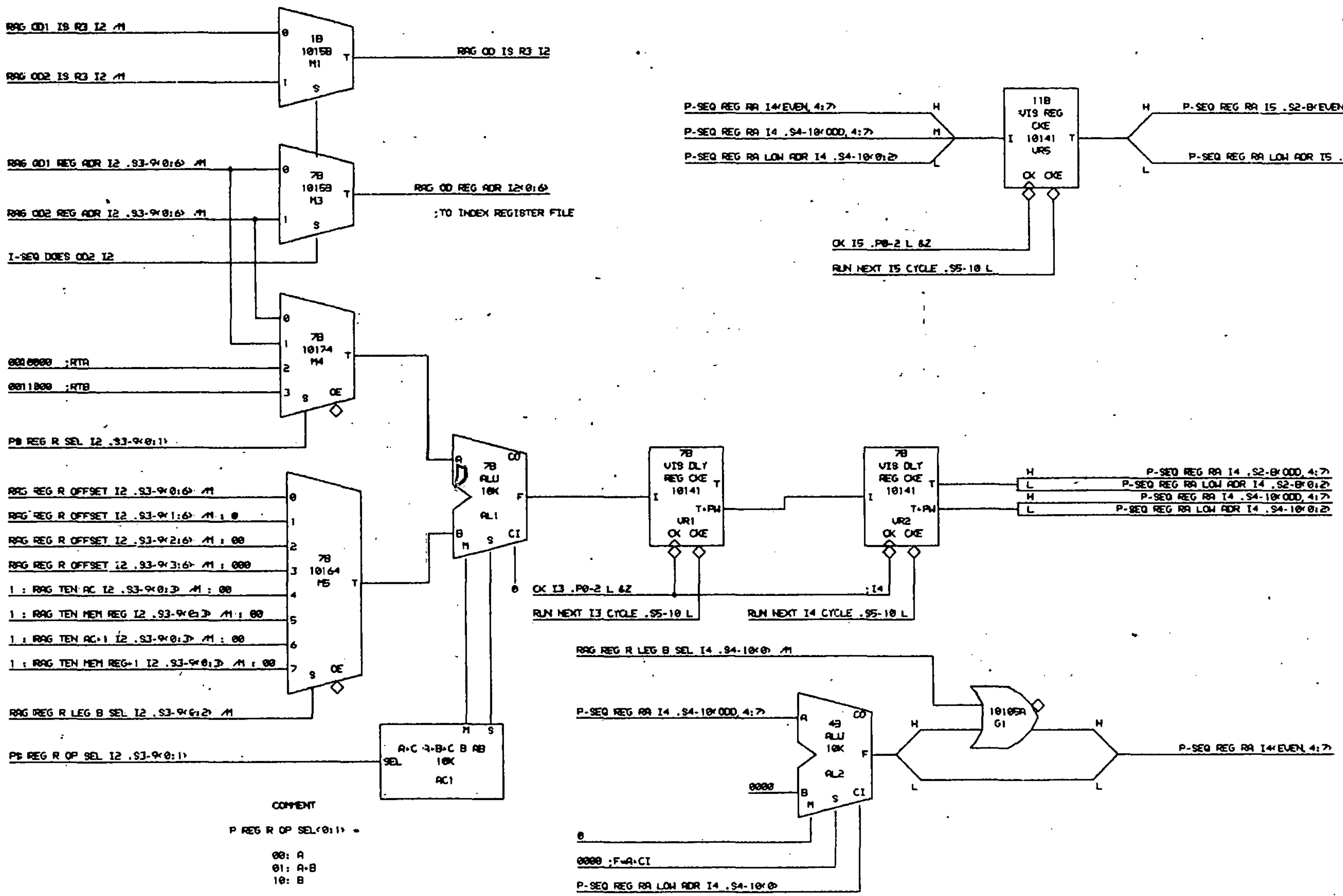
P-SEQUENCER


IR I1 .S2-8<OD1.X>
IR I1 .S2-8<OD1.MODE>
IR I1 .S2-8<OD1.F>
IRX OP1 I1 .S2-8<EXT.TAG>
OD X
OD MODE
OD F
EW TAG
OPERAND DESCRIPTOR DECODE
OD1
OD SA
IR I1 .S2-8<OD2.X>
IR I1 .S2-8<OD2.MODE>
IR I1 .S2-8<OD2.F>
IRX OP2 I1 .S2-8<EXT.TAG>
OPERAND DESCRIPTOR DECODE
OD2
IR I1 .S2-8<TEN.I>
IR I1 .S2-8<TEN.XR>
IR I1 .S2-8<TEN.Y>
I
XR
Y
TEN OPERAND DECODE
TOD
P$ I SA SEL I1 .S4-9<0:1>
P$ I DOES OD2 I1 .S4-9
H
L
P$ I SA I1 .S4-9<0:5>
P-SEQ I SA<0:5>
P$ I SA I1 .S4-9<6:11>
P$ I SA I1 .S4-9<6:8>
0
1
2
3
4
5
6
7
S
OE
6B
10164
M1
T
P-SEQ I SA<6:11>

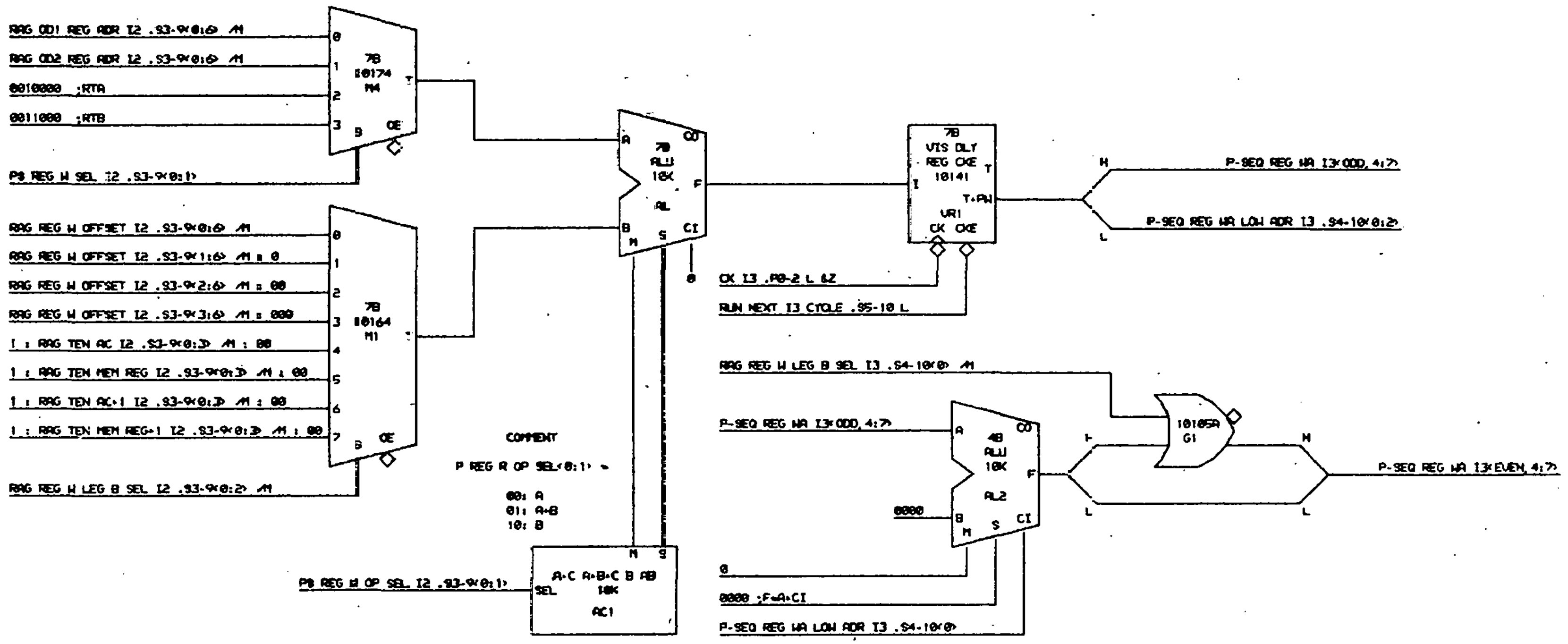
P-SEQUENCER



OPERAND DESCRIPTOR DECODE

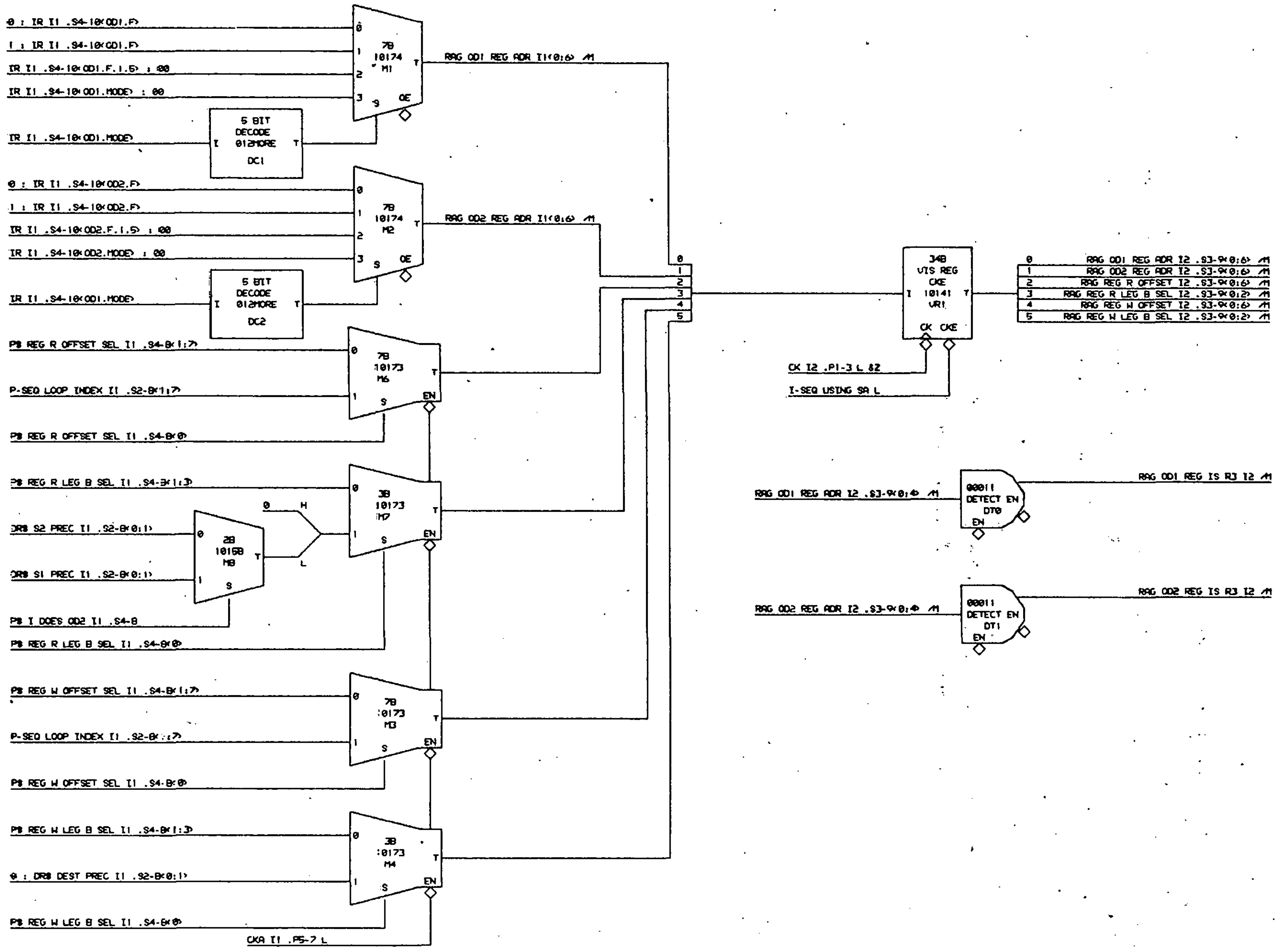

RAG OD1 IS R3 I2 /M
RAG OD2 IS R3 I2 /M
1B 10158 M1
RAG OD IS R3 I2
RAG OD1 REG ADR I2 .S3-9<0:6> /M
RAG OD2 REG ADR I2 .S3-9<0:6> /M
I-SEQ DOES OD2 I2
7B 10158 M3
RAG OD REG ADR I2<0:6>
;TO INDEX REGISTER FILE
0000000 ;RTA
0011000 ;RTB
7B 10174 M4
P0 REG R SEL I2 .S3-9<0:1>
RAG REG R OFFSET I2 .S3-9<0:6> /M
RAG REG R OFFSET I2 .S3-9<1:6> /M : 0
RAG REG R OFFSET I2 .S3-9<2:6> /M : 00
RAG REG R OFFSET I2 .S3-9<3:6> /M : 000
1 : RAG TEN AC I2 .S3-9<0:3> /M : 00
1 : RAG TEN MEM REG I2 .S3-9<0:3> /M : 00
1 : RAG TEN AC+1 I2 .S3-9<0:3> /M : 00
1 : RAG TEN MEM REG+1 I2 .S3-9<0:3> /M : 00
7B 10164 M5
RAG REG R LEG B SEL I2 .S3-9<0:2> /M
P0 REG R OP SEL I2 .S3-9<0:1>
A+C A+B+C B AB 10K AC1
7B ALU 10K AL1
CK I3 .P0-2 L &Z
RUN NEXT I3 CYCLE .S5-10 L
7B VIS DLY REG CKE 10141 VR1
7B VIS DLY REG CKE 10141 VR2
;I4
RUN NEXT I4 CYCLE .S5-10 L
P-SEQ REG RA I4 .S2-8<ODD, 4:7>
P-SEQ REG RA LOW ADR I4 .S2-8<0:2>
P-SEQ REG RA I4 .S4-10<ODD, 4:7>
P-SEQ REG RA LOW ADR I4 .S4-10<0:2>
P-SEQ REG RA I4<EVEN, 4:7>
P-SEQ REG RA I4 .S4-10<ODD, 4:7>
P-SEQ REG RA LOW ADR I4 .S4-10<0:2>
11B VIS REG CKE 10141 VR5
P-SEQ REG RA I5 .S2-8<EVEN:ODD, 4:7>
P-SEQ REG RA LOW ADR I5 .S2-8<0:2>
CK I5 .P0-2 L &Z
RUN NEXT I5 CYCLE .S5-10 L
RAG REG R LEG B SEL I4 .S4-10<0> /M
P-SEQ REG RA I4 .S4-10<ODD, 4:7>
0000
4B ALU 10K AL2
10105A G1
P-SEQ REG RA I4<EVEN, 4:7>
0
0000 ;F=A+CI
P-SEQ REG RA LOW ADR I4 .S4-10<0>
COMMENT
P REG R OP SEL<0:1> =
00: A
01: A+B
10: B

REGISTER ADDRESS GENERATION


RAG OD1 REG ADR I2 .S3-9<0:6> /H
RAG OD2 REG ADR I2 .S3-9<0:6> /H
0010000 ;RTA
0011000 ;RTB
P0 REG W SEL I2 .S3-9<0:1>
7B 10174 M4
RAG REG W OFFSET I2 .S3-9<0:6> /H
RAG REG W OFFSET I2 .S3-9<1:6> /H : 0
RAG REG W OFFSET I2 .S3-9<2:6> /H : 00
RAG REG W OFFSET I2 .S3-9<3:6> /H : 000
1 : RAG TEN AC I2 .S3-9<0:3> /H : 00
1 : RAG TEN MEM REG I2 .S3-9<0:3> /H : 00
1 : RAG TEN AC+1 I2 .S3-9<0:3> /H : 00
1 : RAG TEN MEM REG+1 I2 .S3-9<0:3> /H : 00
RAG REG W LEG B SEL I2 .S3-9<0:2> /H
7B 10164 M1
7B ALU 10K AL
COMMENT
P REG R OP SEL<0:1> =
00: A
01: A+B
10: B
P0 REG W OP SEL I2 .S3-9<0:1>
A+C A+B+C B AB 10K AC1
7B VIS DLY REG CKE 10141 VR1
CK I3 .P0-2 L &Z
RUN NEXT I3 CYCLE .S5-10 L
P-SEQ REG WA I3<ODD, 4:7>
P-SEQ REG WA LOW ADR I3 .S4-10<0:2>
RAG REG W LEG B SEL I3 .S4-10<0> /H
10105A G1
4B ALU 10K AL2
0000
0
0000 ;F=A+CI
P-SEQ REG WA LOW ADR I3 .S4-10<0>
P-SEQ REG WA I3<EVEN, 4:7>

REGISTER ADDRESS GENERATION

REGISTER ADDRESS GENERATION

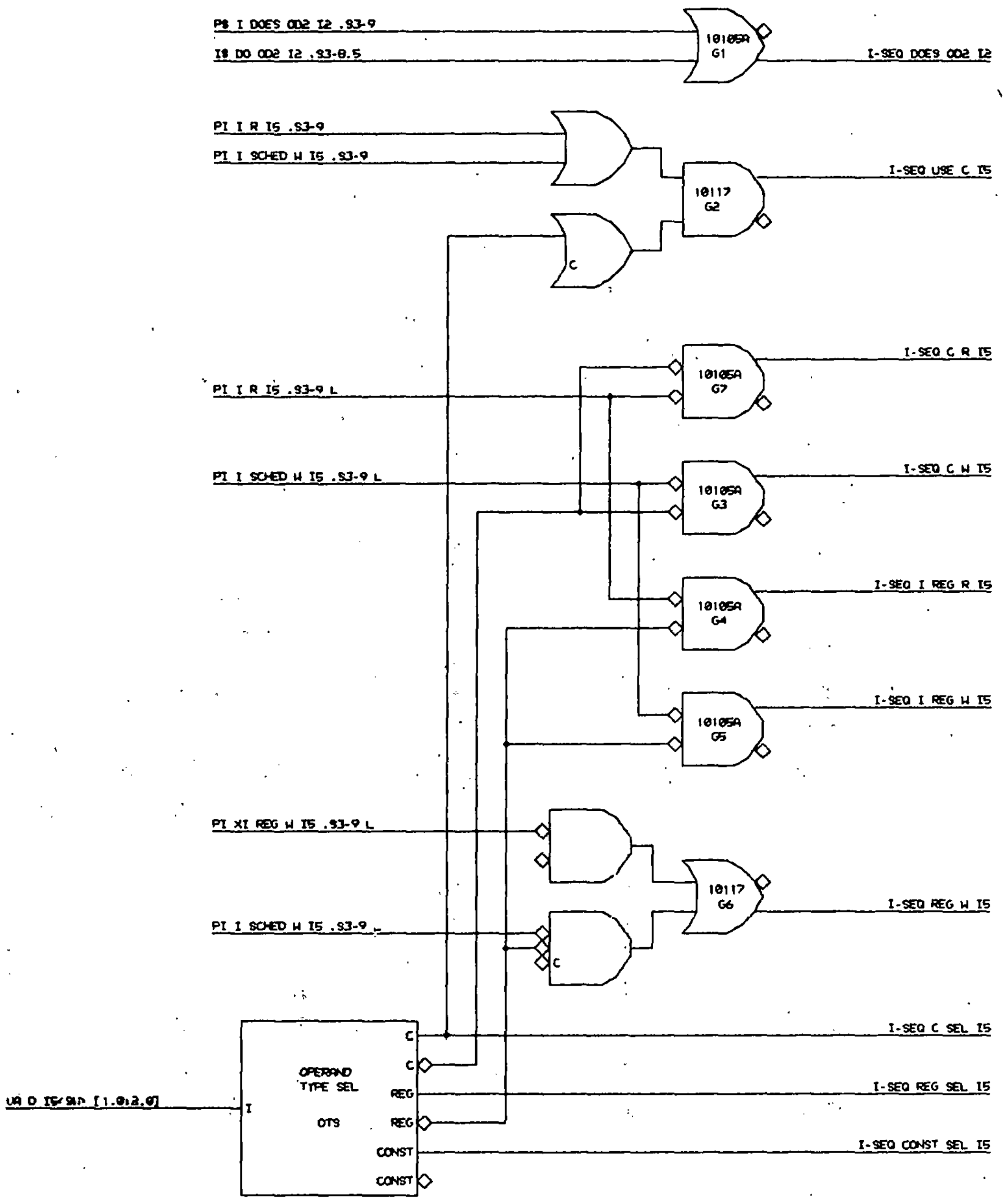

0 : IR I1 .S4-10<OD1.F>
1 : IR I1 .S4-10<OD1.F>
IR I1 .S4-10<OD1.F.1.5> : 00
IR I1 .S4-10<OD1.MODE> : 00
IR I1 .S4-10<OD1.MODE>
5 BIT DECODE 012MORE DC1
7B 10174 M1
RAG OD1 REG ADR I1<0:6> /H
0 : IR I1 .S4-10<OD2.F>
1 : IR I1 .S4-10<OD2.F>
IR I1 .S4-10<OD2.F.1.5> : 00
IR I1 .S4-10<OD2.MODE> : 00
IR I1 .S4-10<OD1.MODE>
5 BIT DECODE 012MORE DC2
7B 10174 M2
RAG OD2 REG ADR I1<0:6> /H
P$ REG R OFFSET SEL I1 .S4-B<1:7>
P-SEQ LOOP INDEX I1 .S2-B<1:7>
7B 10173 M6
P$ REG R OFFSET SEL I1 .S4-B<0>
P$ REG R LEG B SEL I1 .S4-B<1:3>
DR$ S2 PREC I1 .S2-B<0:1>
DR$ S1 PREC I1 .S2-B<0:1>
2B 10158 M8
0 H
L
3B 10173 M7
P$ I DOES OD2 I1 .S4-B
P$ REG R LEG B SEL I1 .S4-B<0>
P$ REG W OFFSET SEL I1 .S4-B<1:7>
P-SEQ LOOP INDEX I1 .S2-B<1:7>
7B 10173 M3
P$ REG W OFFSET SEL I1 .S4-B<0>
P$ REG W LEG B SEL I1 .S4-B<1:3>
0 : DR$ DEST PREC I1 .S2-B<0:1>
3B 10173 M4
P$ REG W LEG B SEL I1 .S4-B<0>
CKA I1 .P5-7 L
34B VIS REG CKE 10141 VR1
CK I2 .P1-3 L 82
I-SEQ USING SA L
0 RAG OD1 REG ADR I2 .S3-9<0:6> /H
1 RAG OD2 REG ADR I2 .S3-9<0:6> /H
2 RAG REG R OFFSET I2 .S3-9<0:6> /H
3 RAG REG R LEG B SEL I2 .S3-9<0:2> /H
4 RAG REG W OFFSET I2 .S3-9<0:6> /H
5 RAG REG W LEG B SEL I2 .S3-9<0:2> /H
RAG OD1 REG ADR I2 .S3-9<0:4> /H
00011 DETECT EN DT0
RAG OD1 REG IS R3 I2 /H
RAG OD2 REG ADR I2 .S3-9<0:4> /H
00011 DETECT EN DT1
RAG OD2 REG IS R3 I2 /H

2.16 ISEQC

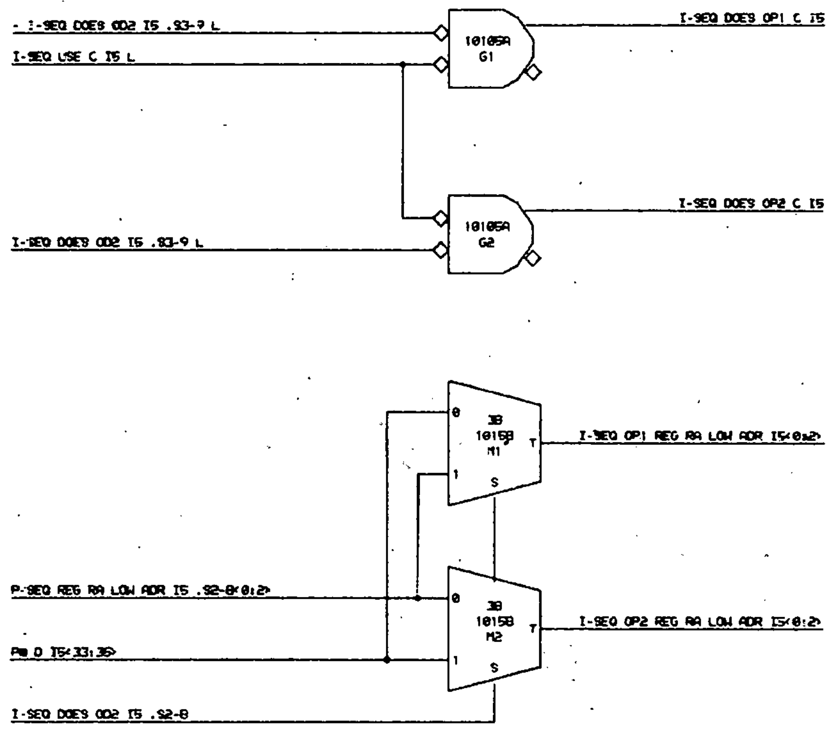
Drawings: ISEQC1, ISEQC2, OPTYPS

This section computes certain I-Sequencer related control signals. It determines whether or not the I-Sequencer will be doing read or write operations for register, cache, or constant operands. It also determines which operand (OP1 or OP2) the I-Sequencer will be concerned with for the current cycle.

Operand type selection is under microcode control. The three principal choices are register, cache, or constant. One additional possibility uses either register or cache depending upon the virtual address supplied. This is useful for emulation of architectures with registers in the address space, like the PDP-10.

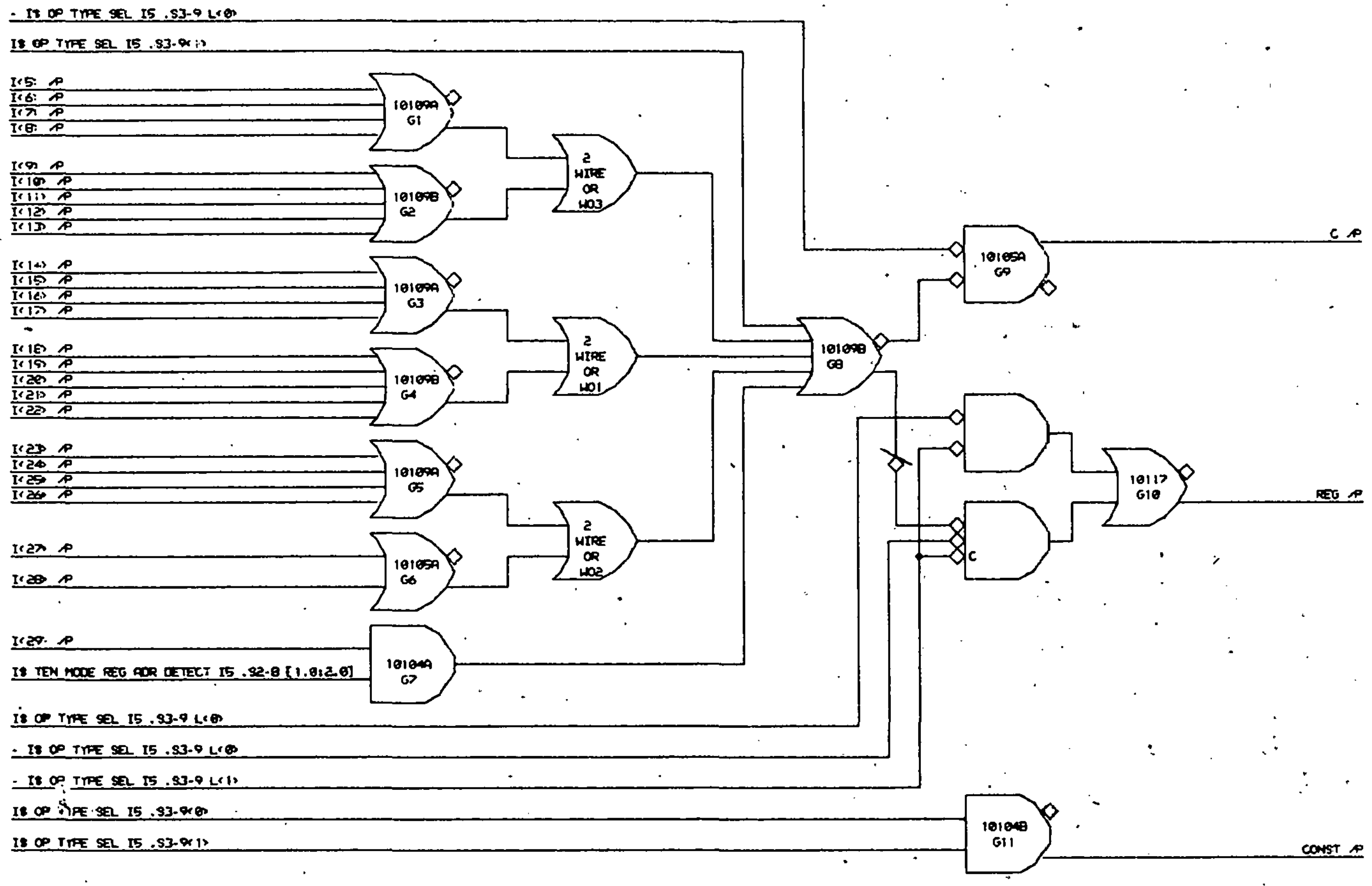


I SEQ CONTROL



- I-SEQ DOES OD2 I5 .S3-9 L
I-SEQ USE C I5 L
I-SEQ DOES OD2 I5 .S3-9 L
10105A
G1
10105A
G2
I-SEQ DOES OP1 C I5
I-SEQ DOES OP2 C I5
P-SEQ REG RA LOW ADR I5 .S2-8<0:2>
P■ D I5<33:35>
I-SEQ DOES OD2 I5 .S2-8
3B
10158
M1
3B
10158
M2
0
1
S
T
I-SEQ OP1 REG RA LOW ADR I5<0:2>
I-SEQ OP2 REG RA LOW ADR I5<0:2>

I-SEQ CONTROL



PARAMETER

I<5:P

C /U
CONST /U
REG /U

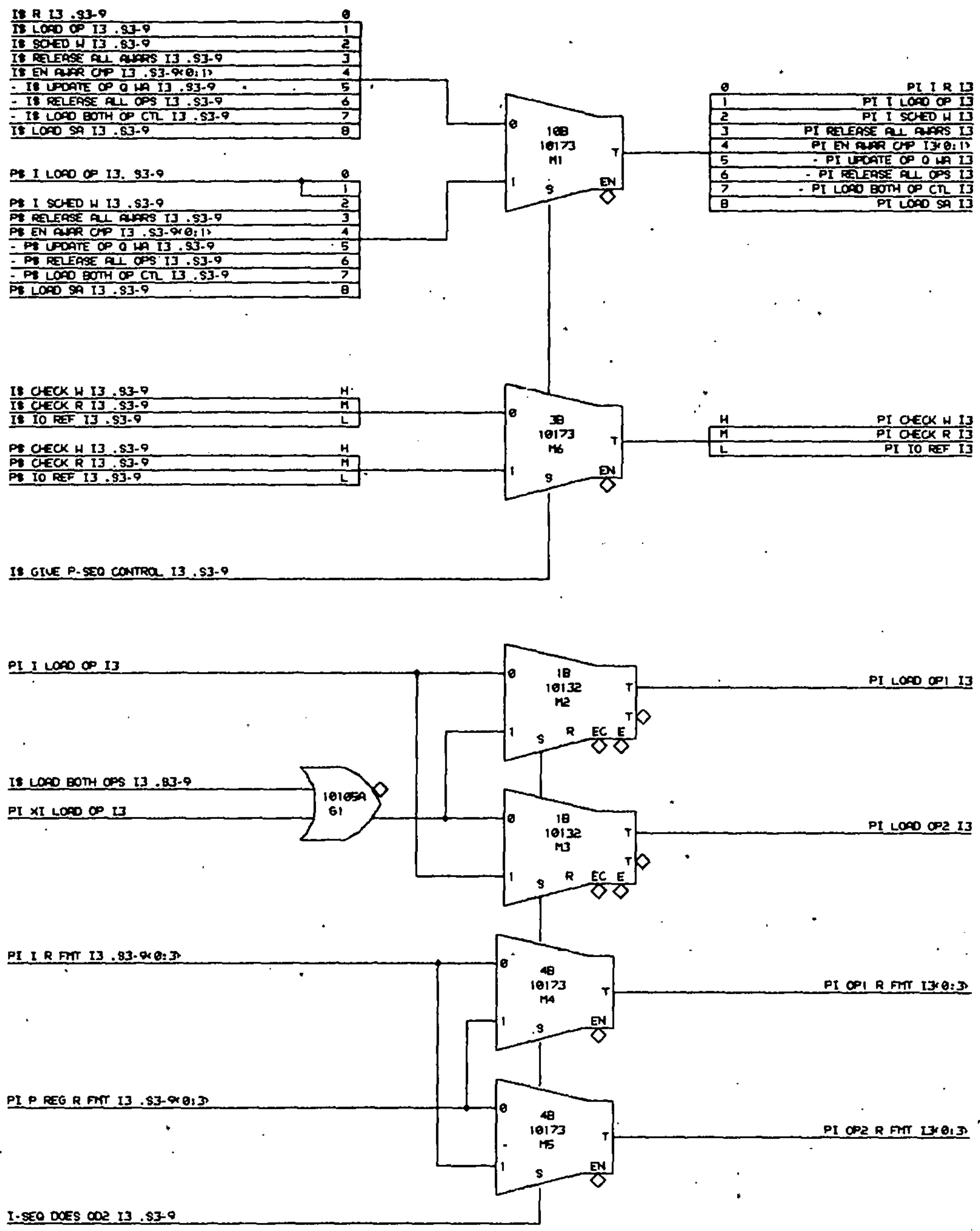
OPERAND TYPE SEL

2.17 PI

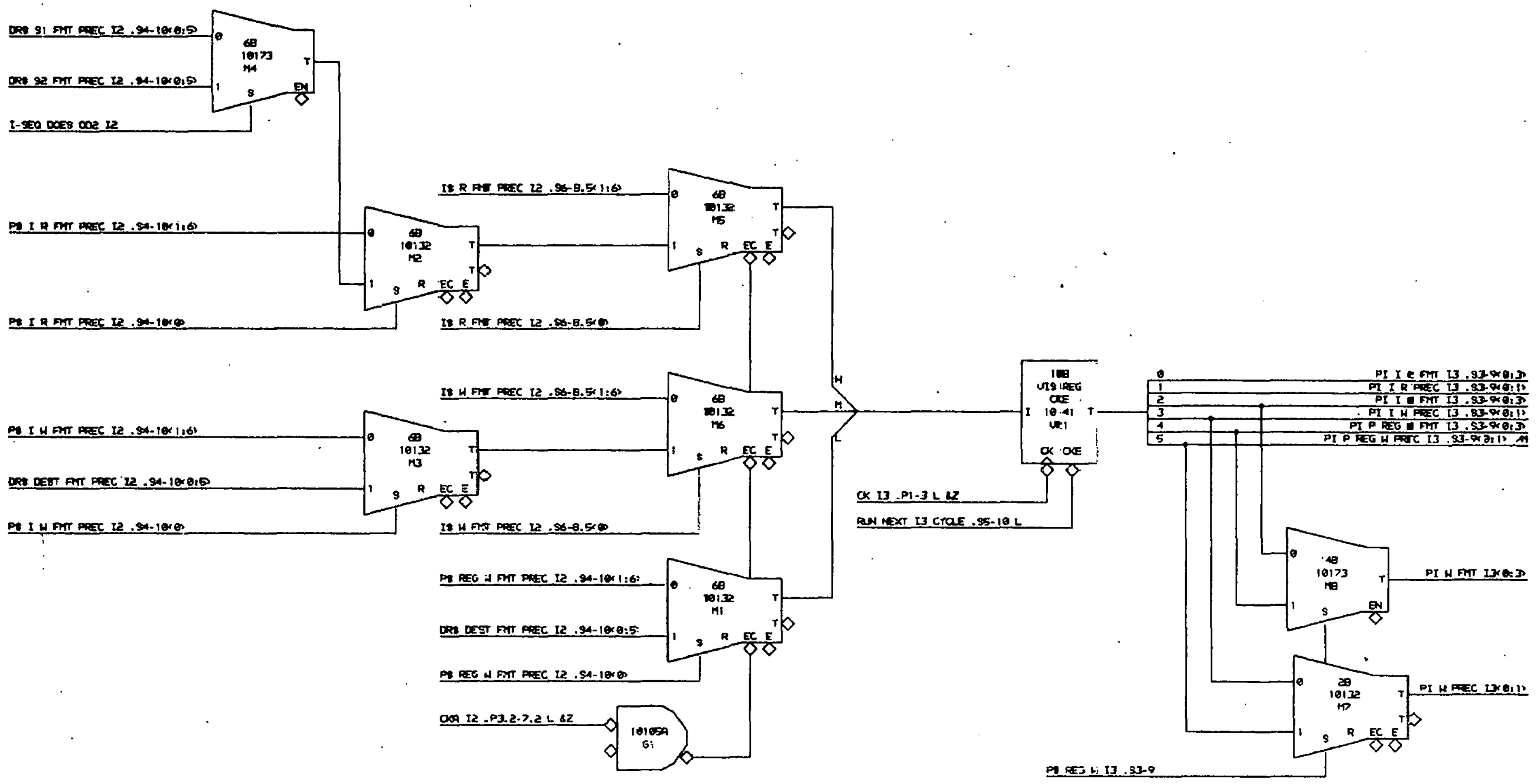
Drawings: PI1, PI2

The PI drawings contain control signals which are generated by combining P- and I-Sequencer generated signals. One class of signals is controlled normally by the I-Sequencer unless it chooses to give control to the P-Sequencer. Control is usually transferred in this way during the I-Sequencer address calculation routines. Since the I-Sequencer does not know during these routines exactly what the P-Sequencer wanted after the address calculation was done, it gives the P-Sequencer control over loading operands into the operand queue and scheduling writes in the write queue.

These drawings are also the site of the generation of the precision and format fields. The precision is a two bit field whose value decodes to quarter (00), half (01), single (10), and double (11). The format is a four bit field indicating the ABOX internal format (see the ABOX description). These format-precision fields are specified for I-Sequencer read and write operations and for P-Sequencer register read and write operations. The I-Sequencer controls the generation for the I-Sequencer read and write format-precisions but may specify that they come from signals generated by the P-Sequencer. The P-Sequencer, in turn, may let the decode RAM be the source of these signals. It may also let the decode RAM generate the register read and write format-precision signals.



PI CONTROL SIGNALS PIPELINE

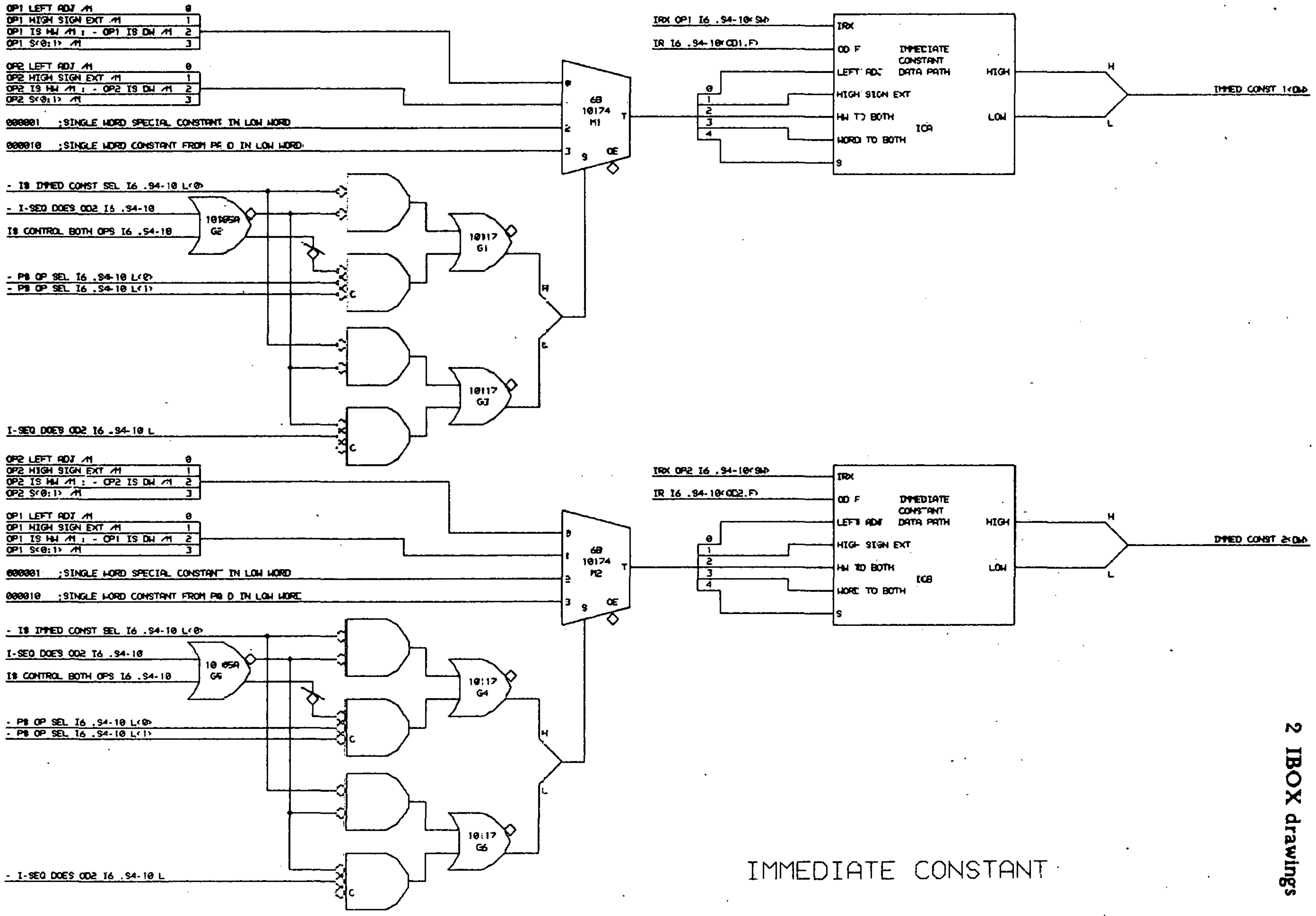

DR0 91 FMT PREC I2 .94-10<0:5>
DR0 92 FMT PREC I2 .94-10<0:5>
I-SEQ DOES OD2 I2
6B 10173 M4
P0 I R FMT PREC I2 .S4-10<1:6>
P0 I R FMT PREC I2 .94-10<0>
6B 10132 M2
I0 R FMT PREC I2 .S6-8.5<1:6>
6B 10132 M5
I0 R FMT PREC I2 .S6-8.5<0>
I0 W FMT PREC I2 .S6-8.5<1:6>
6B 10132 M6
P0 I W FMT PREC I2 .94-10<1:6>
6B 10132 M3
DR0 DEST FMT PREC I2 .94-10<0:6>
P0 I W FMT PREC I2 .S4-10<0>
I0 W FMT PREC I2 .S6-8.5<0>
P0 REG W FMT PREC I2 .94-10<1:6>
6B 10132 M1
DR0 DEST FMT PREC I2 .94-10<0:5>
P0 REG W FMT PREC I2 .S4-10<0>
CKA I2 .P3.2-7.2 L &Z
10105A G1
10B VIS REG CKE 10-41 VR1
CK I3 .P1-3 L &Z
RUN NEXT I3 CYCLE .S5-10 L
PI I R FMT I3 .S3-9<0:3>
PI I R PREC I3 .S3-9<0:1>
PI I W FMT I3 .S3-9<0:3>
PI I W PREC I3 .S3-9<0:1>
PI P REG W FMT I3 .S3-9<0:3>
PI P REG W PREC I3 .S3-9<0:1>
4B 10173 M8
PI W FMT I3<0:3>
2B 10132 M7
PI W PREC I3<0:1>
P0 REG W I3 .S3-9

PI CONTROL SIGNALS PIPELINE

2.18 CONST

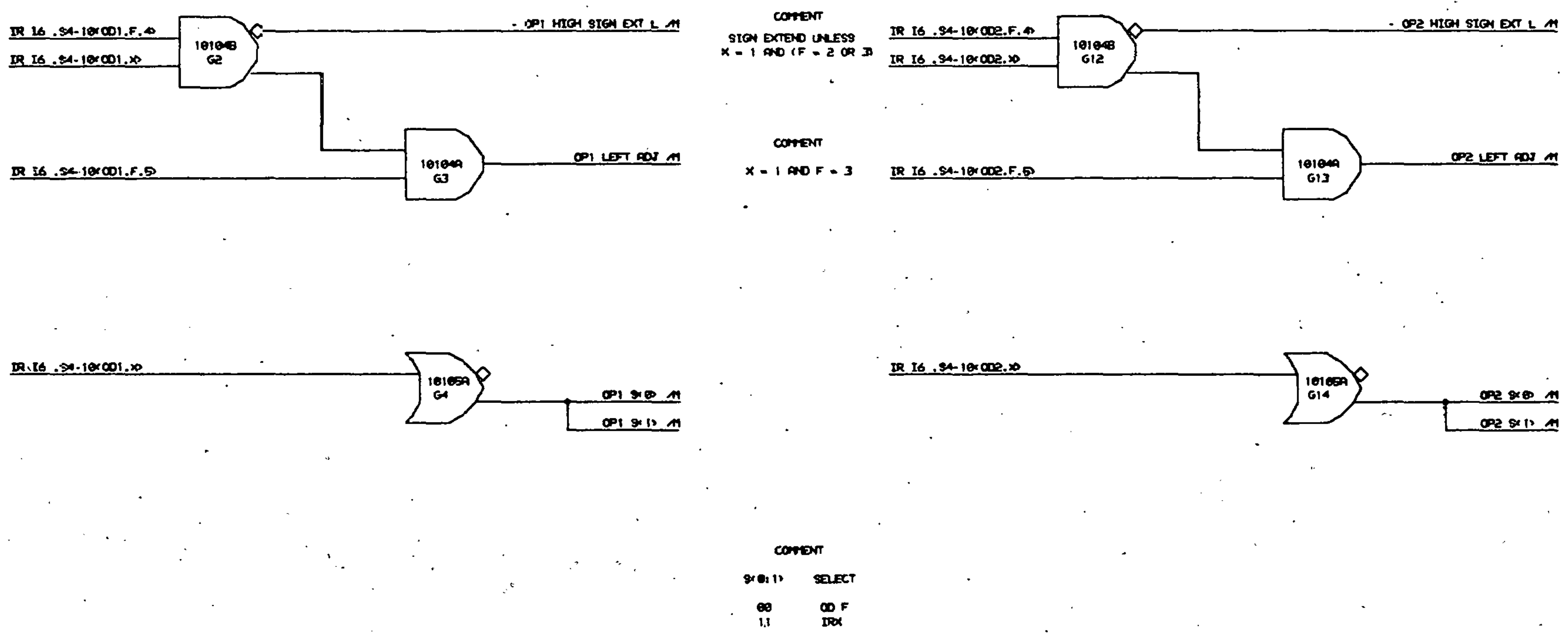
Drawings: IMCON1, IMCON2, IMCDP

These drawings show the data paths and addressing mode decoding for the generation of immediate constants. There are two double word data paths, one for OP1 and one for OP2. The constants may come from the operand descriptors in the case of short constants, from the extended words in the case of long constants, and from the physical address lines in the case of indexed constants. The data paths perform all the left and right adjusting and the sign/zero extending specified by the addressing modes.

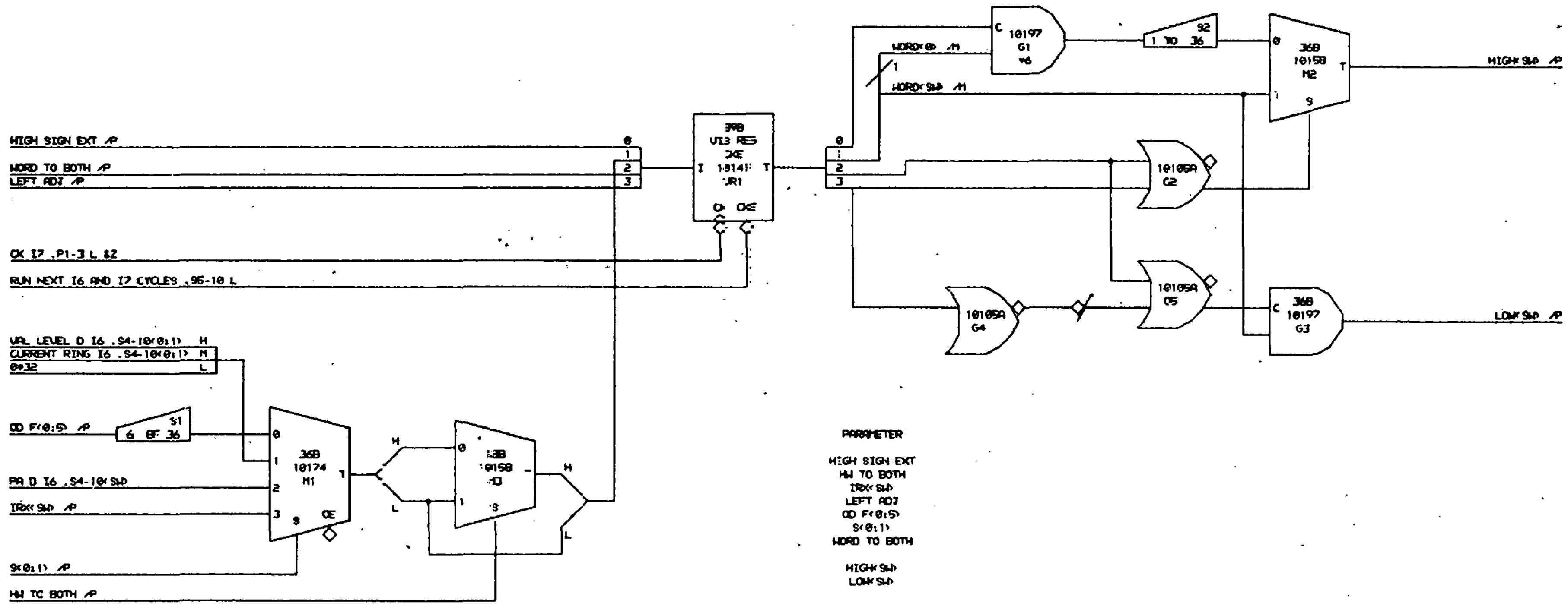


OP1 LEFT ADJ ⁄H
OP1 HIGH SIGN EXT ⁄H
OP1 IS HW ⁄H ; - OP1 IS DW ⁄H
OP1 S<0:1> ⁄H
OP2 LEFT ADJ ⁄H
OP2 HIGH SIGN EXT ⁄H
OP2 IS HW ⁄H ; - OP2 IS DW ⁄H
OP2 S<0:1> ⁄H
000001 ;SINGLE WORD SPECIAL CONSTANT IN LOW WORD
000010 ;SINGLE WORD CONSTANT FROM PC D IN LOW WORD
- I$ IMMED CONST SEL I6 .S4-10 L<0>
- I-SEQ DOES OD2 I6 .S4-10
I$ CONTROL BOTH OPS I6 .S4-10
- P$ OP SEL I6 .S4-10 L<0>
- P$ OP SEL I6 .S4-10 L<1>
I-SEQ DOES OD2 I6 .S4-10 L
10105A G2
10117 G1
10117 G3
6B 10174 M1
OE
IRX OP1 I6 .S4-10<SW>
IR I6 .S4-10<OD1.F>
IRX
OD F
LEFT ADJ
HIGH SIGN EXT
HW TO BOTH
WORD TO BOTH
S
IMMEDIATE CONSTANT DATA PATH
ICA
HIGH
LOW
IMMED CONST 1<DW>
IRX OP2 I6 .S4-10<SW>
IR I6 .S4-10<OD2.F>
6B 10174 M2
ICB
IMMED CONST 2<DW>
10 05A G5
10117 G4
10117 G6
- I-SEQ DOES OD2 I6 .S4-10 L

IMMEDIATE CONSTANT



IMMEDIATE CONSTANT



IMMEDIATE CONSTANT DATA PATH

2.19 AWAR

Drawings: WQ1, WQ2, WQ3, WQ4, WQ5, WQ6, WQ7, CCAM1, CCAM2, REGCAM, XRCAM, REGCER, REGCEW, XRCMP, ABFCMP, SNEM1, SNEM2

The ABOX write address register (AWAR) drawings are somewhat elaborate. They implement the write queue, which is a 16 entry queue of addresses at which results will be written by the ABOX.

Results are first scheduled by the IBOX to be written later by the ABOX. The IBOX must first check that it is legal to write into the addressed locations. Then it makes sure for a memory destination that the locations are present in the cache. These operations are no different than read operations (except for the type of access that is checked) up to this point. However, instead of reading the locations from the cache and writing them into the operand queue, a new entry is used in the write queue which remembers the addresses of the locations to be written, whether they are to be written in the cache or in the registers, and where exactly in the cache they are to be written. The IBOX depends on the fact that a location will remain in the cache until the ABOX generates the result. When a result is actually generated by the ABOX, the next entry of the write queue is read to determine where the result should be written.

There is another queue, built with content addressable memories (CAMs), which is written in parallel with the write queue. The CAMs store the address to which the result will be written later. All of a CAM's entries are compared in parallel to the address of an operand that is being read. Any of the entries matching indicates that the cache or register file does not have the correct value of that operand; rather, the ABOX has yet to write the latest version of that location. It clearly would be incorrect to let such an operand be read before the correct value was generated. There are two possibilities in this case. One is to wait for the location to be written before delivering any more operands. The other is to tell the ABOX somehow that the operand that is being delivered should be ignored and that the true value is one of the last several results that the ABOX has produced. There are certain (common) restricted cases in which this latter option can be exercised.

The I-Sequencer control over the scheduling of writes into the write queue is somewhat similar to the operand queue control but is not as complex. The I-Sequencer can schedule register or cache write operations, release all the unreleased write operations, and enable comparisons on the unreleased write operations. The state information of the write queue is contained in the following signals: WQ ENTRIES USED<0:4> is a count of the number of write operations that have been scheduled (released or not), UNRELEASED COUNT-1<0:4> indicates one less than the number of those used entries that have not yet been released, and CMP UNRELEASED ADRS indicates whether or not comparisons are enabled for the unreleased entries. In addition, there is a current read address and a current write address. Being somewhat more restrictive than the operand queue, the write queue allows only one location to be used up at a time. Also, as in the operand queue, the clocks must be stopped if a request to schedule a write would cause the write queue to overflow.

Most of the complexity of the write queue is in the CAMs. The CAMs are split into three pieces: the index register CAMs, the register address CAMs and the cache address CAMs. All of the CAMs are addressed in parallel for writing and, in fact, all of the CAMs are written at the same time. Any given write is either to the cache or to the registers, so either the index and register

CAMs will have a valid entry written or the cache CAMs will.

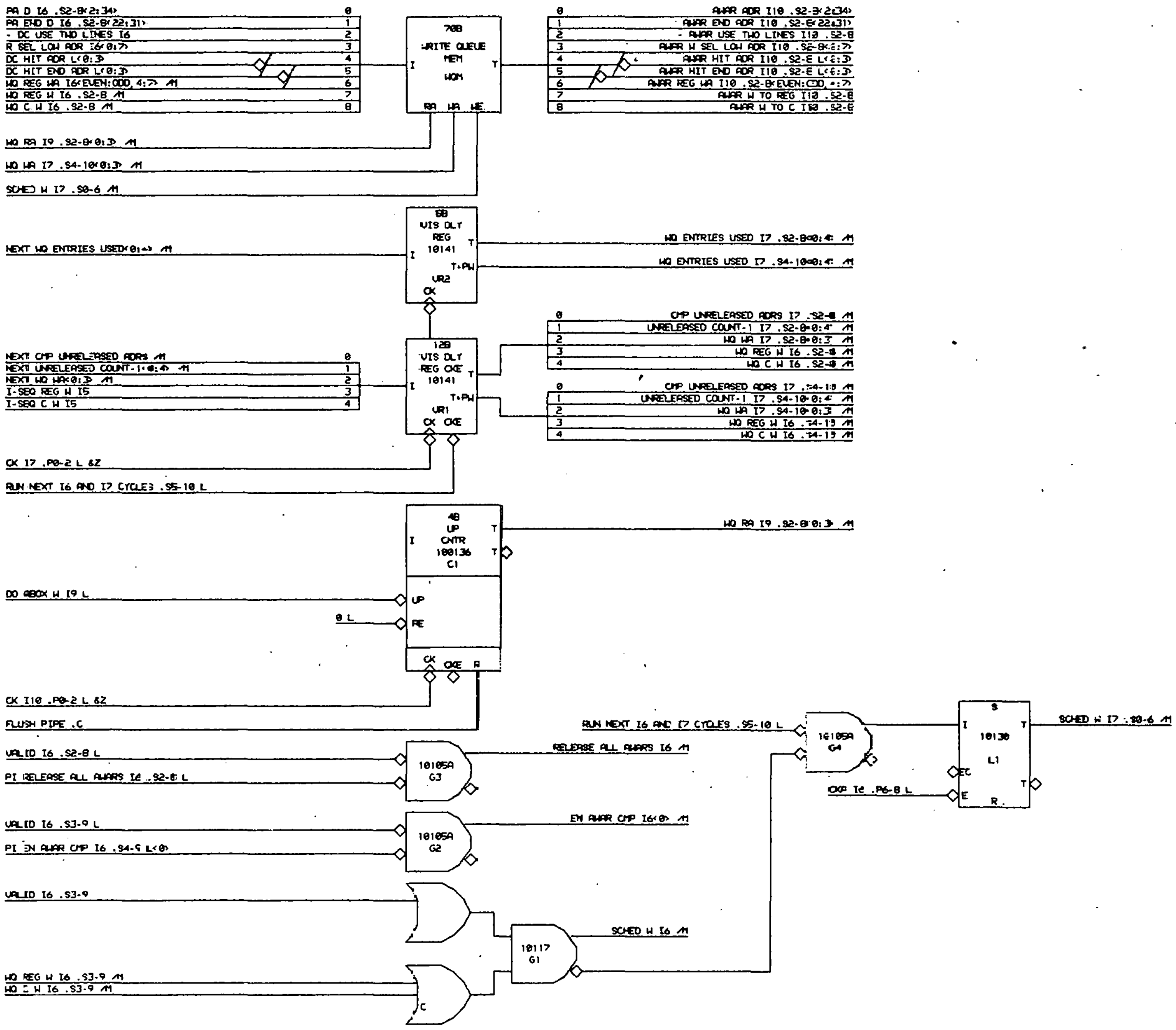
There are two sets of register CAMs, one for OP1 and one for OP2. This is necessary because it is possible to read registers on both operands in one cycle. The read addresses supplied to the CAMs are the appropriate register read addresses for OP1 and OP2. These addresses are double word addresses and more information must be passed in order to determine exactly which registers are being read. Within a double word there are eight quarter words and so an eight bit mask is generated indicating exactly which quarter words are being read. A similar mask is generated for writing. These masks are produced based upon the low order register address bits and the precision.

There are also two sets of index register CAMs, corresponding to the two sets of index registers that can be accessed simultaneously. The outputs of the index register CAMs are used somewhat differently than those of the cache and register CAMs. In the latter case, the question of operand wraparound must be decided. This is not possible for the index registers; if there is an address calculation that needs to index by a value that has been scheduled to be written, then the indexing *must* wait until that value becomes available. It is therefore necessary to stop the pipeline at the I3 stage where indexing occurs if there are any index register CAM matches.

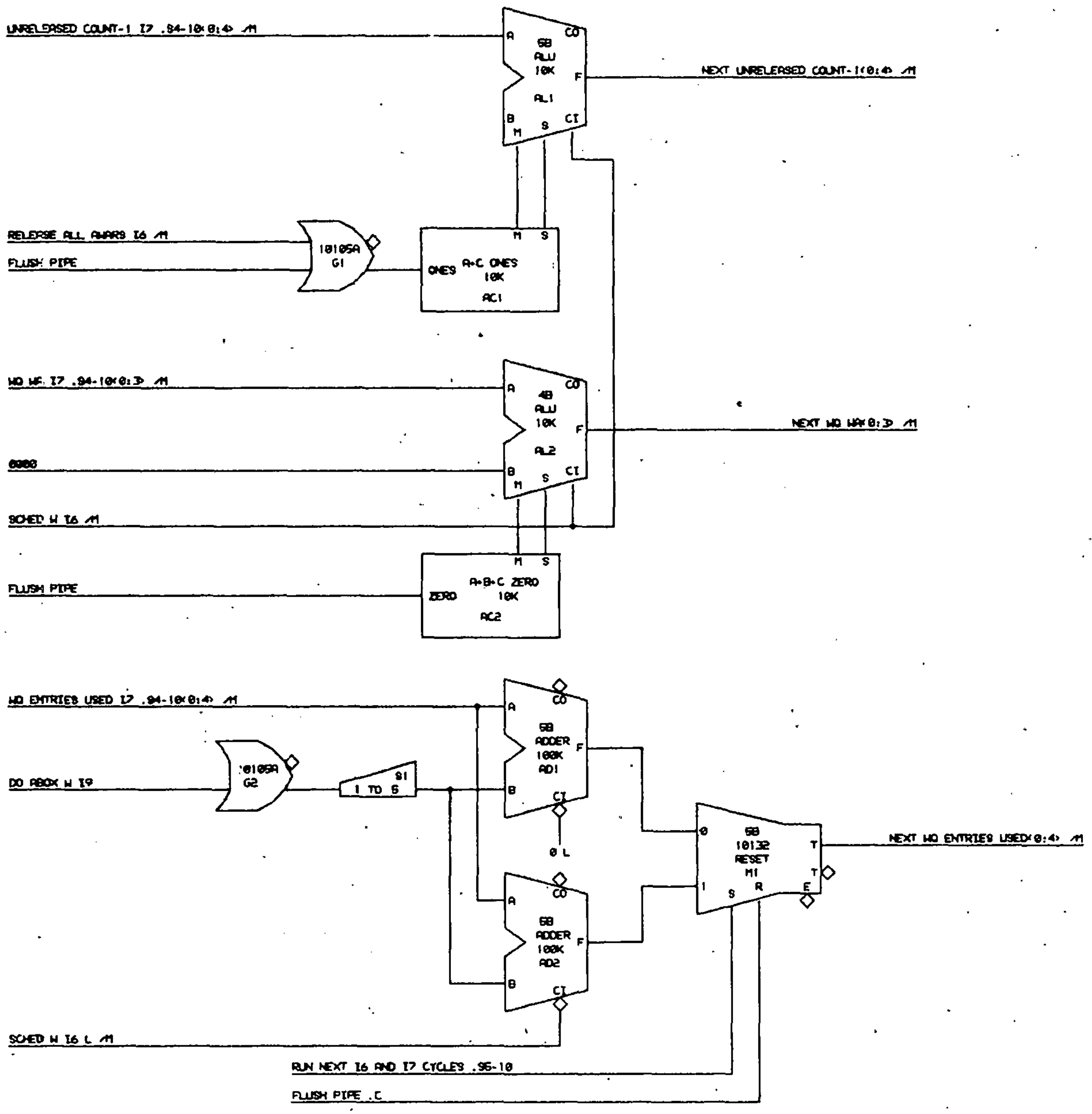
The cache CAMs are the most complicated of all. Memory locations are broken down into quad word blocks aligned on quad word boundaries. The quad word blocks are then further classified as even and odd quad word blocks. Any given cache write (which may be a vector write) will occupy at most a quad word. However, it need not be aligned on a quad word boundary. In any event, it will overlap at most two quad word blocks; an even and an odd one. A pair of CAMs stores the high order address bits indicating which even quad word block and which odd quad word block are being occupied. Corresponding to these CAMs are another pair of CAMs which store 16 bit vectors indicating which quarter words within the even and odd quad word are being read. The first set of CAMs mentioned produces high order address match signals and the second set of CAMs produces overlap signals. A final set of CAMs stores low order address bits, enable bits and format bits. If the format bits do not match, then it is not possible to ask the ABOX to wrap around its results and use them as operands. If the low order address bits and the high order address bits and the format bits match and the enable bits are set, then there is an exact match. When there is an exact match, results may be wrapped around within the ABOX. There is still a problem, however. There may be an exact match (allowing results to be wrapped) but there may be a more recent match that is not exact. That is, there will be a match of the quad word addresses and there will be some overlap but the full addresses don't match. Since this partial match corresponds to a more recent write, it is not correct to wrap around the older value. An exact implementation, then, would wrap around only if there is no more recent partial match. The Mark IIA implementation, however, will wrap only if there are no partial matches at all. This will be only slightly suboptimal as it will be very rare to have partial matches in the first place. If there are no matches (exact or partial) then the operands may be read from the cache. If there are partial matches then the clocks must be stopped. This entire discussion applies equally well to the OP1 and OP2 register CAMs except that double words are used instead of quad words.

Once it is determined that wraparound can occur, it is still necessary to tell the ABOX which result it should select for the operand. This is accomplished by first selecting the appropriate register or cache exact-match lines for OP1 and for OP2 and then rotating them to put them in order by time of scheduling. These outputs are then priority encoded to find the relative distance to the most recently scheduled write that matches. These values are then sent to the operand queue to be forwarded to the ABOX.

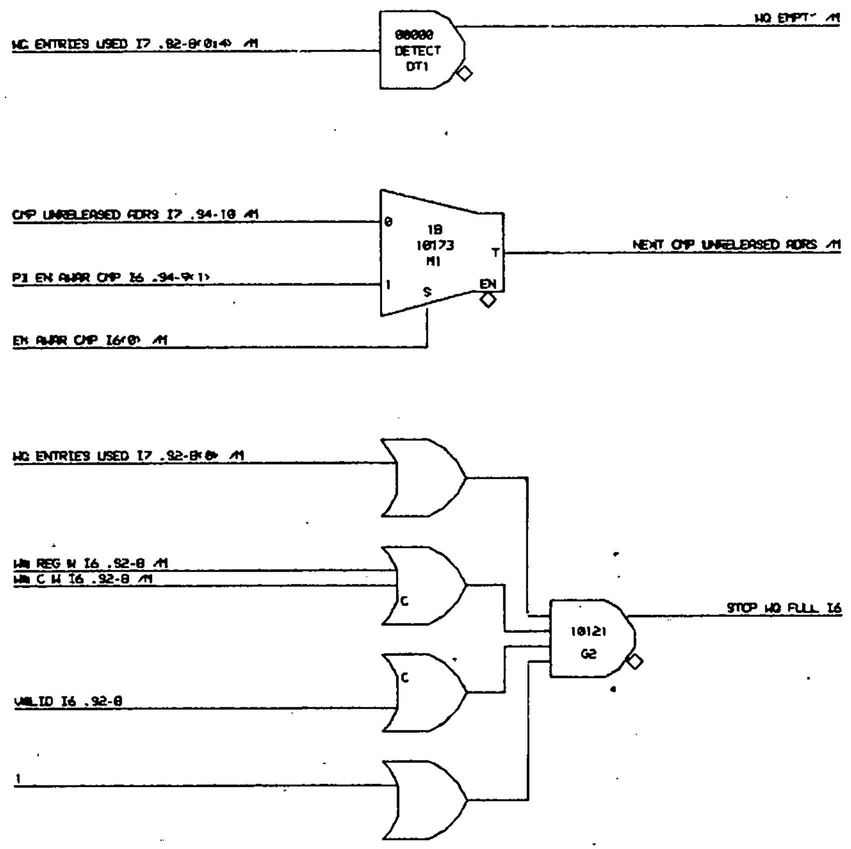
WRITE QUEUE



PA D I6 .S2-8<2:34>
PA END D I6 .S2-8<22:31>
- DC USE TWO LINES I6
R SEL LOW ADR I6<0:7>
DC HIT ADR L<0:3>
DC HIT END ADR L<0:3>
WQ REG WA I6<EVEN:ODD, 4:7> /H
WQ REG W I6 .S2-8 /H
WQ C W I6 .S2-8 /H
70B
WRITE QUEUE
MEM
WQM
RA WA WE
AWAR ADR I10 .S2-8<2:34>
AWAR END ADR I10 .S2-8<22:31>
- AWAR USE TWO LINES I10 .S2-8
AWAR W SEL LOW ADR I10 .S2-8<2:7>
AWAR HIT ADR I10 .S2-8 L<0:3>
AWAR HIT END ADR I10 .S2-8 L<0:3>
AWAR REG WA I10 .S2-8<EVEN:ODD, 4:7>
AWAR W TO REG I10 .S2-8
AWAR W TO C I10 .S2-8
WQ RA I9 .S2-8<0:3> /H
WQ WA I7 .S4-10<0:3> /H
SCHED W I7 .S0-6 /H
5B
VIS DLY
REG
10141
VR2
NEXT WQ ENTRIES USED<0:4> /H
WQ ENTRIES USED I7 .S2-8<0:4> /H
WQ ENTRIES USED I7 .S4-10<0:4> /H
12B
VIS DLY
REG CKE
10141
VR1
NEXT CMP UNRELEASED ADRS /H
NEXT UNRELEASED COUNT-1<0:4> /H
NEXT WQ WA<0:3> /H
I-SEQ REG W I5
I-SEQ C W I5
CMP UNRELEASED ADRS I7 .S2-8 /H
UNRELEASED COUNT-1 I7 .S2-8<0:4> /H
WQ WA I7 .S2-8<0:3> /H
WQ REG W I6 .S2-8 /H
WQ C W I6 .S2-8 /H
CMP UNRELEASED ADRS I7 .S4-10 /H
UNRELEASED COUNT-1 I7 .S4-10<0:4> /H
WQ WA I7 .S4-10<0:3> /H
WQ REG W I6 .S4-10 /H
WQ C W I6 .S4-10 /H
CK I7 .P0-2 L &Z
RUN NEXT I6 AND I7 CYCLES .S5-10 L
4B
UP
CNTR
100136
C1
WQ RA I9 .S2-8<0:3> /H
DO ABOX W I9 L
0 L
UP
RE
CK CKE R
CK I10 .P0-2 L &Z
FLUSH PIPE .C
VALID I6 .S2-8 L
PI RELEASE ALL AWARS I6 .S2-8 L
10105A
G3
RELEASE ALL AWARS I6 /H
VALID I6 .S3-9 L
PI EN AWAR CMP I6 .S4-9 L<0>
10105A
G2
EN AWAR CMP I6<0> /H
VALID I6 .S3-9
WQ REG W I6 .S3-9 /H
WQ C W I6 .S3-9 /H
10117
G1
SCHED W I6 /H
RUN NEXT I6 AND I7 CYCLES .S5-10 L
10105A
G4
CKP I6 .P6-8 L
10130
L1
SCHED W I7 .S0-6 /H

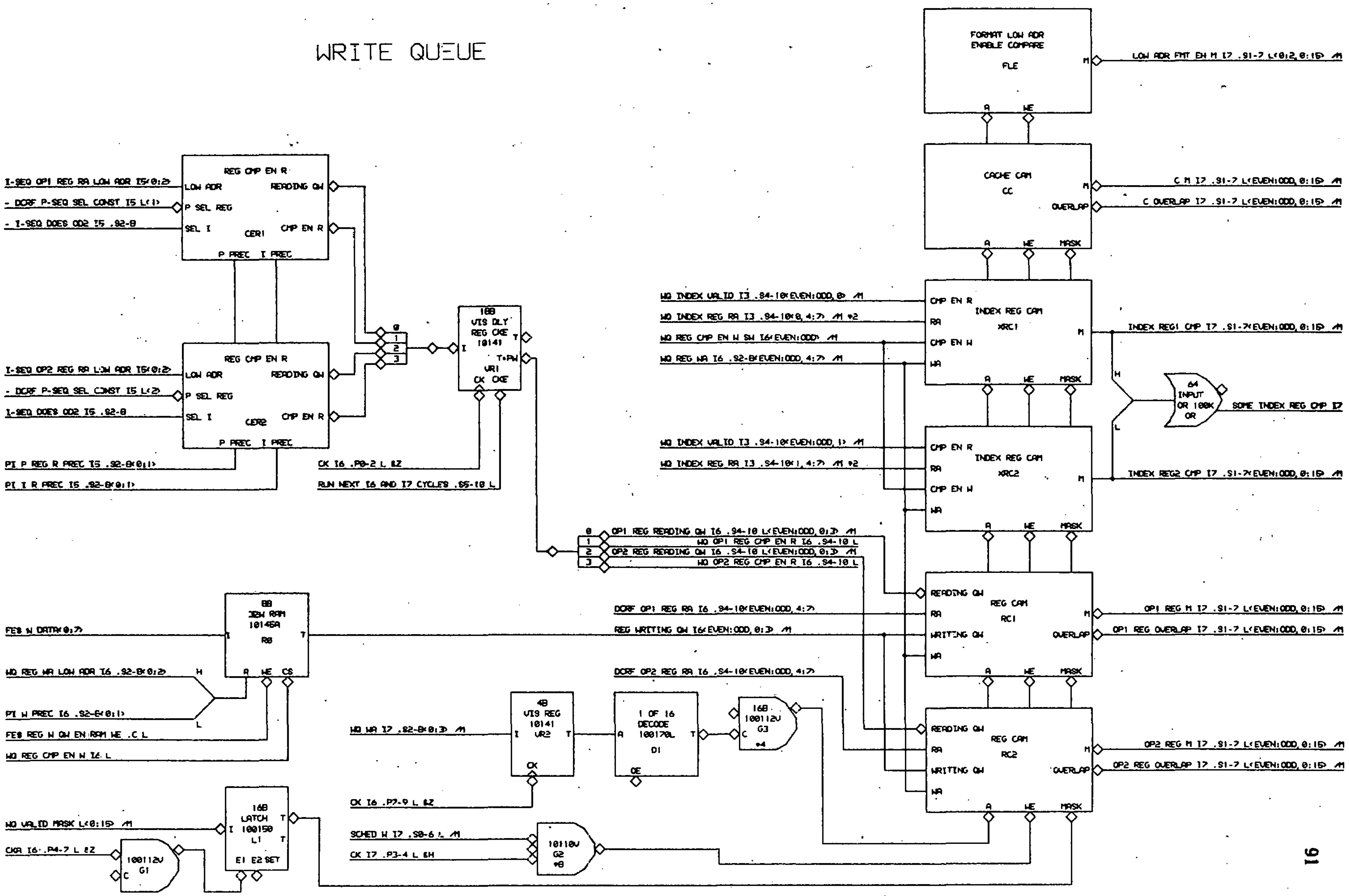

UNRELEASED COUNT-1 I7 .94-10<0:4> /M
A
CO
5B
ALU
10K
F
AL1
B
M
S
CI
NEXT UNRELEASED COUNT-1<0:4> /M
RELEASE ALL AWARS I6 /M
FLUSH PIPE
10105A
G1
M
S
ONES
A+C ONES
10K
AC1
WQ WA I7 .94-10<0:3> /M
A
CO
4B
ALU
10K
F
AL2
0000
B
M
S
CI
NEXT WQ WA<0:3> /M
SCHED W I6 /M
M
S
FLUSH PIPE
ZERO
A+B+C ZERO
10K
AC2
WQ ENTRIES USED I7 .94-10<0:4> /M
A
CO
5B
ADDER
100K
AD1
F
DO ABOX W I9
10105A
G2
S1
1 TO 5
B
CI
0 L
A
CO
5B
ADDER
100K
AD2
F
B
CI
0
5B
10132
RESET
M1
T
T
1
S
R
E
NEXT WQ ENTRIES USED<0:4> /M
SCHED W I6 L /M
RUN NEXT I6 AND I7 CYCLES .95-10
FLUSH PIPE .C

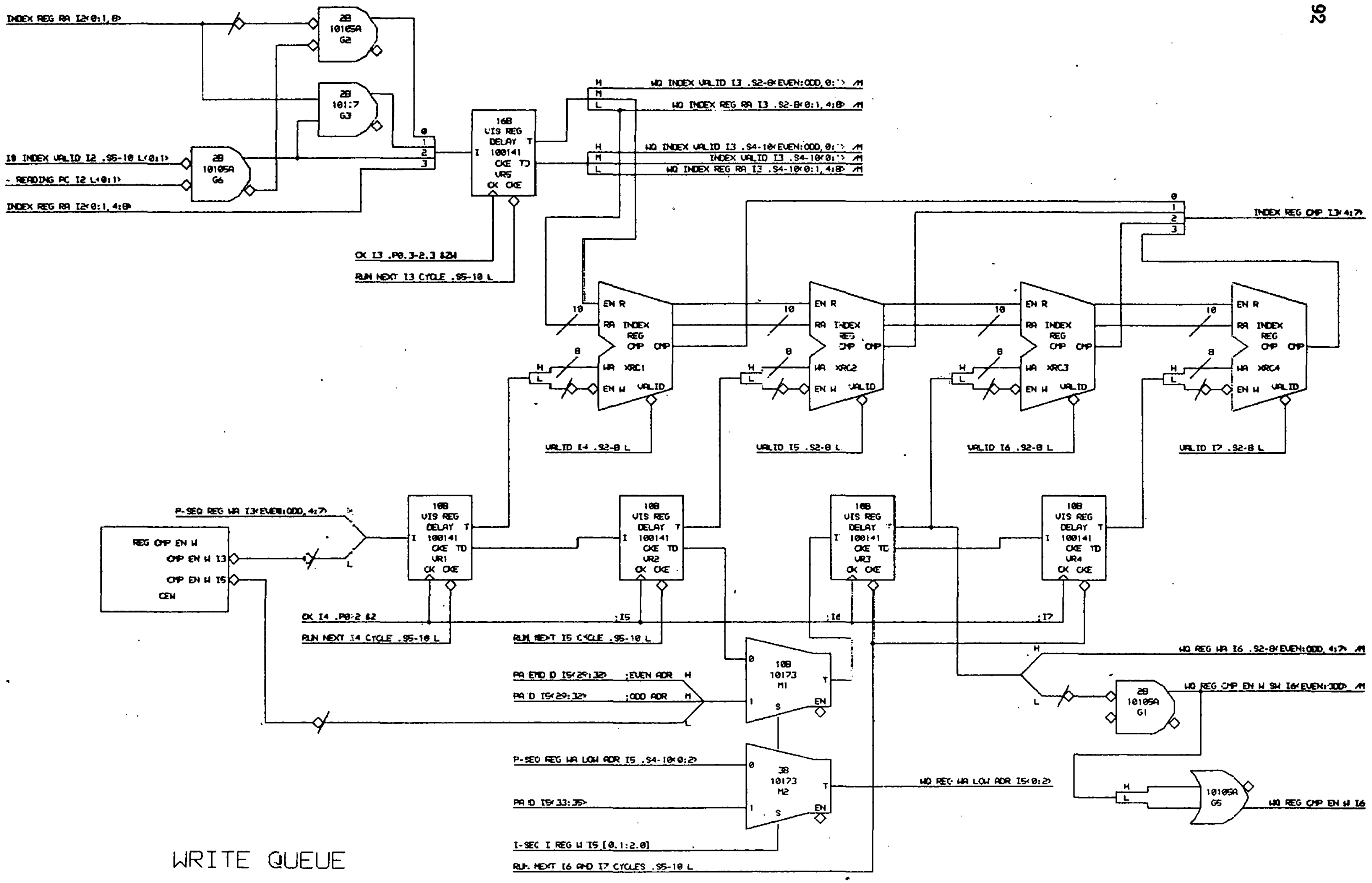
WRITE QUEUE


WQ ENTRIES USED I7 .92-8<0:4> /M
00000 DETECT DT1
WQ EMPTY /M
CMP UNRELEASED ADRS I7 .94-10 /M
PI EN AWAR CMP I6 .94-9<1>
EN AWAR CMP I6<0> /M
0
1
S
1B 10173 M1
T
EN
NEXT CMP UNRELEASED ADRS /M
WQ ENTRIES USED I7 .92-8<0> /M
WM REG W I6 .92-8 /M
WM C W I6 .92-8 /M
C
VALID I6 .92-8
C
1
10121 G2
STOP WQ FULL I6

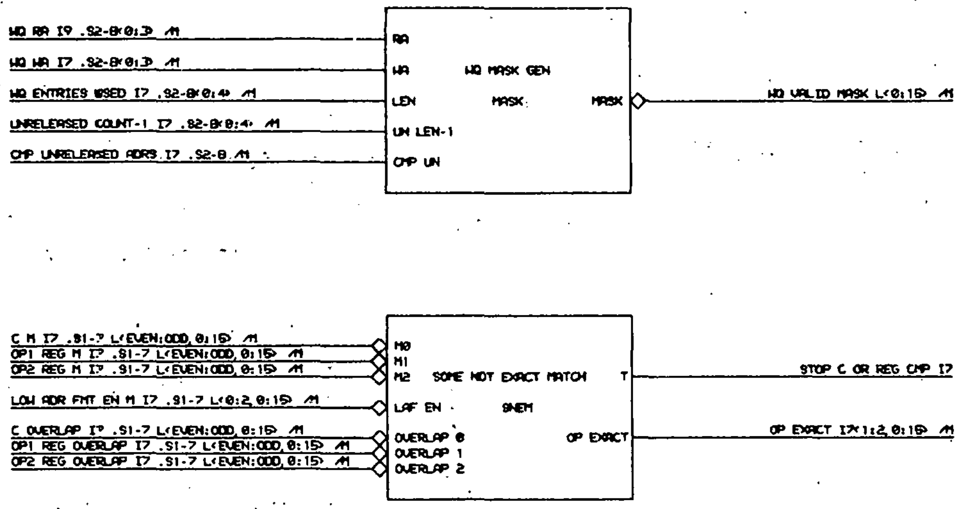
WRITE QUEUE

WRITE QUEUE

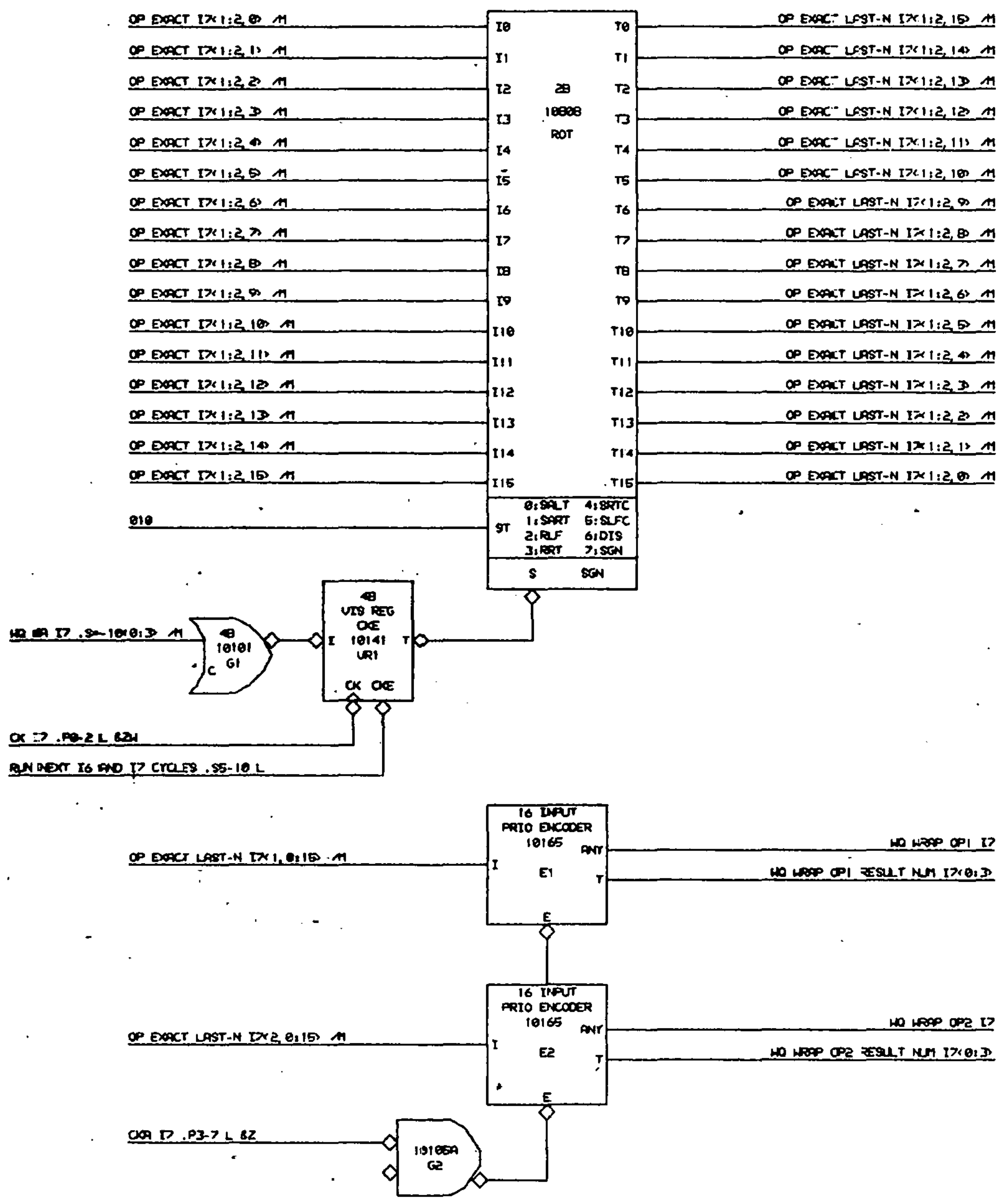

FORMAT LOW ADR ENABLE COMPARE
FLE
LOW ADR FMT EN M I7 .S1-7 L<0:2,0:15> /M
CACHE CAM
CC
C M I7 .S1-7 L<EVEN:ODD,0:15> /M
C OVERLAP I7 .S1-7 L<EVEN:ODD,0:15> /M
REG CMP EN R
I-SEQ OP1 REG RA LOW ADR I5<0:2>
- DCRF P-SEQ SEL CONST I5 L<1>
- I-SEQ DOES OD2 I5 .S2-8
LOW ADR
P SEL REG
SEL I
READING QW
CMP EN R
CER1
P PREC I PREC
I-SEQ OP2 REG RA LOW ADR I5<0:2>
- DCRF P-SEQ SEL CONST I5 L<2>
I-SEQ DOES OD2 I5 .S2-8
CER2
PI P REG R PREC I5 .S2-8<0:1>
PI I R PREC I5 .S2-8<0:1>
16B VIS DLY REG CKE 10141
VR1
T+PW
CK CKE
CK I6 .P0-2 L &Z
RUN NEXT I6 AND I7 CYCLES .S5-10 L
WQ INDEX VALID I3 .S4-10<EVEN:ODD,0> /M
WQ INDEX REG RA I3 .S4-10<0,4:7> /M *2
WQ REG CMP EN W SW I6<EVEN:ODD> /M
WQ REG WA I6 .S2-8<EVEN:ODD,4:7> /M
INDEX REG CAM
XRC1
CMP EN R
RA
CMP EN W
WA
A WE MASK
INDEX REG1 CMP I7 .S1-7<EVEN:ODD,0:15> /M
64 INPUT OR 100K OR
SOME INDEX REG CMP I7
WQ INDEX VALID I3 .S4-10<EVEN:ODD,1> /M
WQ INDEX REG RA I3 .S4-10<1,4:7> /M *2
XRC2
INDEX REG2 CMP I7 .S1-7<EVEN:ODD,0:15> /M
OP1 REG READING QW I6 .S4-10 L<EVEN:ODD,0:3> /M
WQ OP1 REG CMP EN R I6 .S4-10 L
OP2 REG READING QW I6 .S4-10 L<EVEN:ODD,0:3> /M
WQ OP2 REG CMP EN R I6 .S4-10 L
DCRF OP1 REG RA I6 .S4-10<EVEN:ODD,4:7>
REG WRITING QW I6<EVEN:ODD,0:3> /M
REG CAM
RC1
READING QW
WRITING QW
OVERLAP
OP1 REG M I7 .S1-7 L<EVEN:ODD,0:15> /M
OP1 REG OVERLAP I7 .S1-7 L<EVEN:ODD,0:15> /M
DCRF OP2 REG RA I6 .S4-10<EVEN:ODD,4:7>
RC2
OP2 REG M I7 .S1-7 L<EVEN:ODD,0:15> /M
OP2 REG OVERLAP I7 .S1-7 L<EVEN:ODD,0:15> /M
8B 32W RAM 10145A
RB
A WE CS
FE$ W DATA<0:7>
WQ REG WA LOW ADR I6 .S2-8<0:2>
PI W PREC I6 .S2-8<0:1>
FE$ REG W QW EN RAM WE .C L
WQ REG CMP EN W I6 L
4B VIS REG 10141
VR2
WQ WA I7 .S2-8<0:3> /M
CK I6 .P7-9 L &Z
1 OF 16 DECODE 100170L
D1
OE
16B 100112V
G3
*4
SCHED W I7 .S0-6 L /M
CK I7 .P3-4 L &H
10110V
G2
*8
16B LATCH 100150
L1
E1 E2 SET
WQ VALID MASK L<0:15> /M
CKA I6 .P4-7 L &Z
100112V
G1


INDEX REG RA I2<0:1,8>
I0 INDEX VALID I2 .S5-10 L<0:1>
- READING PC I2 L<0:1>
INDEX REG RA I2<0:1,4:8>
2B 10105A G2
2B 101:7 G3
2B 10105A G6
16B VIS REG DELAY 100141 VR5
CK I3 .P0.3-2.3 &2H
RUN NEXT I3 CYCLE .S5-10 L
WQ INDEX VALID I3 .S2-8<EVEN:ODD,0:1> /H
WQ INDEX REG RA I3 .S2-8<0:1,4:8> /H
WQ INDEX VALID I3 .S4-10<EVEN:ODD,0:1> /H
INDEX VALID I3 .S4-10<0:1> /H
WQ INDEX REG RA I3 .S4-10<0:1,4:8> /H
INDEX REG CMP I3<4:7>
INDEX REG CMP XRC1
INDEX REG CMP XRC2
INDEX REG CMP XRC3
INDEX REG CMP XRC4
VALID I4 .S2-8 L
VALID I5 .S2-8 L
VALID I6 .S2-8 L
VALID I7 .S2-8 L
P-SEQ REG WA I3<EVEN:ODD,4:7>
REG CMP EN W
CMP EN W I3
CMP EN W I5
CEW
10B VIS REG DELAY 100141 VR1
10B VIS REG DELAY 100141 VR2
10B VIS REG DELAY 100141 VR3
10B VIS REG DELAY 100141 VR4
CK I4 .P0-2 &2
;I5
;I6
;I7
RUN NEXT I4 CYCLE .S5-10 L
RUN NEXT I5 CYCLE .S5-10 L
PA END D I5<29:32> ;EVEN ADR H
PA D I5<29:32> ;ODD ADR H
10B 10173 M1
P-SEQ REG WA LOW ADR I5 .S4-10<0:2>
PA D I5<33:35>
3B 10173 M2
I-SEC I REG W I5 [0.1:2.0]
RUN NEXT I6 AND I7 CYCLES .S5-10 L
WQ REG WA LOW ADR I5<0:2>
WQ REG WA I6 .S2-8<EVEN:ODD,4:7> /H
WQ REG CMP EN W SW I6<EVEN:ODD> /H
2B 10105A G1
10105A G5
WQ REG CMP EN W I6

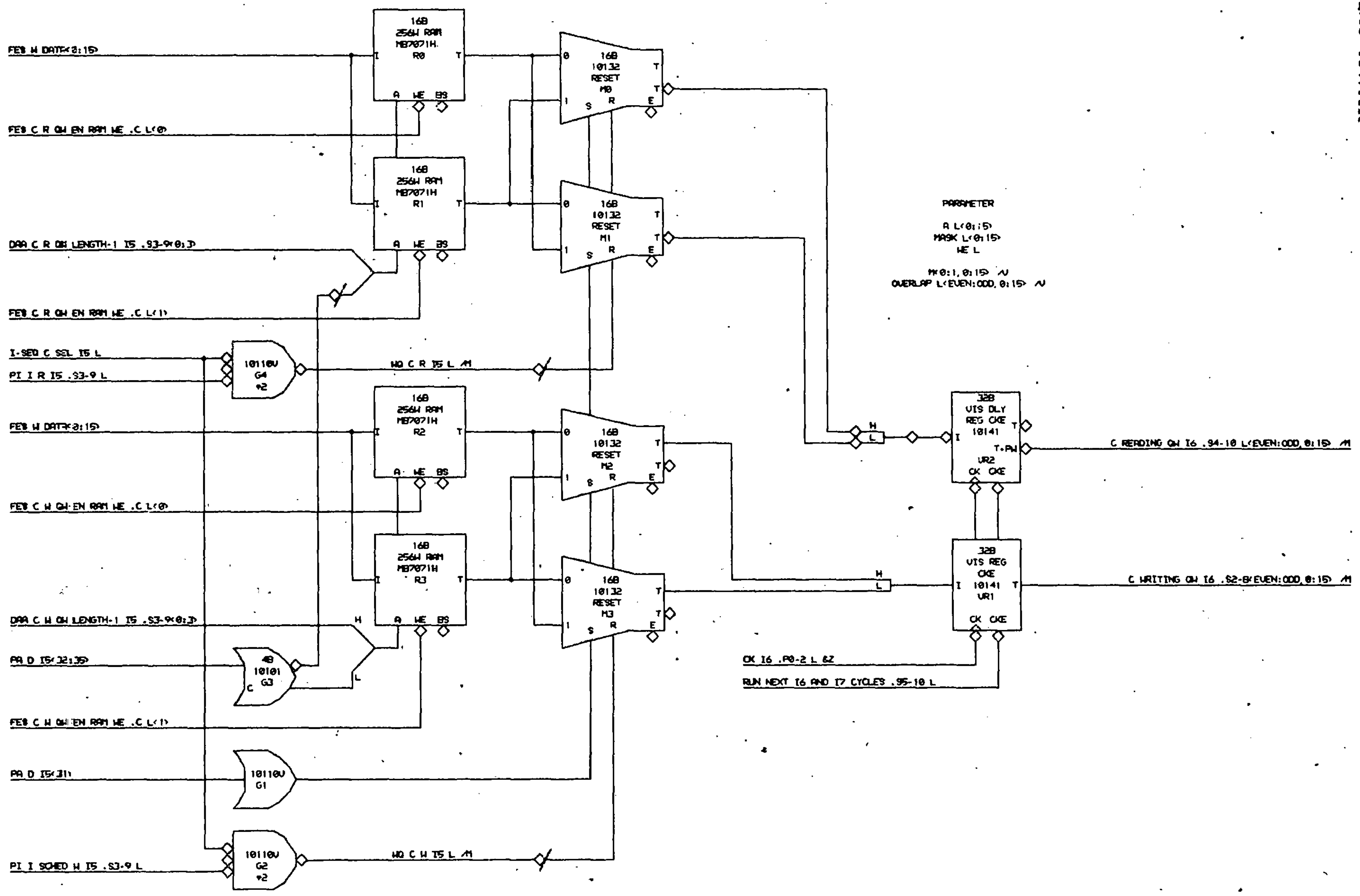
WRITE QUEUE


WQ RA I9 .S2-8<0:3> /M
WQ WA I7 .S2-8<0:3> /M
WQ ENTRIES USED I7 .S2-8<0:4> /M
UNRELEASED COUNT-1 I7 .S2-8<0:4> /M
CMP UNRELEASED ADRS I7 .S2-8 /M
RA
WA
LEN
UN LEN-1
CMP UN
WQ MASK GEN
MASK
MASK
WQ VALID MASK L<0:15> /M
C M I7 .S1-7 L<EVEN:ODD,0:15> /M
OP1 REG M I7 .S1-7 L<EVEN:ODD,0:15> /M
OP2 REG M I7 .S1-7 L<EVEN:ODD,0:15> /M
LOW ADR FMT EN M I7 .S1-7 L<0:2,0:15> /M
C OVERLAP I7 .S1-7 L<EVEN:ODD,0:15> /M
OP1 REG OVERLAP I7 .S1-7 L<EVEN:ODD,0:15> /M
OP2 REG OVERLAP I7 .S1-7 L<EVEN:ODD,0:15> /M
M0
M1
M2
SOME NOT EXACT MATCH
LAF EN
SNEM
OVERLAP 0
OVERLAP 1
OVERLAP 2
T
OP EXACT
STOP C OR REG CMP I7
OP EXACT I7<1:2,0:15> /M

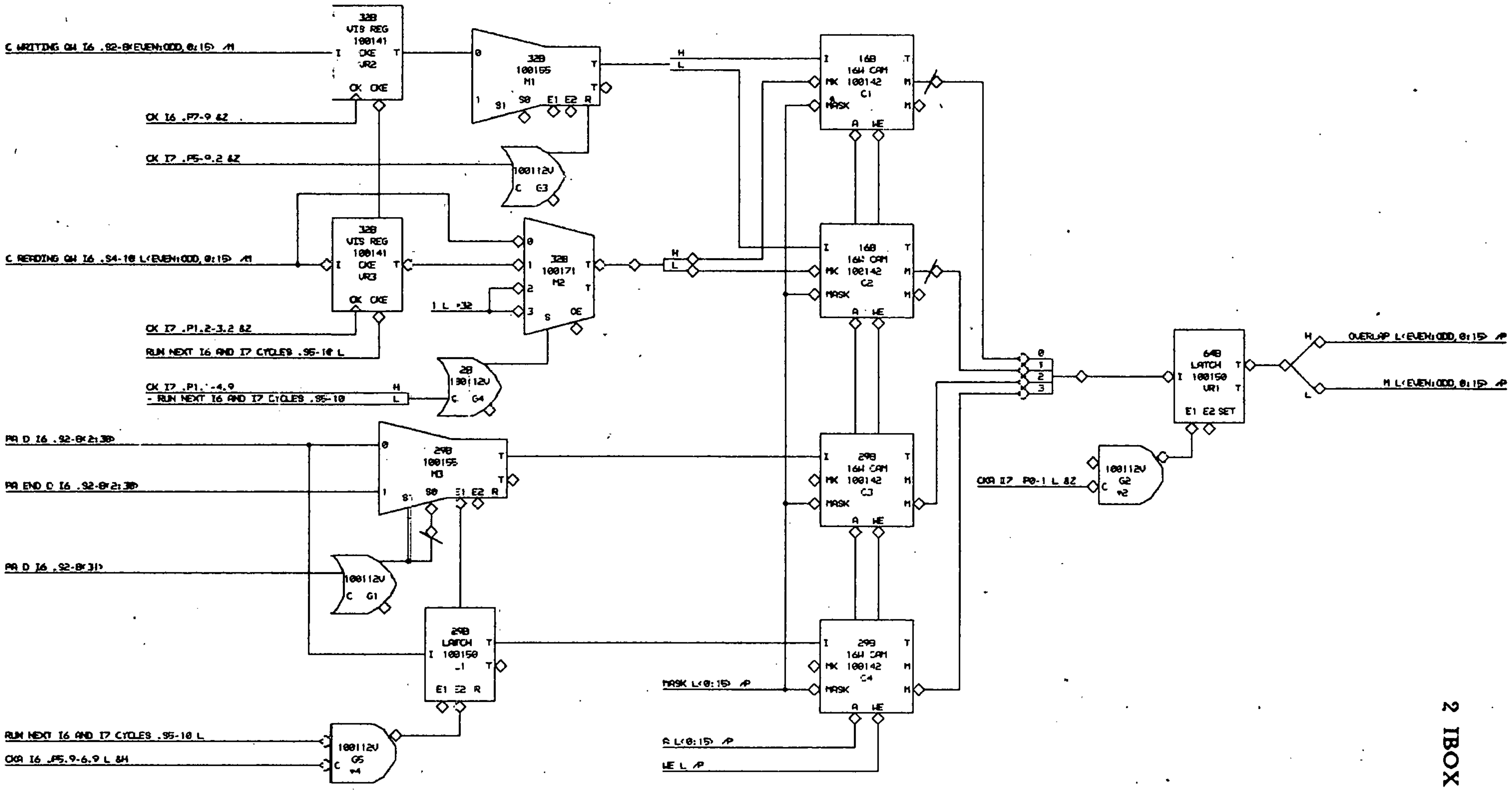
WRITE QUEUE


OP EXACT I7<1:2,0> /M
OP EXACT I7<1:2,1> /M
OP EXACT I7<1:2,2> /M
OP EXACT I7<1:2,3> /M
OP EXACT I7<1:2,4> /M
OP EXACT I7<1:2,5> /M
OP EXACT I7<1:2,6> /M
OP EXACT I7<1:2,7> /M
OP EXACT I7<1:2,8> /M
OP EXACT I7<1:2,9> /M
OP EXACT I7<1:2,10> /M
OP EXACT I7<1:2,11> /M
OP EXACT I7<1:2,12> /M
OP EXACT I7<1:2,13> /M
OP EXACT I7<1:2,14> /M
OP EXACT I7<1:2,15> /M
010
I0 I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11 I12 I13 I14 I15
2B
10808
ROT
T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15
ST
0:SALT 4:SRTC
1:SART 5:SLFC
2:RLF 6:DIS
3:RRT 7:SGN
S SGN
OP EXACT LAST-N I7<1:2,15> /M
OP EXACT LAST-N I7<1:2,14> /M
OP EXACT LAST-N I7<1:2,13> /M
OP EXACT LAST-N I7<1:2,12> /M
OP EXACT LAST-N I7<1:2,11> /M
OP EXACT LAST-N I7<1:2,10> /M
OP EXACT LAST-N I7<1:2,9> /M
OP EXACT LAST-N I7<1:2,8> /M
OP EXACT LAST-N I7<1:2,7> /M
OP EXACT LAST-N I7<1:2,6> /M
OP EXACT LAST-N I7<1:2,5> /M
OP EXACT LAST-N I7<1:2,4> /M
OP EXACT LAST-N I7<1:2,3> /M
OP EXACT LAST-N I7<1:2,2> /M
OP EXACT LAST-N I7<1:2,1> /M
OP EXACT LAST-N I7<1:2,0> /M
WQ WA I7 .S--10<0:3> /M
4B
10101
G1
C
4B
VIS REG
CKE
10141
UR1
I T
CK CKE
CK I7 .P0-2 L 62H
RUN NEXT I6 AND I7 CYCLES .S5-10 L
16 INPUT
PRIO ENCODER
10165
ANY
I
E1
T
E
OP EXACT LAST-N I7<1,0:15> /M
WQ WRAP OP1 I7
WQ WRAP OP1 RESULT NUM I7<0:3>
16 INPUT
PRIO ENCODER
10165
ANY
I
E2
T
E
OP EXACT LAST-N I7<2,0:15> /M
WQ WRAP OP2 I7
WQ WRAP OP2 RESULT NUM I7<0:3>
CKA I7 .P3-7 L 6Z
10106A
G2

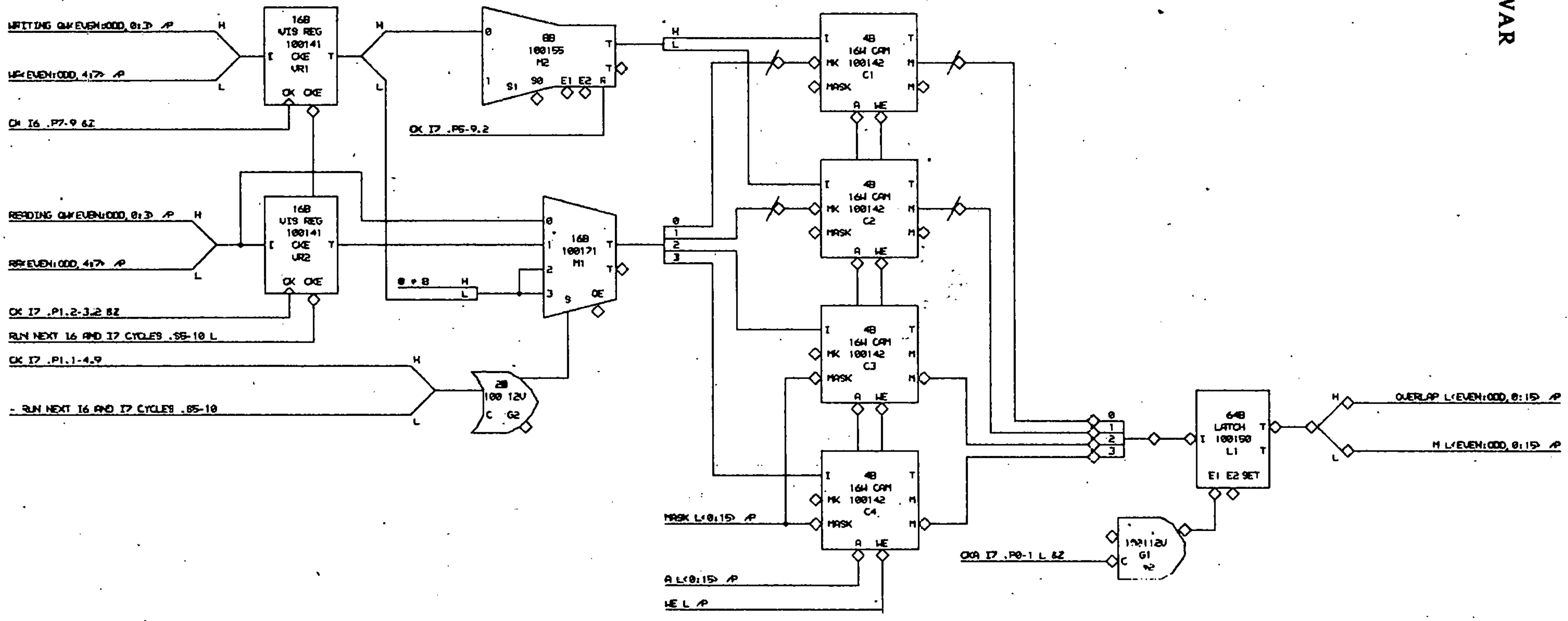
WRITE QUEUE


FE$ W DATA<0:15>
FE$ C R QW EN RAM WE .C L<0>
DAA C R QW LENGTH-1 I5 .S3-9<0:3>
FE$ C R QW EN RAM WE .C L<1>
I-SEQ C SEL I5 L
PI I R I5 .S3-9 L
FE$ W DATA<0:15>
FE$ C W QW EN RAM WE .C L<0>
DAA C W QW LENGTH-1 I5 .S3-9<0:3>
PA D I5<32:35>
FE$ C W QW EN RAM WE .C L<1>
PA D I5<31>
PI I SCHED W I5 .S3-9 L
16B 256W RAM MB7071H R0
16B 256W RAM MB7071H R1
16B 256W RAM MB7071H R2
16B 256W RAM MB7071H R3
A WE BS
I T
16B 10132 RESET M0
16B 10132 RESET M1
16B 10132 RESET M2
16B 10132 RESET M3
0 1 S R T E
10110V G4 ÷2
4B 10101 G3 C
10110V G1
10110V G2 ÷2
H L
WQ C R I5 L /M
WQ C W I5 L /M
PARAMETER
A L<0:15>
MASK L<0:15>
WE L
M<0:1, 0:15> /V
OVERLAP L<EVEN:ODD, 0:15> /V
32B VIS DLY REG CKE 10141 VR2
T T+PW CK CKE
32B VIS REG CKE 10141 VR1
C READING QW I6 .S4-10 L<EVEN:ODD, 0:15> /M
C WRITING QW I6 .S2-8<EVEN:ODD, 0:15> /M
CK I6 .P0-2 L &Z
RUN NEXT I6 AND I7 CYCLES .S5-10 L

CACHE CAM


C WRITING QW I6 .S2-8(EVEN:ODD,0:15) /M
CK I6 .P7-9 &Z
CK I7 .P5-9.2 &Z
C READING QW I6 .S4-10 L(EVEN:ODD,0:15) /M
CK I7 .P1.2-3.2 &Z
RUN NEXT I6 AND I7 CYCLES .S5-10 L
CK I7 .P1.-4.9
- RUN NEXT I6 AND I7 CYCLES .S5-10
PA D I6 .S2-8(2:30)
PA END D I6 .S2-8(2:30)
PA D I6 .S2-8(31)
RUN NEXT I6 AND I7 CYCLES .S5-10 L
CKA I6 .P5.9-6.9 L &H
32B VIS REG 100141 VR2
32B VIS REG 100141 VR3
32B 100155 M1
32B 100171 M2
29B 100155 M3
29B LATCH 100150 L1
100112V C G3
2B 100112V G4
100112V C G1
100112V G5
1 L +32
16B 16W CAM 100142 C1
16B 16W CAM 100142 C2
29B 16W CAM 100142 C3
29B 16W CAM 100142 C4
MASK L(0:15) /P
A L(0:15) /P
WE L /P
64B LATCH 100150 VR1
100112V G2
CKA I7 P0-1 L &Z
OVERLAP L(EVEN:ODD,0:15) /P
M L(EVEN:ODD,0:15) /P

CACHE CAM

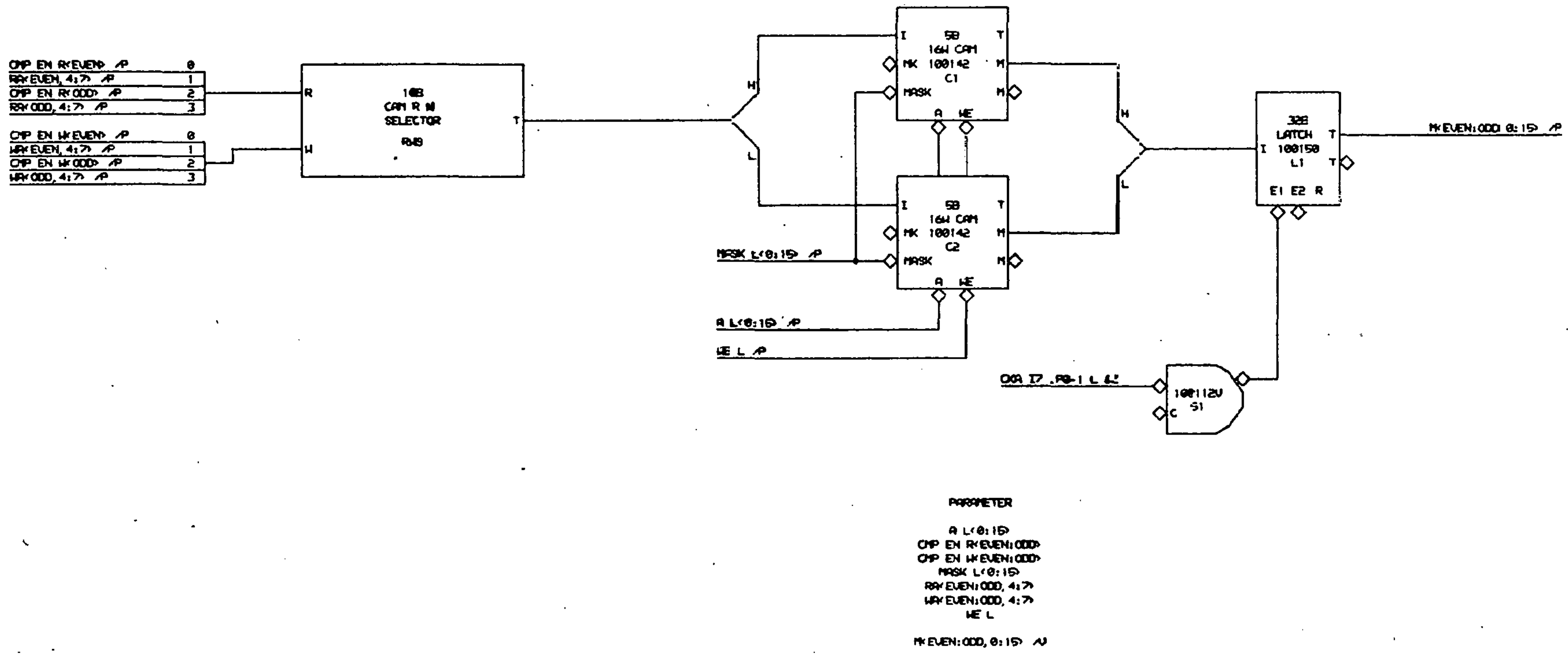


PARAMETER

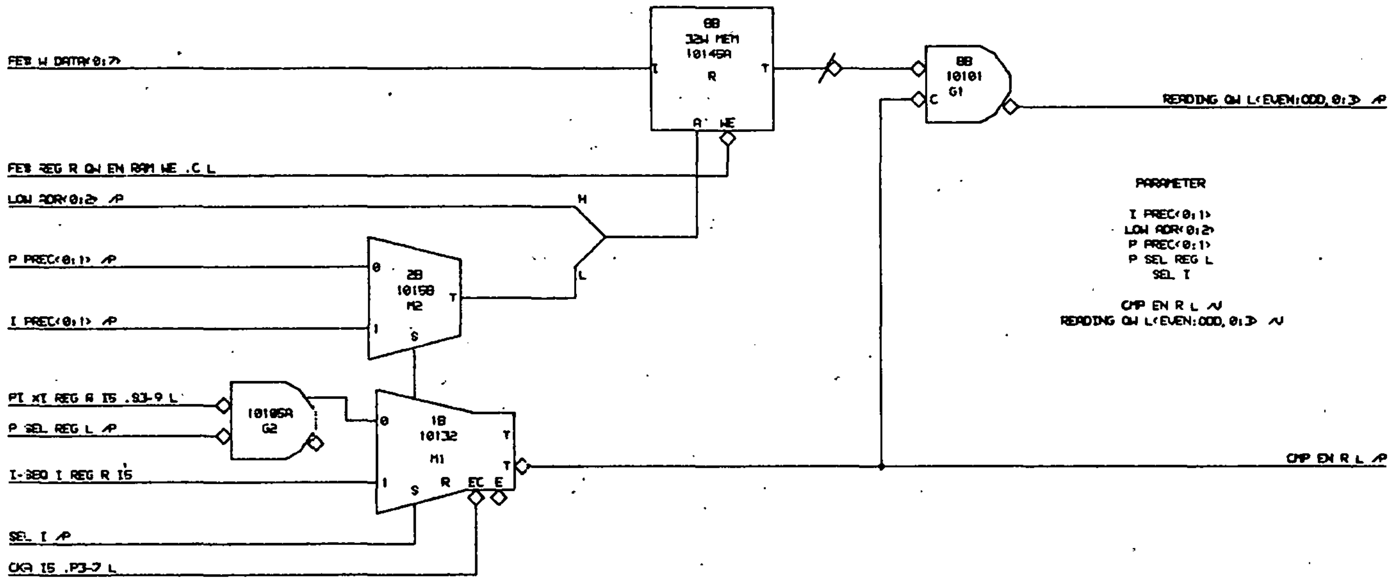
A L<0:15>
MASK L<0:15>
RA<EVEN:ODD, 4:7>
WA<EVEN:ODD, 4:7>
READING QW L<EVEN:ODD, 0:3>
WE L
WRITING QW<EVEN:ODD, 0:3>

M L<EVEN:ODD, 0:15> /U
OVERLAP L<EVEN:ODD, 0:15> /U

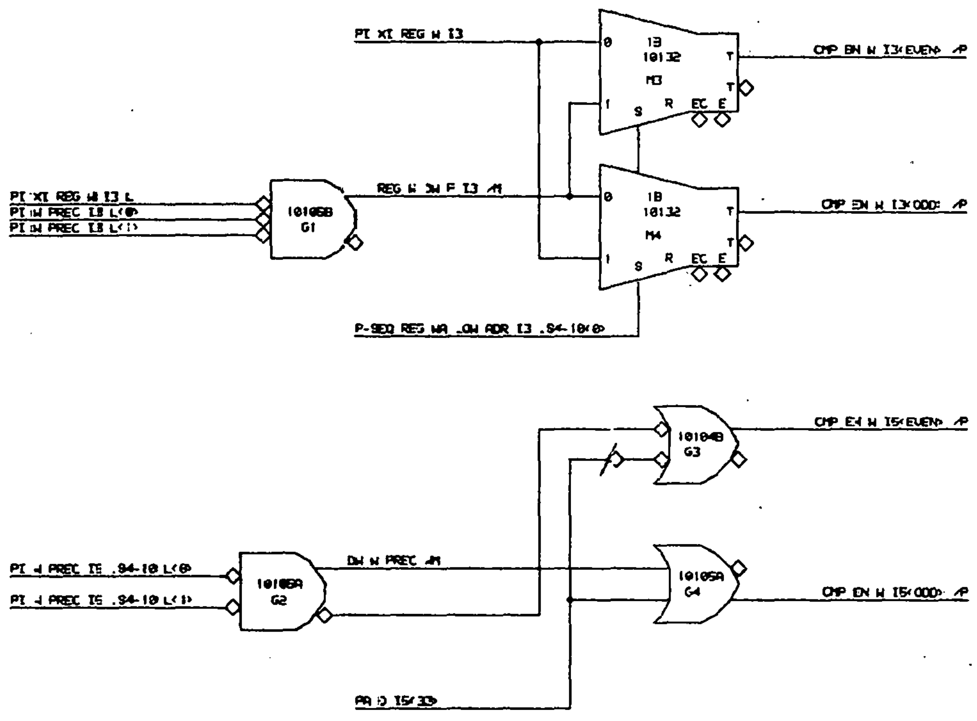
REG CAM



INDEX REG CAM


FE8 W DATA<0:7>
FE8 REG R QW EN RAM WE .C L
LOW ADR<0:2> /P
P PREC<0:1> /P
I PREC<0:1> /P
PI XT REG A I5 .83-9 L
P SEL REG L /P
I-380 I REG R I5
SEL I /P
CKA I5 .P3-7 L
8B
32W MEM
10145A
R
A
WE
I
T
H
L
2B
10158
M2
0
1
S
T
10105A
G2
1B
10132
M1
0
1
S
R
EC
E
T
T
8B
10101
G1
C
READING QW L<EVEN:ODD,0:3> /P
CMP EN R L /P
PARAMETER
I PREC<0:1>
LOW ADR<0:2>
P PREC<0:1>
P SEL REG L
SEL I
CMP EN R L ~/
READING QW L<EVEN:ODD,0:3> ~/

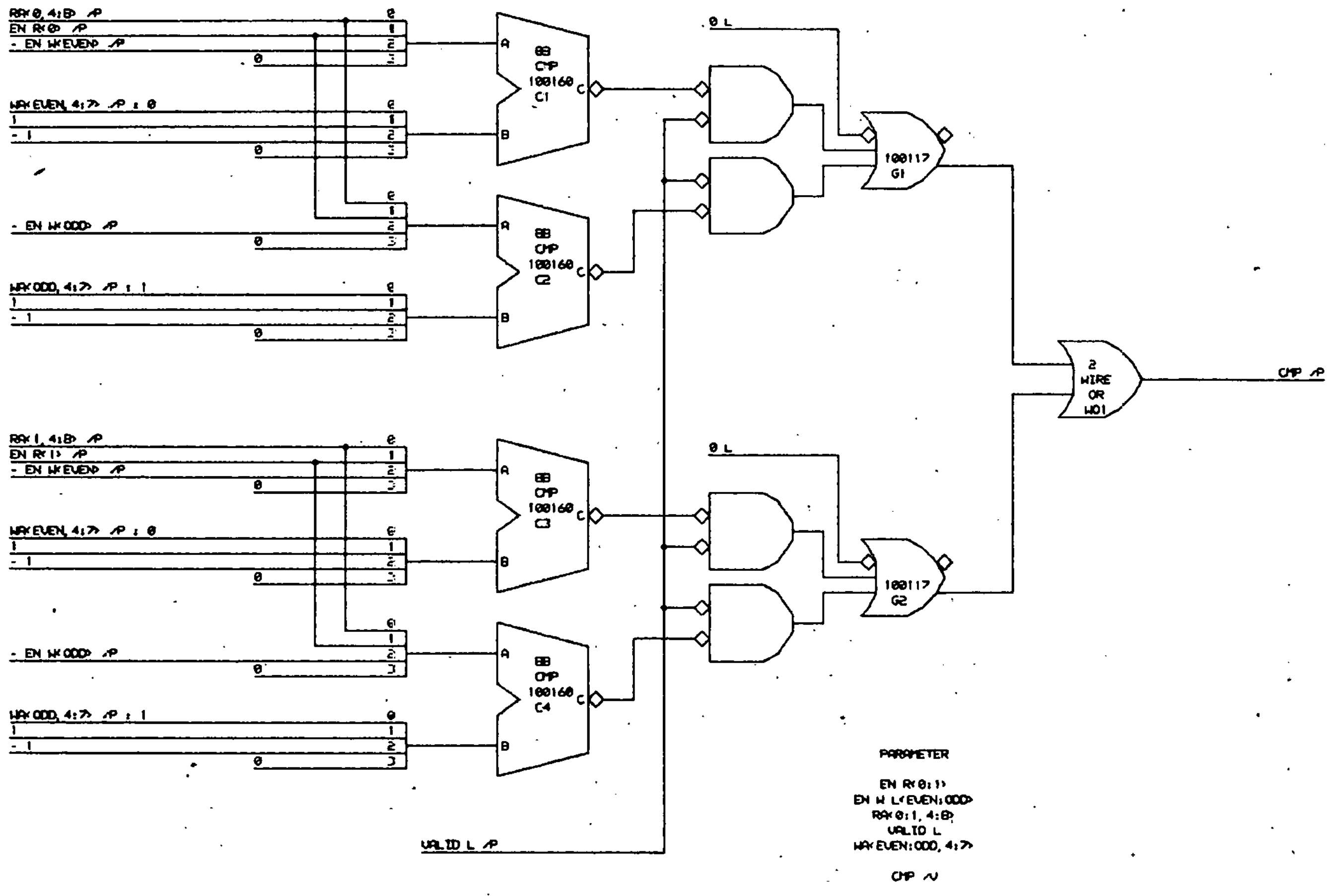
REG CMP EN R



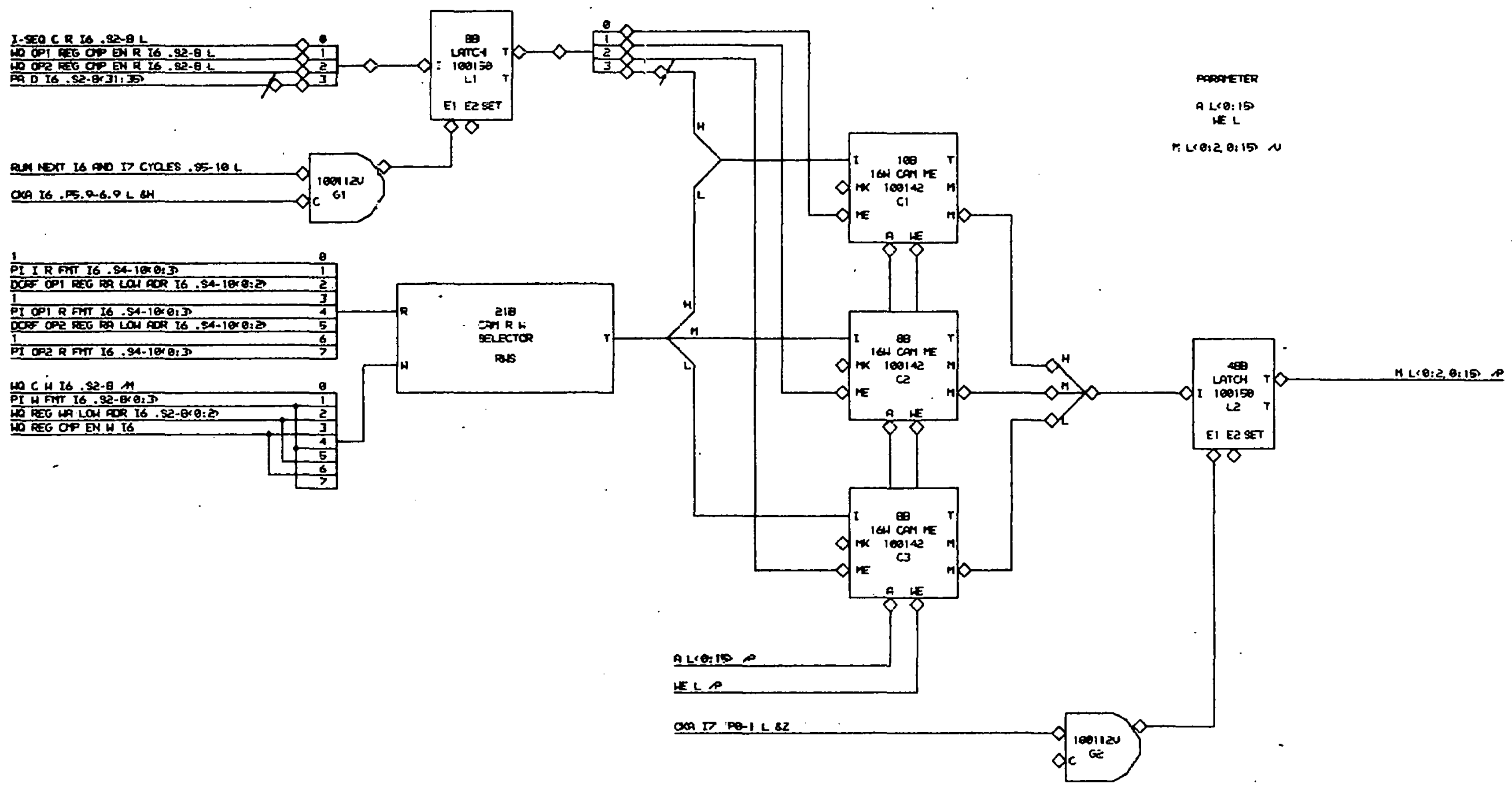
PARAMETER

CMP EN W I3<EVEN:ODD> /V
CMP EN W I5<EVEN:ODD> /V

REG CMP EN W

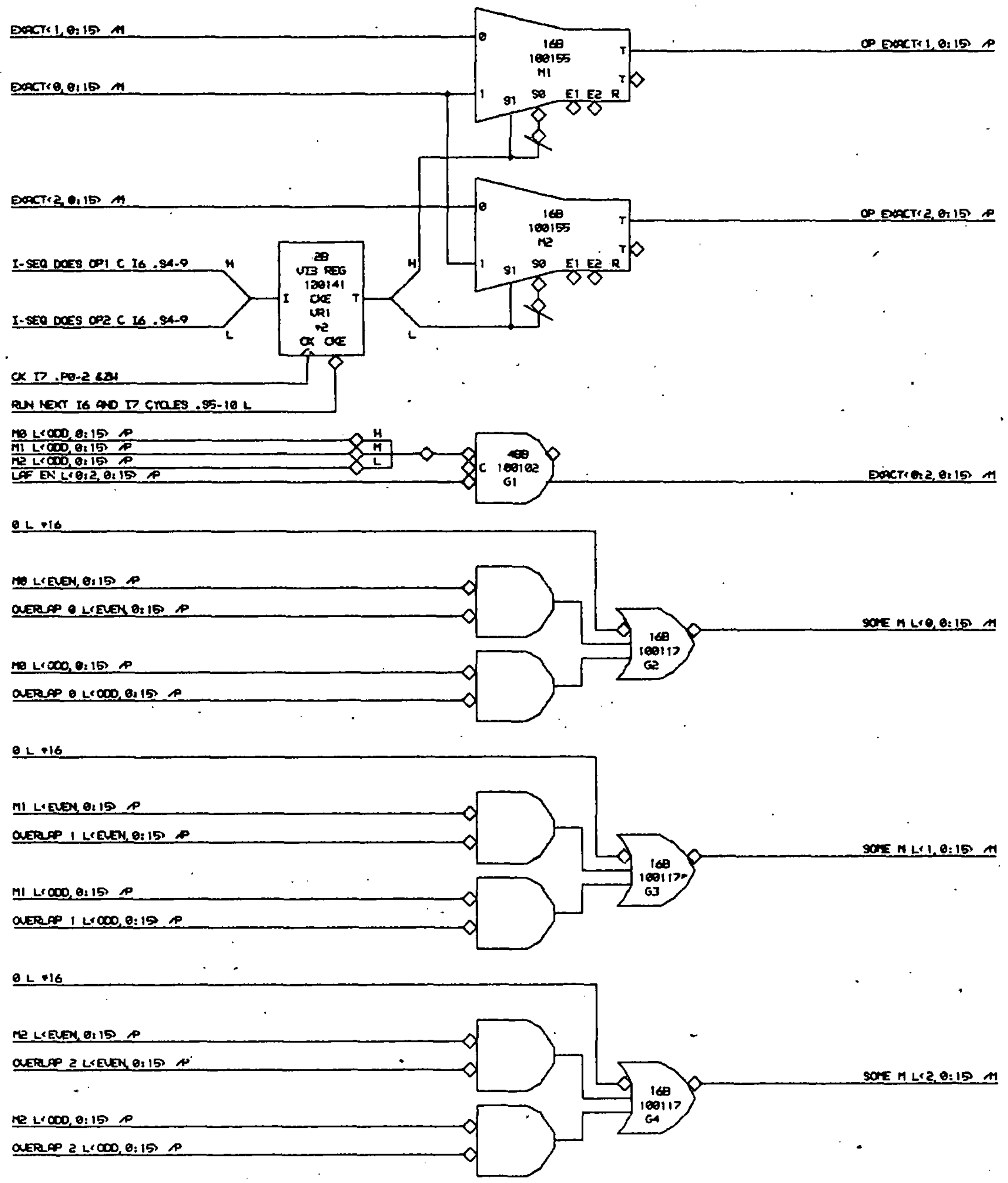


INDEX REG CMP



FORMAT LOW ADR ENABLE COMPARE

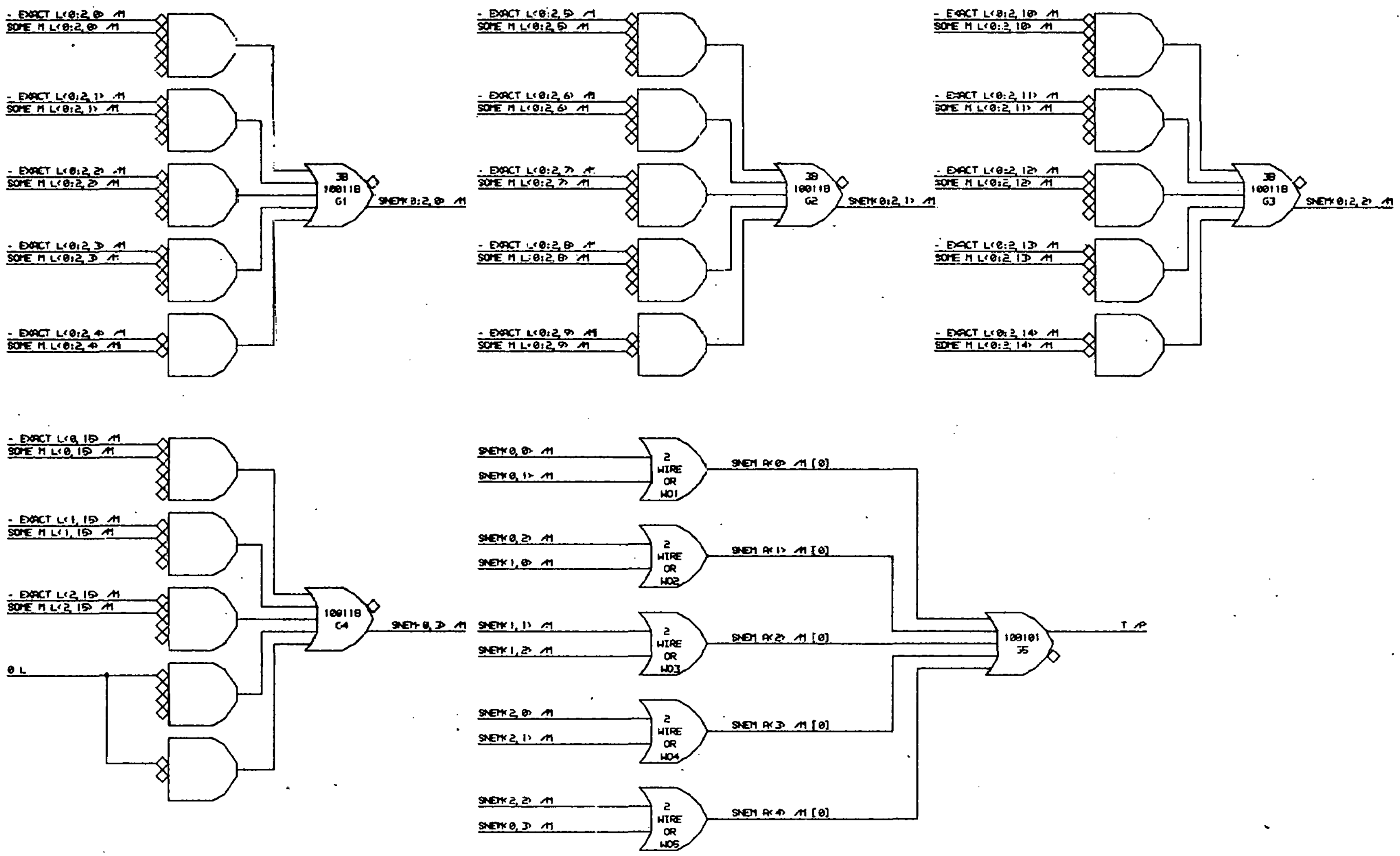
SOME NOT EXACT MATCH



PARAMETER

LAF EN L<0:2,0:15>
M0 L<EVEN:ODD,0:15>
M1 L<EVEN:ODD,0:15>
M2 L<EVEN:ODD,0:15>
OVERLAP 0 L<EVEN:ODD,0:15>
OVERLAP 1 L<EVEN:ODD,0:15>
OVERLAP 2 L<EVEN:ODD,0:15>

OP EXACT<1:2,0:15> /V
T /V



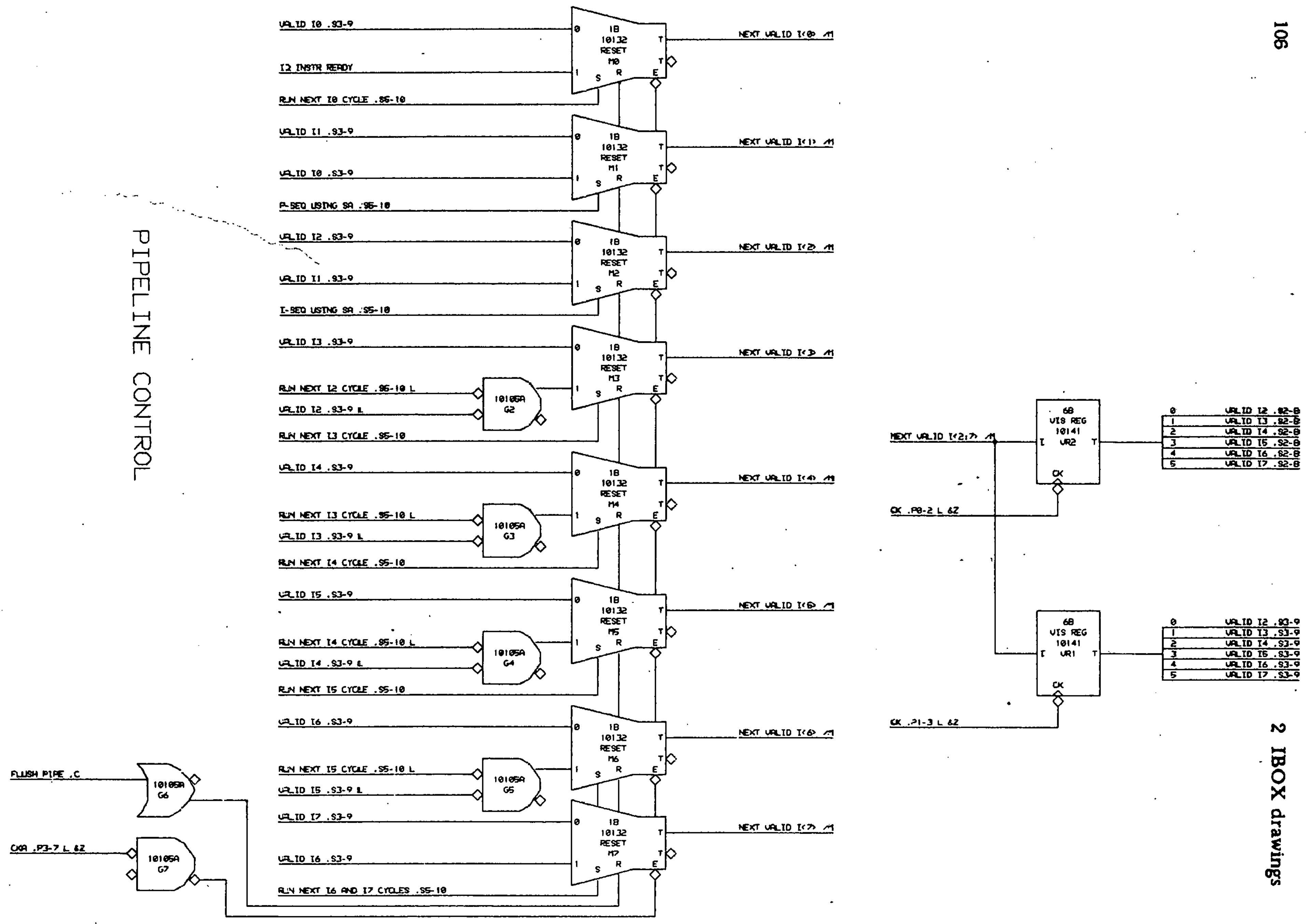
SOME NOT EXACT MATCH

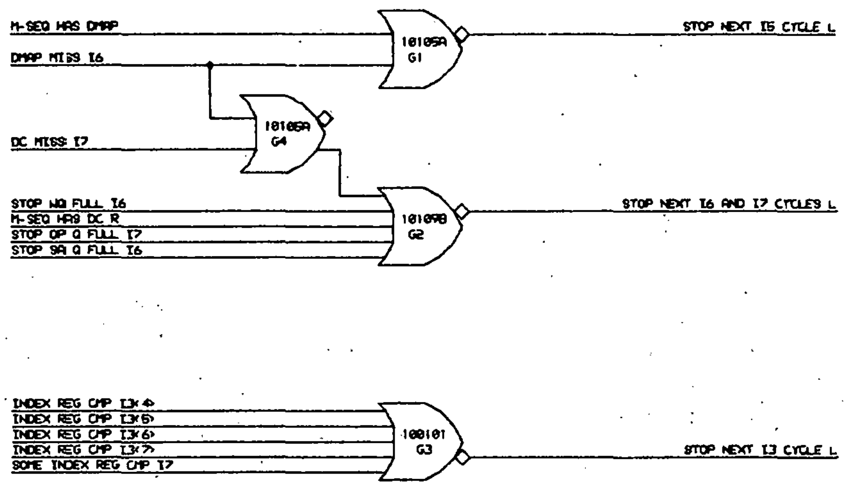
2.20 PIPEC

Drawings: PIPEC1, PIPEC2, PIPEC3

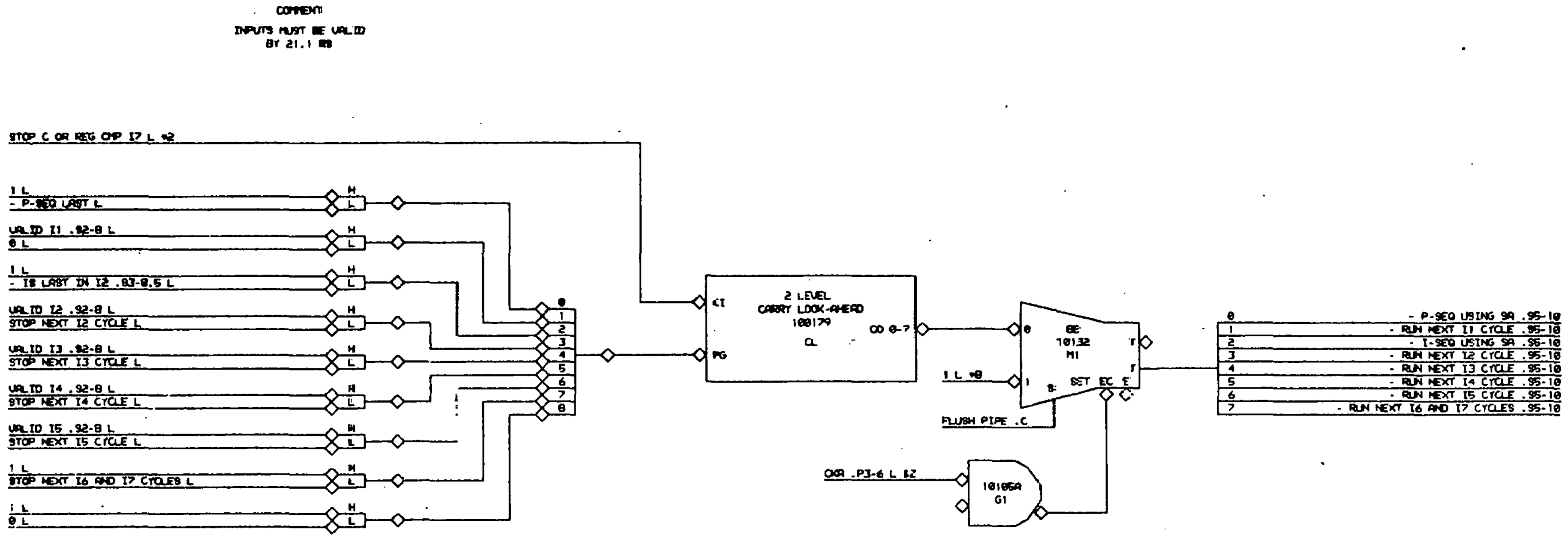
The pipeline control unit is responsible for stopping the clocks when the stop lines are asserted and for keeping track of which pipeline stages have valid signals propagating through them. The clock stopping logic centers on a two-level 100179 carry look-ahead unit. The carry outputs of this unit are the "stop" lines. The least significant output corresponds to the latest pipeline stage (I6 and I7). For a given stop line, the "generate" input is an unconditional stop signal. The "propagate" input is asserted if the next stage being stopped is sufficient to cause the current stage to be stopped. Typically this is the case if and only if the current stage has valid information in it. (Then, if the next stage is stopped, there is no place for the valid information in the current stage to go, so the current stage must be stopped too.)

PIPELINE CONTROL





PIPELINE CONTROL



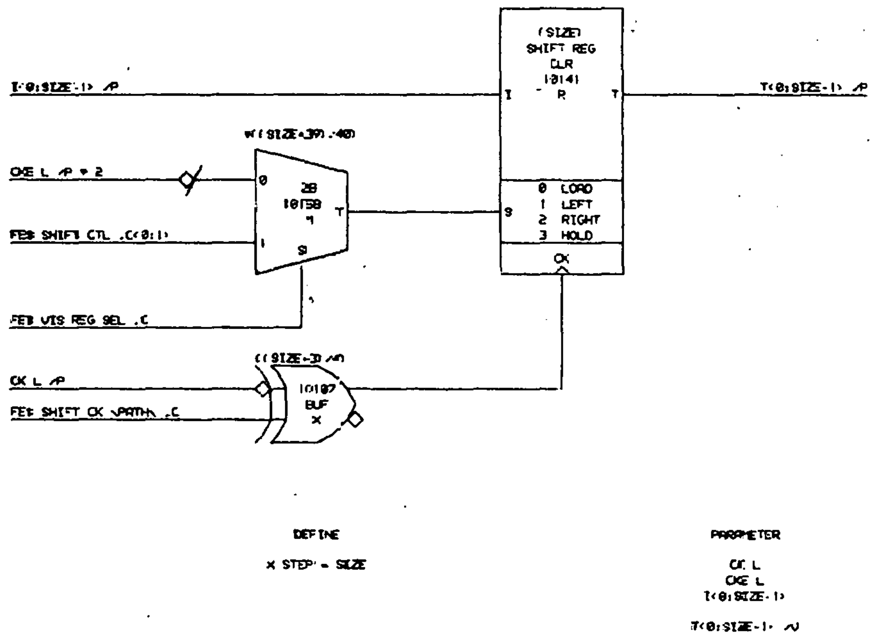
PIPELINE CONTROL

2.21 VREG

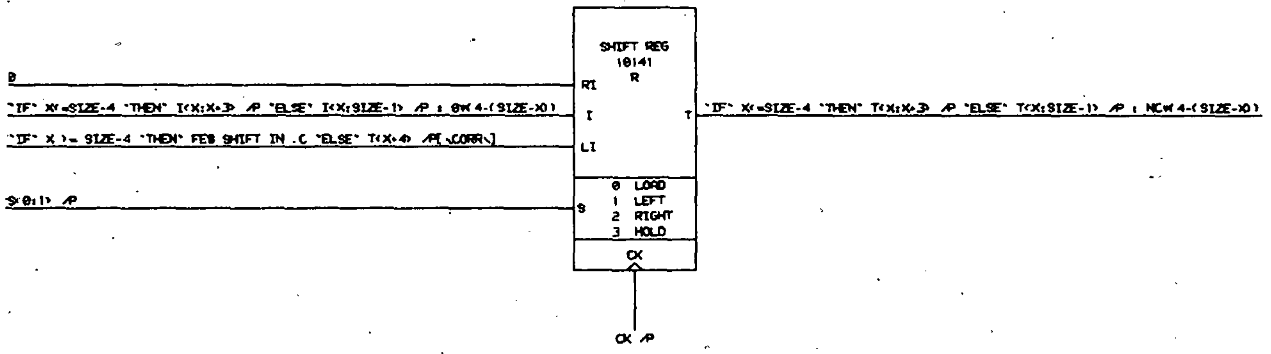
Drawings: VREGE1, SRCLR

The vision registers are part of the low-level documentation but contain certain features that are worth describing in this high-level description. A vision register (VIS REG) acts, as far as the calling macro is concerned, as a simple broadside load register with a clock enable.

The register that is used to implement a VIS REG, however, provides additional features. It can hold or load parallel input data or shift its data one place to the left or right on any given clocking, as determined by a two-bit select line. Normally the select lines are driven from the clock enable signal to switch between load and hold. However, there are special connections which allow the front end maintenance processor to control the select lines. When the front end stops the clocks it can switch the select lines to the shift-left mode and pulse the register's clock line (through an XOR gate with a front end connection). This will shift the register one place to the left, shifting in data from the front end. In this way, the front end can load the contents of any VIS REG. It can also read out the contents of any VIS REG by examining the high bit while shifting all the bits out.



VIS REG CKE 10141



SHIFT REG
10141
R
0
"IF" X<=SIZE-4 "THEN" I<X:X+3> /P "ELSE" I<X:SIZE-1> /P : 0W 4-(SIZE-X)
"IF" X >= SIZE-4 "THEN" FEB SHIFT IN .C "ELSE" T<X+4> /P[.CORR.]
S<0:1> /P
RI
I
LI
S
T
0 LOAD
1 LEFT
2 RIGHT
3 HOLD
CK
CK /P
"IF" X<=SIZE-4 "THEN" T<X:X+3> /P "ELSE" T<X:SIZE-1> /P : NCW 4-(SIZE-X)

DEFINE

X STEP = 4

PARAMETER

CK
I<0:SIZE-1>
S<0:1>

T<0:SIZE-1>

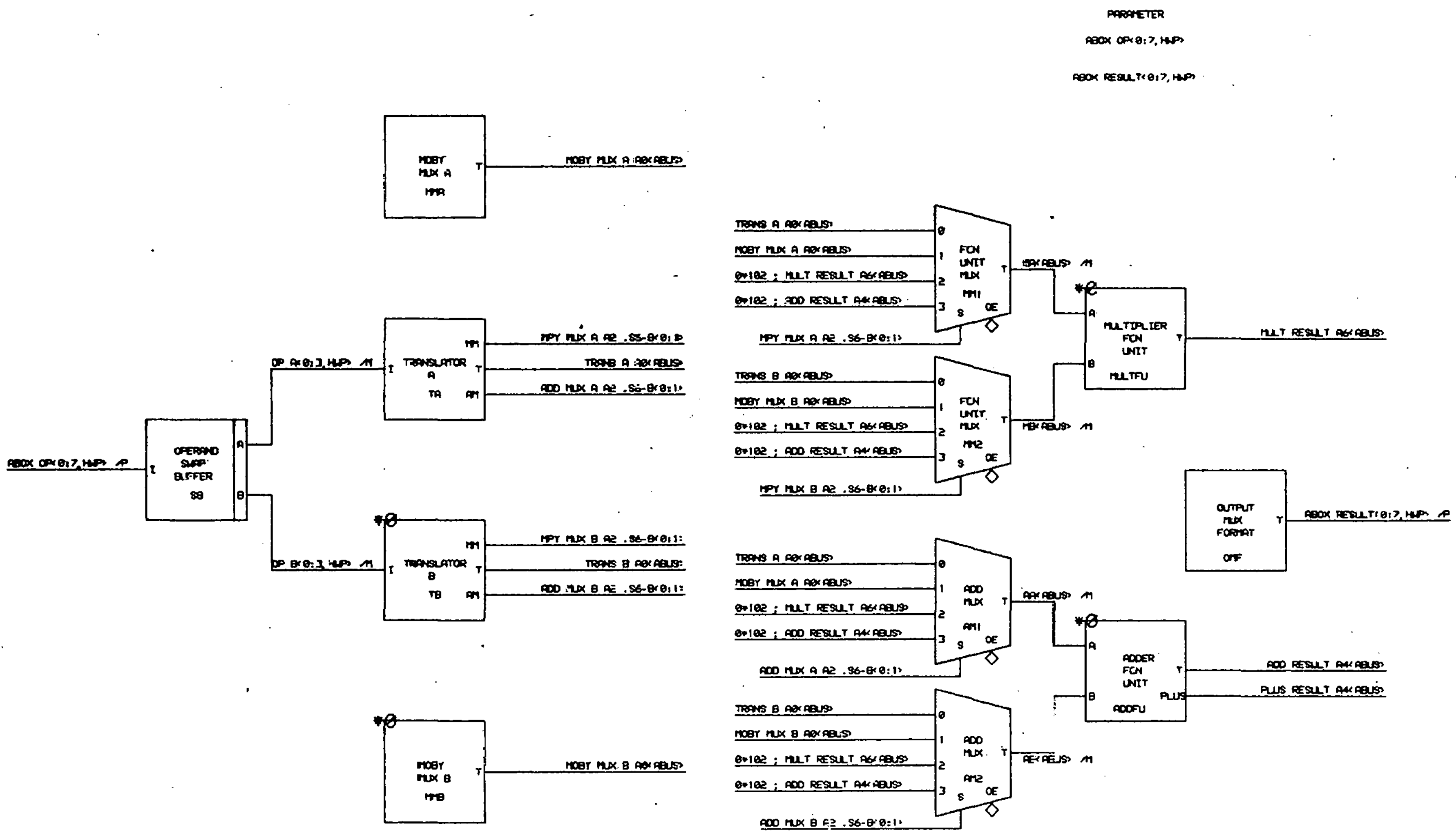
SHIFT REG CLR 10141

3 ABOX drawings

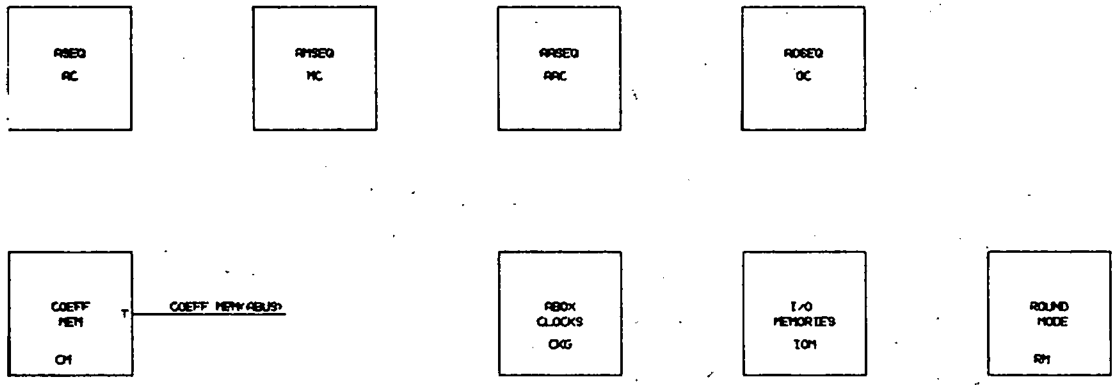
3.1 ABOX

Drawings: ABOX1, ABOX2

This is the defining macro for the ABOX. It is called from the IBOX. The explicit parameters are the two input operands (4 words wide) and the result (also 4 words wide). The main data paths are shown on the first page. The second page contains macros for various pieces of control logic.


PARAMETER
ABOX OP<0:7,HWP>
ABOX RESULT<0:7,HWP>
MOBY MUX A
MMA
MOBY MUX A A0<ABUS>
ABOX OP<0:7,HWP> /P
OPERAND SWAP BUFFER
SB
OP A<0:3,HWP> /M
OP B<0:3,HWP> /M
TRANSLATOR A
TA
MM
AM
MPY MUX A A2 .S5-B<0:1>
TRANS A A0<ABUS>
ADD MUX A A2 .S6-B<0:1>
TRANSLATOR B
TB
MPY MUX B A2 .S6-B<0:1>
TRANS B A0<ABUS>
ADD MUX B A2 .S6-B<0:1>
MOBY MUX B
MMB
MOBY MUX B A0<ABUS>
TRANS A A0<ABUS>
MOBY MUX A A0<ABUS>
0+102 ; MULT RESULT A6<ABUS>
0+102 ; ADD RESULT A4<ABUS>
FCN UNIT MUX
MM1
OE
MPY MUX A A2 .S6-B<0:1>
MA<ABUS> /M
TRANS B A0<ABUS>
MOBY MUX B A0<ABUS>
FCN UNIT MUX
MM2
MPY MUX B A2 .S6-B<0:1>
MB<ABUS> /M
MULTIPLIER FCN UNIT
MULTFU
MULT RESULT A6<ABUS>
OUTPUT MUX FORMAT
OMF
ABOX RESULT<0:7,HWP> /P
ADD MUX
AM1
ADD MUX A A2 .S6-B<0:1>
AA<ABUS> /M
ADDER FCN UNIT
ADDFU
PLUS
ADD RESULT A4<ABUS>
PLUS RESULT A4<ABUS>
AM2
ADD MUX B A2 .S6-B<0:1>
AB<ABUS> /M

ABOX


ASEQ
AC
AMSEQ
MC
AASEQ
AAC
ADSEQ
OC
COEFF
MEM
CM
T
COEFF MEM<ABUS>
ABOX
CLOCKS
CKG
I/O
MEMORIES
IOM
ROUND
MODE
RM

ABOX

3.2 ASEQ

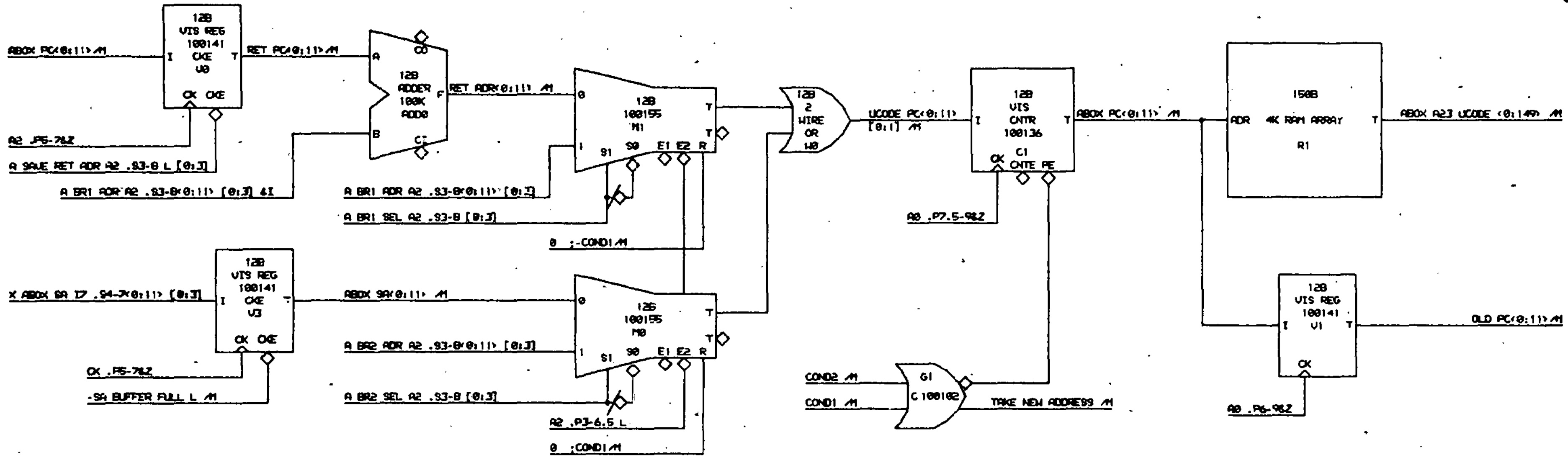
Drawings: ASEQ, ASEQ1, ASEQ2

The A-Sequencer is the main control engine of the ABOX. It provides direct control for itself, the Operand Swap Buffer, the Translators, and the Moby Muxes. In addition it provides starting addresses for the Adder, Multiplier and Output Formatter microengines. The A-Sequencer also handles some of the hand-shaking necessary for communications with the IBOX. The A sequencer contains a 4k word by 150 bit microstore, an address counter capable of parallel loading, branch condition multiplexers, and repeat control.

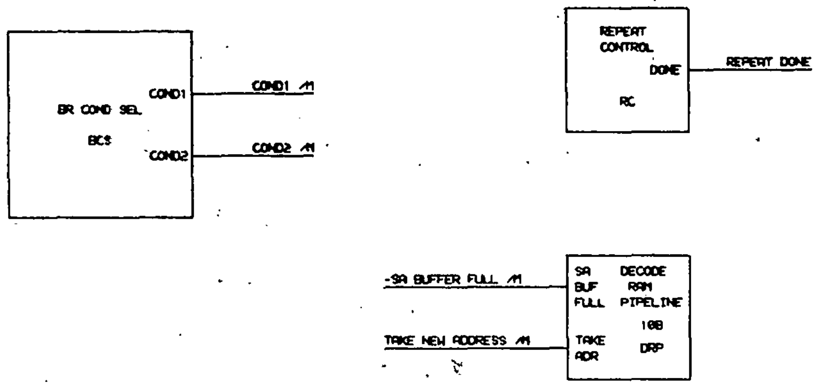
Through use of branch conditions and a pair of 2-input multiplexers, the A-sequencer is capable of taking its starting address from one of four sources. It can take a new starting address from the IBOX, from either of two branch addresses supplied by the current micro instruction, or from a return-from-subroutine address that may be modified by one of the current micro instruction branch addresses. This modification is provided to allow a return to any micro instruction in the microstore.

Repeat control is provided so that any microinstruction can specify the number of times that it is to be repeated. Each time that the instruction is executed the repeat counter is decremented. When the count reaches zero, a branch address is taken based on one of the two microinstruction branch addresses.

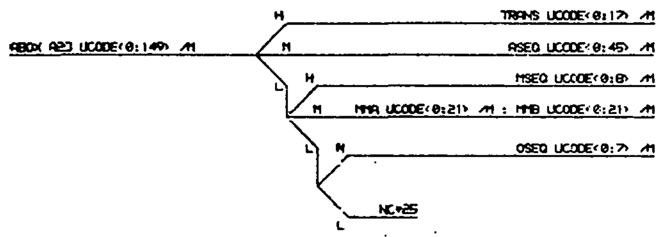
Also included in the A-Sequencer is a pipeline for some of the control bits supplied by the IBOX via the Decode RAM.



		COMMENT	
BR SEL	COND1	-COND1~COND2	-COND1~-COND2
0	RET ADR	SA	PC+1
1	RET ADR	BR ADR 2	PC+1
2	BR ADR1	SA	PC+1
3	BR ADR1	BR ADR2	PC+1



ASEQ



TRANS UCODE<0> /M	A TRANSA HW COMPLEX A2 .S3-8
TRANS UCODE<1:2> /M	A TRANSA PRECISION A2 .S3-8<0:1>
TRANS UCODE<3:5> /M	A TRANSA ROTATE A2 .S3-8<0:2>
TRANS UCODE<6> /M	A TRANSA TYPE A2 .S3-8
TRANS UCODE<7> /M	A TRANSA USE DECODE RAM A2 .S3-8
TRANS UCODE<8> /M	A TRANSB HW COMPLEX A2 .S3-8
TRANS UCODE<9:10> /M	A TRANSB PRECISION A2 .S3-8<0:1>
TRANS UCODE<11:13> /M	A TRANSB ROTATE A2 .S3-8<0:2>
TRANS UCODE<14> /M	A TRANSB TYPE A2 .S3-8
TRANS UCODE<15> /M	A TRANSB USE DECODE RAM A2 .S3-8
TRANS UCODE<16> /M	A NEW RESULT SCHEDULED A2 .S3-8
TRANS UCODE<17> /M	A TAKE OPERANDS FROM IBOX A2 .S3-8

ASEQ UCODE<0> /M	A SAVE RET ADR A2 .S3-8
ASEQ UCODE<1:12> /M	A BR1 ADR A2 .S3-8<0:11>
ASEQ UCODE<13:24> /M	A BR2 ADR A2 .S3-8<0:11>
ASEQ UCODE<25> /M	A BR SEL A2 .S3-8
ASEQ UCODE<26> /M	A BR2 SEL A2 .S3-8
ASEQ UCODE<27:31> /M	A BR1 COND SEL A2 .S3-8<0:4>
ASEQ UCODE<32:36> /M	A BR2 COND SEL A2 .S3-8<0:4>
ASEQ UCODE<37:43> /M	A REPEAT COUNT A2 .S3-8<0:6>
ASEQ UCODE<44:45> /M	A REPEAT COUNTER CONTROL A2 .S3-8<0:1>

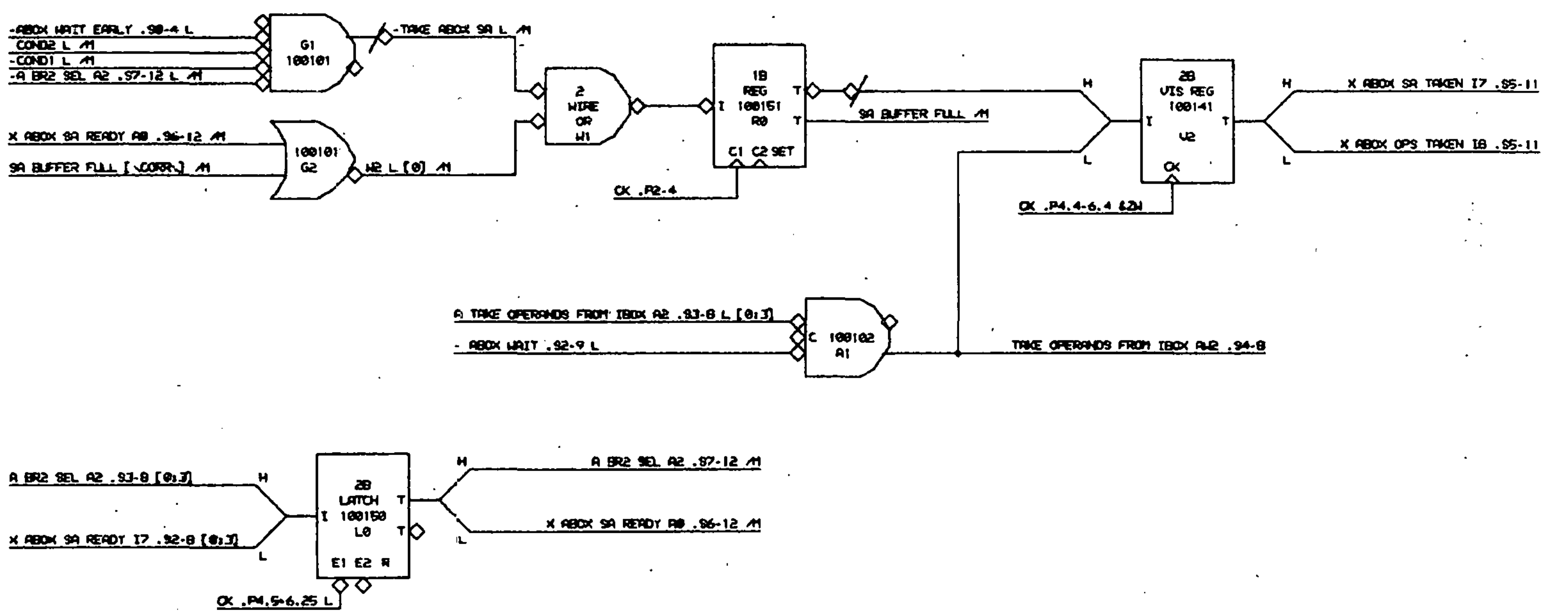
MSEQ UCODE<0:7> /M	A MPY ADR A2 .S3-8<0:7>
MSEQ UCODE<8> /M	A MPY LOW ADR A2 .S3-8

MMA UCODE<0:2> /M	A MMA MUX SEL A2 .S3-8<0:2>
MMA UCODE<3:10> /M	A MMA READ ADR A2 .S3-8<0:7>
MMA UCODE<11> /M	A MMA READ ADR SOURCE SEL A2 .S3-8
MMA UCODE<12> /M	A MMA READ CONSTANT A2 .S3-8
MMA UCODE<13> /M	A MMA RESET FIFO A2 .S3-8
MMA UCODE<14:17> /M	A MMA WRITE ADR A2 .S3-8<0:3>
MMA UCODE<18> /M	A MMA WRITE ADR SOURCE SEL A2 .S3-8
MMA UCODE<19> /M	A MMA WRITE ENA1 A2 .S3-8
MMA UCODE<19:21> /M	MMA DELAY UCODE<0:2>

OSEQ UCODE<0:7> /M	A FORMATTER ADR A2 .S3-8<0:7>

MMB UCODE<0:2> /M	A MMB MUX SEL A2 .S3-8<0:2>
MMB UCODE<3:10> /M	A MMB READ ADR A2 .S3-8<0:7>
MMB UCODE<11> /M	A MMB READ ADR SOURCE SEL A2 .S3-8
MMB UCODE<12> /M	A MMB READ CONSTANT A2 .S3-8
MMB UCODE<13> /M	A MMB RESET FIFO A2 .S3-8
MMB UCODE<14:17> /M	A MMB WRITE ADR A2 .S3-8<0:3>
MMB UCODE<18> /M	A MMB WRITE ADR SOURCE SEL A2 .S3-8
MMB UCODE<19> /M	A MMB WRITE ENA1 A2 .S3-8
MMB UCODE<19:21> /M	MMB DELAY UCODE<0:2>

ASEQ

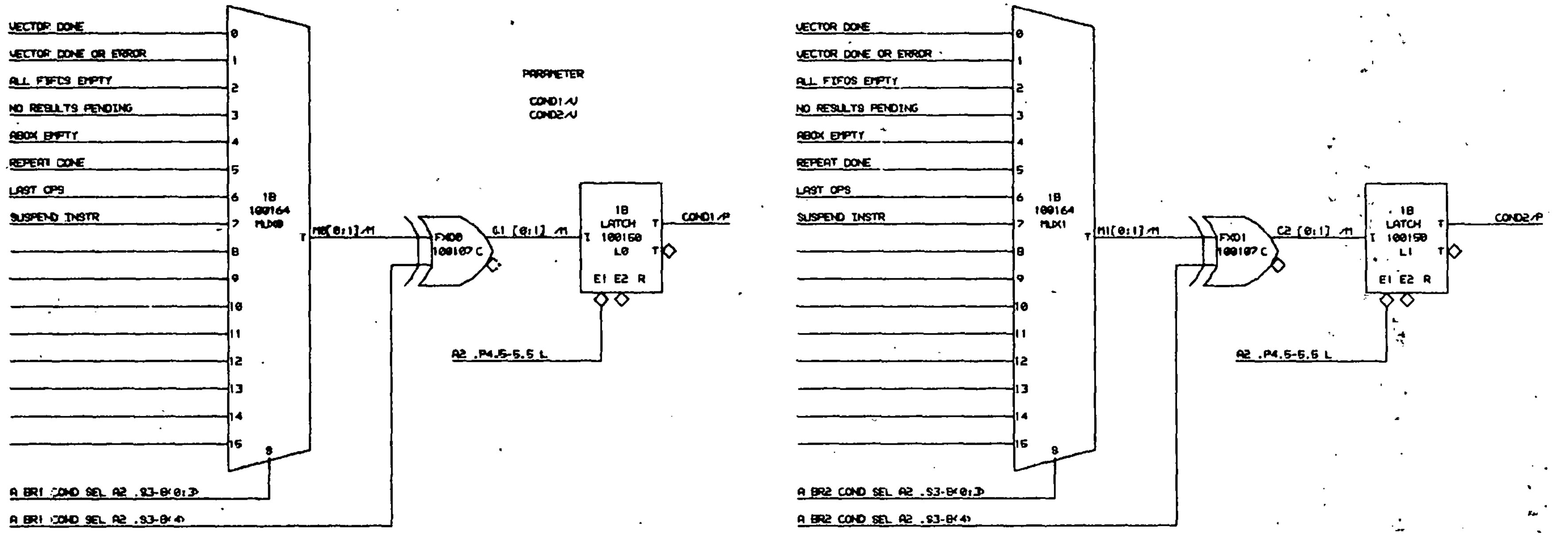

-ABOX WAIT EARLY .90-4 L
COND2 L /M
-COND1 L /M
-A BR2 SEL A2 .S7-12 L /M
G1
100101
-TAKE ABOX SA L /M
X ABOX SA READY A0 .96-12 /M
SA BUFFER FULL [\CORR\] /M
100101
G2
W2 L [0] /M
2
WIRE
OR
W1
1B
REG
100151
R0
I
T
T
C1 C2 SET
CK .A2-4
SA BUFFER FULL /M
H
L
2B
VIS REG
100141
V2
I
T
CK
CK .P4.4-6.4 &2W
H
L
X ABOX SA TAKEN I7 .S5-11
X ABOX OPS TAKEN I8 .S5-11
A TAKE OPERANDS FROM IBOX A2 .93-8 L [0:3]
- ABOX WAIT .92-9 L
C 100102
A1
TAKE OPERANDS FROM IBOX A#2 .94-8
A BR2 SEL A2 .93-8 [0:3]
X ABOX SA READY I7 .92-8 [0:3]
H
L
2B
LATCH
100150
L0
I
T
T
E1 E2 R
CK .P4.5-6.25 L
H
L
A BR2 SEL A2 .S7-12 /M
X ABOX SA READY A0 .S6-12 /M

ASEQ

3.3 BR COND SEL

Drawing: BRCOND

This is the macro that implements branch condition selection for the A-sequencer. It is under direct A-sequencer control.

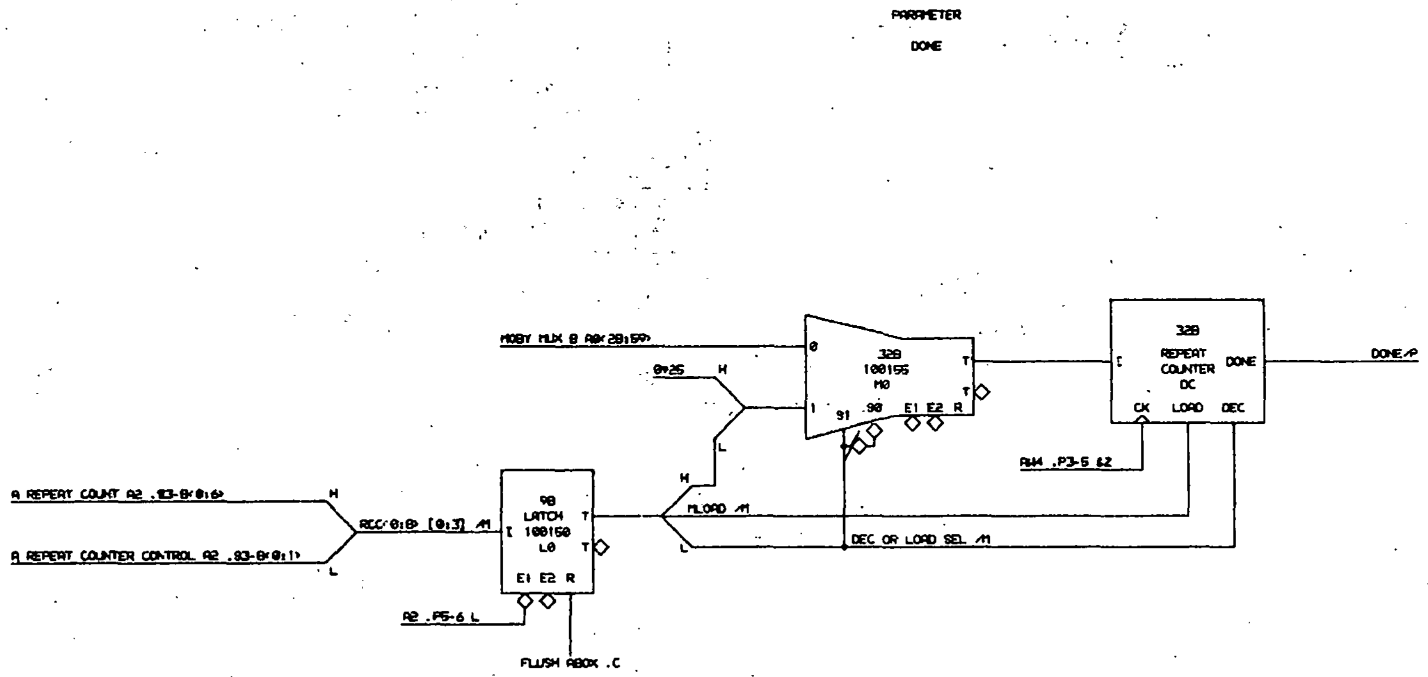

VECTOR DONE
VECTOR DONE OR ERROR
ALL FIFOS EMPTY
NO RESULTS PENDING
ABOX EMPTY
REPEAT DONE
LAST OPS
SUSPEND INSTR
1B 100164 MUX0
M0[0:1]/M
FX00 100107
C1 [0:1] /M
1B LATCH 100150 L0
E1 E2 R
COND1/P
A2 .P4.5-5.5 L
A BR1 COND SEL A2 .S3-B(0:3)
A BR1 COND SEL A2 .S3-B(4)
PARAMETER
COND1/V
COND2/V
1B 100164 MUX1
M1[0:1]/M
FX01 100107
C2 [0:1] /M
1B LATCH 100150 L1
COND2/P
A BR2 COND SEL A2 .S3-B(0:3)
A BR2 COND SEL A2 .S3-B(4)

BR COND SEL

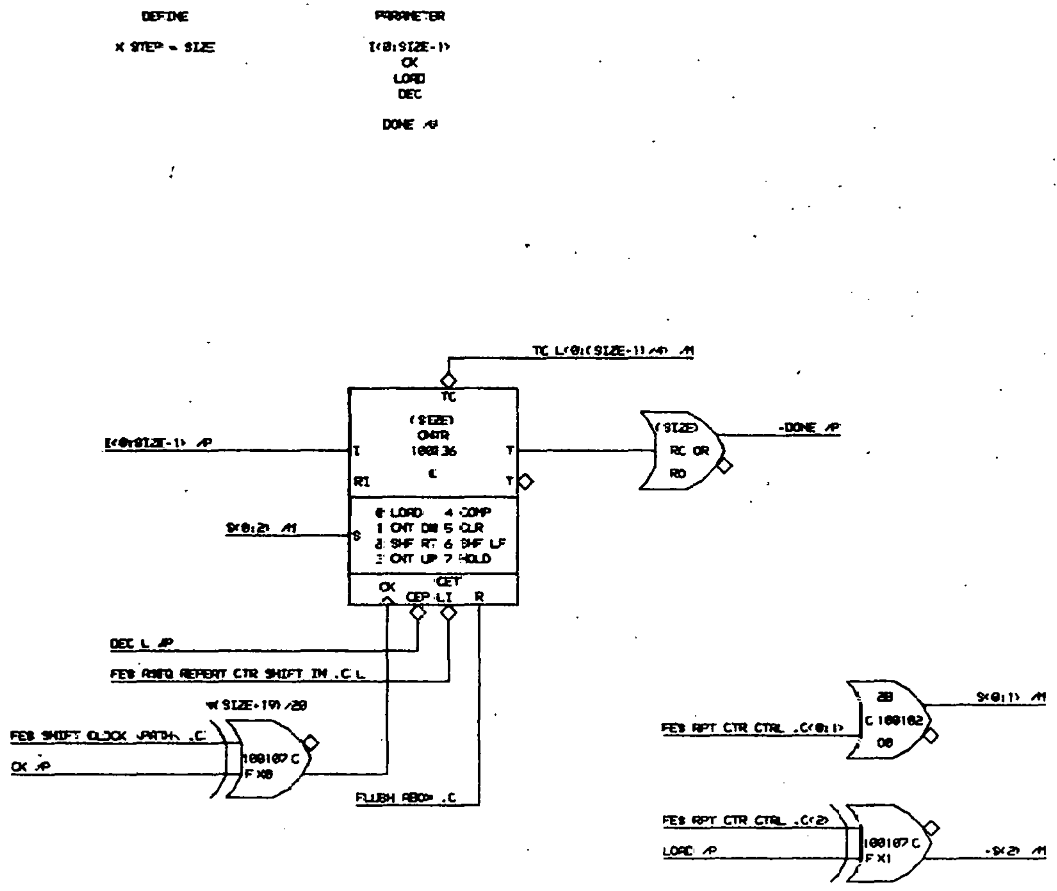
3.4 REPEAT CONTROL

Drawings: RC, RPTCTR

Repeat control provides a branch condition signal "repeat done" to the branch condition multiplexers. The repeat count may be loaded directly from the microcode for small counts or from MOBY MUX A for up to 36 bits of count from any source.



REPEAT CONTROL

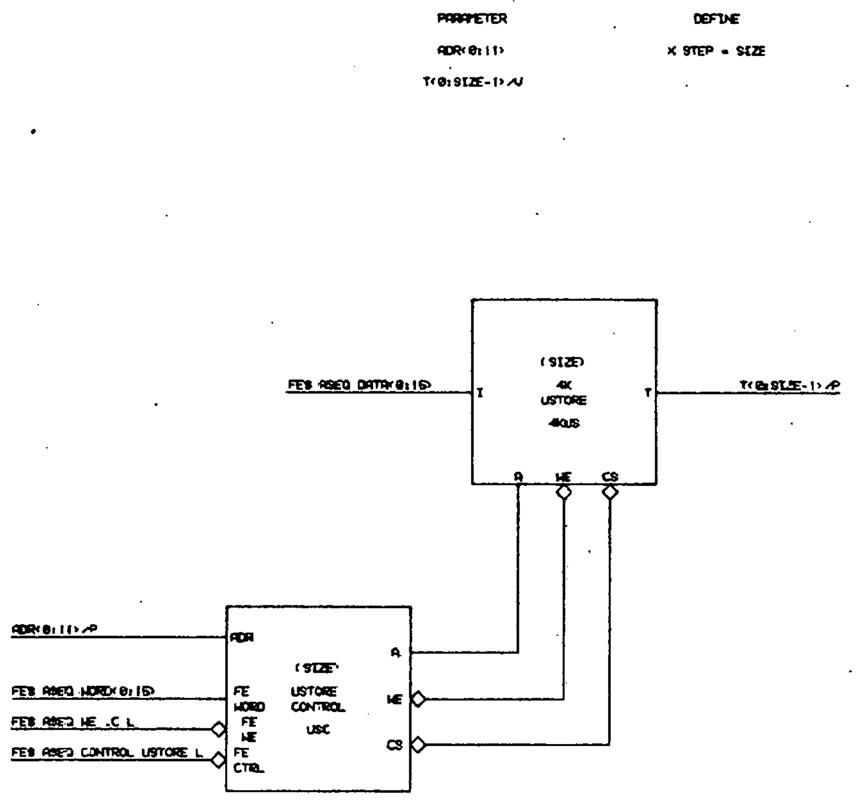


REPEAT COUNTER
SIZE MOD 4 = 0

3.5 4K RAM ARRAY

Drawing: 4KRAMA

This is the RAM array for the A-sequencer. This array is 4k words deep by 150 bits wide. This macro contains two other macros, 4K USTORE and USTORE CONTROL.

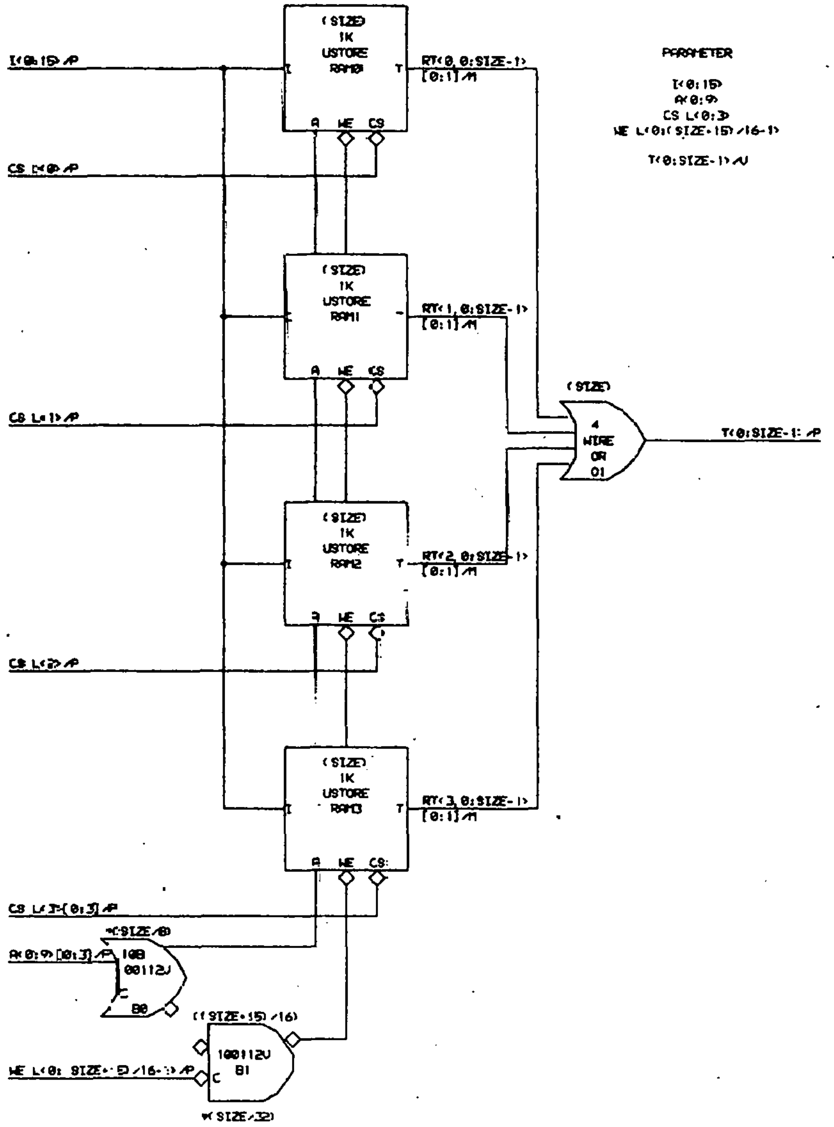

PARAMETER
ADR<0:11>
T<0:SIZE-1>/U
DEFINE
X STEP = SIZE
FE$ ASEQ DATA<0:15>
I
(SIZE)
4K
USTORE
4KUS
T
T<0:SIZE-1>/P
A
WE
CS
ADR<0:11>/P
ADR
A
(SIZE)
FE$ ASEQ WORD<0:15>
FE
WORD
USTORE
CONTROL
USC
WE
FE$ ASEQ WE .C L
FE
WE
FE$ ASEQ CONTROL USTORE L
FE
CTRL
CS

4K RAM ARRAY

3.6 4K USTORE

Drawing: 4KUS

This macro simply provides the buffering necessary to drive the various address and control lines in the RAM array. 1K USTORE is called to provide more of the necessary buffering.



PARAMETER

I<0:15>
A<0:9>
CS L<0:3>
WE L<0:(SIZE+15)/16-1>

T<0:SIZE-1>/U

DEFINE

X STEP = SIZE

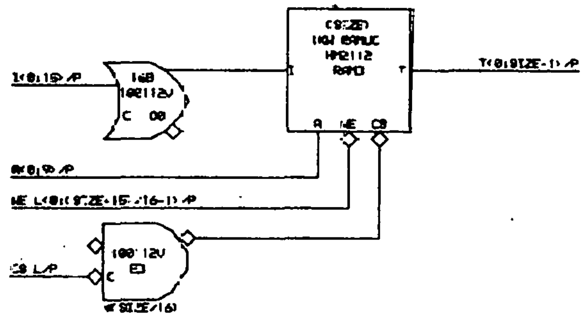
4K USTORE

3.7 1K USTORE

Drawing: 1KUS

This macro provides more of the buffering of the RAM array control lines.

PARAMETER	DEFINE
I<0:15>	X STEP = SIZE
A<0:9>	
CS L	
WE L<0:(SIZE+15)/16-1>	
T<0:SIZE-1>/U	

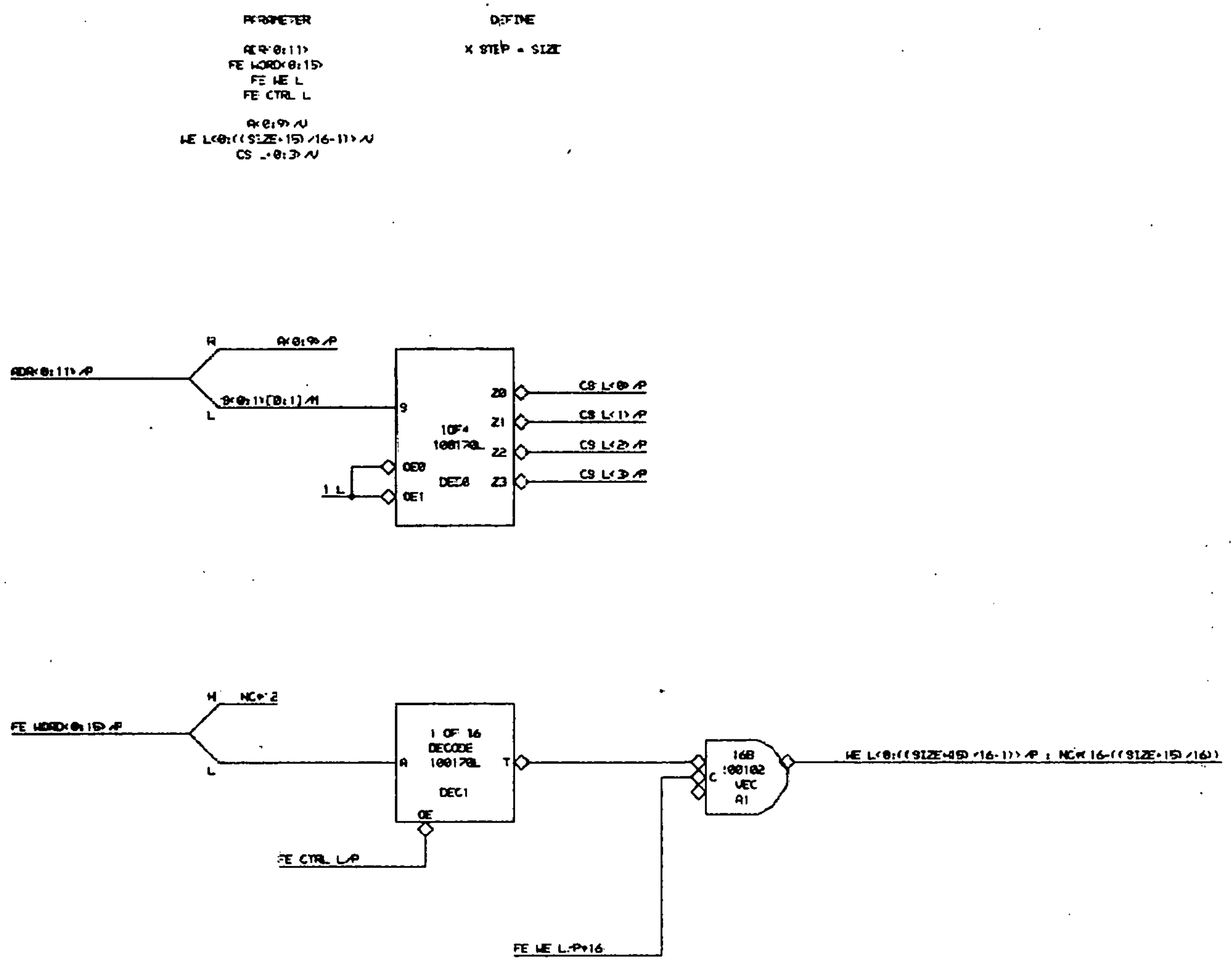


1K USTORE

3.8 USTORE CONTROL

Drawing: AUCTRL

This macro provides the decoding of the low order address bits to provide the chip-select signal to the various RAMs in the RAM array. It also provides the means by which the front end computer can load the microstore during the initialization of the machine.

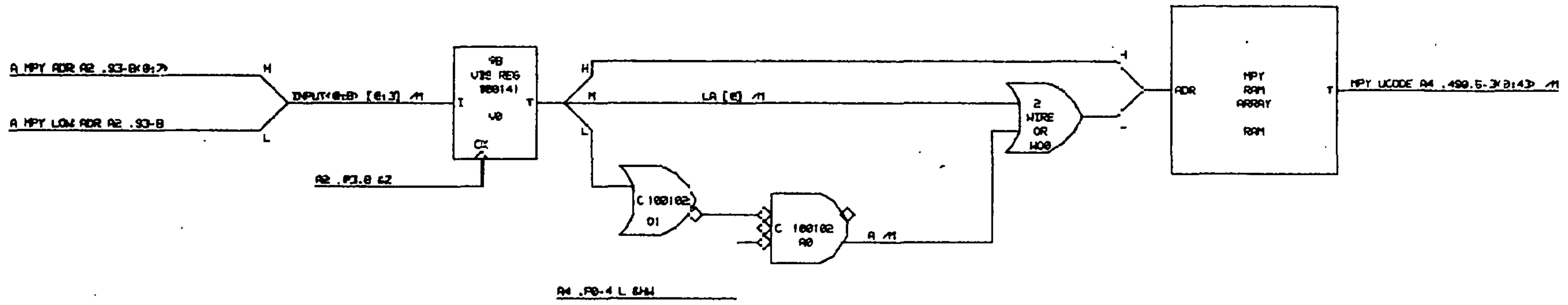


USTORE CONTROL

3.9 AMSEQ

Drawing: AMSEQ

AMSEQ is the definition of the multiplier microsequencer. The multiplier sequencer is very simple. It is composed of a vision register to hold the address, a RAM array called MPY RAM ARRAY, and a means of toggling the low order address bit. The ability to toggle the low address bit is provided so that pairs of addresses can be read out in succeeding 25 ns periods.

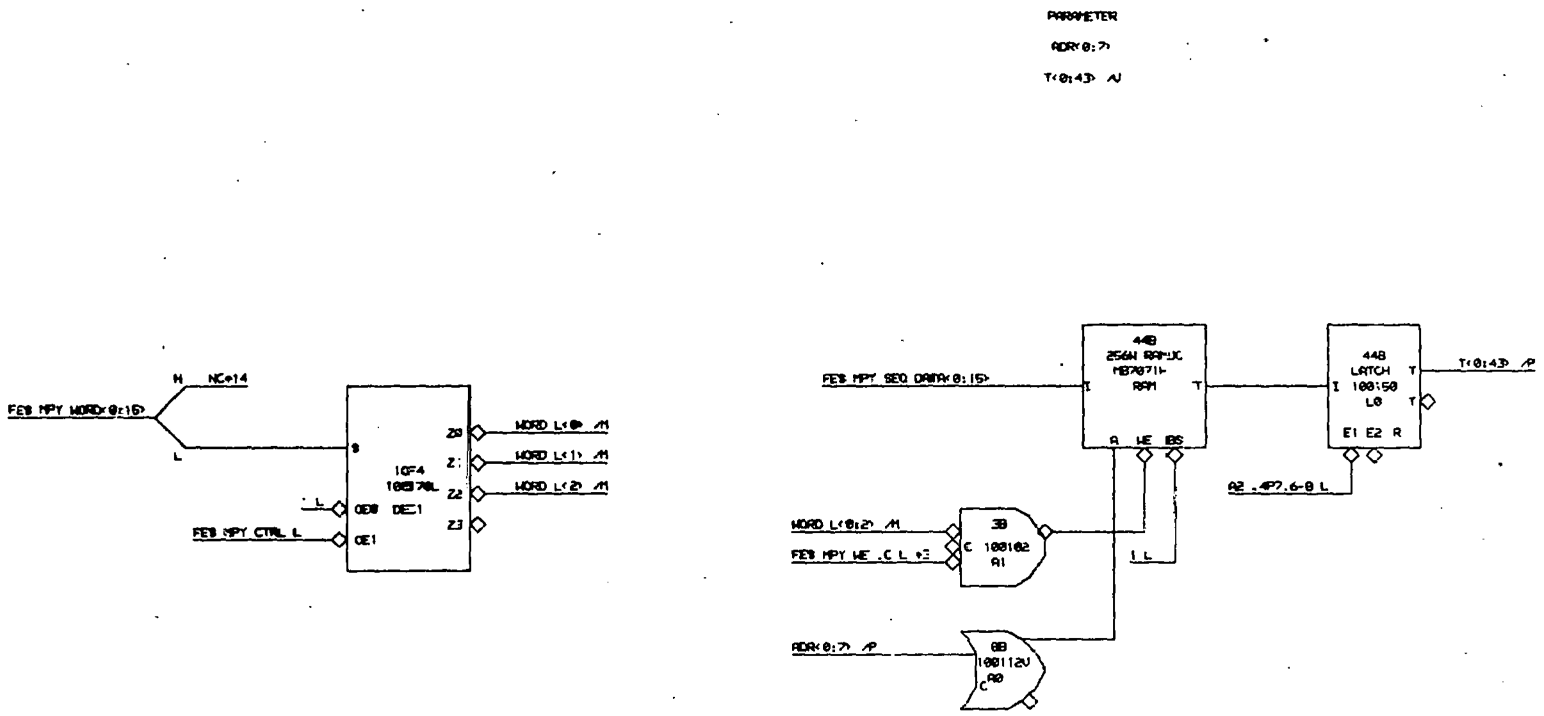

A MPY ADR A2 .S3-B<0:7>
A MPY LOW ADR A2 .S3-B
H
L
INPUT<0:8> [0:3] /M
9B
VIS REG
100141
V0
I
T
CK
A2 .P3.8 &2
H
M
L
LA [0] /M
C 100102
O1
C 100102
A0
A /M
A4 .P0-4 L &HW
2
WIRE
OR
WO0
ADR
MPY
RAM
ARRAY
RAM
T
MPY UCODE A4 .490.5-3<0:43> /M

AMSEQ

3.10 MPY RAM ARRAY

Drawing: MRA

The MPY RAM ARRAYis a 256 word by 44 bit array. The macro MRA provides two things: the buffering necessary to drive the RAM array, and the means for the front end computer to load the microstore during the initialization of the machine.

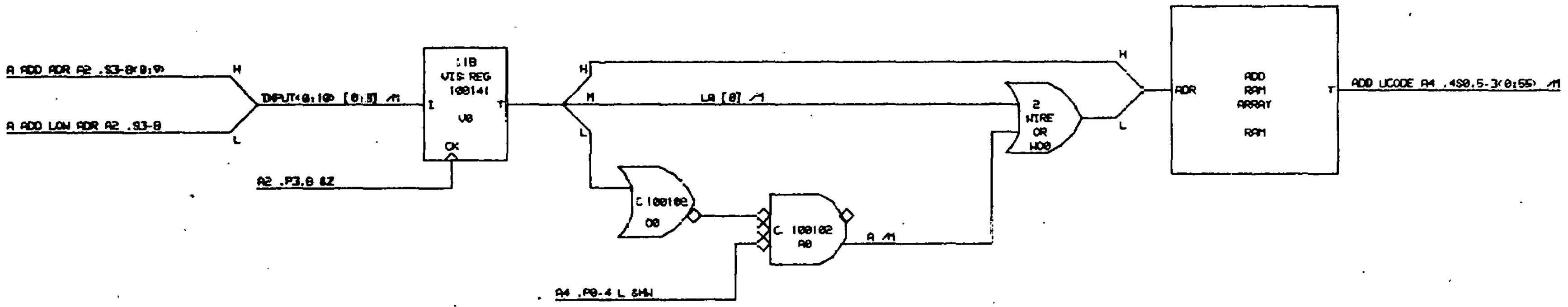


MPY RAM ARRAY

3.11 AASEQ

Drawing: AASEQ

AASEQ is the macro that defines the adder microstore and sequencer. It is identical to the multiplier microsequencer and microstore array except that the microstore is 1k deep and 56 bits wide.

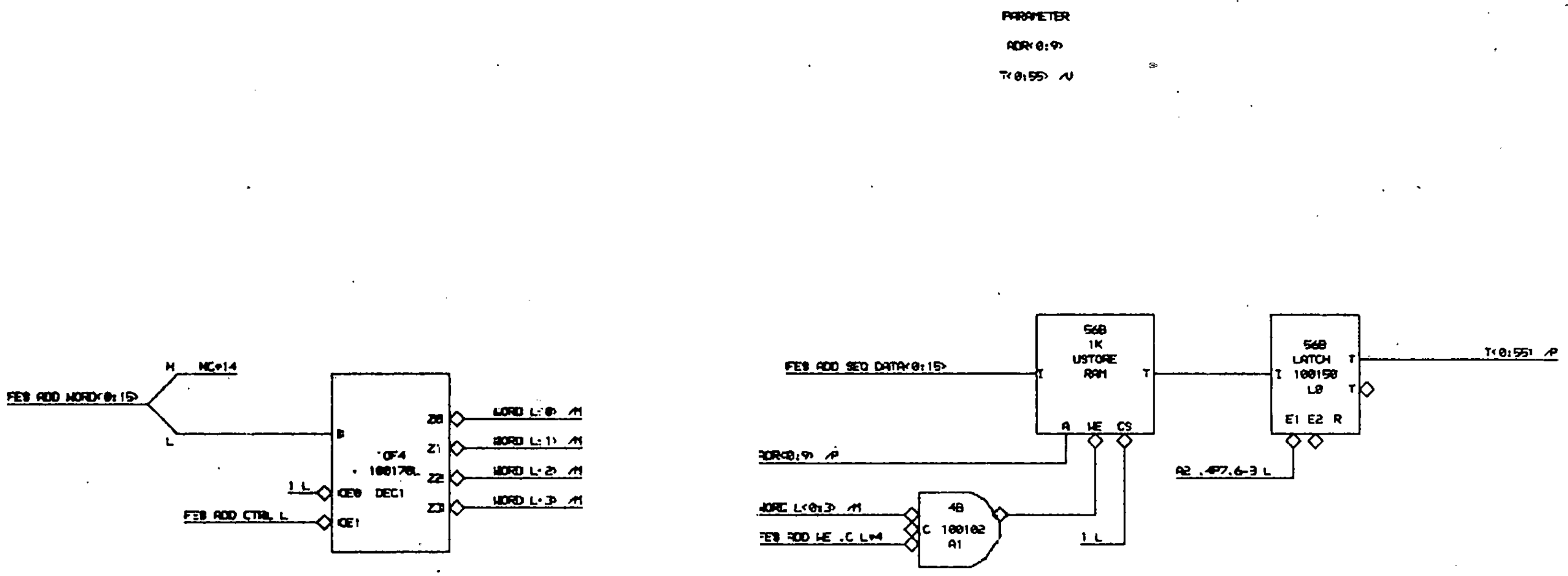

A ADD ADR A2 .S3-8<0:9>
A ADD LOW ADR A2 .S3-8
H
L
INPUT<0:10> [0:9] /M
I
:1B
VIS REG
100141
V0
T
CK
A2 .P3.8 &Z
H
M
L
LA [0] /M
C 100102
O0
C 100102
A0
A /M
A4 .P0-4 L &HW
2
WIRE
OR
W00
H
L
ADR
ADD
RAM
ARRAY
RAM
T
ADD UCODE A4 .4S0.5-3<0:55> /M

AASEQ

3.12 ADD RAM ARRAY

Drawing: ARA

This macro provides for the adder sequencer the function which the multiplier RAM array provides for the multiplier sequencer.



ADD RAM ARRAY

3.13 ABUS DEFINITION

Drawing: ABUS

Internally, the ABOX uses a number format different from that specified by the S-1 architecture. The internal format consists of two exponent fields (the second is used for complex numbers) a 72-bit fraction field and two tag fields. Integers are stored left adjusted, zero filled in the fraction field with exponents of zero. For integer half-word complex, the data is stored in the rightmost two half-words of the fraction. There is a uniform internal floating-point format with the binary point of the fraction being between bits 2 and 3. Most floating-point operations assume that the fraction is between 1 and 2, and all floating point operations produce such fractions. Exception cases (overflow, underflow, zero divide) are detected by the functional units; however, only the tags are modified to reflect this. The tags are propagated correctly (according to the current USER_STATUS modes) by use of RAM look-up tables in the functional units. The output unit decides what to do for the exception cases by looking at the tag or tags and at the current user status.

For integers the tags are:

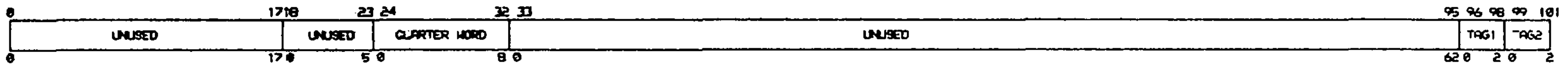
Tag	Use
0	Zero
2	Positive number
3	Positive overflow
4	Divide by zero
5	Negative overflow
6	Negative number

For floating-point the tags are:

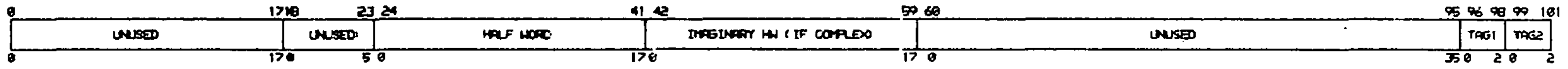
Tag	Use
0	Zero
1	Positive underflow
2	Positive number
3	Positive overflow
4	Not a number
5	Negative overflow
6	Negative number
7	Negative underflow

ABUS DEFINITION

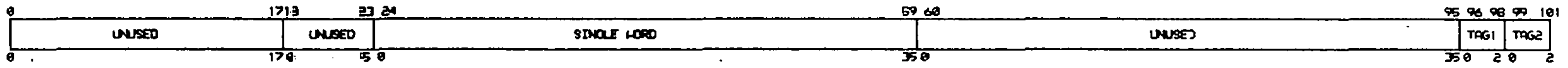
QW INTEGER FORMAT



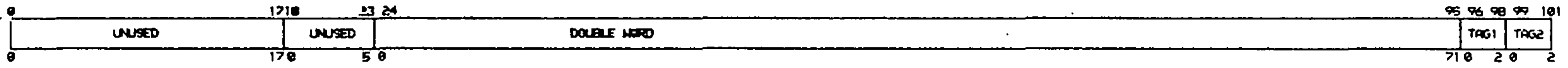
HW INTEGER FORMAT



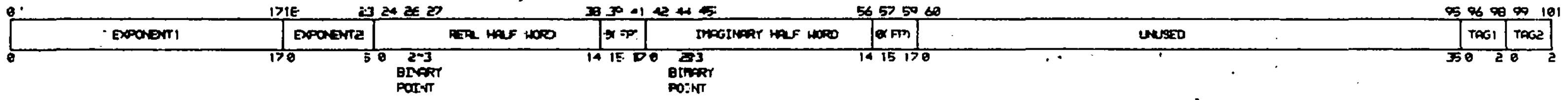
SW INTEGER FORMAT



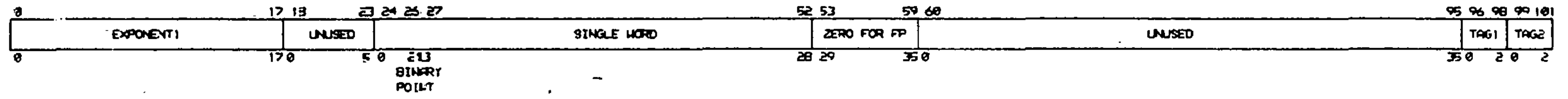
DW INTEGER FORMAT



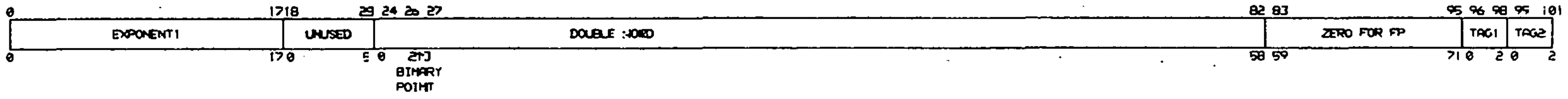
HW FP AND COMPLEX FORMAT



SW FLOATING POINT FORMAT



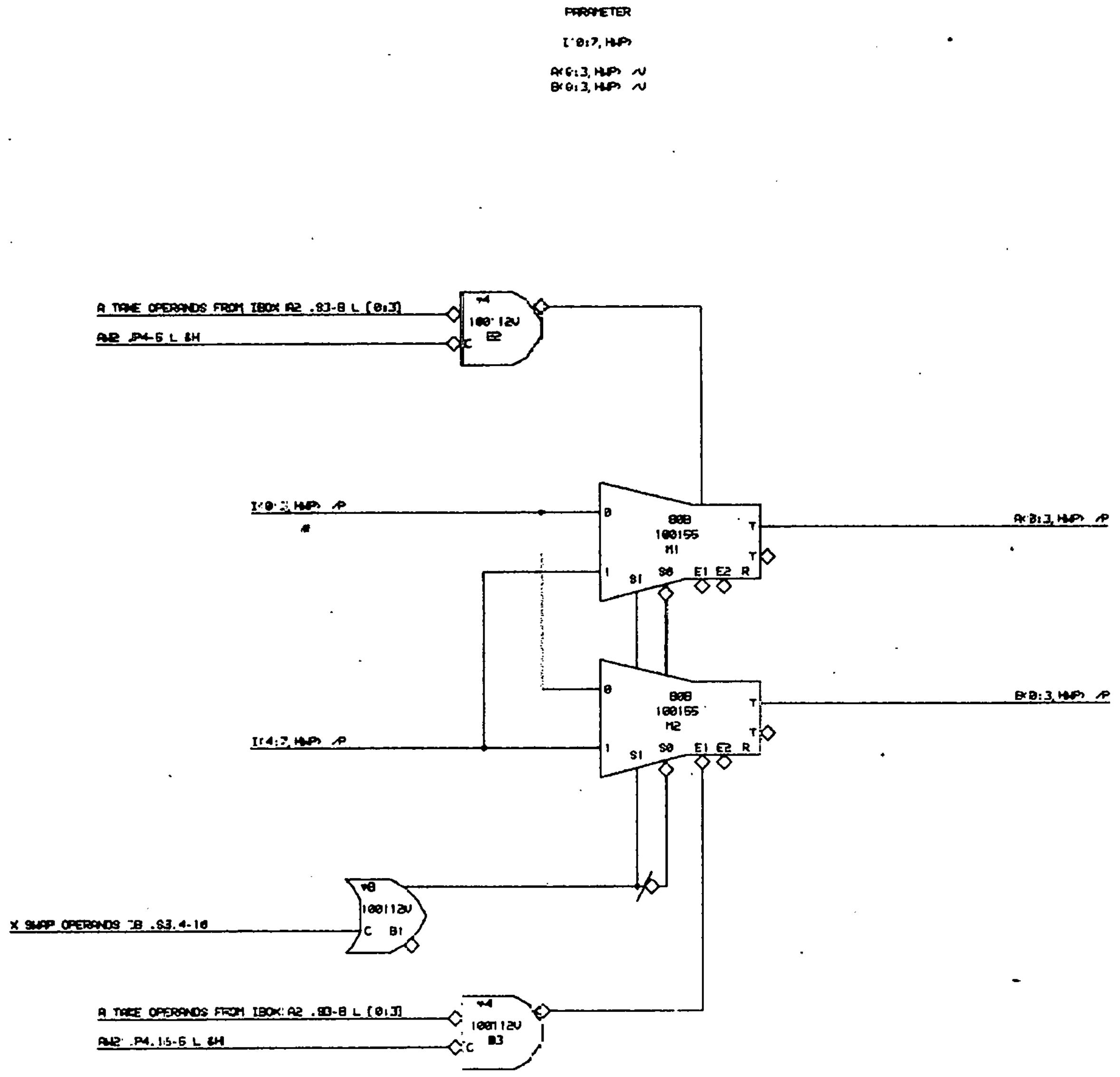
DW FLOATING POINT FORMAT



3.14 OPERAND SWAP BUFFER

Drawing: SWAPBF

To implement the reversed form of TOP intructions, this simply reverses the operands depending upon the signal X SWAP OPERANDS. It also latches the data.

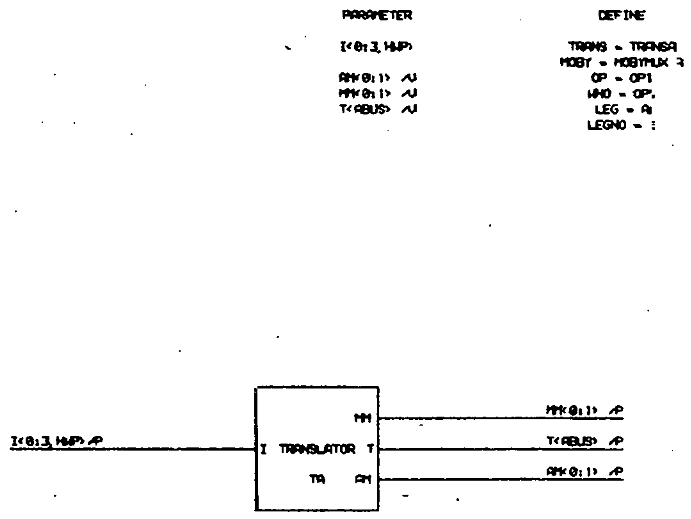


OPERAND SWAP BUFFER

3.15 TRANSLATOR A (and B)

Drawings: TRANSA, TRANSB

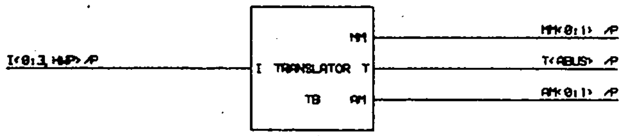
These set a text string TRANS to TRANSA (or TRANSB) and call the common macro TRAN.



TRANSLATOR A

```
PARAMETER                DEFINE

I<0:3,HWP>               TRANS = TRANSB
                         MOBY = MOBYMUX B
AM<0:1> ~/               OP = OP2
MM<0:1> ~/               WHO = OP2
T<ABUS> ~/               LEG = B
                         LEGNO = 2
```



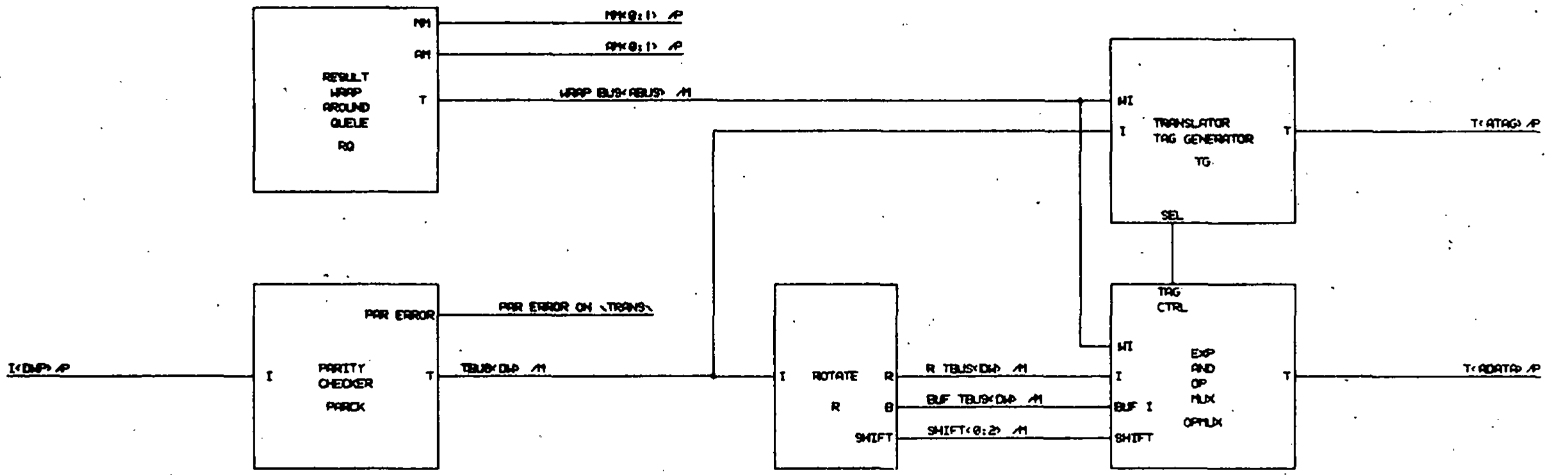
TRANSLATOR B

3.16 TRANSLATOR

Drawing: TRAN

The translator converts an operand (from the IBOX) from external to internal format. The external format is the architecturally defined format (the format the programmer sees). The internal format is defined for convenience and speed from the point of view of the hardware design. For instance, it extracts the exponent from floating-point numbers and puts fractions into a canonical form. The standard internal format for integers is right-adjusted. Also, the translator is responsible for wrap-around (called SHORT-STOP on CDC and Cray machines, this is the use of special data paths to get pending result operands from the ABOX internal state, rather than having to wait until they can come from the IBOX state). Thus the translator needs to keep a queue of PREVIOUS RESULTS (16 deep) and to remember what results are still in the process of being generated in one of the functional units. This queue is stored in internal format for maximum speed. The IBOX can tell the ABOX to use the *n*th previous result. If X WRAP {1,2} is set then X WRAP {1,2} RESULT NUM<0:3> indicates which result to use. Zero indicates the last result—that is, the last result which has come out of the ABOX or will eventually come out if no more instructions are executed—and 15 indicates the fifteenth previous result. Note that result are counted during vector operations even though they cannot wrap. Also, from the point of view of wrapping, the definition of a result is very precise: it is the the number of 4-word X RESULT DATA blocks given to the IBOX, regardless of how much data is stored in the blocks.

PARAMETER	COMMENT
I<DWP>	TRANSLATOR TIMES
	EXPONENT 14.15NS
AM<0:1> ∧/	DATA 11.05NS
MM<0:1> ∧/	TAG 19.35NS
T<ABUS> ∧/	

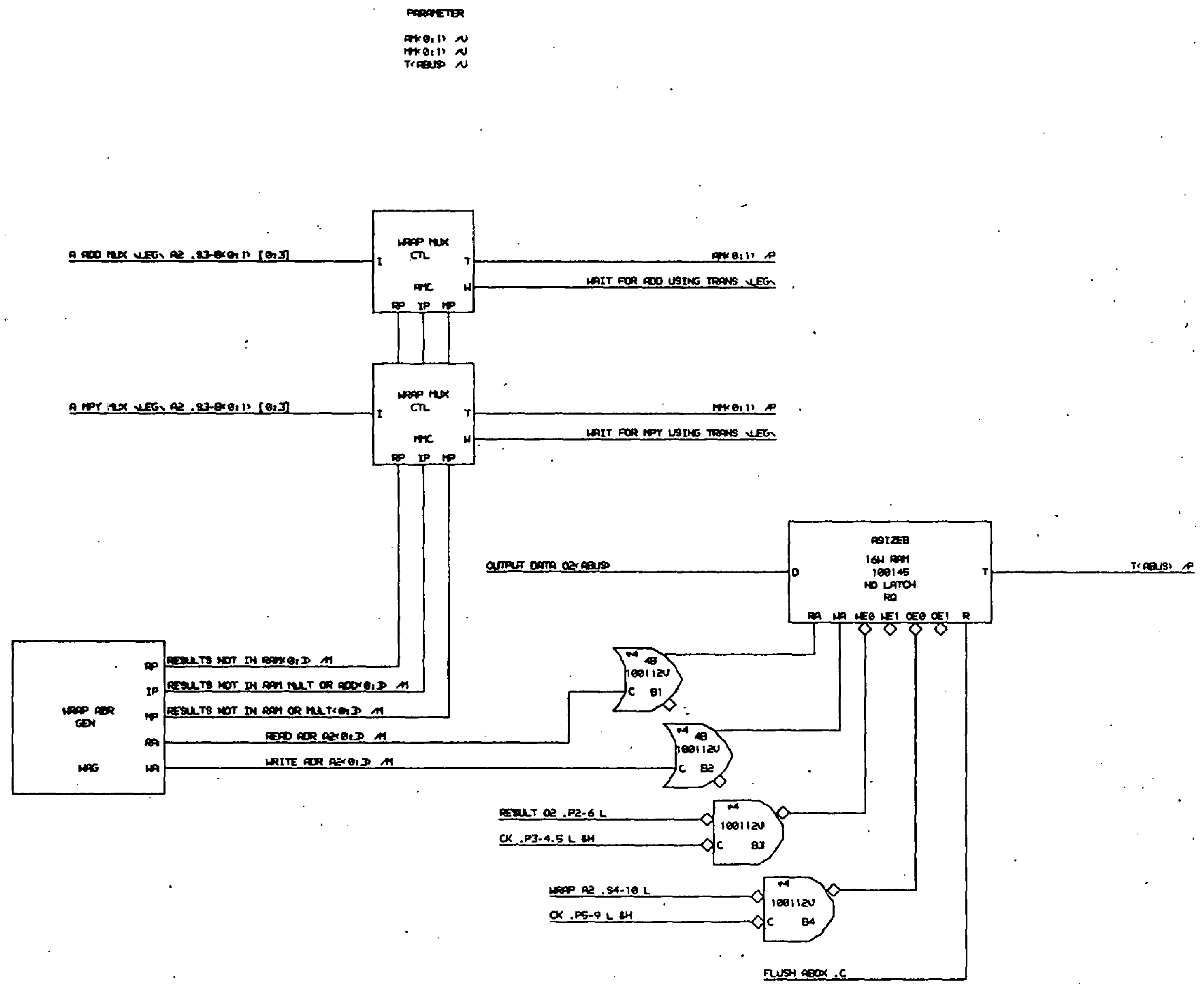


TRANSLATOR

3.17 RESULT WRAP AROUND QUEUE

Drawing: RWAQ

This implements the result wrapping as described in Section 3.15. A 16 by ABUS RAM stores the data. Every scalar result is written into this RAM (for vectors the garbage is written and the pointer is advanced). If X WRAP {1,2} is set then the RAM output is enabled and the read address will be set to the correct address by the WRAP ADR GEN. It is possible that the result is not yet in the RAM, in which case it is either on the output of a functional unit or is still in the process of being computed. The WRAP MUX CTL logic decides this for both the multiplier and the adder and sets the input multiplexers (and/or stop logic) accordingly.

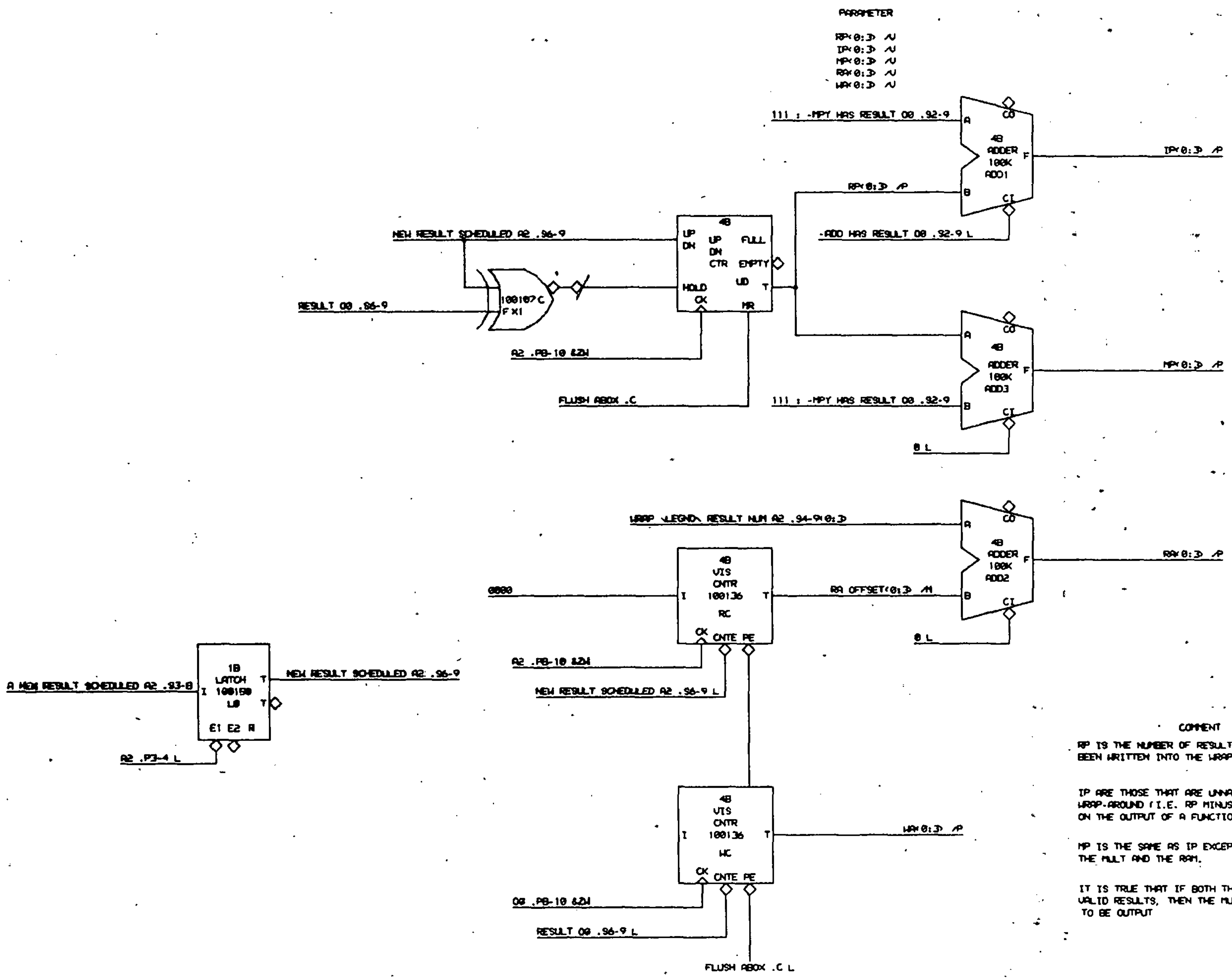


RESULT WRAP AROUND QUEUE

3.18 WRAP ADR GEN

Drawing: WRAPAG

The counter RC keeps track of the location of the "last" result in the wrap RAM. Thus it is simple to calculate the address of any previous result. To decide whether the result is in the RAM, on the output of a functional unit, or not yet available, there is a up/down counter to keep track of the number of results which have yet to be written into the RAM.



COMMENT

RP IS THE NUMBER OF RESULTS THAT HAVE NOT BEEN WRITTEN INTO THE WRAP RAM

IP ARE THOSE THAT ARE UNNAVAILABLE FOR WRAP-AROUND (I.E. RP MINUS THOSE THAT ARE CURRENTLY ON THE OUTPUT OF A FUNCTIONAL UNIT)

MP IS THE SAME AS IP EXCEPT THAT IS ONLY COUNTS THE MULT AND THE RAM.

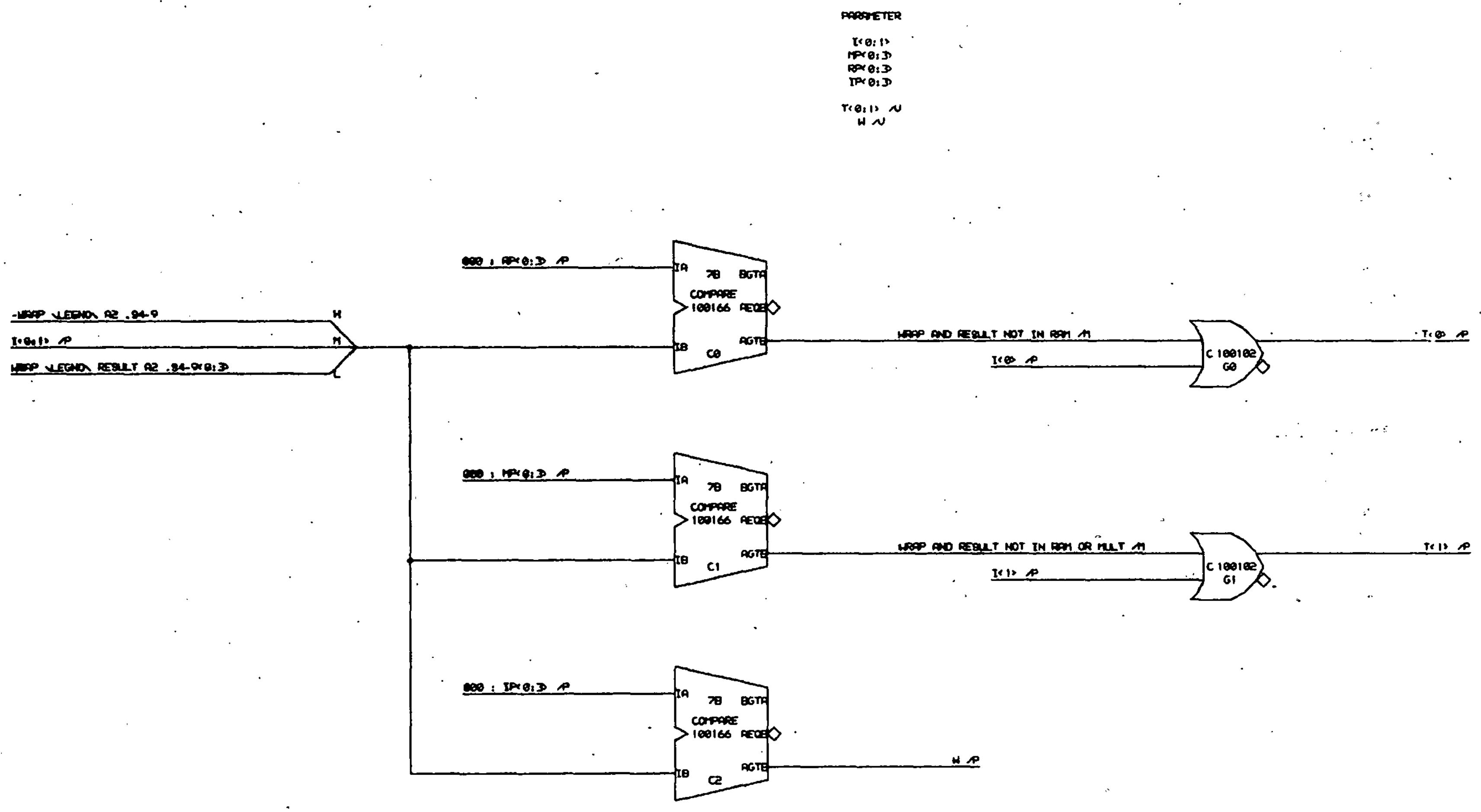
IT IS TRUE THAT IF BOTH THE MULT AND THE ADDER HAVE VALID RESULTS, THEN THE MULT IS ALWAYS FIRST TO BE OUTPUT

WRAP ADR GEN

3.19 WRAP MUX CTL

Drawing: WRAPMC

This generates signals to drive the select lines for the input multiplexers for the multiplier and the adder. If the microcode is selecting the translator and we are wrapping results, then we need to select the correct source to get the data there in the fastest possible way (or set WAIT if the data is not yet available.) If X WRAP {1,2} RESULT NUM is less than the number of results not yet in the RAM then we need to select either the adder or the multipler. Similiarly if X WRAP {1,2} RESULT NUM is less than the number of results not yet in the RAM or on the output of the multiplier, then we should select the adder. Finally if it is less than the number of results anywhere (in the RAM and on the output of the adder or multiplier) we should wait until the result is available.


PARAMETER
I<0:1>
MP<0:3>
RP<0:3>
IP<0:3>
T<0:1> ~
W ~
-WRAP \LEGND\ A2 .94-9
I<0:1> /P
WRAP \LEGND\ RESULT A2 .94-9<0:3>
H
M
L
000 : RP<0:3> /P
000 : MP<0:3> /P
000 : IP<0:3> /P
IA
IB
7B
COMPARE
100166
BGTA
AEQB
AGTB
C0
C1
C2
WRAP AND RESULT NOT IN RAM /M
WRAP AND RESULT NOT IN RAM OR MULT /M
I<0> /P
I<1> /P
C 100102
G0
G1
T<0> /P
T<1> /P
W /P

WRAP MUX CTL

3.20 PARITY CHECKER

Drawing: PARCK

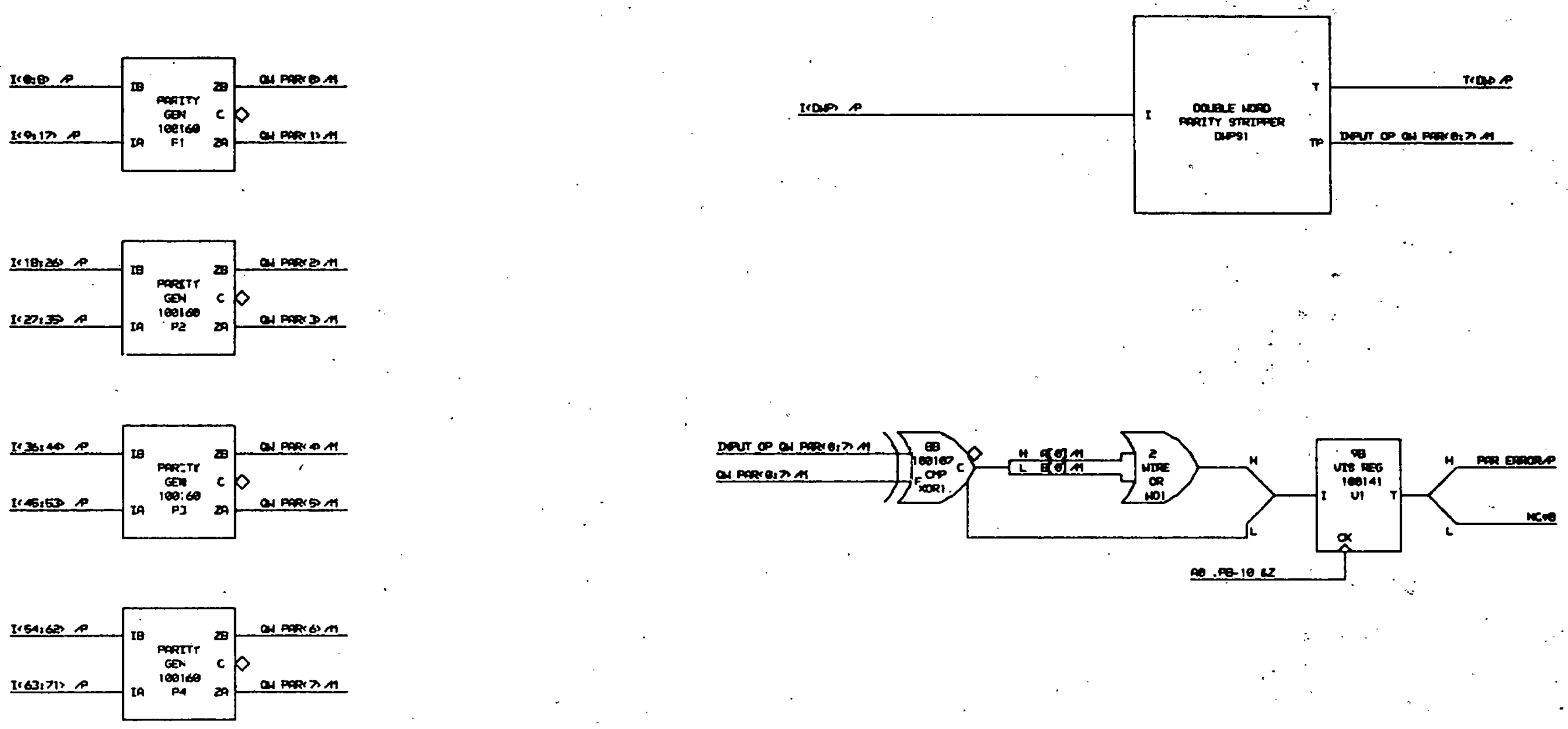
This strips off the parity put on X OP1 and X OP2 and sets some global error flags if the parity is incorrect.

PARAMETER

I<DWP>

T<DW>/J
PAR ERROR/#

COMMENT
IT IS ASSUMED THAT THERE IS
CORRECT PARITY EVEN WHEN
ONLY PART OF THE INPUT OP IS
VALID DATA.

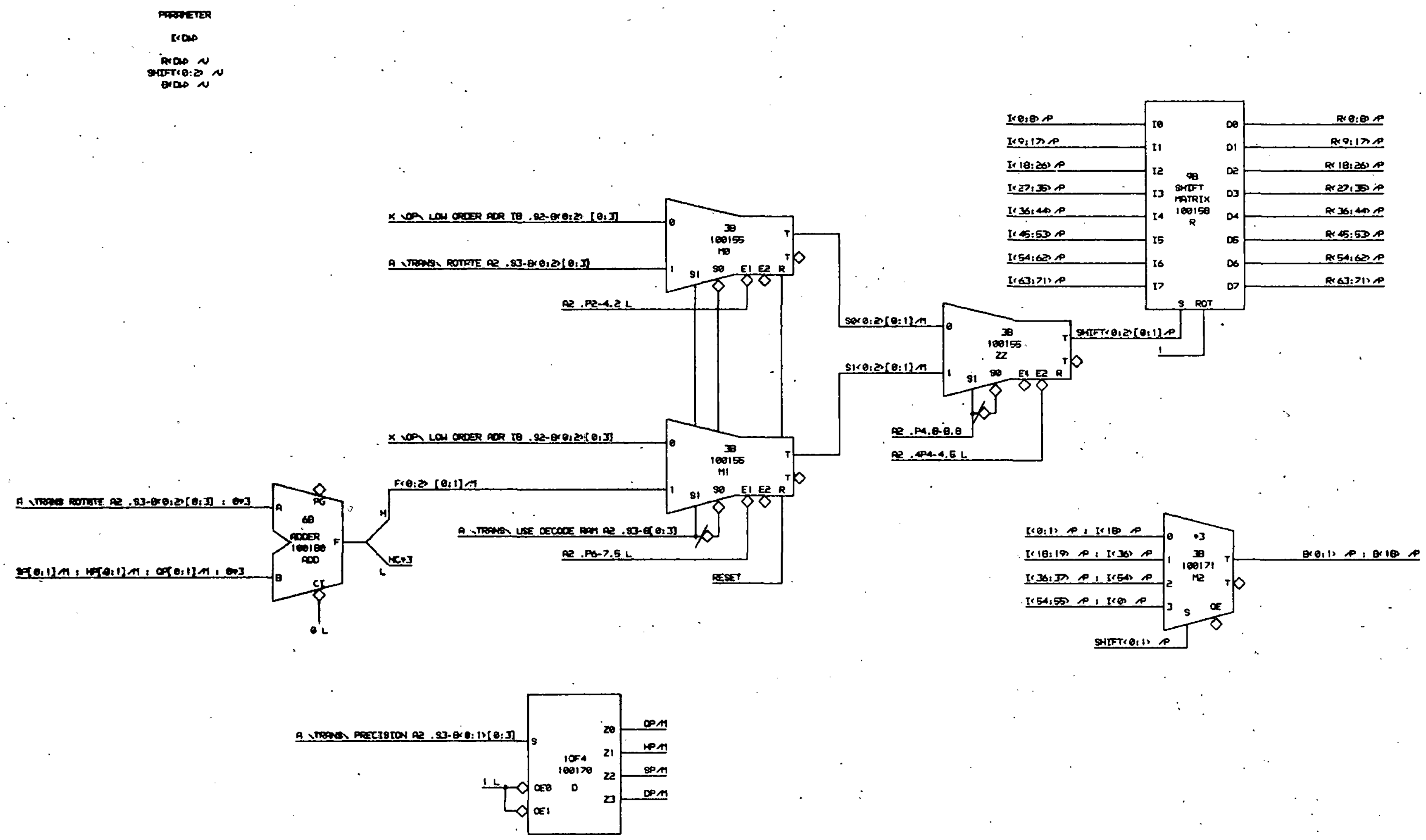


PARITY CHECKER

3.21 ROTATE

Drawing: ROT

This allows the translator to align the data such that any quarterword of an operand is at the right. For vectors, the second half-cycle can do a different rotation to select the "next" operand in a vector. The amount of rotation can come from the low order bit of the address (for scalars) or from ABOX top-level microcode. Scalar data is doubleword aligned coming into the rotator. For instance, the doubleword starting at quarterword address 4 comes in with the singlewords reversed, i.e. 8-11 in the leftmost word and 4-7 in the rightmost. Thus we need to reverse (rotate by 4) if bit 0 of X OP1 LOW ORDER ADR<0:2> is set. Similar considerations apply to quarterwords, halfwords, and singlewords.


PARAMETER
I<D>P
R<D>P ~
SHIFT<0:2> ~
B<D>P ~
X \OP\ LOW ORDER ADR IB .S2-B<0:2> [0:3]
A \TRANS\ ROTATE A2 .S3-B<0:2>[0:3]
3B 100155 M0
A2 .P2-4.2 L
X \OP\ LOW ORDER ADR IB .S2-B<0:2>[0:3]
3B 100155 M1
F<0:2> [0:1]/M
A \TRANS\ USE DECODE RAM A2 .S3-B[0:3]
A2 .P6-7.5 L
RESET
A \TRANS ROTATE A2 .S3-B<0:2>[0:3] : 0*3
SP[0:1]/M : HP[0:1]/M : QP[0:1]/M : 0*3
6B ADDER 100180 ADD
PG
CI
0 L
M
MC*3
L
S0<0:2>[0:1]/M
S1<0:2>[0:1]/M
3B 100155 ZZ
A2 .P4.8-8.8
A2 .4P4-4.5 L
SHIFT<0:2>[0:1]/P
1
I<0:8>/P
I<9:17>/P
I<18:26>/P
I<27:35>/P
I<36:44>/P
I<45:53>/P
I<54:62>/P
I<63:71>/P
9B SHIFT MATRIX 100158 R
S ROT
R<0:8>/P
R<9:17>/P
R<18:26>/P
R<27:35>/P
R<36:44>/P
R<45:53>/P
R<54:62>/P
R<63:71>/P
I<0:1> /P : I<18> /P
I<18:19> /P : I<36> /P
I<36:37> /P : I<54> /P
I<54:55> /P : I<0> /P
*3 3B 100171 M2
SHIFT<0:1> /P
B<0:1> /P : B<18> /P
A \TRANS\ PRECISION A2 .S3-B<0:1>[0:3]
1OF4 100170 D
1 L
OE0
OE1
QP/M
HP/M
SP/M
DP/M

ROTATE

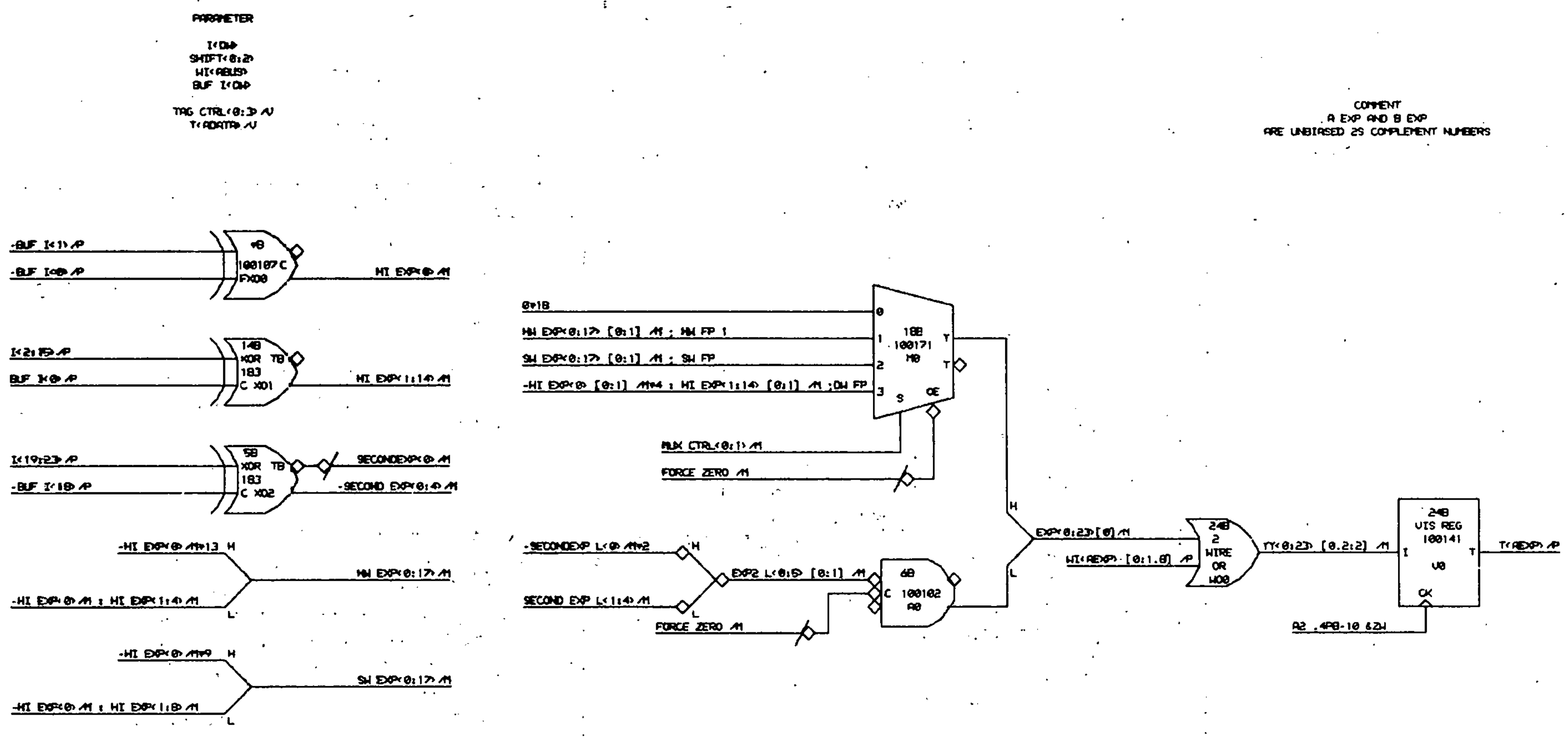
3.22 EXP AND OP MUX

Drawings: TMUX1, TMUX2, MUXCTR

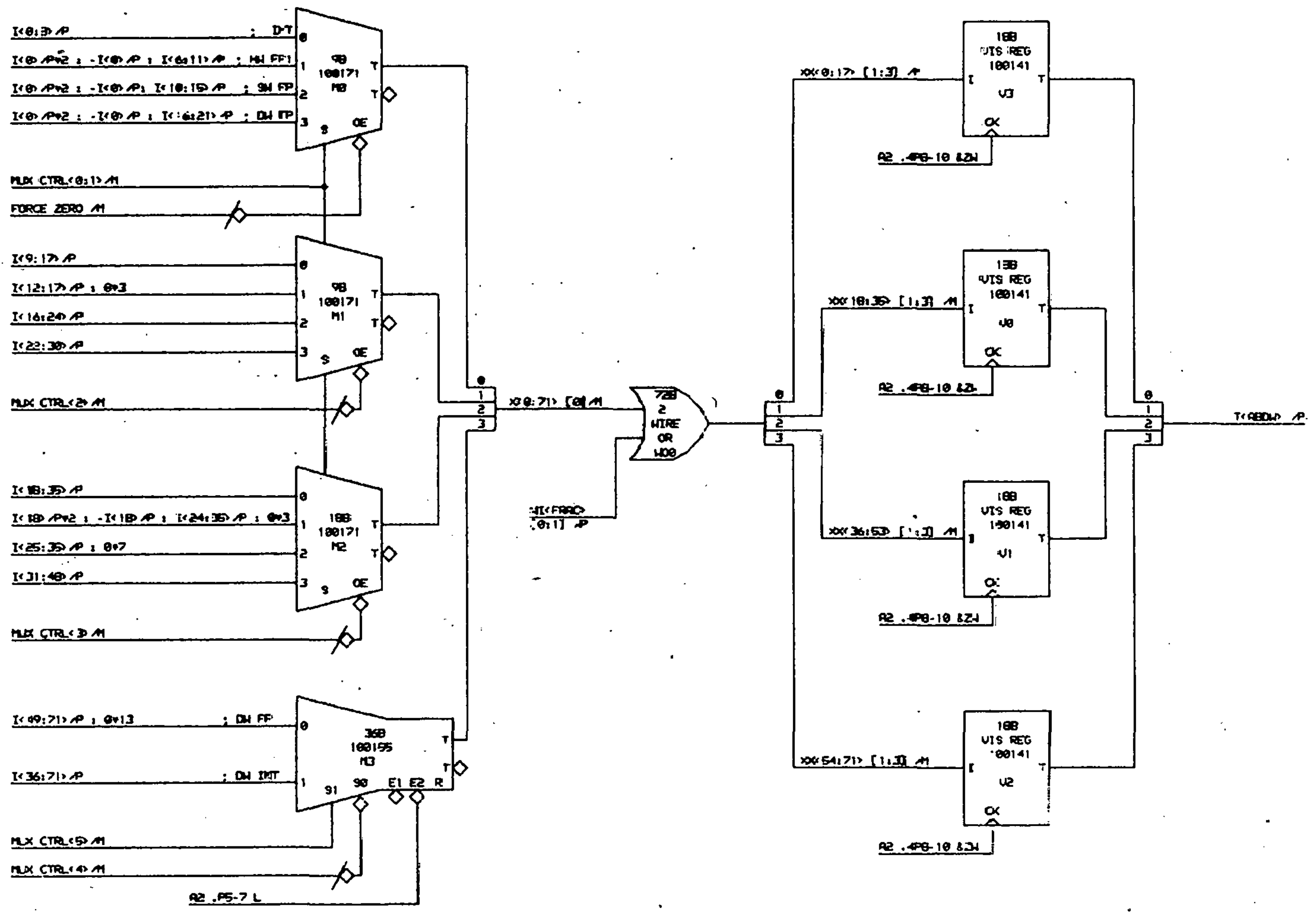
These macros do the unpacking necessary to convert to internal format. Since the precisions of integers are taken care of by the rotator, there are only four modes of unpacking:

1	Integer
2	Half-word floating point
3	Single-word floating point
4	Double-word floating point

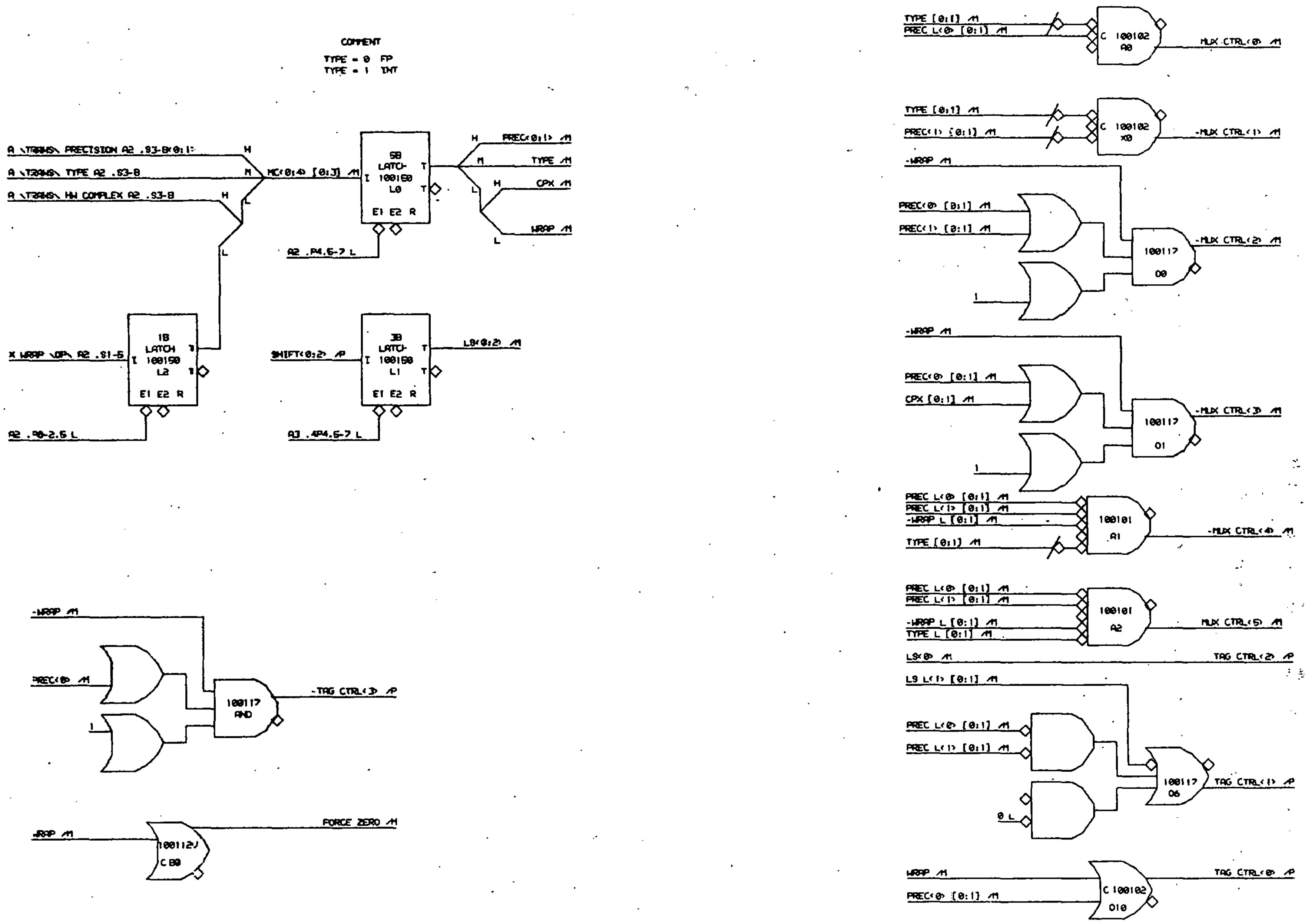
Halfword floating point always translates two halfwords into the halfword complex format. If the second halfword is not desired, the hardware later ignores it.



EXP AND OP MUX



EXP AND OP MUX

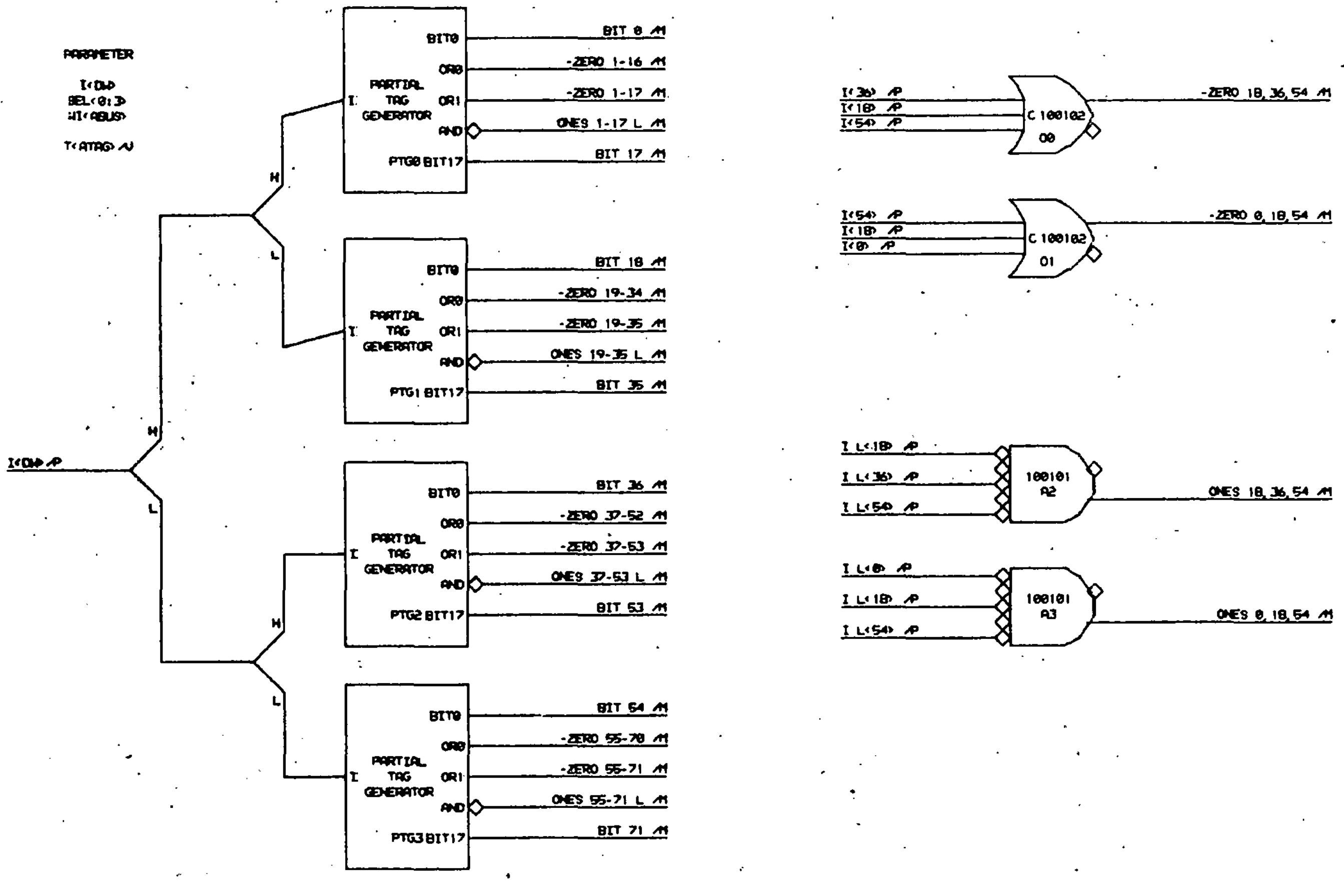


EXP AND OP MUX

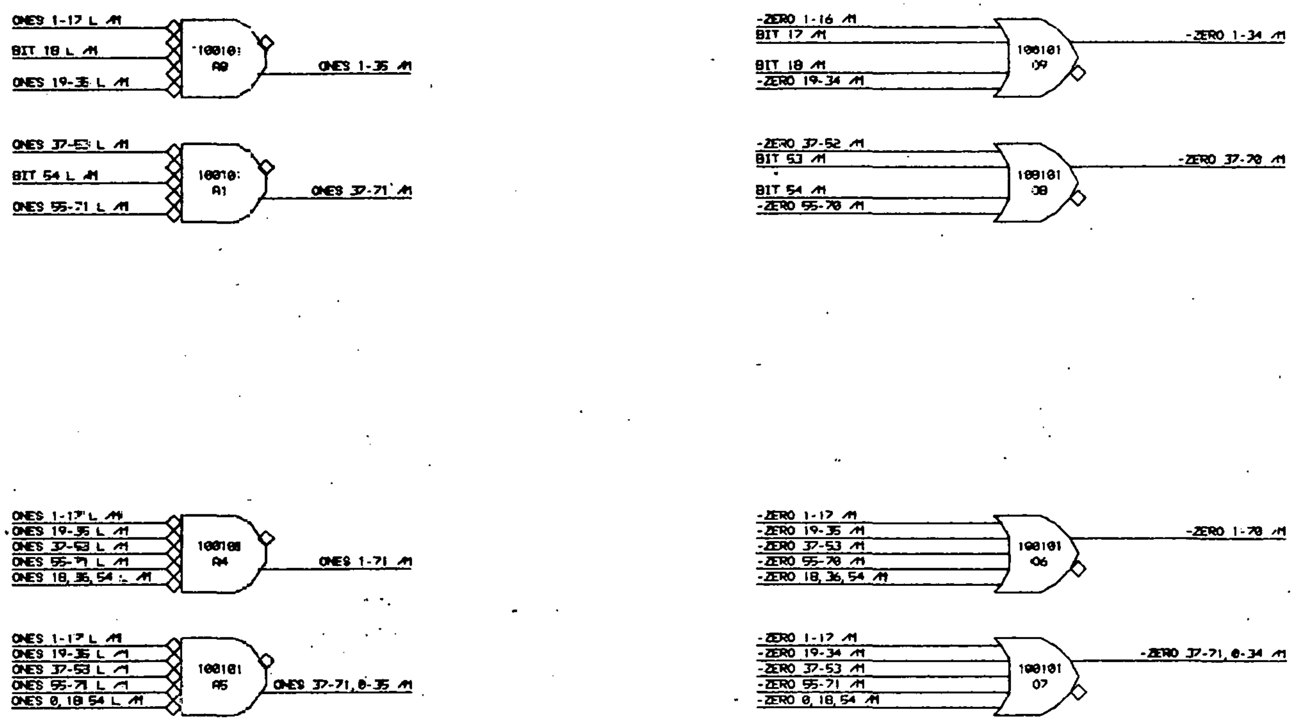
3.23 TRANSLATOR TAG GENERATOR

Drawings: TTGEN1, TTGEN2, TTGEN3, TTGEN4, PTGEN

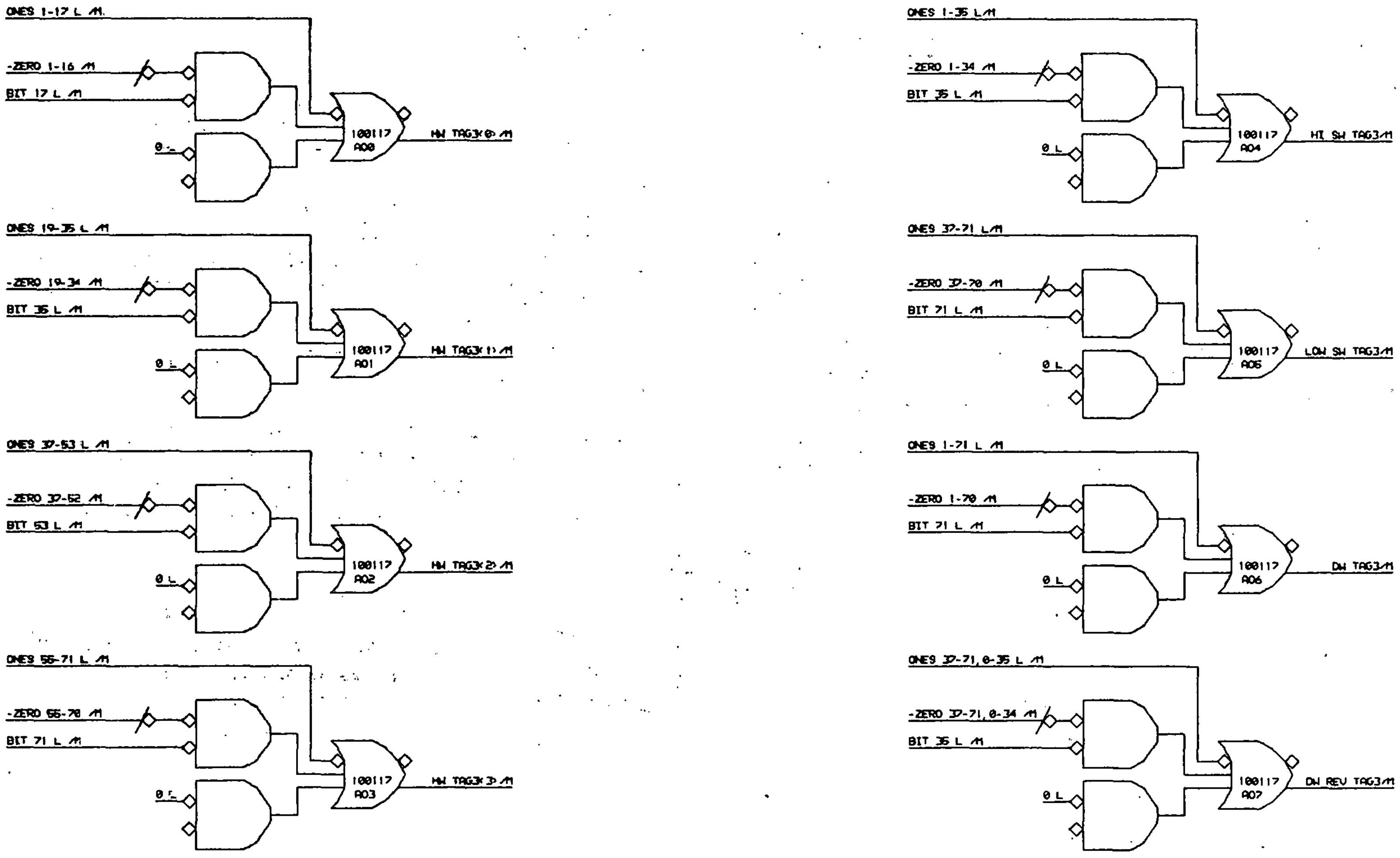
This generates the tags used by the internal format as described earlier.



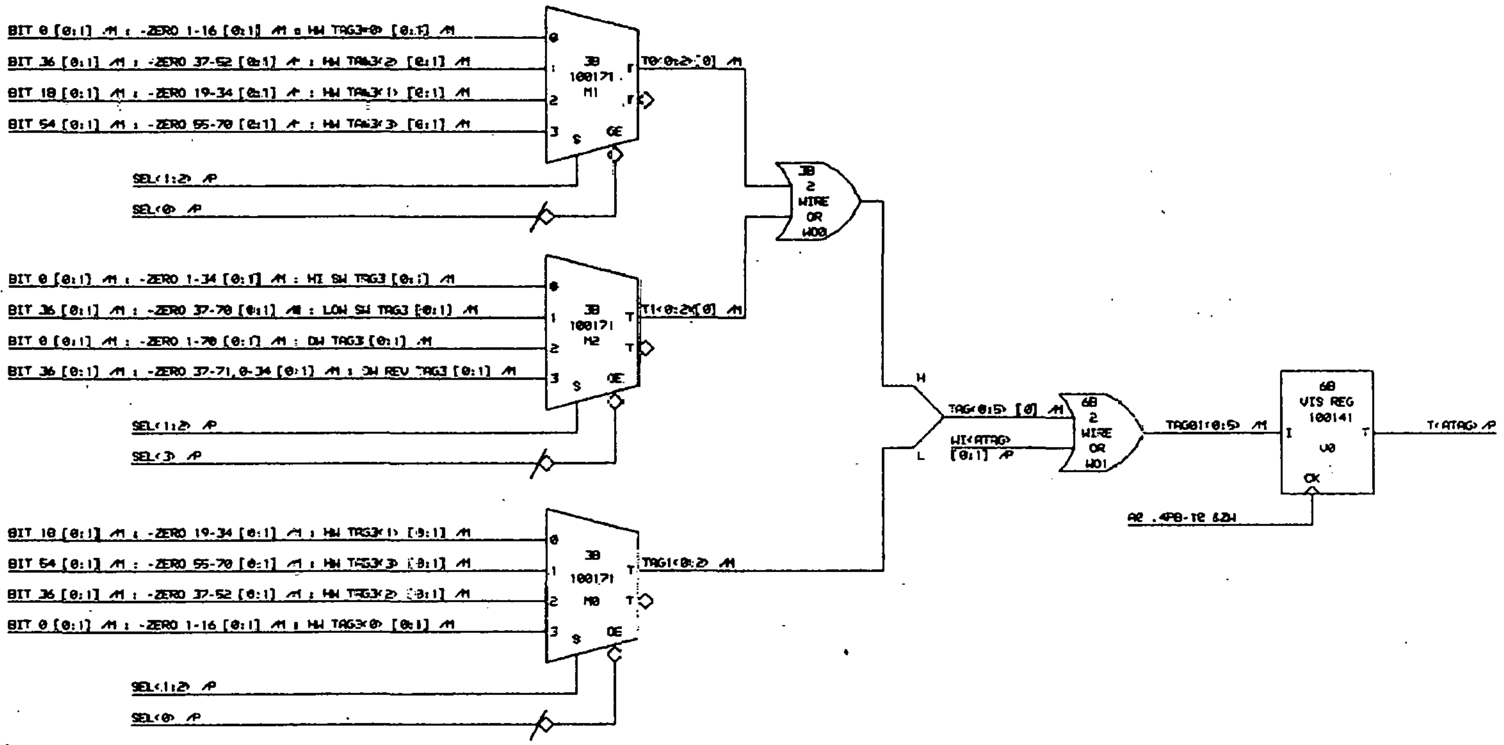
TRANSLATOR TAG GENERATOR



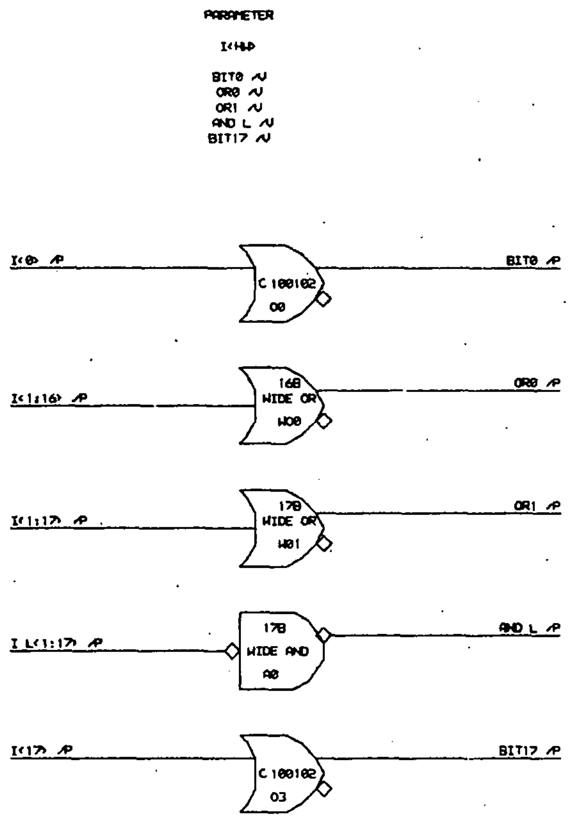
TRANSLATOR TAG GENERATOR



TRANSLATOR TAG GENERATOR


BIT 0 [0:1] /M : -ZERO 1-16 [0:1] /M : HW TAG3<0> [0:1] /M
BIT 36 [0:1] /M : -ZERO 37-52 [0:1] /M : HW TAG3<2> [0:1] /M
BIT 18 [0:1] /M : -ZERO 19-34 [0:1] /M : HW TAG3<1> [0:1] /M
BIT 54 [0:1] /M : -ZERO 55-70 [0:1] /M : HW TAG3<3> [0:1] /M
SEL<1:2> /P
SEL<0> /P
3B
100171
M1
T0<0:2>[0] /M
BIT 0 [0:1] /M : -ZERO 1-34 [0:1] /M : HI SW TAG3 [0:1] /M
BIT 36 [0:1] /M : -ZERO 37-70 [0:1] /M : LOW SW TAG3 [0:1] /M
BIT 0 [0:1] /M : -ZERO 1-70 [0:1] /M : DW TAG3 [0:1] /M
BIT 36 [0:1] /M : -ZERO 37-71,0-34 [0:1] /M : DW REV TAG3 [0:1] /M
SEL<1:2> /P
SEL<3> /P
3B
100171
M2
T1<0:2>[0] /M
BIT 18 [0:1] /M : -ZERO 19-34 [0:1] /M : HW TAG3<1> [0:1] /M
BIT 54 [0:1] /M : -ZERO 55-70 [0:1] /M : HW TAG3<3> [0:1] /M
BIT 36 [0:1] /M : -ZERO 37-52 [0:1] /M : HW TAG3<2> [0:1] /M
BIT 0 [0:1] /M : -ZERO 1-16 [0:1] /M : HW TAG3<0> [0:1] /M
SEL<1:2> /P
SEL<0> /P
3B
100171
M0
TAG1<0:2> /M
3B 2 WIRE OR W00
H
L
TAG<0:5> [0] /M
WI<ATAG> [0:1] /P
6B 2 WIRE OR W01
TAG01<0:5> /M
6B VIS REG 100141 V0
CK
AR .4P8-12 &2W
T<ATAG> /P

TRANSLATOR TAG GENERATOR



PARTIAL TAG GENERATOR

3.24 MOBY MUX, MOBY MUX A (and B)

Drawings: MM, MMA, MMB

MOBY MUX A (and B) set parameter strings and call the MOBY MUX macro.

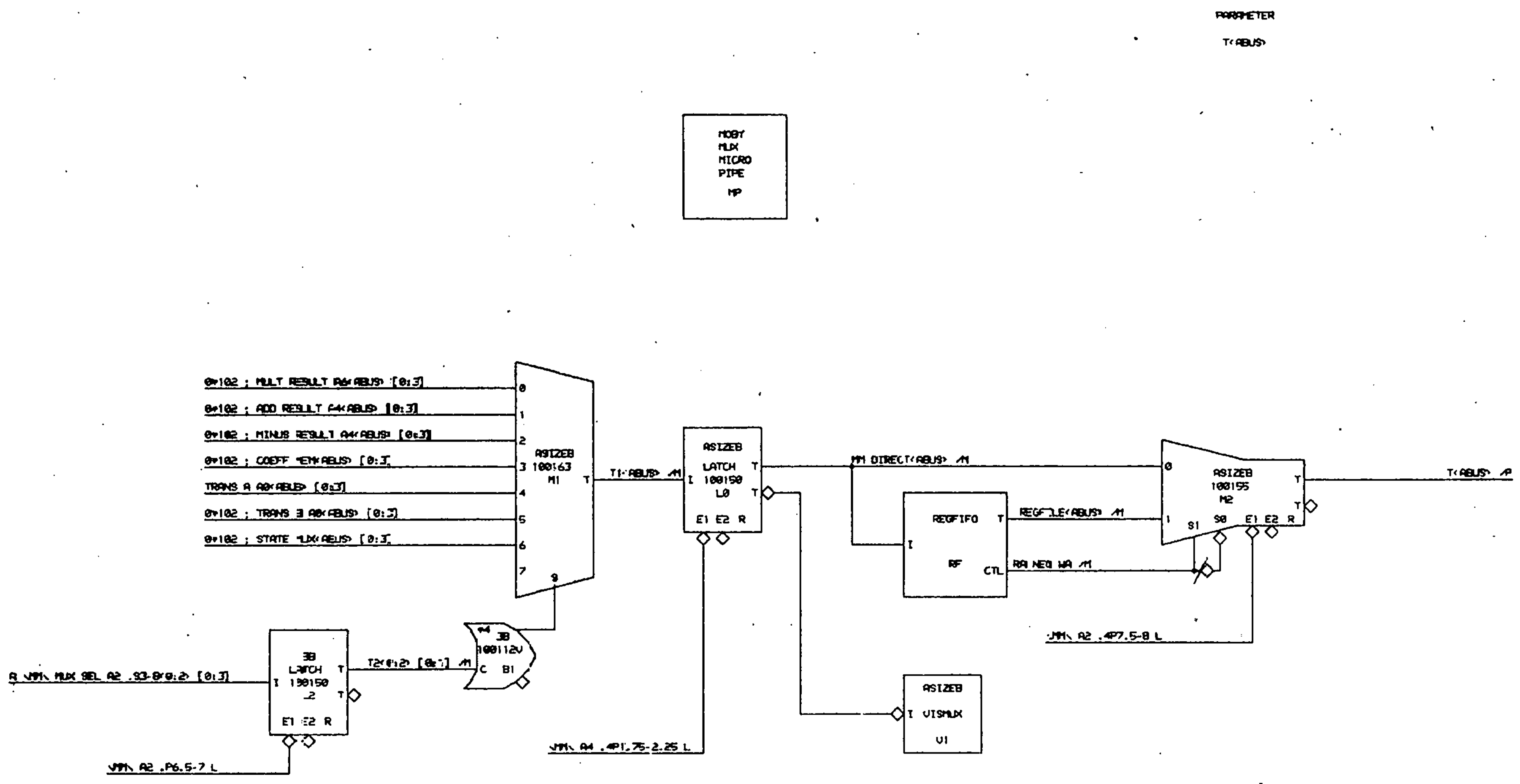
The MOBY MUX is a combination register file, constant table and giant multiplexer. The input multiplexer is capable of reading all of the functional unit results and most of the interesting fields in the ABOX. On every cycle, the ABOX can give a pair of read addresses and a pair of write addresses to the MOBY MUX. The pairs must be aligned, i.e. the possible reads (or writes) are:

Address First 25 ns.	Address second 25 ns.
2X+0	2X+1
2X+1	2X+0
2X+0	none (hold 2X+0)
2X+1	none (hold 2X+1)

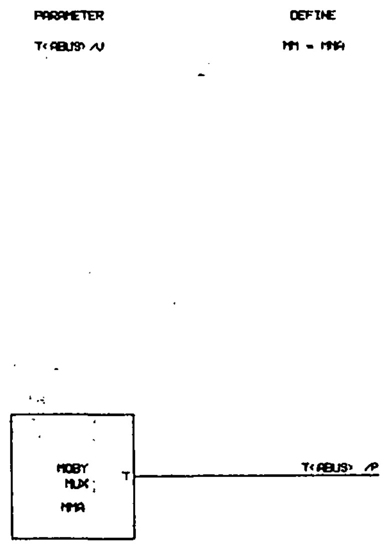
Read and write addresses are directly controlled by the top level microcode. If necessary, there is a path around the RAM so that writing one cycle and reading the next introduces no unnecessary delays (and/or incorrect results).

The MOBY MUX can be convenently used as a FIFO. There is a counter associated with both the read and write addresses. The value of the counter can be used instead of the top level register address. The counter is incremented every time it is used. A minor trick is that the register address from the top level microcode is ORed with the counter value. This allows any table in the upper part of the register file whose size is a power of two to be used for the FIFO.

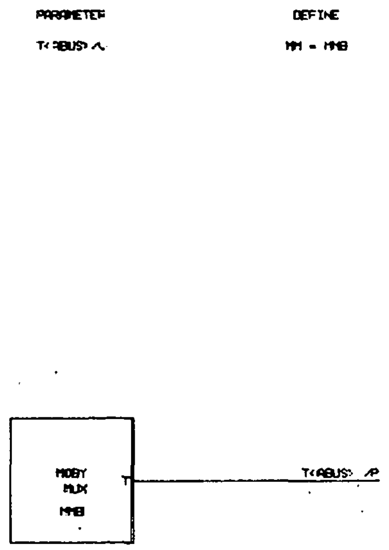
The read/write portion of the register file itself is 16 ABUS words deep. However, there is a read-only portion comprised of 256 additional words for use as constants. Which portion of the file is read is controlled by a top level microcode bit.


PARAMETER
T<ABUS>
MOBY MUX MICRO PIPE MP
0+102 ; MULT RESULT A6<ABUS> [0:3]
0+102 ; ADD RESULT A4<ABUS> [0:3]
0+102 ; MINUS RESULT A4<ABUS> [0:3]
0+102 ; COEFF MEM<ABUS> [0:3]
TRANS A A0<ABUS> [0:3]
0+102 ; TRANS B A0<ABUS> [0:3]
0+102 ; STATE MUX<ABUS> [0:3]
ASIZEB 100163 M1
T1<ABUS> /M
ASIZEB LATCH 100150 L0
MM DIRECT<ABUS> /M
REGFIFO RF
REGFILE<ABUS> /M
RA NEQ WA /M
ASIZEB 100155 M2
T<ABUS> /P
VMM A2 .4P7.5-8 L
ASIZEB VISMUX V1
VMM A4 .4P1.75-2.25 L
3B LATCH 100150 L2
R VMM MUX SEL A2 .S3-8<0:2> [0:3]
T2<0:2> [0:7] /M
3B 100112V B1
VMM A2 .P6.5-7 L

MOBY MUX



MOBY MUX A

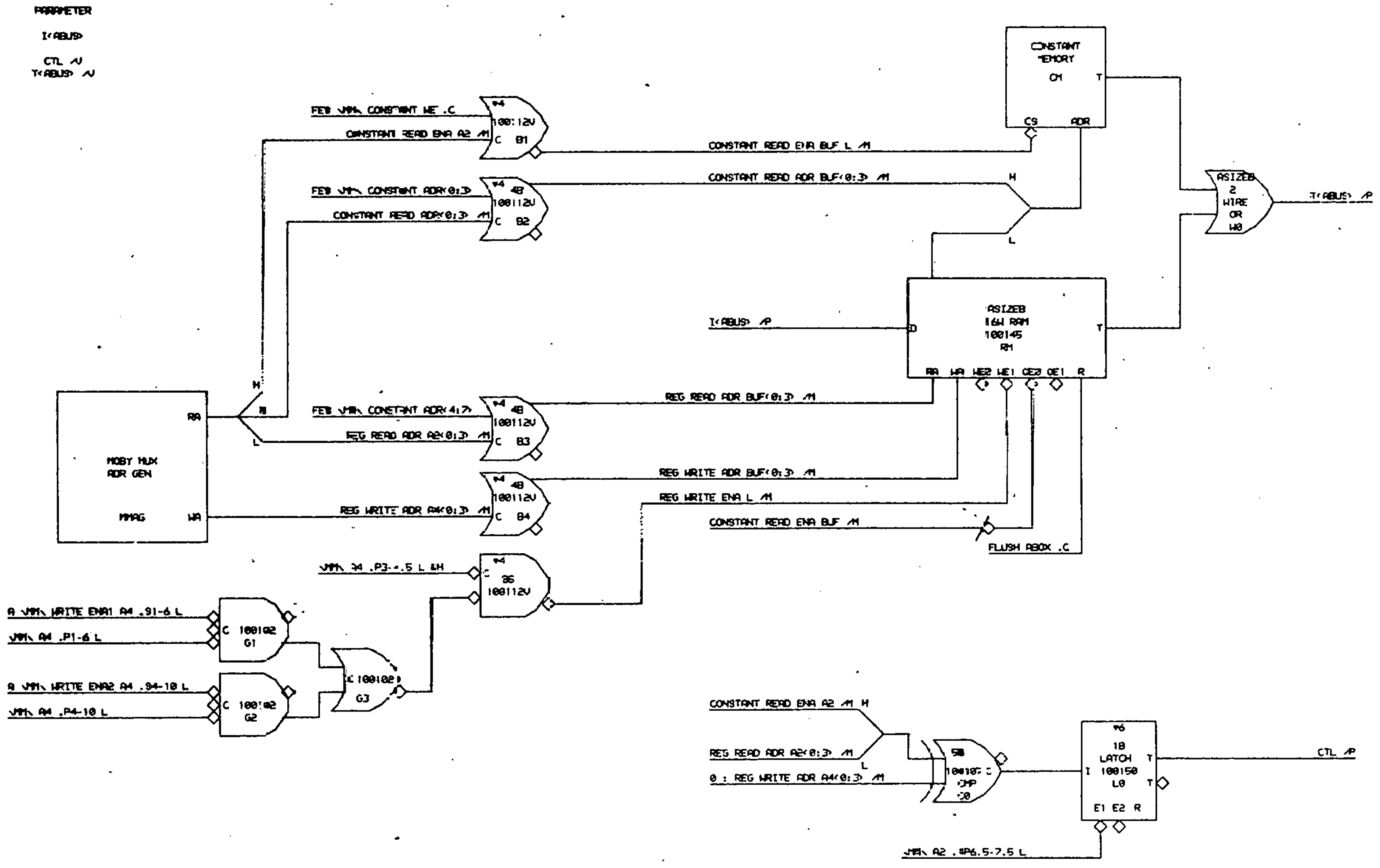


MOBY MUX B

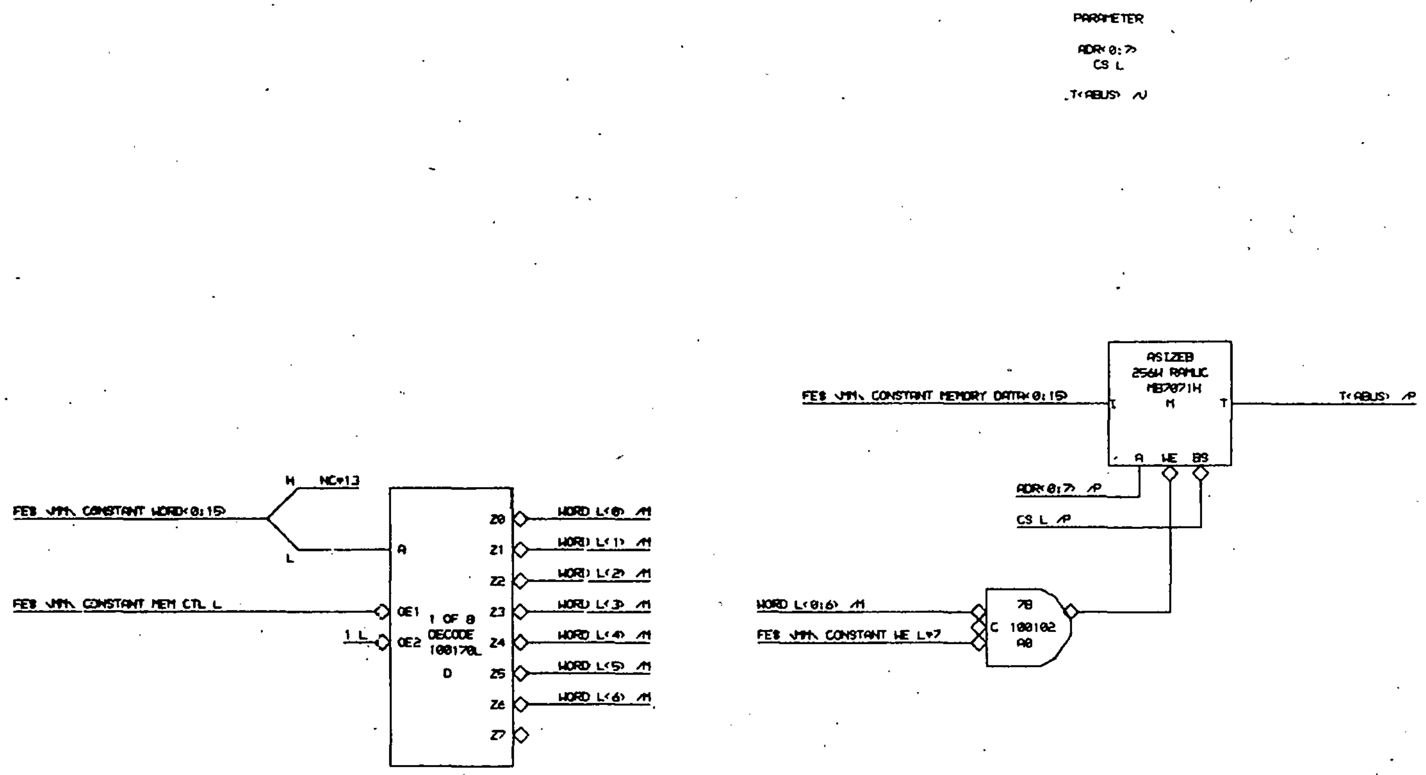
3.25 REGFIFO

Drawing: REGFIF, MMCM

This is the register file for a MOBY MUX. It has a 16 by ABUS RAM which can be both read and written every half-cycle as well as a constant RAM which can be read every half-cycle.


PARAMETER
I<ABUS>
CTL /V
T<ABUS> /V
CONSTANT MEMORY CM
CS ADR T
FE% VMA CONSTANT WE .C
CONSTANT READ ENA A2 /M
100112V B1
CONSTANT READ ENA BUF L /M
FE% VMA CONSTANT ADR(0:3)
CONSTANT READ ADR(0:3) /M
4B 100112V B2
CONSTANT READ ADR BUF(0:3) /M
ASIZEB 2 WIRE OR W0
T<ABUS> /P
I<ABUS> /P
ASIZEB 16W RAM 100145 RM
RA WA WE2 WE1 CE2 OE1 R
MOBY MUX ADR GEN
RA
MMAG WA
FE% VMA CONSTANT ADR(4:7)
REG READ ADR A2(0:3) /M
4B 100112V B3
REG READ ADR BUF(0:3) /M
REG WRITE ADR A4(0:3) /M
4B 100112V B4
REG WRITE ADR BUF(0:3) /M
REG WRITE ENA L /M
CONSTANT READ ENA BUF /M
FLUSH ABOX .C
VMA A4 .P3-4.5 L &H
B6 100112V
A VMA WRITE ENA1 A4 .91-6 L
VMA A4 .P1-6 L
100102 G1
A VMA WRITE ENA2 A4 .94-10 L
VMA A4 .P4-10 L
100102 G2
100102 G3
CONSTANT READ ENA A2 /M H
REG READ ADR A2(0:3) /M
0 : REG WRITE ADR A4(0:3) /M
5B 100107 CMP C0
1B LATCH 100150 L0
E1 E2 R
CTL /P
VMA A2 .%P6.5-7.5 L

REGFIFO

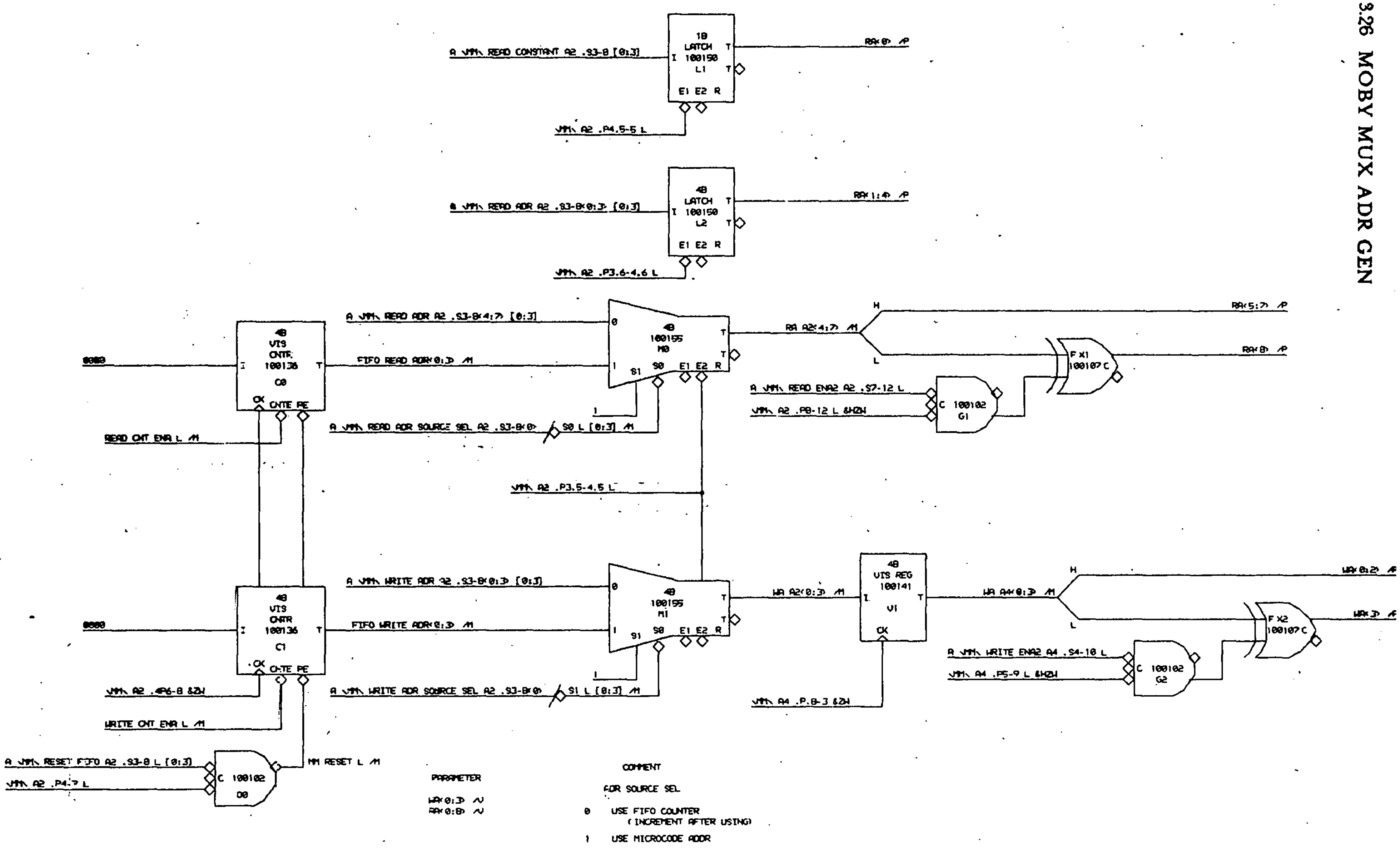


CONSTANT MEMORY

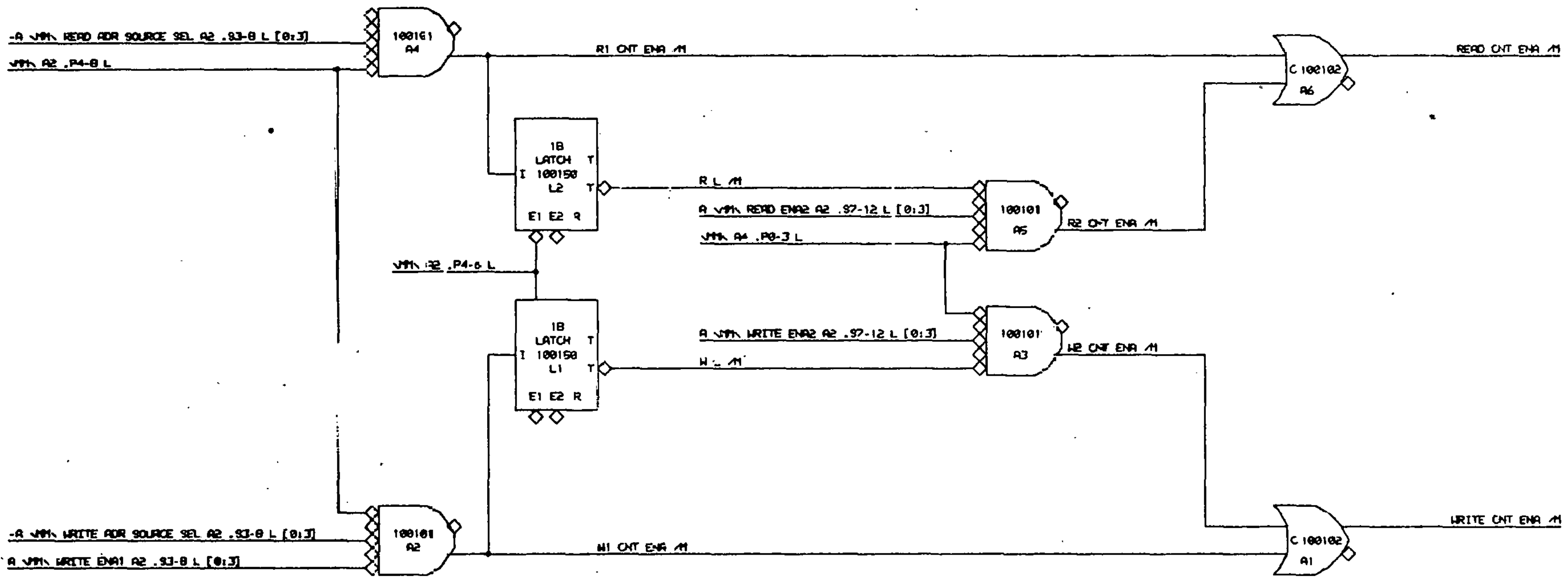
3.26 MOBY MUX ADR GEN

Drawing: MFIFO1, MFIFO2

This generates the addresses used by the REGFIFO. It contains the read and write counters necessary to use a MOBY MUX as a FIFO.

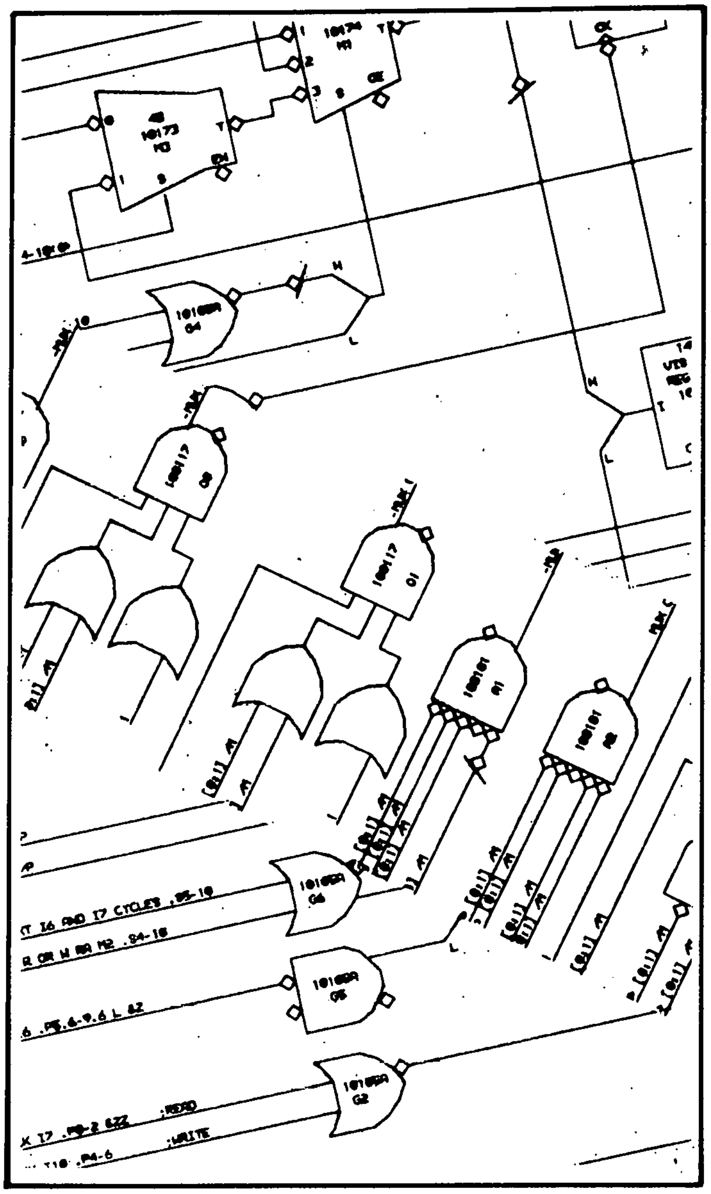


MOBY MUX ADR GEN



MOBY MUX ADR GEN

2



SCALD II User's Manual

(SCALD-2)

Thomas M. McWilliams, Jeffrey B. Rubin,
L. Curtis Widdoes, and Steven Correll

1 What SCALD does and why

SCALD (the acronym stands for "structured computer-aided logic design system") cuts the cost and time required to design logic. It does this by letting the logic designer express ideas as naturally as possible, and by eliminating as many errors as possible--through consistency checking, simulation, and timing verification--before the hardware is built.

This manual describes SCALD II, intended for use in the design of the S-1 Mark IIA processor. The original version, SCALD I, was used in the design of the S1 Mark I processor.

Designing hardware with SCALD is in many respects analogous to programming in a high level language. First, the designer uses a graphics editor to draw logic circuit diagrams on a CRT screen, just as a progammer would use a text editor to compose a source program. The diagrams form a hierarchy in which general, high level drawings are defined in terms of more specific, lower level drawings, just as the top level procedures in a well structured program call more specific, low level procedures. (Actually, each drawing represents a *macro* which can be replicated as often as necessary within the design.)

Then the designer feeds the drawings to the SCALD macro expander, which translates the *logical* design into a detailed *physical* design just as a compiler would translate source language into machine code. In the process, the macro expander can find many errors by checking syntax and design rules.

The designer then uses the SCALD layout programs and physical design programs to map the output of the macro expander onto actual circuit boards, just as a programmer uses a linker and loader to map the compiler-generated code onto the actual computer hardware. And, just as a programmer can use a symbolic debugger to find runtime errors, the designer can use the SCALD simulator and timing verifier to check the behavior of the hardware before building it.

Ultimately SCALD produces tapes and listings that permit assembly of a prototype either automatically or by hand.

SCALD brings to hardware the top-down design principles that programmers have adopted for software. At the top level of a well-structured program, a programmer does not deal with loops and branches and assignments, but with two or three procedures that divide the program's task logically into major subtasks. Similarly, at the top level of a digital circuit designed with SCALD, the designer does not deal with gates and signal polarity and fanout, but with two or three functional blocks that divide the circuit's task logically into major subtasks.

Each of the major blocks is then defined in terms of other blocks, and each of those in terms of still other blocks, and so on, forming the hierarchy. Successive definitions become increasingly specific, until finally the lowest level drawings correspond to actual integrated circuits. Those integrated circuits are themselves defined in terms of a few primitive logic elements--gates, flip-flops, multiplexers, adders, and so on—to permit simulation.

This approach to logic design has a number of advantages. (Readers who are already convinced should skip the following sales pitch and start with Section 1.1.) Some of them stem from the hierarchical structure, others from the basic use of a computer to automate the task, and still others from specific features in the SCALD programs.

Advantages of hierarchical structure--In hardware design as in programming, a top-down approach lets the mind tackle the most important and far-reaching questions first, deferring the rest. At any point, the designer confronts a manageable number of decisions. Structured design makes it easier to apportion work among a group of designers, since splitting the task into subtasks along functional lines provides a set of relatively independent chunks of work. Structured design makes it easier for a newcomer or outsider to understand the design by progressing from a general overview toward fine details.

Some advantages of top-down design apply uniquely to hardware. In a design requiring many individual drawings, structure reduces the confusion caused by wires running from one drawing to the next on the basis of paper size rather than meaning.

Further, structured designs are subject to less trauma as technology advances. The upper levels of the hierarchy tend to be general enough that they remain independent of the specific technology or logic family the designer uses. And as circuit packages come to hold increasing amounts of logic, the bottom level of the hierarchy may simply vanish because each frequently-used macro which was formerly defined in terms of a network of a dozen integrated circuits can be implemented with a single gate array chip.

Advantages of automation--Other advantages result simply because SCALD maintains the design on a computer in machine-readable form.

- It imposes uniform conventions on the design team.

- The computer's normal procedures handle mundane concerns like sharing drawings between designers, archiving old drawings, placing drawings in safekeeping, and so forth.

- The designs are readily available to programs for simulation, error-checking, cost estimating, parts counting, and so forth.

- Handling post-design changes by computer makes it more likely they'll be systematic and well documented.

Advantages of SCALD itself--The SCALD family of programs provides a number of specific services to make design easier.

- A timing verifier and logic simulator help test the design before constructing it.

- Semiautomatic layout and automatic routing speed construction.

- Extensive error-checking reduces the number of bugs before construction even begins. For example, SCALD checks the assertion level of signals against the expected inputs to each functional block; it finds a source for the inverse of a signal when needed; it lets the designer specify rules to handle fanout problems automatically; and it checks for undefined signals, unconnected signals, outputs tied together unintentionally, and undefined inputs.

1.1 The Structure of SCALD

SCALD itself is, as hinted earlier, a family of programs rather than a single program, making it easier to alter the system to suit different needs. For example, changes in the graphics input hardware affect only the graphics editor. Changes in the wiring technology employed affect only the packager programs.

For portability, all programs except the graphics editor and system-dependent utilities are written in PASCAL, and generally allow configuration for varying memory usage.

SCALD divides into a logical design system (programs which apply regardless of the technology used to implement the design) and a physical design system (programs which implement the logical design using a particular technology).

Important parts of the logical design system are:

> **D, the graphics editor**--This program lets the designer define macros by drawing networks of logic elements on a CRT display using a special keyboard or a light pen. One of its outputs is a file listing all the logic elements and the connections among them.
>
> **Macro expander**--This program takes in the logical design (a set of hierarchical macros defined by graphics editor drawings) and transforms it into the first stage of the physical design, outputting a set of actual IC functions and a list of the connections among them.
>
> Alternatively, for simulating the design before construction, this program can further expand the actual IC functions into the logical primitives which the simulator works with.
>
> **Logic simulator**--Using a typical value for the logic delays, this program simulates the design. In the case of a processor, it can even run small programs to check the processor's ability to execute various instructions.
>
> **Timing verifier**--This program takes into account a range of logic delays, from minimum to maximum, along with timing skews. It checks all the combinations of timing and signal paths necessary to assure that the design meets worst case timing constraints.
>
> Unlike the simulator, it does not fully simulate the network; it concerns itself with whether a signal is true or false only to the degree necessary to determine the interval within which that signal is stable. This division of labor between the simulator and verifier allows SCALD to assure a thorough simulation of large designs in a reasonable time.

Important parts of the physical design system are:

> **Layout**--Within constraints specified by the designer, this program automatically positions parts on circuit boards.
>
> **Packager**--This group of programs routes wires among parts on the circuit boards,

calculates waveforms of signals propagating along those wires, and manages post-design changes.

1.2 How to use this manual

This edition of the manual covers the graphics editor, macro expander, and layout program, but not the packager programs.

In some cases it gives two different views of the same material: a "How-to" section with a concise description, followed by a "Guided tour" through illustrative (that is, blatantly contrived) examples.

Installation-dependent information such as how to start a program running tends not to appear at all. Information of interest to those modifying the programs rather than to those using them, such as the formats of files, appears in appendices.

2 How to use D (the Graphics Editor)

This chapter is an abridged description of D, the graphics editor, describing a minimal subset of commands needed to create drawings for SCALD. For a description of many more commands, see the SUDS manual listed in Section 8.

2.1 Preliminaries

Terminology—We will assume use of the Stanford keyboard, which has keys labelled CONTROL, META, TOP, and SHIFT. This keyboard operates differently from that of either a typewriter or an ASCII computer terminal:

Pressing a key without holding any shift key gives the lower case version of the bottom symbol printed on the key.

Pressing a key while holding SHIFT gives the upper case (capital) version of the bottom symbol, *not* the top symbol as it would on a typewriter. If the symbol in question isn't a letter and thus can't be capitalized, then SHIFT has no effect.

Pressing a key while holding TOP gives the top symbol printed on the key.

Holding CONTROL or META in addition to some other combination of keys affects the *flavor* of the character but not its identity. For example, holding SHIFT changes "a" to "A"; but holding META in addition merely produces a special version of "A" which the program regards as a command, not some entirely new character. Generally, CONTROL gives the weaker or more ordinary version of that command while META gives a stronger or more exotic version.

We'll use the following notation throughout this manual:

α<character>	says to hold down CONTROL while pressing the <character>
β<character>	says to hold down META while pressing the <character>
αβ<character>	says to hold down both CONTROL and META while pressing the <character>

(The program never requires use of the characters "α" or "β" themselves, so there's no danger of confusion; throughout this document they always represent the CONTROL and META keys.)

When using D, latch the SHIFT LOCK key down to avoid having to lean on the SHIFT key constantly, thus freeing all ten fingers to manipulate SHIFT, TOP, CONTROL, and META.

Whenever the program expects a multiple character string--the name of a file or logic element, for example--it permits use of the DEL key to backspace and erase mistakes.

The commands that consist of a character with META and/or CONTROL held down will, however, execute immediately, giving you no chance to use DEL. If, as a result of the command, the program then prompts for additional information, the ALT key will generally abort the command; otherwise, you must simply figure out a way to undo the results of the command.

Files, libraries, and bodies--The program stores drawings in files with names of the form "<name>.DRW". At the top of the screen, it constantly displays an equation "∃=<name>" which tells the name of the file (if any) that it is currently editing.

In the most general sense, the program can do two things: it can develop templates for "bodies", and it can draw circuits by first drawing bodies based on those templates and then connecting lines between those bodies. For SCALD's purposes, a body generally represents either a macro or a logic primitive such as a gate or adder. A drawing generally defines a macro in terms of additional bodies connected together.

The templates for bodies hide in the background until the designer either uses a template to place a body in a drawing or enters a special mode (Edit mode) capable of creating or modifying body templates.

When editing a drawing, the program operates on a copy of the drawing in a special area called a workspace. A particular drawing file can be copied into the workspace in three distinct ways:

1. The first drawing file copied in after clearing the workspace becomes the one named in the "∃=" line.

2. (One rarely uses this feature.) If you copy in any additional drawing files, their body templates are added to the repertoire of templates in the workspace, and their drawings become sets of elements superimposed on your existing drawing, just like the sets you yourself can create as described in Section 2.5. You may then move those sets around and

add them to the existing drawing.

3. If you bring in a drawing file as a *library*, its body templates are added to the repertoire of templates in the workspace, but its drawings (if any) are not used. There is nothing special about a file used to hold libraries of body templates; any drawing can look like a library if brought in as one.

Other files--The program can produce two other files corresponding to "<name>.DRW": "<name>.PLT" is useful for making a paper copy of the drawing, and "<name>.WD" is a list of bodies and interconnections which ultimately becomes the input to the SCALD macro expander.

Moving the cursor--The program will always display (though sometimes at the very edge of the screen) a set of crosshairs which serves as the *cursor*. Four keys above the RETURN key move it incrementally to the left, to the right, up, and down:

(	Left
)	Right
/	Up
\	Down

(On some keyboards, the sequence is "[ ] \ /" instead; in any case, no matter what is marked on them, use the four keys immediately above the RETURN key, and associate the directions with fingers rather than with the markings on the keys.)

These are obviously intended to be convenient, not mnemonic; since you will probably use them more heavily than any others, it's easy to become accustomed to placing four fingers over them without looking at the keyboard or thinking about the symbols on the keys.

Holding down various shift keys multiplies the distance these keys cause the cursor to move:

CONTROL	x2
META	x4
TOP	x16

Using several shift keys at once multiplies the factors. Holding down both CONTROL and META, for example, multiplies the fundamental cursor motion by 8.

Enlargement, reduction, and moving the paper--When the program starts, it shows a x16 enlargement of the "paper" it will draw on. That is a convenient scale for seeing everything clearly, but the entire paper will not fit on the screen at once, so the screen acts as a sort of "window" through which you view the drawing.

To move the paper to see a different part of it through this window, use the "↑", "↓", "→", and "←" characters. Typing such a character once moves the drawing by 1/8 of the window dimension. As with the cursor, the CONTROL key multiplies this motion by 2 and the META key by 4. (The

TOP key isn't available as a multiplier in this case, because it is needed to obtain any of those characters in the first place.)

As the paper moves, the cursor sticks to it until the cursor hits the edge of the screen. You can continue to move the paper further in that direction, but the cursor will remain at the edge of the screen until you move the paper in a different direction. If you move the paper far enough, you'll reach its edge and see a line representing the perimeter (assuming the SHOWBOX feature described in Section 2.2.2 is enabled).

You can also reduce and enlarge the paper to see more or less of it through the screen, but this is inconvenient because the text remains the same size while the bodies and lines shrink and grow. The "*" key reduces, the "⊗" key enlarges, and once again CONTROL and META multiply the effect.

If part of the drawing spills off the edge of the paper, the "X PICCEN" command (Section 2.2.2) will recenter it, but the system automatically recenters the drawing anyway just before plotting a hard copy of it. If the centered drawing won't fit on the paper, however, the plot will clip it at the edges.

Modes--At any time, the program is in one of several major modes, each of which may have one or more submodes. Only certain modes and submodes are essential to drawing circuits for SCALD. To describe bodies one uses edit mode; to draw circuits with them one switches back and forth among body mode, point mode, and set mode. The top line of the screen will always contain "MODE=" followed by one or two letters. The first letter generally tells the current mode and the second the current submode.

The program begins in body mode. The following commands change back and forth among body, point, and set modes:

αβB	Select body mode (MODE=B)
αβP	Select point mode (MODE=P)
αβS	Select set mode (MODE=S)

Many commands work equally well in any mode, others don't, and still others mean slightly different things in different modes. Unless noted otherwise, assume the commands given here work in any mode.

Attaching the cursor and moving objects--When the program enters body, point, or set mode, the cursor is detached from all objects so it may move at will without affecting the drawing. The program will superimpose a large flashing letter on the object of the appropriate type--a body if it is in body mode, a point in point mode, or a set in set mode--which is closest to the cursor. *Attaching* the cursor to that object forces the object to follow wherever the cursor moves. The following commands accomplish this:

αM Move the cursor to the object and attach the cursor to it. The large flashing

	letter identifying the object will vanish.
α/βM	Move the object to the cursor and attach the cursor to it. The large flashing letter identifying the object will vanish.
<SPACE>	Detach the cursor from the object. The large flashing letter identifying the object will reappear.

It's not critical to understand them, but the large flashing letters do convey meanings:

B	Body
P	Point
PL	Point with line(s)
TL	Line (and usually a point, too) with text
PA	Point to which you may attach a line. On a body, this is usually an input or output pin
PLA	Same as PA, but there's already a line attached
BT	Body text
BTP	Body property name/text pair

Extended commands--A special set of commands beginning with "X" sets options and performs functions without regard to mode. When you see a command in this chapter described like this:

```
X CLEAR
```

it means that after you type "X" the program will type "WELL?" and wait for you to put in the remainder of the command ("CLEAR" in that example). Since operating systems supporting the graphics editor generally offer "typeahead" (that is, they will save up characters if you happen to produce them faster than a program can use them), it's usually safe to type the entire string without waiting for the "WELL?".

In addition, you may combine these extended commands. The following example

```
X
WELL? EW,CLEAR
```

shows how to perform "X EW" and "X CLEAR" together. Note that the program executes them in the order specified, so this would (with considerably more kindness than one has any right to expect from a machine) save the workspace before clearing it.

Text--The graphics editor provides two kinds of text: simple text and property name/text pairs. Text is usually associated with a point or body, and thus appears, disappears, and moves around whenever the point or body does. To deal with a piece of text, move the cursor close to it and give the appropriate command as described later in this chapter. Property name/text pairs give an additional means of access, a name associated with the text. Because the name is merely an access key, it's invisible on the drawings.

The editor distinguishes between text created as part of the drawing and text that is copied from a body template. Though the editor provides a mode that can manipulate both kinds, we won't discuss that because you won't need to use it. Throughout the modes we'll describe, template text is sacrosanct: because you didn't type it in, you can't touch it. A default property name/text pair is semi-sacrosanct: you can replace its text completely, and thereafter you can edit the replacement, but you can't edit the default text.

2.2 Commands for manipulating drawing files

2.2.1 Getting and saving drawings

X CLEAR	Clear the workspace, deleting drawings and body definitions but not macros. This also resets the editor to MODE=B, LEVEL=0, SCALE=16, and ∃=<nothing>.
βI	Bring in a new drawing file. (The program will prompt for the name.) If the workspace is clear, the new file becomes the "∃=" file; otherwise, it becomes a set within the existing drawing. Specifying "∃" as the new file clears the remembered name (the "∃=" feature) at the top of the screen.
αW	Save the workspace into a drawing file and change the "∃=" line, if necessary, to point to that file. (The program will prompt for the name. If the "∃=" already shows the filename, reply "∃" and the drawing will automatically go back into that file, which is a lot safer than attempting to retype the name.)
αL	List all body templates in the workspace (both bodies described in this drawing file and those described in any libraries you are using).
X GETLIB	Get a file and use it as a library. The program will prompt for the filename. Once a particular drawing knows about a certain library, it will remember it, so you need not repeat the command the next time you edit that drawing.
X TITLE	Invent a title for the drawing. (The program will prompt for the first line of the title and then for the second.) The title is a label that appears at the bottom of the drawing, and is quite distinct from the name of the file containing the drawing.
X PAGE	The SCALD programs do not require it, but for documentation purposes you may paginate drawings. This command prompts for SHEET (the current page number) and OF (the total number of pages). The numbers appear in the form "Page X of Y" at the bottom of the printed version of the drawing.
X PROJECT	Specify which project the drawing belongs to. (SCALD does not require this, but you may wish to partition your design into projects, with a certain number of macros in each project. If you do, SCALD will print the project name on listings to help you mentally sort macros into categories.)
X EW	Identical with "αW" followed by "∃". It's a good idea to use this command periodically as you edit, just as insurance against a system crash.

X EP Write a plot file called "<name>.PLT" based on this drawing, provided the "∃=" line contains a name.

X EL Write a wirelist file called "<name>.WD" based on this drawing, provided the "∃=" line contains a name.

2.2.2 Initializations

We recommend using the following commands to set up initial conditions in the program. The most painless approach is to make them into an editor macro called INIT as explained in Section 2.8. Then create a dummy drawing with nothing in it but the INIT macro. To create a new drawing (as opposed to editing an existing one), make it a practice always to start by clearing the workspace and bringing in the dummy.

X -LOCS Disables the displaying of a feature that SCALD doesn't use.

X BOARD Sets a number of characteristics that in general don't matter to SCALD. The program will prompt for the board type, and you should reply "DECPC". Consistency throughout all your drawings in this respect will spare you countless annoying, but harmless error messages.

X SHOWBOX Tells the graphics editor to provide a border around the drawing similar to those conventionally used for engineering drafting. Boxes attached to the border have room for the title, date, site, engineer's name, signature of approval, and so on. Aside from the title, SCALD needs none of these, though it will print page numbers and project names on its output listings for documentation purposes. Various "X" commands (explained in the SUDS manual) exist to specify each of these items. Paper plots show the entire border, but on the screen, all that appears is a simple rectangle defining the "edge" of the "paper".

The program will prompt for the type of box, the drawing scale, and the plot scale. Reply "A 16/1".

X PICCEN Centers the drawing within the box provided by "X SHOWBOX". This is primarily a convenience, since the program which produces a paper copy of the drawing centers it within the SHOWBOX anyway.

X UNDERLINE Positions signal names so that signal wires always go under them, not at the end of them.

X DIAMONDS Engineering drafting is afflicted with one great unanswerable question: when two lines cross, are they meant to connect? The DIAMONDS option causes D to supply a diamond at the intersection point whenever the lines are indeed meant to connect, but only on the copy of the drawing that gets plotted on paper.

The program will also ask now whether to plot a diamond whenever only three lines connect. Answer "Y". (Terminology gets confusing here. What looks like two lines crossing and connecting is, as far as D is concerned, a matter of four lines--two pairs of colinear segments. And what looks like one line meeting another in a "T" may, in some cases, be three discrete segments in the eyes of the program.)

2.2.3 Finishing a drawing

When you are satisfied with a drawing and plan to make a paper copy, we recommend using the following commands, which prepare it for plotting and clear the workspace in preparation for the next drawing. We use a macro called PLOT (Section 2.8 explains macros) to perform them automatically.

X SCALE Prompts for the scale of the drawing. Reply "16".

X -DEFPIN, EW,EP,CLE Described individually elsewhere.

2.2.4 Looking for errors

The following command looks for errors which occur when lines appear connected on the graphics display screen, but are not connected from the point of view of the program.

X DANGLE Mark all dangling points. Once they're marked, you can enter point mode and use "αF" repeatedly to move the cursor from one marked point to the next.

A dangling point is one which:

1. Has no lines or text associated with it (in which case you should delete the point), or

2. Has two colinear lines associated with it (in which case you should, within point mode, type "βD" to delete the point and merge the lines), or

3. Has exactly one line associated with it but no text (in which case you should label the point with some text), or

4. Has text but no line (in which case you should either supply one or more lines, or delete the point and its text), or

5. Lies atop another point (in which case you should type "αA". The program will put a star atop the twin points and ask, "This one?" Reply "Y" and the program will combine the points.

2.3 Commands for Body Mode

In body mode, all the commands implicitly refer to bodies in the drawing itself. They can create a copy of a body based on a specified body template and place the new copy in the drawing; they can move a body about, delete a body, rotate a body, or label a body.

Typically one first draws bodies in this mode and then switches to point mode to connect lines between them.

αP	Place a new copy of a body at the cursor position, leaving the cursor attached to it. (The program will prompt for the body name.)
αD	Delete the body closest to the cursor.
αY	Create or replace the text of a property name/text pair for the body closest to the cursor. The command works whether the text came from a body template or from one of the commands that let you type in text. The program will prompt for the property name and, if it does not already exist, will establish a new one. Then it will prompt for a text to go with the property name. That text may be any string, including embedded blanks and using the character "↔" to break the string across multiple lines. To replace an existing property text rather than to create a new property name/text pair, the "βY" command is safer.
βY	Replace the text for an existing property name/text pair. (This command works just like "αY", but requires you to type only enough of the property name to identify the property unambiguously. In addition, this command won't create a new property if you happen to mistype the old property name.)
αO	Rotate the body closest to the cursor by 90 degrees counterclockwise. After rotating it 360 degrees, the program will replace it with its mirror image.
αβY	Enter text/property submode for the body closest to the cursor, showing "MODE=BT" at the top of the screen. This submode of body mode lets you manipulate text or properties of that body by attaching and detaching the cursor, moving the cursor, and issuing commands. Until you attach the cursor, a large flashing letter or letters will identify the text or property closest to the cursor.

This mode will not alter text derived directly from a body template, but only text you have created or replaced yourself.

Note that everything you do within property/text submode of body mode applies to the body that was closest to the cursor on entering the submode. Once inside the submode, moving the cursor to another body doesn't alter this; before working with the properties or text or another body, you must get out of the submode, move the cursor to the other body, and get back in.

The following commands apply to text/property submode (while the "βY", and "αY" commands for body mode work equally well within text/property submode, it's a bad idea to use them here because if you forget which body the cursor was closest to when you entered the submode, you can easily create a piece of text that appears within one body but belongs to another as far as the graphics editor is concerned):

αD	Delete the property or text closest to the cursor but belonging to the current body
αβA	Use Alter submode to edit the text or property text closest to the cursor but belonging to the current body. You will see "MODE=BA" at the top of the screen. Note that this command cannot change text obtained directly from a body template. Only after you have used "αY", "βY", or "αT" to replace or produce text can you use "αβA" to alter it.
αβB	Return to normal body mode.

2.4 Commands for Point Mode

While point mode does deal with points, its principal use is to create lines by drawing *between* points. The commands in point mode which explicitly create or delete a point are used far less than those which draw lines, implicitly creating points as they do so.

αP	Create a point at the current cursor position.
αD	Delete the point at the current cursor position, along with any lines or text associated with that point. If two or more points coincide, the program deletes whichever it likes. (If the point is really a pin on a body, the program deletes lines and text associated with the point, but not the point itself.)
αT	Like "βT", but doesn't provide offset.
βT	Label the point closest to the cursor with text and offset the text if necessary to make it pretty. (The program will prompt for text.) You may use the "↔" character to separate the text into multiple lines.
αK	Delete the text (if any) labelling the point closest to the cursor.
αβA	Use Alter submode to edit the text, if any, labelling the point closest to the cursor. You will see "MODE=PA" at the top of the screen.
βA	Starting at the point closest to the cursor, draw a perpendicular line toward the line closest to that point. The program will place a star on the line it proposes to connect to and will ask you to confirm the command.
+	Draw a blinking line or pair of lines connecting the cursor with the point closest to it. (The program wants to avoid slanted lines, so if necessary it will use two perpendicular lines, one vertical and one horizontal). These lines are temporary and will stretch and contract to follow the cursor wherever it goes. As the cursor travels, the program puts a star on its current favorite "point of attachment" (that is, the point to which it will extend the lines automatically if you so choose.)

The command leaves you in a line-drawing submode of point mode, from which you can issue the following commands:

<ALT>	Delete the blinking line(s) and return to ordinary point mode.
<SPACE>	If there are two lines blinking, swap them. In other words, if the program previously chose to go from the point to the cursor by drawing first horizontally and then vertically, it will now draw first vertically and then horizontally.
+	If only one line is blinking, make it permanent by creating a

new point at the end. If two lines are blinking, make the one furthest from the cursor permanent by creating a new point at their intersection. Once it is permanent, the line stops blinking.

In either case, leave the program in line-drawing mode so that blinking lines continue to follow the cursor.

Extend the blinking line(s) as necessary to reach the current favorite point of attachment (that is, the point marked with a star) and then make the lines permanent.

2.5 Commands for Set Mode

Set mode manipulates groups of bodies and lines; it duplicates, moves, or deletes the entire group as a single entity.

<SPACE>, +, <ALT>, and -	These work more or less as described under point mode in Section 2.4, but are used to draw a box around a group of bodies and points. The two important differences are that you *must* use "-" to close the box, and that the lines of the box will continue to flash after you've used "+" to make them permanent. Then the box will vanish and the bodies and points become members of a set, which you can manipulate as a single entity, attaching the cursor to it, moving it, deleting it, and so on. The program will flag the bodies and points with large flashing "B" and "P" characters to indicate they're members, and will place a flashing "S" at the center of the set. When you attach the cursor, all the flashing letters vanish, and when you detach it, they reappear.
αD	Release all members of the set which is closest to the cursor. The points and bodies still exist, but don't belong to the set.
βD	Delete all members of the set which is closest to the cursor. The points and bodies no longer exist, so various lines and text associated with them must vanish too.

2.6 Commands for Edit Mode

Edit mode creates body templates, which you typically put into libraries and use to create bodies for drawings. This mode differs from point, body, or set mode in certain basic ways. First, the drawing temporarily hides in the background while you create or modify the template. Second, you must type a specific command to leave edit mode before switching to one of the other modes.

Edit mode performs three functions: drawing vectors to represent the body, defining pins on the body, and labelling the body and pins with text. For each function, there is a particular submode: insert submode, grab-body submode, pin submode, and text/property submode.

When inside edit mode but not inside any of these submodes, the program will place a star at the point on the body closest to the cursor. Resist the temptation to treat this like the star that appears in normal point mode; to draw the body you *must* use insert submode.

$\alpha\beta$E Enter edit mode. "MODE=E" will appear at the top of the screen. The program will prompt for the name of the body template to be edited and, if that template doesn't already exist, will create a new one.

While edit mode creates and modifies body templates, it does not delete them. Instead, the command "X DELTYP", which may be used only *outside* of edit mode, will ask you for a body name and then delete that body's template along with all occurrences of that body in your drawing.

αE Leave edit mode and return to body mode. (You may safely do this even from within one of the submodes of edit mode.) Note that if bodies derived from the template just edited exist in the drawing, the changes are reflected in the drawing immediately. In particular, deleting pins can cause havoc because lines formerly attached to them will vanish.

αI Enter insert submode within edit mode so as to draw vectors to make the body template. "MODE=EI" will appear at the top of the screen. Within this mode the commands resemble those used in point mode to draw lines; they are just similar enough to instill a false sense of confidence.

When you enter this submode, the cursor is resting at the point which will be the "origin" for the body—that is, the point the cursor will move to when you attach it to the body, and the point upon which the program will superimpose the flashing "B" when appropriate. By convention, we draw a body so its origin is at the upper left, though nothing in the program requires this.

From the origin, proceed to draw visible and invisible vectors forming a single path around the body. Provided it is not invisible, the vector you are currently working with will appear a bit brighter than the others.

These vectors *must* form a single path. To make three lines meet at a point, for example, you must draw through the point, then use an invisible vector to backtrack to it, and finally start a new visible vector headed outward from the point. Trying to attach the cursor to the intersection of two vectors and then moving the cursor to start a new vector will either overlay two visible vectors or move the intersection point without creating a new vector at all.

Here are the commands allowed within insert submode:

α+	End the pending vector (if any) and start a visible vector that will follow the cursor wherever it goes.
α-	End the pending vector (if any) and start an invisible vector that will follow the cursor wherever it goes.
<RUBOUT>	End the pending vector (if any) and move the cursor back along the path of vectors, toward the origin. Each time you press <RUBOUT>, the cursor travels the length of the preceding vector and lands at its starting point. The vectors on either side of that starting point appear extra bright (unless they're invisible), and if you move the cursor using the up/down/right/left keys, the vectors will stretch to follow it. When <RUBOUT> causes the cursor to reach the origin, it stops there, and further use of <RUBOUT> has no effect.
<SPACE>	Just like <RUBOUT>, but moves the cursor forward along the path of vectors, away from the origin. It's a good idea to use <RUBOUT> and <SPACE> to travel the path, checking for any duplicate or unwanted vectors, before leaving edit mode. If it takes two <RUBOUT>s to pass a certain point, for example, then you have inadvertently placed a zero length vector there, and should delete it.
αD	Delete a vector. If the cursor lies at a point on the path with one vector preceding it and another following it, this command deletes the preceding vector and stretches the following one to take its place, maintaining an unbroken path. If the cursor lies at the last point on the path, this command deletes the vector preceding it and makes the previous point into the last point on the path. If the cursor lies at the origin, then this command deletes the

first vector on the path and moves the cursor to the next point. This becomes the first point on the path, but doesn't change the origin—though <RUBOUT> will not move the cursor back to the origin, the origin is still there.

<ALT>	End the pending vector (if any), leave insert mode, and return to normal edit mode.

αC Grab a copy of an existing body template and add it at the cursor position to the body template being edited. The program will ask for that body's name. "MODE=EG" will appear at the top of the screen. For example, this command allows you to define a diamond or "bubble" body template and grab that body whenever necessary to show that a pin expects its signal to assert low.

The new body arrives with the cursor attached, and you can move it by moving the cursor. The following commands are valid within this submode:

<SPACE> Detach the cursor from the new body, incorporate it into the body template being edited, and return to normal edit mode. Once detached, the cursor cannot be reattached; attempting to do so will put you into insert submode, dealing with its path of vectors.

The reason will become clear if you enter insert submode and use <SPACE> and <RUBOUT> to travel the path; the program has already converted the new body into a series of vectors inside the path.

<ALT> Delete the body just grabbed and return to normal edit mode.

αO Rotate the body just grabbed, exactly as you would in body mode.

αβP Enter pin submode within edit mode. "MODE=EP" will appear at the top of the screen. Within this submode, use the cursor as you would within ordinary point mode to attach to pins, move them around, delete them, and so on. Important commands within this mode are:

αP Create a pin at the cursor position. The program will ask for a pinname (actually, this "name" must begin with a number.) To create an invisible duplicate of a pin for bus-through purposes, end the name with "/B": thus, a pin called "1" and a pin called "1/B" are electrically identical though physically they appear in two separate places on the body.

(You can actually create pins outside of pin submode, but it's a disorderly sort of practice since you must then get into pin submode to do anything else with them.)

αD Delete the pin closest to the cursor.

<ALT> Leave pin submode and return to normal edit mode.

X DEFPIN Display the pinname next to each pin, within edit mode and in the normal drawing modes. It's handy to turn this feature on while working with pins within edit mode, but one customarily turns it off in the normal drawing.

X -DEFPIN Don't display the pinname next to each pin.

αβT Enter text/property submode within edit mode. "MODE=ET" will appear at the top of the screen. Here you can label the body with text, create properties for the body or for pins, attach the cursor to text or properties, move them around with the cursor, delete them, alter them, and so on. Important commands within this submode are:

αT Create text at the cursor position. Such text merely labels the body, as a sort of comment that has no more significance to SCALD than does the shape of the body itself.

αY Create or replace the text of a property name/text pair just as in normal drawing modes.

βY Replace the text for an existing property name/text pair just as in normal drawing modes.

You can actually use αT, αY, and βY in edit mode without getting into text/property submode, but that's a disorderly sort of practice since you must then get into the submode to do anything else with the text you've created.

αD Delete the text or property name/text pair closest to the cursor.

αK Kill the text or property closest to the cursor. If it's simple text, this deletes it. If it's a property name/text pair, this doesn't delete it, but simply hides the text so it doesn't appear when you use the body in a drawing. This is handy because it allows you to label pins as SCALD requires while avoiding clutter on simple bodies, such as gates, where the purpose of each pin is understood by convention.

	If you use αK by mistake on a property, the only way to undo the damage is to delete the property in question and create it anew.
α/βA	Use Alter submode to edit the property or text closest to the cursor. "MODE=EA" will appear at the top of the screen.
<ALT>	Leave text/property submode and return to normal edit mode.

2.7 Using Alter Submode to Edit Text

Alter submode is a text editor into which you may momentarily descend from within a drawing mode or submode. When you leave alter submode, you return to whatever you were doing before. If, for example, you were in the text/property submode of body mode when you decided to alter something, you'll be back in text/property submode when you return.

Invoking alter submode from point mode edits the text associated with the point closest to the cursor, or creates text if that point has none.

Invoking alter submode from a text/property submode edits the text closest to the cursor. If that text is part of a property name/text pair, then alter mode affords you the side benefit of finding out the property name associated with that text, which is otherwise invisible.

Within alter submode, the program displays the text with the "↔" character indicating any point at which the text breaks into multiple lines. Underneath the text, an L-shaped line serves as a pointer.

In the list of commands that follows, <-> indicates that placing "-" before the command reverses its operation—backward instead of forward or forward instead of backward. <n> indicates that placing a digit in front of the command causes it to repeat itself the specified number of times.

α,βA	Enter alter submode.
<ALT>	Leave alter submode.
<->\<n><SPACE>	Move the pointer forward one character.
<->\<n><RUBOUT>	Move the pointer backward one character.
<->\<n>S\<char>	Move the pointer forward past the next occurrence of character \<char>. If \<char> doesn't occur, leave the pointer at the end of the text. With <->, the pointer will move backward and come to rest before the character, or at the beginning of the text if the character doesn't occur.
<->\<n>D	Delete the character to the right of the pointer.
<->\<n>K\<char>	Delete characters to the right of the pointer up to and including the next occurrence of \<char>. If \<char> doesn't occur, leave the pointer at the end of the text without deleting anything. With <->, the program will delete characters to the left of the pointer through the next occurrence of the character. If the character doesn't occur, the pointer will land at the beginning of the text without deleting anything.
I	Insert text at the pointer position. The program will prompt you by asking

"INSERT TEXT←". Type the characters you want to insert and press <RETURN>. (To put a carriage return inside the text, use the "↔" character.)

<->\<n>R Replace characters. Equivalent to a "<-><n>D" command followed by an "I" command.

2.8 Defining and Using Editor Macros

To speed repetitive tasks, you can collect together into an editor macro any set of commands you could have performed individually. Such a macro can even define or use another macro. Note that editor macros, which are convenient ways to reduce the amount you must type, are quite different from SCALD macros, which are drawings representing functional blocks of circuitry.

Macros are not associated with particular drawings, but rather with the session at the editor. Clearing the workspace doesn't delete them, and saving a drawing doesn't necessarily save them unless the "X SMACRO" command described later is in effect.

The macro commands actually need only begin with "α;" when used within a macro definition; outside, the "α;" is optional.

Note that after you type the initial "α;" for any of the following commands, the program will print "←" to prompt for the rest of the command. The "α; C" command will also print "TYPE MACRO NAME" to prompt for the <id>.

As you enter and exit macros, the number to the right of "LEVEL=" at the top of your screen will keep track of their nesting. If LEVEL is 0, no macros are pending.

α; P	Define a temporary, unnamed macro. After you type "α; P", each command you type will execute within the drawing and also become part of the macro. This will continue until you use "α; S" to abort the macro or "α; R" to call it repeatedly. After executing the proper number of times, the macro vanishes.
α; S	Abort all macros currently pending. This is the command to use when you're inside one or more levels of macros and realize you've made a mistake or lost track of the situation.
α; R<num>	Stop adding commands to the current macro, end it, and execute it the number of times specified by <num>. That number should include the first execution, which for "α; P" or "α; M" has already taken place within the drawing. For example, "α; R4" will execute the macro three *additional* times.
α; M<id>	Like "α; P", this begins by executing commands as it collects them into a macro and ends by executing the entire macro enough *additional* times to satisfy the closing "α; R" command. But it also gives the macro the name <id> so you can call it again with "α; C". The name "INIT" gives a macro two special properties that make it useful for initializing various aspects of the program. First, the program will save this macro in the "<name>.DRW" file along with the drawing whether or not the "X SMACRO" command is in effect. Second, it will execute the macro automatically when you bring the file into your workspace.

If, for example, you want to set the scale to "x17" whenever you begin a drawing but don't want to have to remember to use the "*" command, create a macro called INIT containing that command.

α; C<id>	Call the macro named <id> and execute it the number of times specifed by the "α; R" command used to close the macro when you originally defined it.
α; A<s>,<i>	This command puts a counter inside a macro. It is valid anywhere inside a macro definition--even partway through a string of characters. Every time the macro executes, the "α; A" expression replaces itself with the text representing a number, starting with the number <s> and incrementing by <i>.
X SMACRO	Associated with each macro is a flag telling the program whether that macro should be saved in the drawing file whenever you perform a "αW" or "X EW" command. When you bring in a drawing file that contains a macro, or when you use such a drawing file as a library, you acquire the macro and retain it, even if you clear your workspace, until you leave the program or use the "X DMACRO" command to get rid of it. This command asks you for a macro name and sets the flag for that particular macro.
X -SMACRO	Clears the flag for a particular macro, thus telling the program not to save the macro in the drawing file when you use "X EW" or "αW".
X DMACRO	Deletes a macro from the work area. (The program will prompt for the name of the macro.)
X MACRO	Lists all macros associated with this editing session.

3 A guided tour of D

This section proceeds step by step through an entire session with the graphics editor, showing how to create a typical drawing. It makes a number of assumptions which--if true--will make it much easier to learn to use the program:

- We assume you're using a Stanford keyboard, whose distinguishing features are shift keys labelled "TOP", "CONTROL", and "META". If not, consult a friendly local wizard, or refer to the SUDS manual mentioned in Section 8, for the conversion procedure.
- We assume you know, or can find out from a friendly local wizard, how to start the program running at your installation.
- We assume someone has already described and placed in libraries called "SIMLB" and "STDLB" the bodies your drawing will need, and that they've given you a blank drawing called "BLANK" that initializes the appropriate options for you.
- We ask you to assume the complete drawing was revealed to you in some mysterious flash of insight, so we can concentrate on the graphics editor, and postpone discussion of the SCALD language.

Before you start, you should read the first few pages of the preceding chapter--Section 2.1 should be plenty.

In the examples that follow, we use italic type for the characters you produce and normal type for the characters the computer produces. We use "α" and "β" as explained in Section 2.1, and use <ALT> to represent the key labelled "ALT" or "ALTMODE"; <SPACE> to represent the space bar; and <RETURN> to represent the key marked "RETURN".

3.1 Running the program

Get the program running by whatever means, fair or foul, your local wizards have taught you. You should see something similar to Figure 3-1. The program devotes most of the screen, below the "MODE=" line, to your drawing. On the bottom quarter of the screen it superimposes the character-by-character dialog between the program and the keyboard. Soon you'll probably find yourself focusing on the drawing rather than the characters you type, since the drawing is a lot more fascinating, and it will seem as if your fingers control the image directly. When the keys don't seem to be working, however, you can often tell from the character-by-character dialog what's wrong.

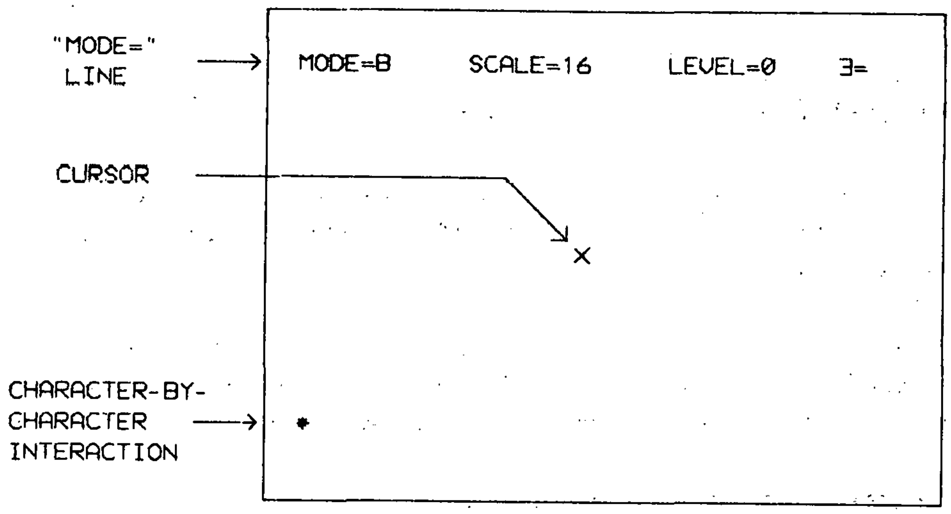


Figure 3-1
An empty screen

The top line shows you're in body mode ("MODE=B") with scale set to 16, no macros pending ("LEVEL=0"), and no file brought into your workspace("∃="). Throughout the rest of the chapter, we'll show only the drawing portion of the screen, leaving the top line implicit.

To get accustomed to moving the cursor, place the four fingers of your right hand on the keys marked "[", "]", "\", and "/" above the RETURN key. Press with your index finger and you should see the cursor move left. The long finger should make the cursor move right. The next finger should move the cursor up, and the little finger should move it down.

Experiment with holding down the shift keys--CONTROL, META, and TOP--first by themselves and then in combination--to make the cursor move further with each keystroke. With CONTROL alone, it should move twice as far as it does without any shift key; with META, four times as far; with TOP, 16 times as far. With CONTROL, META, and TOP together, it should whiz across the screen 128 times as far as it does with none of the shift keys.

Practice moving the cursor around till you feel bored or comfortable with it. Soon, you will automatically associate the four fingers with the four directions, without thinking about the keys they're pressing.

3.2 Initializing the workspace

Before you can start drawing, you need to initialize certain options and to gather bodies from the libraries.

At our installation, we keep around a drawing called "BLANK" whose function is to bring in bodies from a library and to call an INIT macro which performs without toil or strain on your part the initializations covered in Section 2.2.2. Remember that, as explained in Section 2.1, "β" means you should hold down the META key while you press the succeeding character. Thus, the command for bringing *in* a new drawing is written "βI" and stands for "META I":

```
*βI FILENAME? BLANK
READING BLANK.DRW[MK2,S1]
  LIBRARY STDLB.DRW[MK2,S1]
PLOT
INIT
LEAVING MACRO LEVEL 1
```

The reply from the program (which may vary slightly from that shown here indicates that the it found the file you wanted (BLANK[MK2,S1]), brought in one of the libraries you'll need bodies from (STDLB[MK2,S1]), and carried in with it a couple of macros. Among the macros was INIT, which is unique in that it executes as it enters your workspace, performing the initializations you need.

To *list* the bodies you received use of through that deal, use the "αL" (remember--"CONTROL L") command:

```
*αL

STDLB.DRW[MK2,S1]
8WOO    3WO    5WO    R8MERGEO        8MERGEO
(and so on...)
```

You'll need bodies from a second library, too, so bring the drawing SIMLB into your workspace as a library:

```
*X
WELL?GETLIB
GETLIB
LIBRARY FILENAME?SIMLB
  LIBRARY SIMLB.DRW[MK2,S1]
INIT
```

Now if you try "αL" again, you'll see a lot more bodies:

```
*αL
```

```
SIMLB.DRW[MK2,S1]
2 ANDO  2 AND   5 ORO   4 ORO   3 ORO   5 OR    4 OR
(and so on...)
STDLB.DRW[MK2,S1]
8WOO    3WO     5WO     R8MERGEO        8MERGEO
(and so on...)
```

Before you draw anything, it's not a bad idea to *write* your workspace into a file, just to get the file established. Since the "∃=" in the header makes it clear that the program has no idea what you want to call the file, you'll have to tell it. For this example, we want to call the file "10016":

```
*αWFILENAME?10016
WRITING 10016.DRW[MK2,S1]
```

Notice that the top line of the screen now says "∃=10016[MK2,S1]". The program now remembers which file it is dealing with, so from now on you can use a shortcut to save your workspace into that file without your having to retype the filename:

```
*X
WELL?EWRITE
EWRITE
FILENAME?10016
WRITING 10016.DRW[MK2,S1]
```

We won't mention it, but it's a good idea to use the command periodically--just after you've done something particularly difficult, or just before you leave the keyboard to answer the telephone or a call from nature—so that even if your computer system crashes, you won't lose all of your work.

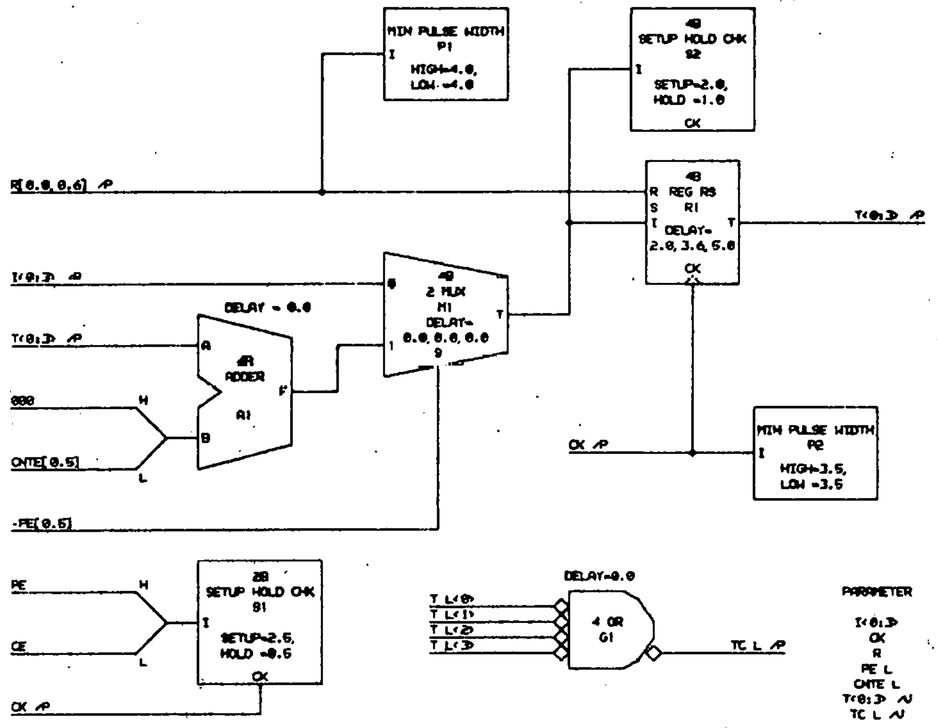


Figure 3–2
Our goal is a drawing like this

This seems a good time to take a look at Figure 3-2, which shows the drawing we will practice upon. It is the definition of an ECL 10016 IC in terms of the primitive bodies that the SCALD logic simulator understands. We'll split the work up systematically: first position all the bodies, then draw lines between them, and finally add text.

3.3 Positioning Bodies

To position bodies in your drawing, you must get into *body* mode, and to do that you should type:

> **αβB*

(Of course, this isn't necessary this time, because you were already in body mode by virtue of having just started to run the program, but the command will prove useful in the future.)

Whereas the commands we've shown you so far apply more or less anywhere in the program, you'd better assume that the ones that follow will live up to our promises for them only within the proper mode. For example, in body mode the "αP" command we're about to introduce places bodies, but it has an entirely different effect in point mode.

Let's start with the body called "MIN PULSE WIDTH" at the top of the drawing. Each body has a short location parameter below its name; in this case, it's "P1", so we can refer to the body as MIN PULSE WIDTH at P1 to distinguish it from the copy of MIN PULSE WIDTH at P2 on the right side of the drawing. To *place* a copy of a body at the cursor, type:

```
*αPTYPE BODY NAME
MIN PULSE WIDTH
SEARCHING FOR MIN PULSE WIDTH IN SIMLB.DRW[MK2,S1]
```

and presto, you'll see the body before you, with the cursor at its top left corner. The cursor happens to be attached to the body--that's always the case when you first place a body--so that wherever you move the cursor, the body will follow. Try it.

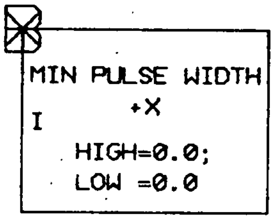


Figure 3-3
Your first body, with cursor detached

To detach the cursor, press <SPACE>. You'll immediately see a big flashing "B" atop the body (Figure 3-3). Now try moving the cursor, and observe that the body doesn't follow. There are two ways to reattach the cursor. "αM" moves the cursor to the body and reattaches it, while "αβM" moves the body to the cursor and reattaches it. Once reattached, you can once again move the body by moving the cursor, and then detach the cursor by pressing <SPACE>.

This sort of thing works throughout body mode. Once you have more than one body on the screen, the program operates on the body closest to the cursor; it alone will have the flashing "B". Experiment with moving this body until you feel jaded, then put the body back near the center of the screen and detach the cursor by pressing pressing <SPACE>, but leave the cursor in position on the body.

Let's place the body SETUP HOLD CHK in location S2 next. Move the cursor to the right by one CONTROL-TOP. (That is, while holding down both CONTROL and TOP, make one stroke with your long finger to move the cursor to the right.) Now place the body, whose name is, as far as the program's concerned, simply "SETUP HOLD":

```
*αPTYPE BODY NAME
SETUP HOLD
SEARCHING FOR SETUP HOLD IN SIMLB.DRW[MK2,S1]
```

Now detach the cursor from the body by pressing <SPACE>.

Why did we emphasize that you should move the cursor by one CONTROL-TOP? Obviously it doesn't make any difference to the final circuit where you place a body. Conventionally, however, it's considered good drawing style to place bodies so that the lines connected to them lie a uniform distance apart. And it's considered better to use a few large increments than several assorted small ones.

There are two reasons for this. First, uniform spacing makes it easier to apply editor macros to reduce repetitive typing, as you'll see later in the chapter. Second, it just plain takes fewer keystrokes to get from one body or line to another when they're CONTROL-TOP apart rather than a TOP plus a META plus a CONTROL apart.

Unfortunately, this empyrean goal of style is tough for a beginner to achieve, particularly because one can't always tell precisely where on a body the program will want to attach a particular line. Just keep the goal in mind as you position the bodies, and comfort yourself with the knowledge that you can always move things around later to repair any irregularities you cause now.

Next, place a REG RS body at position R1, right under the previous body. To do this, first type a space to detach the cursor from the previous body, then move it down by a META plus a TOP, (first hold down META and move the body, then hold down TOP and move it again) and then place the new body:

```
*αPTYPE BODY NAME
REG RS
SEARCHING FOR REG RS IN SIMLB.DRW
```

Because this body is narrower than the previous one, it's not centered beneath it (Figure 3-4), so move it to the right by one CONTROL before you type a space to detach the cursor (Figure 3-5).

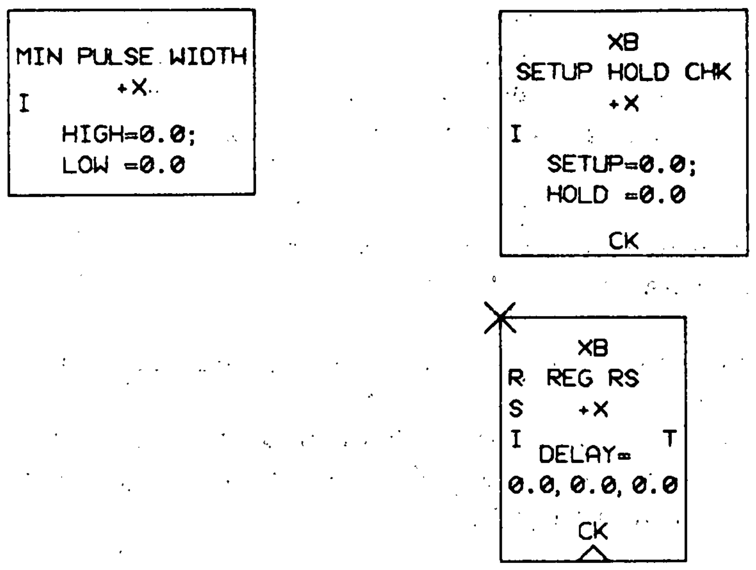


Figure 3-4
Third body off center

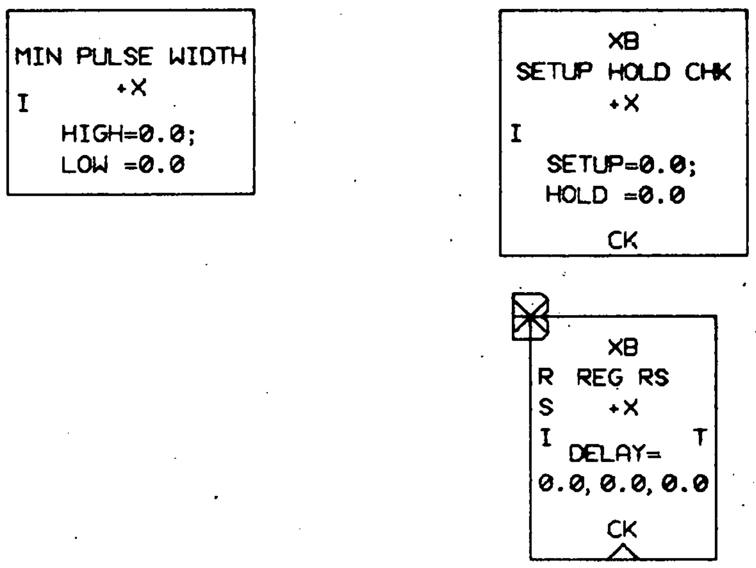


Figure 3-5
Third body centered and cursor detached

Moving the paper—Before you proceed to place the rest of the bodies in your drawing, there are a few loose ends to clear up.

Sooner or later, for example, you're going to run out of room on the screen. Fortunately, the "paper" you're drawing on is much larger than the screen; at any time, you effectively look through the screen at a small area of it. To move the paper to the right, press the "→" key (you'll have to use TOP to produce this character). Do this repeatedly until the bodies you've drawn disappear. Keep doing it, and pretty soon a vertical line will emerge from the left side of your screen, representing the left edge of the paper.

Now press the "←" key repeatedly until your bodies come back onto the screen. You can use "↑" and "↓" similarly to move the drawing up and down.

You already know how to move a body around if you accidentally put it in the wrong place--simply bring the cursor close to it, use the "αM" or "αβM" command to attach the cursor, and move the body by moving the cursor. You also need to know how to get rid of a body entirely if you need to. First move the cursor so it's closer to that body than to any other (the big flashing B will appear atop the potential victim) and then type "αD".

Now go ahead and position the rest of the bodies. Some of them are pretty obvious--the adder is called "ADDER", the multiplexer is called "2 MUX" and the gate is called "4 OR"—but others are a little tricky. The parameter list at the lower right corner needs a body called "PAR", and the two Y-shaped gizmos at the lower left which look like lines are actually bodies called "W2MERGE". When you finish placing bodies, your drawing should look something like Figure 3-6.

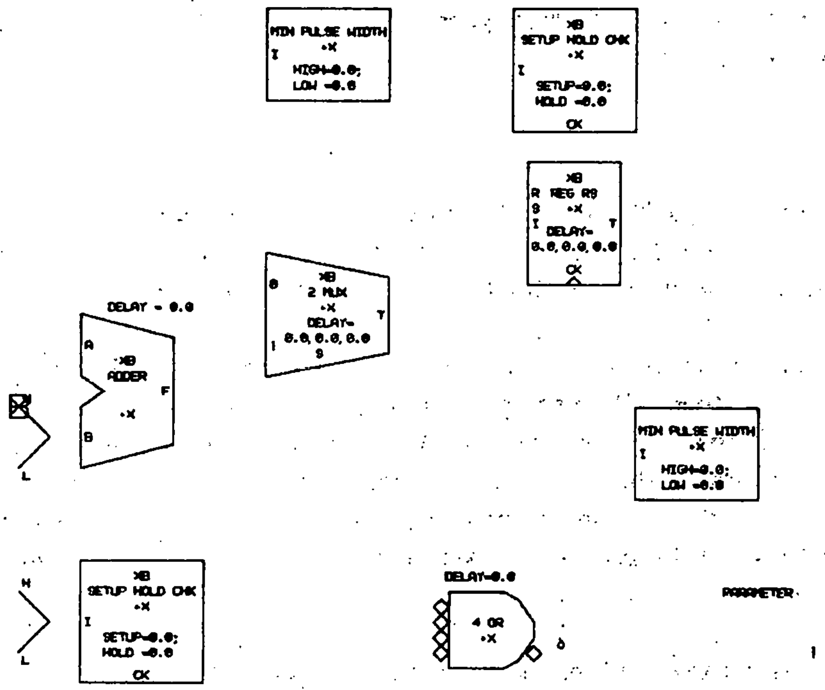


Figure 3–6
All the bodies

3.4 Drawing lines

In the graphics editor, as in high school geometry, any two points define a line, and therefore you must get into *point mode* to draw lines:

$\ast\alpha\beta P$

At the top of the screen, you'll now see "MODE=P", and the flashing "B" on the nearest body will vanish. Instead, you'll see a flashing letter "P" or string of letters beginning with "P" atop the nearest point. Try moving the cursor around; you'll see letters hop from one point to another.

Every pin on a body provides a point you may connect to, and you will create points implicitly as you draw lines between them. Whereas geometry tells us that points are everywhere--lines consist of an infinite number of them side by side, and planes consist of giant smorgasbords of points spread out endlessly—the graphics editor takes a more manageable view: aside from the points provided free with bodies for the purpose of attaching lines, points exist only where you explicitly or implicitly create them. Like geometry, however, it will let you put two points in the same spot, usually to your own distress.

You're about to learn three different ways to create a line: drawing from one existing point to another, drawing from an existing point into midair, and drawing from an existing point to the closest point on an existing line. Those three techniques will cover every situation you'll encounter in this drawing, and in just about any other.

Point to point–First, let's try a line from pin "T" on the multiplexer at M1 to pin "T" on the "SETUP HOLD CHK" body at S2.

Move the cursor close enough to pin "T" on body M1 so you see a big "PA" flashing atop the pin (Figure 3–7).

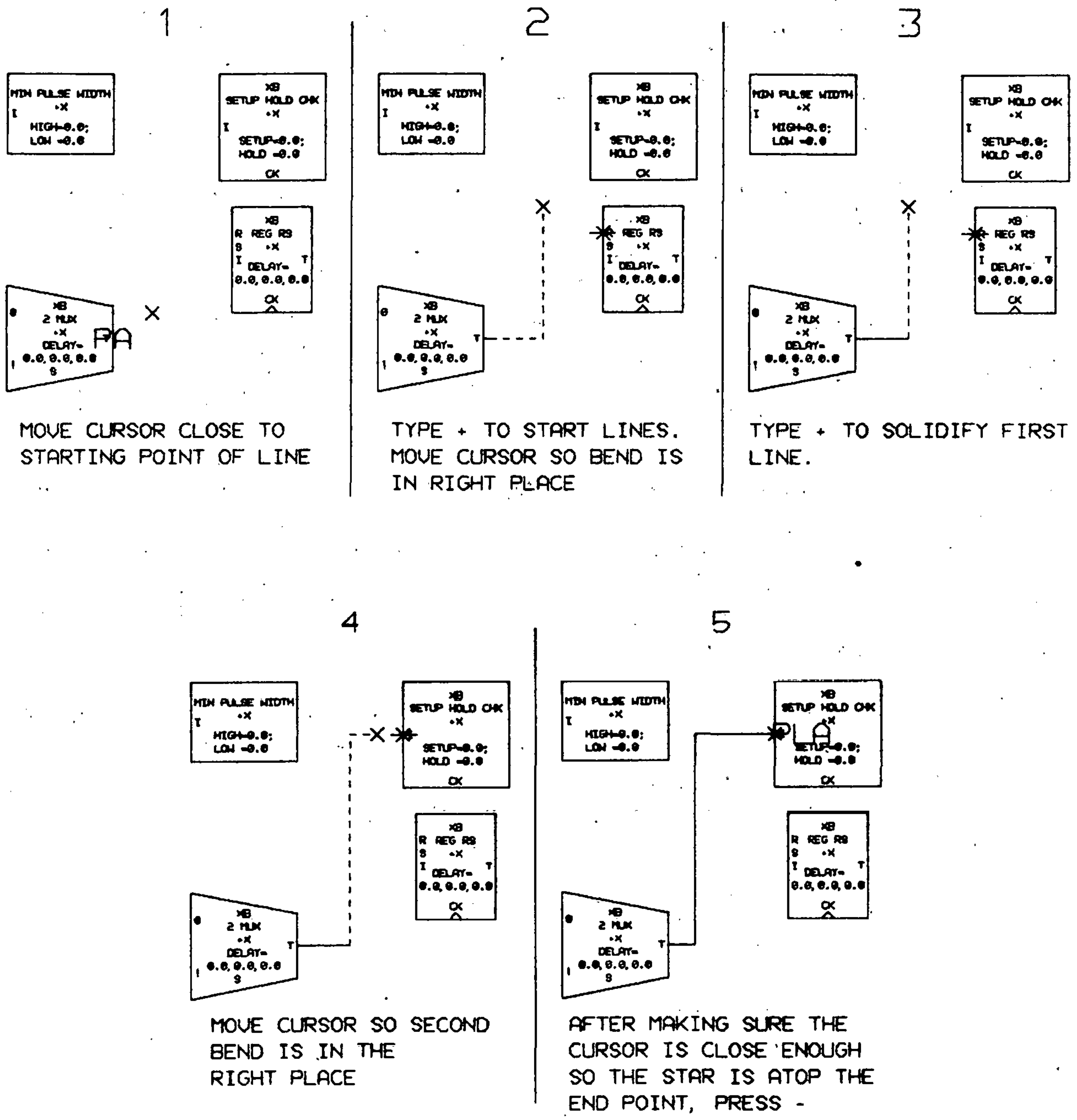

1
MIN PULSE WIDTH
SETUP HOLD CHK
REG RS
2 MUX
MOVE CURSOR CLOSE TO STARTING POINT OF LINE
2
TYPE + TO START LINES. MOVE CURSOR SO BEND IS IN RIGHT PLACE
3
TYPE + TO SOLIDIFY FIRST LINE.
4
MOVE CURSOR SO SECOND BEND IS IN THE RIGHT PLACE
5
AFTER MAKING SURE THE CURSOR IS CLOSE ENOUGH SO THE STAR IS ATOP THE END POINT, PRESS -

Figure 3–7
Drawing a line from point to point

Now press "+" and you'll see a flashing line or lines from the cursor to pin "T". Try moving the cursor around; the lines will stretch and move to follow it wherever it goes. If the cursor happens to be directly in line with pin "T" vertically or horizontally, you'll see a single line, but otherwise the program draws two lines intersecting at a right angle, so as to avoid having to draw a sloping line between the pin and the cursor.

With the cursor positioned so you see two flashing lines, try pressing <SPACE>. Every time you do so, the lines will trade places. The program tries to guess whether the vertical or horizontal should come first--it knows, for example, that lines customarily attach to bodies perpendicularly--but sometimes it isn't too bright, and you must then use <SPACE> to help it along.

Now position the cursor so that the first bend in the line is where you want it, and type "+". The flashing line attached to pin "T" will stop flashing; you've just made it a fixed, permanent line and implicitly created a point at the end where it intersects the other line.

Now try moving the cursor and you'll see a second right angle, with an additional flashing line helping to follow the cursor wherever it goes. In general, every time you press "α+" in this line-drawing submode within point mode, you solidify the oldest flashing line and make it possible to add a new flashing line at the cursor.

While you're in this mode, you'll see a star flashing atop the point which is closest to the cursor but also eligible to have a line attached to it. Move the cursor close enough to pin "I" so that the star appears on that pin, type a space if necessary to put the second bend in the line roughly where you want it, and type "α-" (or just "-"). In one fell swoop, the program will extend the flashing lines to reach the star, attach them to that point, make them permanent, and free the cursor to move without dragging any lines around behind it.

And that, in essence, is the technique for drawing a line between two existing points. Move the cursor close to one point and type "α+" to get a pair of stretchable, flashing lines. Move the cursor around, and whenever you need a new flashing line, type "α+" to solidify the oldest flasher and give you an additional one. When you get the last pair of flashing lines you need, make sure the cursor is close enough to the destination point that the star appears atop it, and type "α-" to finish the job.

The line from pin "CK" on the body at S2 to the unnamed pin at the top of the body at R1 is even easier, since it has no bends. Move the cursor close to pin "CK" so that "PA" flashes atop the pin. Press "α+" to start the flashing lines. Move the cursor close enough to the unnamed pin that the star appears atop it (you're probably so close that the star is already there) and press "α-" to finish it off.

Correcting mistakes in lines--To get rid of a line, you simply delete the points that define it. Fortunately, the program is intelligent about this. When you delete an ordinary point in midair, it vanishes together with all the lines attached to it, but when you delete a point that represents a pin on a body, only the line vanishes; the pin remains intact for future use.

If you discover a mistake while you're still drawing the line, press <ALT> to escape. The flashing lines will vanish, leaving the cursor free. With the cursor free, you simply move it close enough to the point you want to zap so that large flashing letters appear over that point, and then type "αD".

To illustrate this, let's deliberately draw a line from pin "F" on the adder to pin "0" (rather than pin "1") on the multiplexer. Move the cursor close to pin "F" so "PA" flashes above it, and type "α+" to get stretchable lines. When you have the first bend where you want it, type "α+" again. Move the cursor close enough to pin "0" that the star appears atop the pin, and type "α-" to finish off the line.

Now that you've successfully committed a blunder, how do you undo it? Notice that you want to wipe out both the horizontal line attached to pin "0" *and* the vertical line, because the latter is longer than it should be. The easiest way to blow both of them away at once is to delete their point of intersection. So move the cursor close enough to that point so that large letters "PL" flash above it (Figure 3–8) and then type "αD". Both lines (and the point at which they intersect) will vanish.

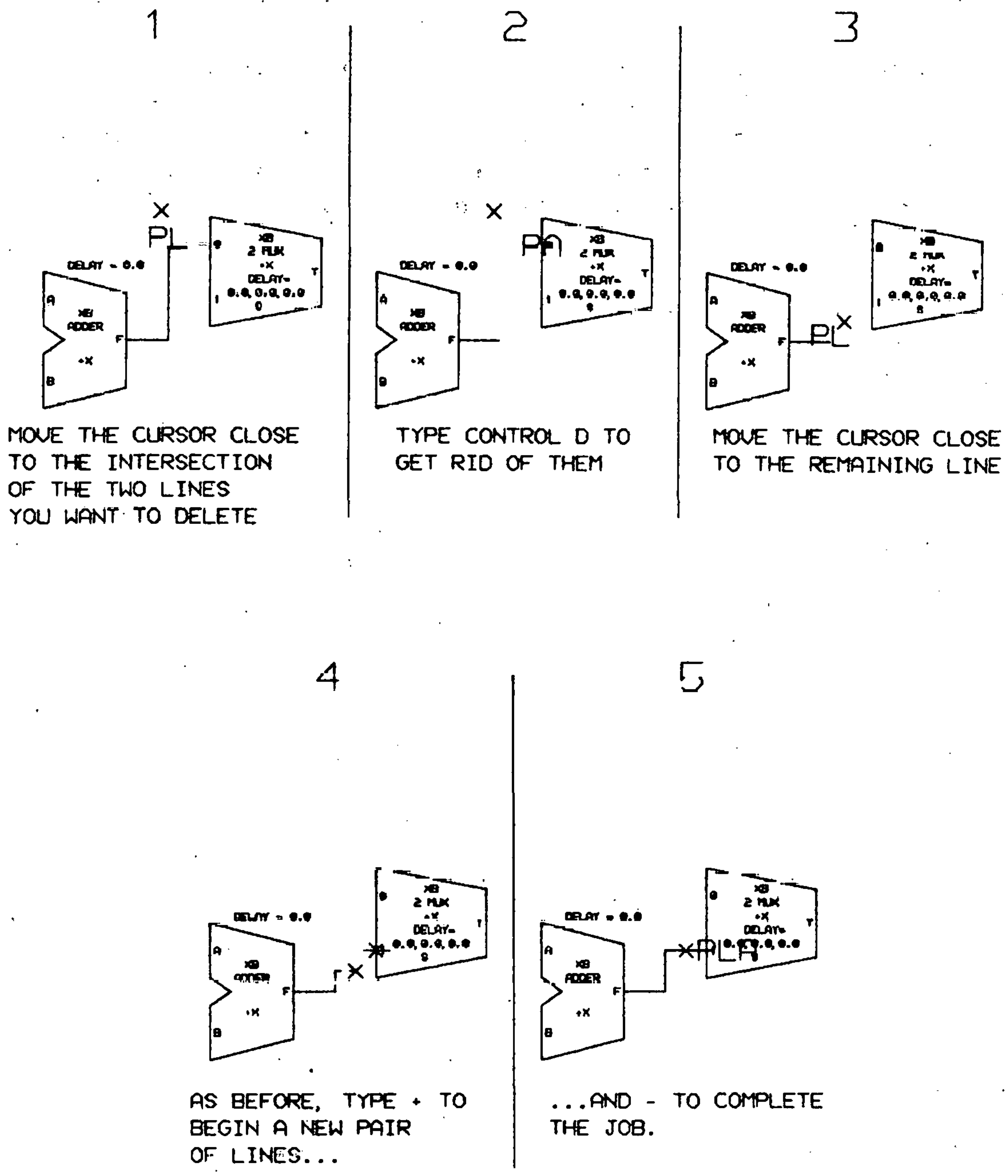

1
2
3
MOVE THE CURSOR CLOSE TO THE INTERSECTION OF THE TWO LINES YOU WANT TO DELETE
TYPE CONTROL D TO GET RID OF THEM
MOVE THE CURSOR CLOSE TO THE REMAINING LINE
4
5
AS BEFORE, TYPE + TO BEGIN A NEW PAIR OF LINES...
...AND - TO COMPLETE THE JOB.
DELAY = 0.0
XB ADDER
XB 2 MUX
DELAY=
PL

Figure 3–8
Correcting an erroneous line

To finish repairing the damage, proceed as you would when drawing from body to body, but use the end of the line that's dangling in midair as the starting point. Move the cursor close to it so "PL" appears over the point, type "α+" to start flashing lines, type a space if necessary to get the bend to go in the proper direction, bring the cursor close enough to the "1" pin to place the star over that pin, and type "α-" to finish.

Just for practice, draw the two remaining point-to-point lines: the line from the adder to the upper MERGE body and the line from the body at S1 to the other MERGE body. Each is easy compared with the lines we just finished, because neither has any bends; in fact, the instant you type "α+" to start the line, the star will probably appear on the destination point so you can type "α-" to finish it. Don't try to draw lines from the register to the body at P1, or to the body at P2; we'll use other techniques for those. When you're finished, the drawing should look like Figure 3-9.

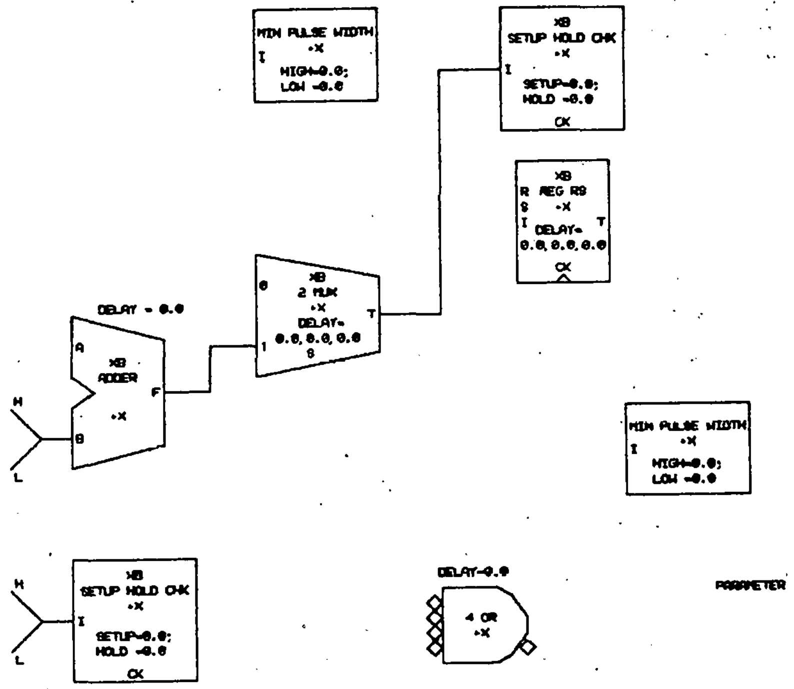


Figure 3-9
After finishing the point-to-point lines

Point-to-midair lines--When a line originates at an existing point but terminates in midair, you must use a second, slightly different technique to draw it. To illustrate, let's draw the line that begins at pin "S" on the multiplexer at M1 and ends with the label "-PE[0.5]" at the left edge of the drawing.

(You may need to use the "→" key to shift the entire paper to the right so you have room to work.)

The first part of the procedure will look familiar. Move the cursor close enough to pin "S" so that the letters "PA" flash atop it (Figure 3-10). Type "α+" to get a pair of flashing lines, and move the cursor down until the bend is in the proper place. Then move the cursor to the left until the horizontal line is the length you want.

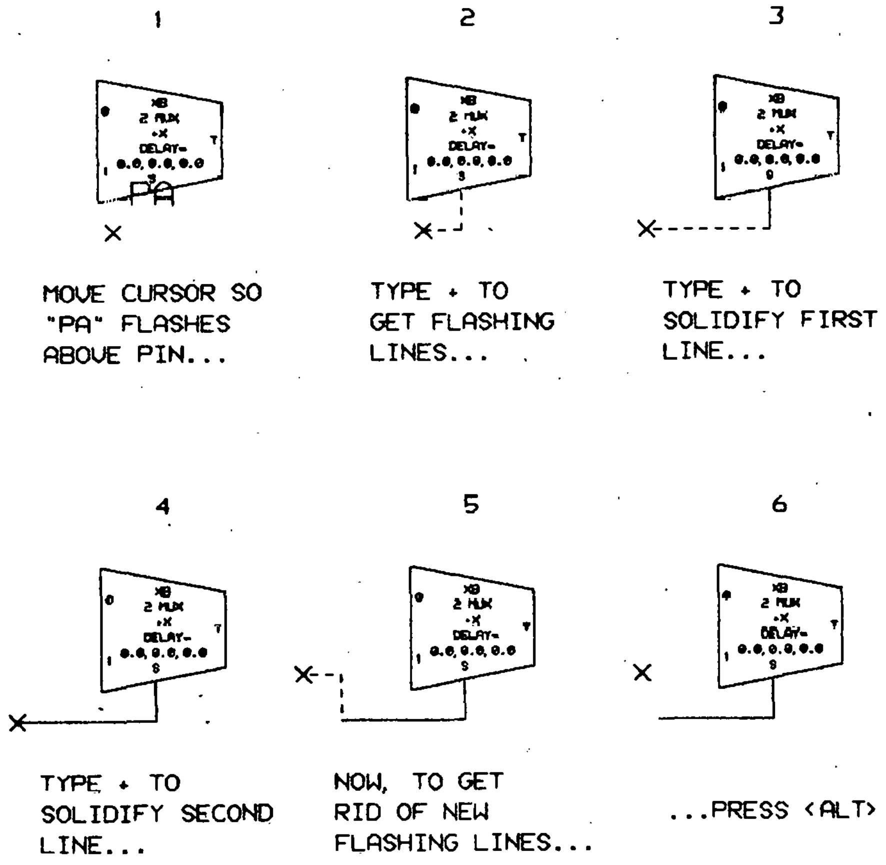

1
2
3
MOVE CURSOR SO "PA" FLASHES ABOVE PIN...
TYPE + TO GET FLASHING LINES...
TYPE + TO SOLIDIFY FIRST LINE...
4
5
6
TYPE + TO SOLIDIFY SECOND LINE...
NOW, TO GET RID OF NEW FLASHING LINES...
...PRESS <ALT>

Figure 3-10
Drawing from a point to midair

Now type "α+" once to solidify the vertical line, and again to solidify the horizontal one. Actually, you've just created two new stretchable, flashing lines from the left endpoint to the cursor. But since the cursor is atop the endpoint, you don't see them. Move the cursor a bit (try it) and there they are.

To get rid of those unwanted flashing lines, simply press <ALT>, rubbing them out and freeing the cursor.

Now you know how to create a point-to-midair line. Note that you've implicitly created two points: one where the two segments of the line intersect, and another at the endpoint in midair. That agrees with what we said earlier: once you're inside this line-drawing submode, every time you use "α+" you solidify a line and create a point at the end of it, too. This has two implications. First, if you ever decide to delete the segment that ends in midair, you must make sure to delete the midair endpoint. You can get the line to vanish by deleting its other endpoint, but that will leave an unused (and invisible) point in midair.

Second, you must not use <ALT> in place of "α-" to finish up a point-to-point (body-to-body) line even if you have the cursor directly atop the destination point, becase it will create a second point atop the existing destination point.

Just for practice, draw the rest of the point-to-midair lines: the two attached to pins "T" and "R" of the register at R1, the one atached to pin "0" of the multiplexer at M1, the one attached to pin "A" of the adder, the four attached to the merge bodies, the one attached to pin "CK" of the body at S1, the one to the right of gate G1, and the one from pin "I" of the body at P2 to the endpoint labelled "CK /P". That's an impressive enough list, so for now don't bother to draw any of the four lines to the left of gate G1. When you're finished, the drawing should look like Figure 3-11.

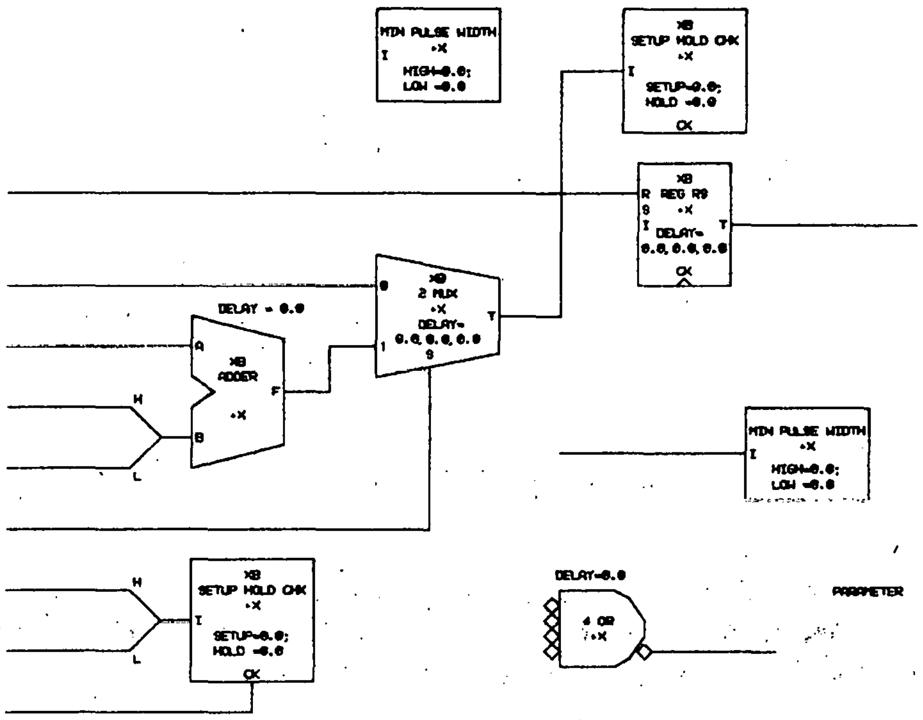


Figure 3-11
Drawing with point-to-midair lines finished

Point-to-line lines--The third and last way to draw a line is to go from an existing point to the closest spot on an existing line, and to connect to the line by creating a new point there. (Thus, this is *not* the way to connect a new line to a bend on an existing line; because a bend always provides an existing point, you would use the point-to-point technique for that.)

To illustrate, let's draw a line from pin "I" of the body at P1 to the horizontal line below it. As you may suspect, the opening moves will be the same as those you've used for the last two kinds of lines; only the endgame is different.

Move the cursor close enough to pin "I" so that "PA" flashes above the point, and type "α+" to get a pair of stretchable, flashing lines. Move the cursor and, if necessary, type a space to put the bend where you want it, and then move the cursor down close to the place on the existing line where you'd like to connect the new lines. Type "βA". The program will put a star on the line where it plans to make the new connection (Figure 3-12) and ask you whether that's the right place:

```
*βATHIS ONE?
```

Answer "Y" and the program will complete the connection; answer "N" and the program will decide not to connect the lines, giving you a chance to move the cursor closer to the precise spot where you'd like the connection before you try again.

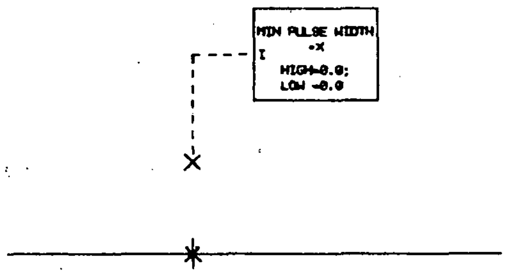


Figure 3-12
Connecting a line to an existing line

For practice, draw the two remaining point-to-line lines: one from the "CK" pin of the register at R1 to the line below it, and the other from the "I" pin of that register to the line to the left of it. When you're finished, the drawing should look like Figure 3-13.

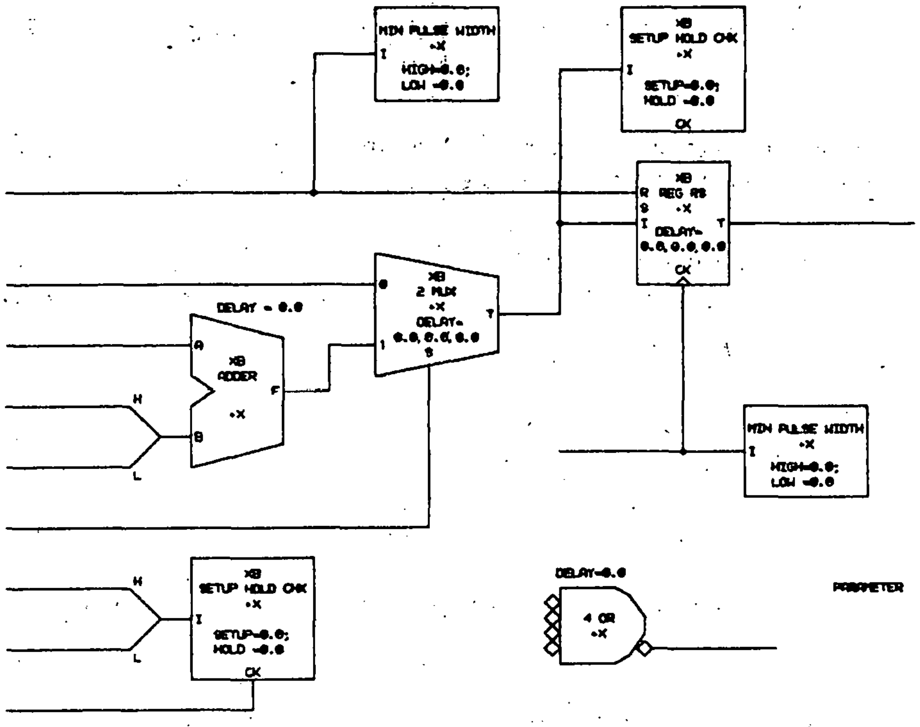


Figure 3–13
Drawing with (almost) all lines

3.5 Putting text on your drawing

The text you'll add to your drawings belongs to either of two categories: signal names and body parameters.

Signal names–Before you can add signal names, the program must be in point mode. (It is probably already in point mode if you've been following these instructions, but if not, type "αβP".)

Typically, you put text on a line near a point where the line ends in midair. To illustrate, let's label the line at the lower left corner of the drawing. First, move the cursor close enough to the midair point that "PL" flashes atop the point. Type:

```
*βTTEXT?
CK/P
```

And that's all there is to it.

If you make a mistake, simply repeat the command and retype the text; the new version will replace the old.

For practice, move the cursor upward and label the line above that one:

```
*βTTEXT?
CE
```

and move the cursor upward once again to label the next line, too:

```
*βTTEXT?
PE
```

This works fine so long as the signal names are short and you are a fairly good typist. When both of those conditions cease to be true and the probability of making an error every time you retype the signal name to correct an error therefore approaches unity, it's lucky that the graphics editor provides for you a simple text editor.

This text editor is called *alter submode*. To illustrate its use, let's deliberately put the wrong text on the fourth line up from the lower left corner:

```
*βTTEXT?
NOW IS THE TIME
```

Now type "αβA" to enter alter submode, which will show you the text plus a *pointer*, a horizontal line under the characters which bends upward at its right end to mark the current editing position. On the screen, you'll see the text you're editing in large letters at the top and the characters you type in small letters near the bottom. To make the following discussion more compact, we'll act as if they appeared together on alternate lines:

```
*αβA
_NOW IS THE TIME
```

To move the pointer forward to the next occurrence of a character, type "S" followed by the character (with no intervening <RETURN>). It will stop just beyond that character:

```
*S
←M
NOW IS THE TIME
```

To move the pointer backward to the previous occurrence of a character, type "-S" followed by the character. It will stop just in front of that character.

```
*-
*S
←E
NOW IS THE TIME
```

In addition, you can type a space to move the pointer forward one character at a time or a <RUBOUT> to move it backward one character at a time. To delete characters to the right of the pointer, type the number you'd like to delete, followed by "D":

```
*2
*D
NOW IS THTIME
```

To delete characters to the left of the pointer, use a negative number instead:

```
*-
*2
*D
NOW IS TIME
```

To insert characters at the pointer, type I followed by the characters you'd like to insert, ending with a <RETURN>:

```
*IINSERT TEXT?SUPPER
NOW IS SUPPERTIME
```

There are a number of other, more powerful commands within Alter submode, some of which are described in Section 2.7, but the ones you just saw should suffice for now. After you've eradicated the damage we just did and you feel satisfied with the result, press the <ALT> key to leave alter submode and return to point mode:

```
*4
*D
NOW IS SUPPER
*-
*1
*3
*D
*/INSERT TEXT?-PE[0.5]
-PE[0.5]
*<ALT>
```

Just for practice, put text on the rest of the signal lines that end in midair, using alter submode if you find it helpful in correcting mistakes. When you finish, the drawing should look like Figure 3-14.

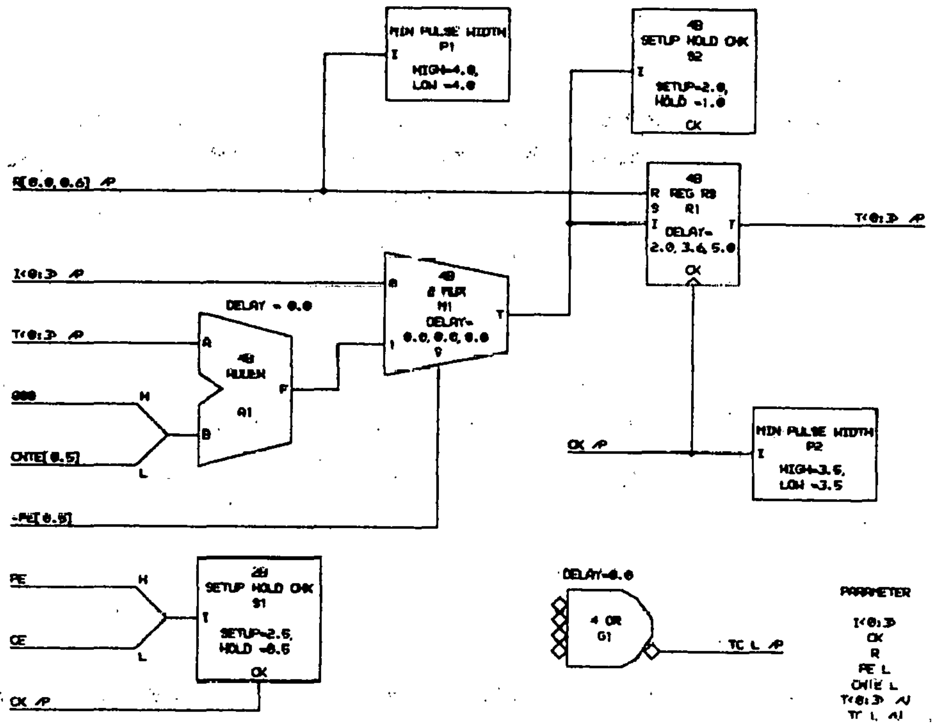


Figure 3-14
Drawing with (almost) all signals labelled

Text for bodies--The text you see on a body can be either of two kinds: simple text which, like a signal name, consists of a string of characters at a particular place; or the text portion of a property name/text pair, a piece of text which has an invisible name that you can use to access it.

On your drawings, however, you'll need to deal only with property name/text pairs. Usually the body comes to you from the library with these name/text pairs already created. To change one, you simply ask for it by name and tell the program what to use for the text. Sometimes, you have to create the property name/text pair yourself.

For most bodies, you'll deal with three properties:

SIZE	usually appears above the body name and arrives from the library set to "XB". You'll want to change it to reflect the number of bits the body is supposed to deal with, such as "4B".
LOC	usually appears below the body name and arrives from the library set to "+X". You'll want to change it to the location code which, as mentioned earlier, helps differentiate between multiple occurrences of the same kind of body in one drawing: something like "G7" or "A2".
VAR	is additional information about the body for later use by SCALD. It can begin with "DELAY=" or "SETUP=" or "HIGH=" followed by a series of numbers.

To illustrate, let's start with a body for which all three property name/text pair already exist, but need changes: the one at the lower left corner of the drawing.

To work with body text, you must first get into body mode by typing "αβB". You will see "MODE=B" on the top line of your screen.

Now move the cursor close enough to the body at the lower left corner of the drawing so that a large "B" flashes atop it. Now when you ask to work with a particular property, the program knows it must be a property associated with that body.

First, you must change the "XB" to "2B". As we just explained, this is doubtless the property called "SIZE", so type the following command to replace the text associated with that property name:

```
*βYPROPERTY NAME (ENOUGH TO UNIQUELY SPECIFY IT)?
SIZE
SIZE
NEW TEXT?
2B
```

You'll see the "X" magically change to a "2". Actually, the program tolerates shortcuts when you type the property name. Since no other property begins with "S", you could have typed "S" instead of "SIZE". Notice that the program echoed "SIZE". If you do get into the habit of using this shortcut, it's not a bad idea to check the echo to make sure you really get the property you want. If not, you can escape from the command by pressing the <ALT> key.

Now do the same sort of thing for the "LOC" property:

```
*βYPROPERTY NAME (ENOUGH TO UNIQUELY SPECIFY IT)?
LOC
LOC
NEW TEXT?
S1
```

and magically the "+X" will change to an "S1". Finally, change the "VAR" property, noting that you use the "↔" character instead of the <RETURN> key to break the text across two lines:

```
*βYPROPERTY TEXT (ENOUGH TO UNIQUELY SPECIFY IT)?
VAR
VAR
NEW TEXT?
SETUP=2.5;↔HOLD =0.5
```

If you make a mistake, simply repeat the command and type the text again, correctly. The new version will replace the old.

Sometimes you'll have to create a property name/text pair for yourself. On this drawing, a good example is the PAR body, where each signal name in the list requires a separate property. To create these, move the cursor over toward the word PARAMETER in the lower right corner and place it where you'd like the center of the first name, "I<0:3>", to be. The name you give to each property isn't important, but by convention we use "0", "1", and so on. Type the following command to create a new property name/text pair:

```
*αYPROPERTY NAME?
0
NEW PROPERTY, TEXT?
I<0:3>
```

Now move the cursor down by a CONTROL and create the next property:

```
*αYPROPERTY NAME?
1
NEW PROPERTY, TEXT?
CK
```

The "αY" command will actually edit an existing property if the name you give has already been used, so it's a good idea to make sure the program prints "NEW PROPERTY" as those examples showed. If not, you can escape from the command by pressing the <ALT> key.

If you mistype a piece of text, you can simply repeat the command and retype it correctly. If you create a property you don't want, or if you inadvertently put a property in the wrong place, you must get into text/property submode of body mode to repair the damage.

To illustrate, suppose there's something wrong with the "CK" text. First, make sure the cursor is close enough to the PARAMETER body so that "B" flashes above it. Type "αβY" and you'll see "MODE=BT" on the top line of the screen, indicating you're in the submode. Now as you move the cursor around, you will see large letters flash atop whichever property is closest to the cursor. Move it so that the letters are atop "CK" and type "αM" to attach the cursor to that property. Now wherever the cursor moves, the property will follow. Try it. When you have that property in an appropriately ridiculous place, press <SPACE> to detach the cursor. Now move the cursor and you'll see that the property no longer follows it.

You may also use alter submode on a property once you're within text/property submode. Simply move the cursor close enough so that the big letters flash above the text you want to edit, and type "αβA". Then you can proceed as you did when editing signal names. When you press the <ALT> key to leave alter submode, you'll find yourself back in text/property submode as you were before.

Suppose you want to get rid of the property altogether. Make sure the cursor is close enough that the big flashing letters are atop our intended victim, and type "αD" to vaporize it. Now that you're finished playing, type "αβB" to return to ordinary body mode.

You should observe two important limitations about body text/property submode. First, you can edit and delete only the text that you yourself have created or at least replaced, not text that arrived along with the body from the library. For example, you cannot edit a SIZE property that still has its original "XB", but you can edit it once you have replaced that "XB" with "2B". Second, everything you do within the submode applies to the body that was closest to the cursor when you entered that submode. The program will let you move the cursor to another body while you're still within text/property submode, but before you can deal with properties associated with that body you must get out of the submode and back in again.

Now that we've enticed you into destroying the perfectly good "CK" property you just created, practice your property creating and replacing skills by completing the properties for the rest of the drawing. When you're finished, the drawing should look like Figure 3-15.

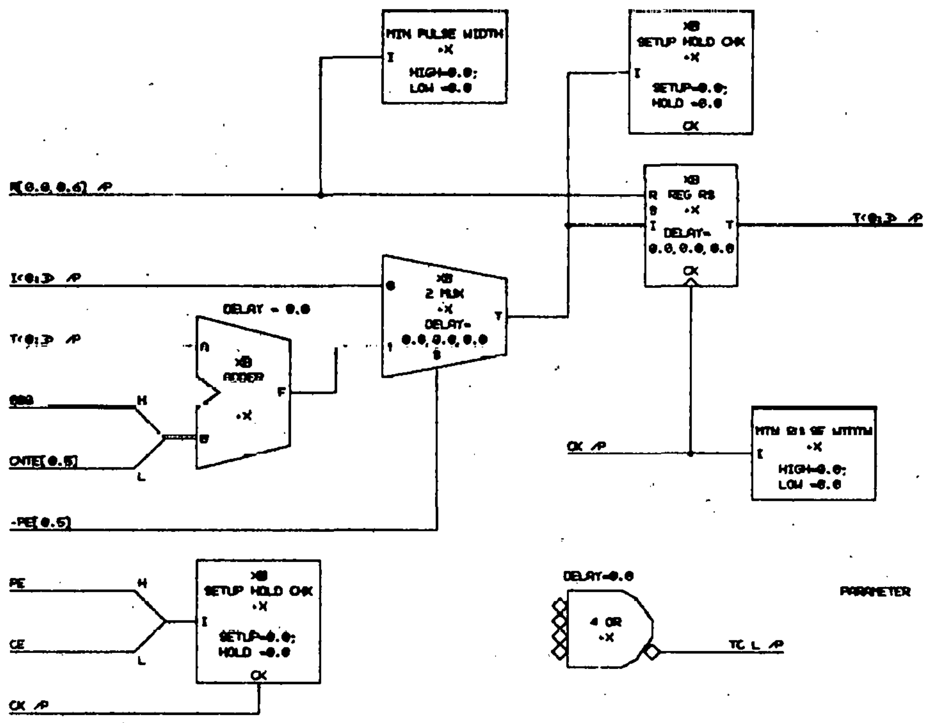


Figure 3-15
Drawing with body text finished

3.6 Editor macros

By now you are no doubt wondering why we have postponed so long drawing the four lines to the left of gate G1.

The reason is that they're an excellent way to demonstrate the use of graphics editor macros to eliminate repetitive typing. If you were to draw those four lines and label them in the obvious fashion, you would wind up doing almost exactly the same thing four times in a row.

An editor macro lets you draw and label one line, and then tell the program to repeat the process three additional times for you.

To see how it works, get into point mode by typing "αβP". Move the cursor close enough to the top diamond so that "PA" flashes above the diamond, type "αM" to attach the cursor to that point, and press <SPACE> to detach it. The attaching and detaching simply assures that the cursor is really directly atop the point; precise alignment is important when you're using macros.

To begin the macro, type "α;P" (use the CONTROL key on the ";" but not on the "P"). From now on, each command you type will execute, changing the drawing; but the program will also save each command into the macro for future use. On the top line of the screen, you'll see "LEVEL=1", showing that you're one level deep inside a macro. If you get confused or make a mistake while inside the macro, type "α;S" to escape. You can then delete whatever the macro has done so far and start over.

Type "+" to start a line, move the cursor left by one TOP, and type "+" to solidify the line. Then press the <ALT> key to finish off the line.

Type "βT" to label the midair end of the line with its signal name. Now we have a slight problem: the signal name ought to be slightly different for each line we want the macro to draw. The first line represents bit 0, the second represents bit 1, and so on. Fortunately, the program's macro facility provides a "α;A" command that puts a counter in the middle of the macro for you. The first number after the "A" gives the initial value for the counter, and the second number gives the increment. Thus, the part of the macro that creates text will look like this:

```
*βTTEXT?
T L<α;
←A0
END ;A
→>
```

Now move the cursor down by a CONTROL and right by a TOP. That puts it back where it was when we started the macro, except that it's now on the second diamond rather than the first. Now we want to stop adding commands to the macro and to have the program repeat the commands three additional times to produce a total of four lines. To accomplish this, type "α;R4" and press <RETURN>. You'll see three more lines appear below the one you just drew. On the top line of the screen, "LEVEL=0" shows you that you're no longer inside the macro.

Now that you've used it, the macro vanishes. Section 2.8 explains several other useful macro commands, including one which allows you to create a macro that remains after you've used it, but for now this should suffice.

3.7 Using sets

As you'll discover if you try, the program is very obliging about patching up the damage that occurs when you move a body that has signals attached to it. Often it manages to stretch lines and introduce bends carefully enough that the result is still pretty.

When you want to move a group of objects including one or more bodies and one or more points and one or more lines, however, you can avoid this interobject stretching by defining a *set* that includes the objects, and then moving the entire set together.

To illustrate this, let's move gate G1 around a bit. First, get into set mode by typing "αβS". On the top line of the screen you'll see "MODE=S".

Now the strategy is to draw a box around the part of the drawing you want to move (in this case, encompassing the lines cconnected to the inputs of gate G1 in addition to the gate itself). Every body and point inside the box belongs to the set, and if both endpoints of a line are inside the box then effectively the line belongs, too.

Move the cursor to the spot you'd like to become the upper left corner of the box and type "α+" to start a line. Move the cursor to the right until the line is long enough and type "α+" to make the line permanent. (In set mode, by contrast with point mode, the line won't stop flashing when you do this.) Move the cursor downward until the right side of the box is long enough and type "α+" again. Move the cursor left until the bottom of the box is long enough and type "α+" again. Finally, move the cursor up to complete the box and this time type "α-" to finish it off. (Actually, the program offers you a shortcut. After you've drawn the first two sides, you can simply type "α+α-" without moving the cursor and the program will complete the third and fourth sides for you.)

As soon as you complete the box, a big flashing letter will appear atop each point or body in the set, and a big flashing "S" will appear near the center of the set. Type "αM" to attach the cursor to the set. Now try moving the cursor; you'll see the entire set move with it. As with individual points and bodies, you press <SPACE> to detach the cursor and leave the set where it is. You can then go define another set elsewhere in the drawing; within set mode, as within body or point mode, commands always refer to the set closest to the cursor.

Once the cursor is detached you can delete the set definition by typing "αD". This doesn't delete any points or bodies, but merely releases them from membership in the set. Typing "βD" (don't try it!) deletes each point and body belonging to the set closest to the cursor.

When you're tired of playing with sets, you can type "αβB" to return to body mode or "αβP" to return to point mode.

3.8 Final touches to your drawing

Just a few steps remain before your drawing is finished.

Because it's so easy to place one point atop another, the program provides a command to check for this and other faults. Type "αβP" to get into point mode, and then type:

```
*X
WELL?DANGLE
DANGLE
```

If the program tells you "NO MATCHES FOUND", congratulations—it found no errors. Otherwise, press "αF" and the program will place the cursor atop the first error. Section 2.2.3 explains possible errors and the usual solutions. After you correct the first one, type "αF" to move the cursor to the next (if any), and so on until typing "αF" ceases to move the cursor.

When you think you've fixed them all, try the "X DANGLE" command again, just to be sure.

Now the drawing is fine as far as the graphics editor is concerned. You must give it a title so that SCALD knows what body this drawing defines. Type:

```
*X
WELL?TITLE
TITLE
NEW TITLE LINE 1?10016
NEW TITLE LINE 2?SIZE=4B
```

Finally, write the files that hold the drawing, allow you to plot it on paper, and provide a wiring list for ultimate use by SCALD:

```
*X
WELL?EW, EP, EL, CLE
```

You've finished the drawing. In the next section of this chapter, you'll learn how to describe a body to represent the drawing.

3.9 Creating a body template

Now that you've drawn the definition of an ECL 10016 IC, you need a template for a body that you can use in your drawings to represent that IC and invoke its definition. In a realistic situation, you'd probably add the template for this body to an existing library, perhaps one called "E10K", and then place the body on a menu drawing, perhaps one called "E10K1", that shows people what bodies are available from that library. But since this is the first body template you've created, we'll assume you want to create a brand new library file and menu file.

3.9.1 Getting started

First, imagine that a flash of inspiration tells you that the body should look like Figure 3-16. Now clear your workspace and perform a few necessary initializations to establish the file "E10K" which will become your library:

```
*X
WELL?-LOCS
-LOCS
*X
WELL?BOARD
BOARD
BOARD TYPE?DECPC
*αW
FILENAME?E10K
WRITING E10K.DRW[MK2,S1]
*X
WELL?GETLIB
GETLIB
LIBRARY NAME?STDLB
  LIBRARY STDLB.DRW[MK2,S1]
```

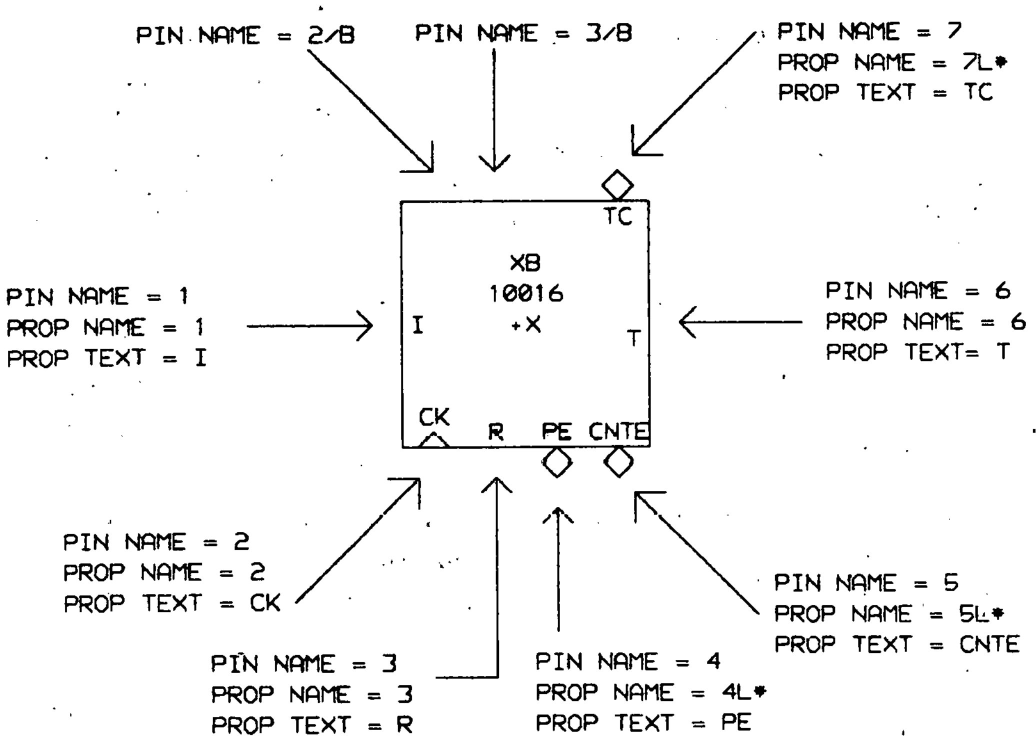


Figure 3-16
This is what you want to create

Body templates are hidden in the background of a file, entirely separate from the drawing portion of the file, if indeed there is one. To create or edit a body template, you enter *Edit mode*, in which the drawing vanishes temporarily. Of course, you won't notice this because you haven't started a drawing in this workspace (and you probably never will since it's supposed to be used as a library). Type:

```
*αβETYPE BODY NAME?
10016
NEW BODY.
```

On the top line of your screen you'll see "MODE=E". At the center of the screen you'll see the familiar cursor. The initial position of the cursor is special because it will become the *origin* of the body. When, in the process of making a drawing, you attach the cursor to a body, it moves to the origin; and when you detach the cursor, the flashing letter "B" appears atop the origin. By convention, though nothing in the program demands it, we always draw a body so the upper left corner is the origin.

Within edit mode, four submodes let you perform four different tasks: add or insert lines to define the body shape, grab bodies from elsewhere to add to this one, create pins to which you can connect signals, and label the body and pins with text.

3.9.2 Drawing the box

First, get into insert submode by typing "αI". You'll see "MODE=EI" on the top line of the screen.

Now we want to draw lines. Edit mode is deceptively similar to point mode, but with two important differences. First, you're not drawing lines between existing points, but creating a series of brand new vectors. Some of the vectors are visible and others are invisible, but each starts where the previous one leaves off. Second, the program is perfectly willing to let you draw slanted lines, and thus doesn't bother with pairs of lines at right angles as it does in point mode.

Type "α+" to start a visible vector and move the cursor to the right by TOP. Type "α+" a second time to finish that vector and begin another, and move the cursor down by TOP. Type "α+" a third time to finish that vector and begin a third, and move the cursor left by TOP. Finally, type "α+" a fourth time and move the cursor back up to the origin by TOP.

To finish off that fourth vector without starting another, press <ALT>. You'll find yourself out of insert submode and back in plain edit mode.

Actually, there was no need to get out of insert submode just then, because we're about to draw more vectors, but we wanted to illustrate the use of <ALT>, which is the only way to end a vector without starting a new one.

3.9.3 Ornaments

That produced a fine box, but we're missing the triangle that represents the clock input, and the three diamonds that represent inputs and outputs that assert low. Once you've drawn the outline of your body there are two ways to add details to it: use invisible vectors to skip around, or grab bodies from elsewhere.

Based on your experience with the program so far, you might guess you could move the cursor to the point at which you'd like to add something, enter insert submode, and start using the cursor to draw vectors. If you try that, however, you'll find that the cursor hops back to the origin on its way into the submode. The program insists that a body consist of a single path of vectors, so the only way to add to the body is to follow the path to its end and append vectors there. If you want to go back to a point along the path, you must go to the end of the path and append an invisible vector that jumps back to the desired point.

Type "αI" to enter insert submode again. The cursor will appear at the origin. Now press <SPACE> repeatedly. The cursor will travel one vector's length along the path every time you do so. When it stops moving and the drawing doesn't even blink at <SPACE>, you've reached the end of the path and can add more vectors.

Invisible vectors--To start an invisible vector, type "α-". Now move the cursor downward by TOP and to the right by a single unit (that is, an unshifted keystroke), and it will rest where the left edge of the triangle should be (Figure 3-17).

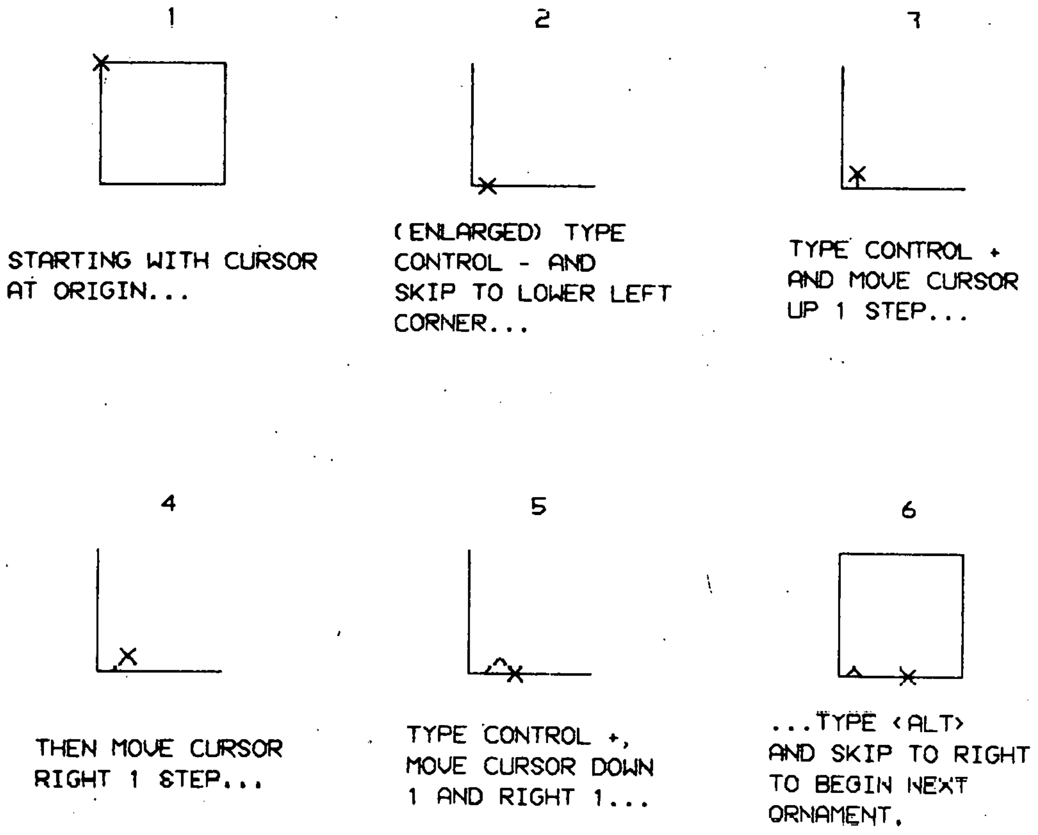

1
STARTING WITH CURSOR AT ORIGIN...
2
(ENLARGED) TYPE CONTROL - AND SKIP TO LOWER LEFT CORNER...
3
TYPE CONTROL + AND MOVE CURSOR UP 1 STEP...
4
THEN MOVE CURSOR RIGHT 1 STEP...
5
TYPE CONTROL +, MOVE CURSOR DOWN 1 AND RIGHT 1...
6
...TYPE <ALT> AND SKIP TO RIGHT TO BEGIN NEXT ORNAMENT.

Figure 3-17
A closeup view of drawing the triangle

To draw the left slant, type "α+", move the cursor up by a single unit and to the right by a single unit. To draw the right slant, type "α+", move the cursor down by a single unit and to the right by a single unit.

Obviously, one could now skip to the point at which the first diamond should appear, and draw it in the same fashion. But there's an easier way, so press <ALT> to leave insert submode.

Grabbing bodies from elsewhere--The easier way is to grab a copy of an existing body and add it to this one. Suppose that one of your libraries contains a body called DIAMOND which is useful for indicating that a pin asserts low.

To grab it, move the cursor so it rests on the bottom line of the box, five CONTROLs from the lower left corner. Then type:

```
*αGTYPE BODY NAME
DIAMOND
SEARCHING FOR DIAMOND IN STDLB.DRW[MK2,S1]
```

First you'll see "MODE=EG" appear on the top line of the screen and then you'll see the diamond appear at the cursor position. The cursor is actually attached so that the diamond will follow it wherever it goes. Experiment with moving it. When you have the diamond back where it belongs, press <SPACE> to release the cursor. The diamond will become part of the body, and you will find yourself back in normal edit mode.

To add the next diamond, move the cursor to the proper position and use the "αG" command again. Finally, move up to the top line and add the third and last diamond.

3.9.4 Defining pins

Corresponding to the seven inputs and outputs listed under PARAMETERS in the drawing we just made, the body has seven labelled pins. In addition, it has two invisible "bus-through" pins at the top. Pin "2/B", for example, is identical electrically with pin 2, as if any signal you connect to pin 2 travelled underneath the body and reemerged on the opposite side (Figure 3-18).

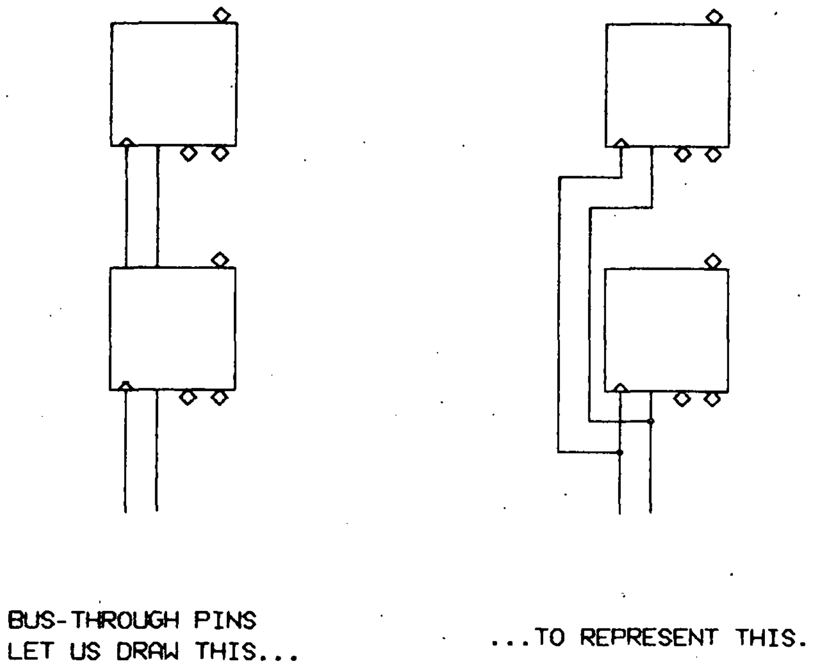


Figure 3-18
Using bus-through pins

To define the pins, type "αβP" to enter pin mode. You'll see "MODE=EP" on the top line of the screen. Ordinarily we don't want pin numbers on drawings, because they represent unneeded clutter, but until you have a chance to create labels near the pins, pin numbers make it easier to find the pins, so type:

```
*X
WELL?DEFPIN
DEFPIN
```

As you create each pin, you will have to supply its pin number (which the program refers to as a "pin name"). To begin, move the cursor to the midpoint of the left side of the box and type:

```
*αPPIN NAME?1
```

Now move the cursor down to the bottom line of the body, directly under the apex of the triangle, and type:

```
*αPPIN NAME?2
```

Continue until you've created all the pins. If you make a mistake, move the cursor to the erroneous pin, type "αD" to delete it, and then use "αP" to create it anew.

When you're finished, hide the pinnames once again so they won't clutter your drawings:

```
X
WELL?-DEFPIN
-DEFPIN
```

Press <ALT> to leave pin submode and return to plain edit mode.

3.9.5 Creating body text

The last step in describing your body is to create a property name/text pair for each of the pins, plus a few pairs for the body as a whole. Type "αβT" to enter text/property submode. You will see "MODE=ET" on the top line of the screen. Within this submode you can create, alter, delete, and move text for a body template just as you would edit text in ordinary body mode or body text submode for the drawing as a whole.

Each visible pin has a property name/text pair associated with it. To begin, move the cursor to the center of the area where you'd like the label for pin 1, "I", to appear. Create a property name/text pair :

```
*αY
PROPERTY NAME?
1
NEW PROPERTY, TEXT?
I
```

Now move the cursor to the spot where you'd like the label for pin 4, "PE" to appear. As Figure 3-16 shows, the property name for this pin is more elaborate. We'll postpone explaining the reason until chapter 4, but note that the difference is important to SCALD:

```
*αY
PROPERTY NAME?
4L*
NEW PROPERTY, TEXT?
PE
```

For the sake of practice, create the property name/text pairs for the rest of the visible pins—R, CK, CNTE, T, and TC. When you're finished, the body should look like Figure 3-19.

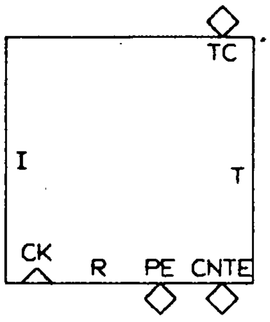


Figure 3-19
Body with all pin properties

As you've probably guessed, the "XB" in the middle of the body is the text of property SIZE and the "+X" is the text of property LOC. The name of the body, "10016", is the text of a property called MNAME. For each of these, position the cursor to the center of the place you'd like to put the text and use "αY" to create the property. Remember that if you make a mistake, you can proceed as we did with properties in the drawing as a whole: attach the cursor to the one you want to correct and either move it, delete it, or use alter submode to edit it.

When you're satisfied with the text, the body is done. Press <ALT> to escape from text/property submode and return to normal edit mode. Then type "αE" to escape from edit mode.

Now save the file and clear your workspace:

```
*X
WELL?EW, CLE
EWRITE
WRITING E10K.DRW[MK2,S1]
CLEAR
```

3.10 Making a menu file

If you were to plot the library file "E10K" which you just created, you'd see nothing. You've described the body "10016", but you won't see it until you use it in a drawing. It turns out to be convenient to put the body templates in one file for use as a library and to put a rogues' gallery of their portraits in another file for use as a menu.

Now that your workspace is clear, type the following to create a new file "E10K1" to serve as a menu:

```
*αW
FILENAME?E10K1
WRITING E10K1.DRW[MK2,S1]
*X
WELL?-LOCS, GETLIB, SHOWBOX
-LOCS
GETLIB
LIBRARY NAME?E10K
SHOWBOX
A16/1
```

Now move the cursor to an appropriate point--near the upper left, for example--and place a copy of body "10016" in the drawing:

```
*αPTYPE BODY NAME
10016
SEARCHING FOR 10016 IN E10K.DRW[MK2,S1]
```

Press <SPACE> to detach the cursor from the body. In this case, it's obvious what name to use when you want to fetch a copy of the body from the library: 10016. Sometimes, however, the name of the body may differ from the text of the MNAME parameter that appears on the body, so it's a good idea to put the name above the body on the menu drawing. To do this, move the cursor to an appropriate place near the body and create some text:

```
*αTTEXT?
10016
```

In a realistic case, you would fill the menu with many different bodies belonging to the ECL 10K family, but since you have defined only one so far, save the file and you're finished:

```
*X
WELL?EW, EP, CLE
```

And that's the end of the tour. Congratulations. You have made a drawing to define ECL IC 10016, a body template to represent it, and a menu drawing to advertise its existence to the world.

4 How to use the macro expander

Like the graphics editor, the SCALD macro expander deals with bodies and the lines connecting them.

A body represents a logic element, and the pins on the body to which you may connect signals represent inputs and outputs.

A line between bodies represents a signal, whose characteristics are determined partly by the bodies it connects and partly by the name, if any, used to label it on the drawing.

A body may represent a macro—a functional block which must be expanded into the logic elements that comprise it—in which case an additional drawing must exist to define it in terms of other bodies connected with signals. Or a body may represent a *terminal*--a fundamental, irreducible logic element—in which case the program looks for entries in a special text file called the TERM file (or *terminal file*), which describes the body's inputs and outputs.

Visually, connecting a line from one body to another with the graphics editor "feeds" the signal from the output of one body to the input of the other. Effectively, this calls the two macros, using the output parameter of one macro as the input parameter of the other. The task of the SCALD macro expander is to replace each body which represents a macro with the set of bodies and signals which define that macro. Because a macro may be defined in terms of additional macros, the program repeats the process until it obtains a network of bodies and signals in which all the bodies are terminals.

Exactly what constitutes a terminal depends on whether you want to obtain a wirelist for actually building a prototype, or whether you simply want to simulate the logical design. When building a prototype, you regard a macro as a terminal if it corresponds to an actual IC or chip to be used in the prototype. By using a TERM file containing entries for these chips, the macro expander produces a list of chips and interconnections for use by the SCALD physical design system.

When simulating a design, however, you define each IC or chip with a drawing that uses only *logical primitives*--that is, idealized gates, adders, latches, multiplexers and so on--which the simulator can deal with. By using a TERM file containing entries for the logical primitives, the macro expander can expand the design past the IC or chip level, producing a network of logic primitives for use by the SCALD simulator and timing verifier. (The behavior--the truth or state table--of each logical primitive is built into the simulator and verifier.)

Thus, the choice between an *IC terminal file* and a *primitive terminal file* determines the operation of the macro expander.

Incidentally, SCALD has no trouble dealing with an IC or chip which contains several copies of a particular logic function--a quad latch or dual flip-flop, for example. In such a case, the body representing that terminal, the entries for it within the IC terminal file, and the drawing defining the terminal in terms of logical primitives all pertain to a single copy of the function. A file called CHIPS (which also contains electrical characteristics of the IC inputs and outputs) takes care of telling the physical design system that it can obtain multiple copies of the function from a single package.

4.1 Typical design procedure

The remainder of the chapter is full of rules for the syntax of bodies and signals that makes up the input language for the SCALD macro expander. Those rules may make better sense after an outline of the typical procedure for designing a large project with the macro expander:

1. For each kind of IC to be used in the design, make entries in the IC Terminal File.

2. If you will want to simulate the design, make a drawing for each IC that defines it in terms of the logical primitives--adders, gates, and so on--available in the graphics editor library "SIMLB". (These primitives are sufficiently general to apply regardless of the actual logic technology used to implement the design, though the numbers for timing will of course vary.) If the IC contains multiple units in one package--a quad latch, for example--the drawing should define a single unit.

3. Develop a graphics editor library containing a body template for each of the ICs. The S-1 Mark II designers, for example, developed a library called "E10K" containing bodies representing ECL 10K parts. Note that the body template for a multiple-unit IC should represent a single unit.

   Since body templates are invisible until used to place bodies in drawings, it helps to make one or more menu drawings for each library. A menu drawing simply shows each IC available in the library, and next to it gives the proper name for its macro. SCALD itself doesn't use these menus, but they aid designers in picking out the proper bodies to use. For the ECL 10K library, for example, the menu drawings are called "E10K1", "E10K2", and so on.

4. Now define any macros expected to be used frequently throughout the design, invent body templates for them, and place those bodies in a library so designers can find them easily. As work progresses, designers can add new templates as needed.

   For example, the S-1 Mark II design frequently uses vision registers, registers with auxiliary logic that accesses the register contents for diagnostic purposes. Placing in a library a set of body templates representing vision registers makes it as easy for a designer to incorporate one of them as it is to design with an ordinary register.

5. Now the designers can start at the top level of the machine and proceed hierarchically down toward the bottommost, detailed level. At each level, the designer makes a drawing by connecting signals between bodies representing macro calls and/or terminals. The designer can obtain bodies for the terminals from the templates in the library described above. For a body representing a macro call, the designer must invent a body template and then make a further drawing defining that body in terms of additional bodies; thus the process recurs.

6. SCALD does not consider the top level drawing in the hierarchy as a special case, so to start the macro expansion process, someone must invoke that drawing through a macro

call. The usual approach is to make a dummy drawing of the "universe" consisting of appropriate drivers and receivers attached to a single body representing the entire design. When it comes time to lay out hardware to implement the machine, simply allocate the contents of this drawing to a separate circuit board which never actually gets built.

While this outline suggests proceeding hierarchically from the top level of the design toward the bottom, SCALD is actually quite flexible in this respect. If it becomes obvious at some point that the design calls for additional types of ICs or that some functions occur so frequently that it is worth repackaging them as standard macros, it is quite easy to change these aspects.

It is possible to expand the upper levels of the design to check for syntax errors and design rule violations even if the lower levels are not finished--simply ignore the errors generated by the missing drawings. Similarly, it is possible to expand a subsection of the design--a subtree within the hierarcy—without expanding the design, simply by concocting a dummy "universe" file that calls the topmost drawing of the subsection rather than the topmost drawing of the entire design.

4.2 General Rules for the macro expander language

Expressions--Wherever the macro expander accepts an integer, it will generally accept an expression instead. The expression syntax is that of a subset of PASCAL, which includes the following operators (where "0" indicates the highest precedence):

Symbol	Meaning	Precedence
NOT	Logical NOT	0
-	Unary minus	0
+	Unary plus	0
*	Multiplication	1
/	Integer division	1
MOD	Modulo	1
AND	Logical AND	1
+	Addition	2
-	Subtraction	2
OR	Logical OR	2
=	Equals	3
<>	Not equals	3
<=	Less than or equal to	3
>=	Greater than or equal to	3
>	Greater than	3
<	Less than	3

Parentheses override precedence as usual in Algebra.

When the macro expander needs to convert a logical value to an integer, it treats "FALSE" as 0 and "TRUE"as 1. When it needs to convert an integer to a logical value, it treats "0" as false and anything else as "TRUE". Thus, the following example evaluates to either "SIGNAL<0:5>" or "SIGNAL<1:5>":

```
SIGNAL<ASIZE=15:5>
```

Note that within a bit subscript (Section 4.6.5), which normally uses "<" and ">" as brackets, you must parenthesize an expression that uses ">" to mean "greater than", or the macro expander (which parses with limited lookahead) will think it has reached a right bracket:

```
SIGNAL<(ASIZE>15):5>
```

Throughout the the macro expander language, integers can end in "X" (for "times") or "B" (for "bits") to improve readability; thus "5B" and "5X" are the same as "5".

Signal, pin, and macro names--While most programming languages prohibit blanks or spaces within identifiers, the macro expander permits them in signal names, pin names, and macro names. And while most languages require identifiers to begin with an alphabetic character, the macro expander permits digits. Thus, it's perfectly legal to use the kind of multiple word signal names and numeric part names that designers are accustomed to:

```
P SEQUENCER
PARITY CHECK INHIBIT
54LS181
CLK ENABLE
```

This freedom is possible because the graphics editor, with its "text" and "property" features, takes care of specifying where one chunk of text begins and ends, so the macro expander does not need to reserve blanks for use in delimiting such chunks.

In general, the macro expander deletes leading and trailing blanks in names, and reduces several consecutive blanks to a single blank.

As noted in chapter 2, the graphics editor also allows the use of the "↔" character to split a piece of text across two lines. It converts that character to a blank before sending the text to the macro expander, however.

4.3 Inventing Bodies to Represent Macros

Parameters--A SCALD macro accepts two kinds of parameters. Pin parameters represent signal inputs and outputs, while body parameters specify some general characteristics of the macro.

Now, most programming languages match actual parameters (the values or variables you plug into a macro or procedure) with formal parameters (the dummy arguments that specify what inputs and outputs the macro wants to see) strictly by their position in a list. The macro expander, by contrast, matches actual parameters with formal parameters by name; whenever you feed a parameter to a macro, you implicitly or explicitly state the name of the formal parameter you're dealing with. The property name/text feature of the graphics editor helps accomplish this.

For a pin parameter, the pinname points to a property name, and the text paired with that property name gives the formal parameter name.

For a body parameter, the property name holds the formal parameter name and the property text holds the actual parameter. Thus, to set the SIZE body parameter to "14B" for a particular macro, use the graphics editor to create or modify the property named SIZE and then specify "14B" as the property text.

4.3.1 Body Parameters

By convention, SCALD macros have up to five standard body parameters; whereas signal parameters are invented by the designer and vary from one macro to another, body parameters are concepts built into the macro expander which govern the way it expands each macro. Body parameters are somewhat unusual in that some of them have an initial value which will appear in the drawing until you supply a value.

MNAME Actually not a parameter, but rather the name of the macro. To find the definition of the macro, the macro expander will search for a drawing with this name in the first line of its title. (It may also use a selection equation as explained in Section 4.4.)

SIZE Basically, an integer specifying how many times the macro should occur. This is useful for creating several independent copies of a macro--for example, to generate 36 copies of a flip-flop to build a register to store data from a 36 bit bus, set SIZE to 36.

A more precise explanation is that the macro expander invokes any macro repeatedly in a loop using a special counter variable "X", which starts at X FIRST, increments by X STEP, and quits at SIZE-1. You can set X FIRST and X STEP using the DEFINE list described later in this chapter.

X FIRST defaults to 0. X STEP defaults to 1 if SIZE is 1, but otherwise you must (as a safety feature) explicitly set X STEP. Failing to do so produces an error message and sets X STEP equal to SIZE.

The variable "X" is available for use within the macro definition, and will be replaced with successive loop-counter values when the macro expander expands and replicates the macro.

The initial value of SIZE is "XB", deliberately chosen to be nonnumeric and therefore invalid so that the system will produce an error message if you forget to specify a size.

TIMES An integer telling the macro expander to invoke the macro repeatedly to obtain multiple copies, and then to tie together the corresponding inputs on all the copies while leaving the outputs independent. This is useful as shown in Figure 4-1 when you'd like several different gates to produce the same signal because a single gate doesn't have enough fanout capability. If not specified, the TIMES parameter defaults to 1. If TIMES is 0, the macro expander ignores the body instead of expanding it.

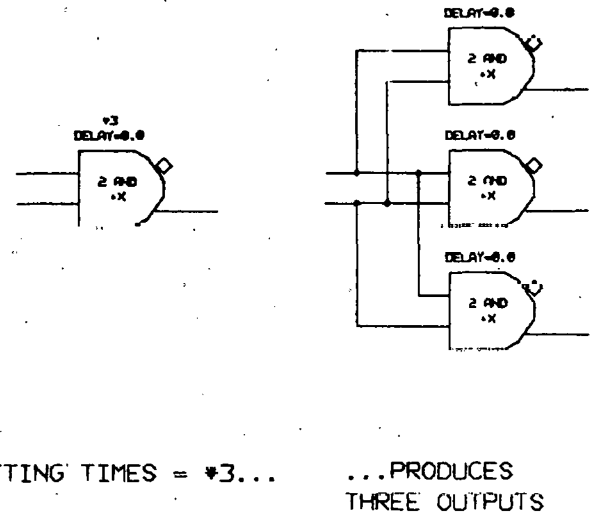


Figure 4-1
The body TIMES parameter

When the macro expander invokes a macro repeatedly due to the TIMES parameter, it sets a special variable called TIMES to a different value on each invocation, starting at 1 and incrementing by 1. You may use this variable to distinguish one invocation from another if you wish.

LOC Within a drawing, every body must have a unique alphanumeric location label; the actual labels don't much matter, but conventionally we label gates as G1, G2, G3... and registers as R1, R2, R3... and so on. It's quite safe to use the same label for two bodies in two different drawings. (These labels are used internally by the macro expander to make local signal names unique when the same macro is invoked in two or more places. Section 4.6.13 recites the details).

The initial value for LOC is "+X", a deliberately invalid choice which will produce an error if you forget to specify a location.

VAR This parameter passes information through the SCALD macro expander to the logic simulator and timing verifier. Its exact purpose varies from one body to another--sometimes it specifies setup and hold requirements and other times it specifies delays—but by convention the initial value will always be something like "DELAY=" or "SETUP=" which explains what the parameter is for.

Figure 4-2 shows two versions of the same body, first exactly as it comes from the library, with parameters set to initial values; and then with values specified by a user.

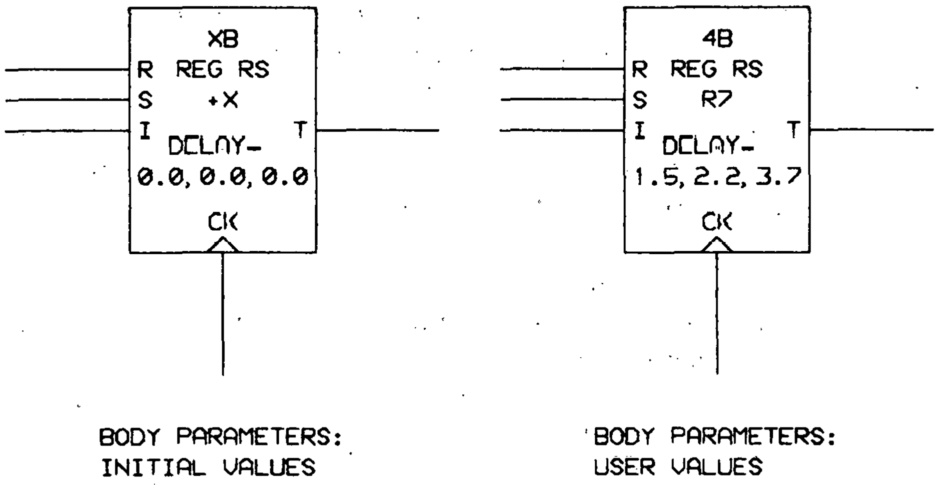


Figure 4-2
Body parameters

Note that while the property names (that is, the formal parameter names) don't appear explicitly on the drawings, the graphics editor will identify any of them within Alter submode as explained in Section 2.7.

One point concerning SIZE and TIMES deserves mention. When you use these parameters on a terminal macro, the physical design system will ultimately generate the specified number of copies of the macro's function.

But when you use these parameters on a nonterminal macro, the definition of the macro determines whether replication actually occurs. If a signal inside the definition has its number of bits expressed in terms of SIZE and TIMES, or if a body has its *own* SIZE and TIMES parameters expressed in terms of SIZE and TIMES, then replication will take place. Otherwise, a signal or body inside the definition is a constant, independent of SIZE and TIMES.

A good analogy is a procedure in a high level language which accepts an integer parameter and then doesn't use that parameter anywhere in its body. Only when the procedure *uses* a parameter does it have an effect.

4.3.2 Pin parameters

The graphics editor always associates a "pinname" (actually a number) with each pin on a body. For each pin, the macro expander requires a property name/text pair that ties the pinname to the corresponding signal parameter name.

If the body in question is a terminal, then by convention each pinname should be the number of the corresponding pin on the actual IC. (If there are multiple units within one IC, use the pin numbers for the "first" unit--the one that has the lowest numbered pin. The CHIPS file will take care of mapping the remaining units onto the first.) If the body represents a macro, the numbering can be arbitrary, but each pin must have a unique number.

The property name/text pair for a pin is derived from the pinname. For the property name, start with the pinname and append an "L" if the corresponding signal parameter inside the macro asserts low (see Section 4.6.4). Then append a "*" if the pin has a diamond or "bubble" on it, telling the macro expander to check to be sure that any signal connected to this pin invokes low.

The property text should include the <Class>, <Simple Name>, and <Timing Spec> portions of the signal parameter name. Essentially, it should be identical with the version of the signal name that appears in the parameter list (Section 4.6.8) but without the "/V" appendage or <Bit Subscripts>. Figure 4-3 is an example.

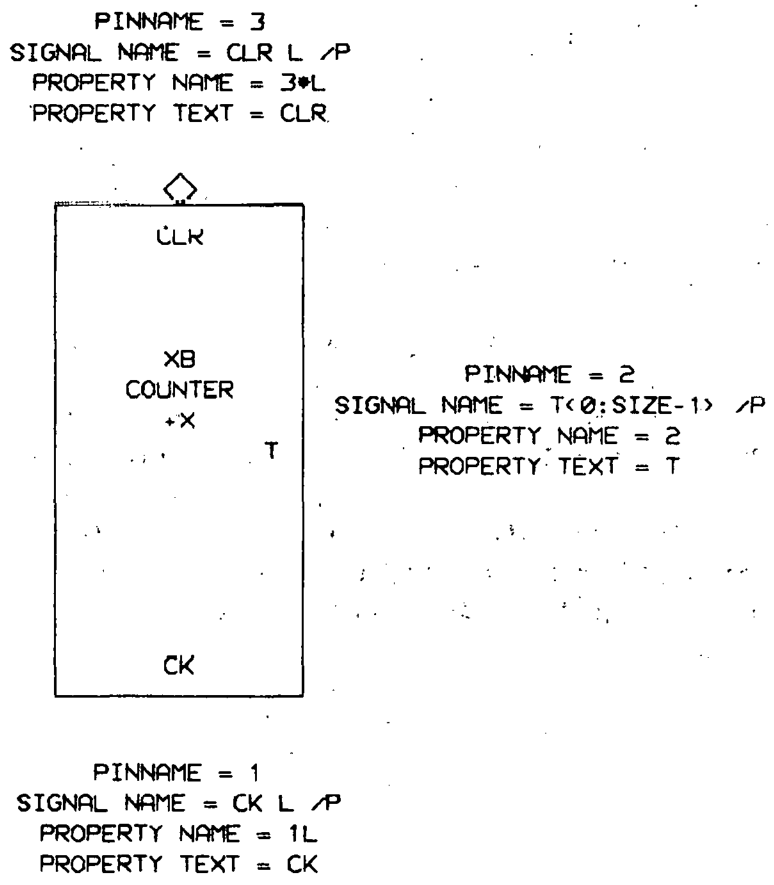


Figure 4-3
Pin properties

Removing the "L" from the property text and putting it in the property name allows the property text to label the pin on drawings; the "L" is customarily omitted in such labels.

Why have a separate "*" to tell the signal checker that the signal asserts low when you already have an "L"? Because the macro's internal notions about the signal polarity may have nothing to do with the outside world. Consider the case of an AND gate which could just as well be represented as an OR gate for inputs and outputs that assert low. A single macro defines both gates equally well, but as Figure 4-4 shows, one body expects its inputs to assert low and the other doesn't.

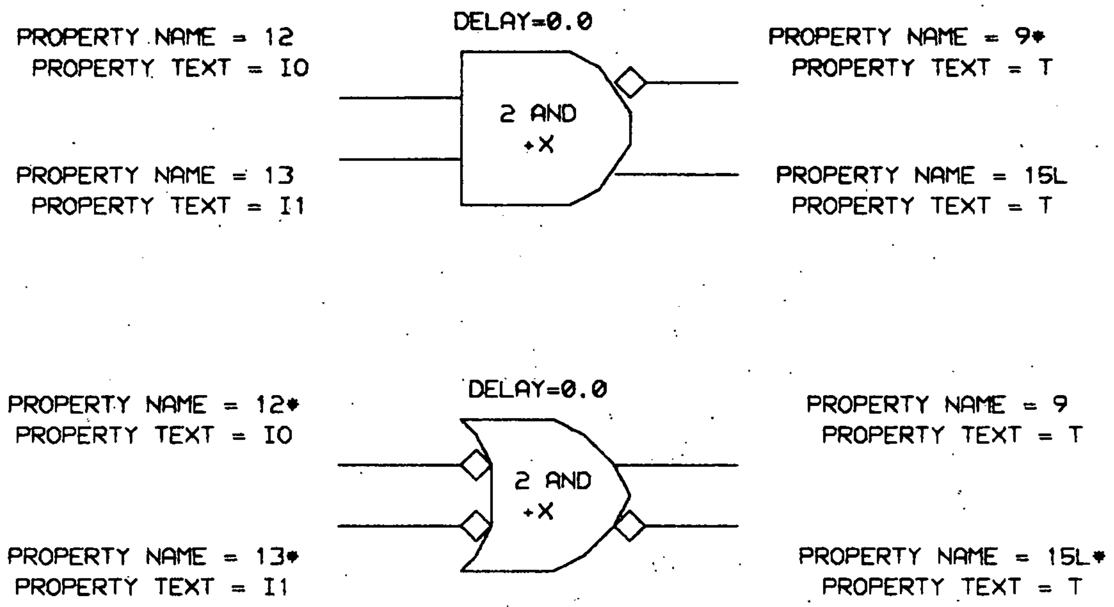


Figure 4–4
Gates for high and low assertion

4.4 How the macro expander binds bodies to drawings

When the macro expander encounters a macro body in a drawing, it takes the text from the property MNAME and looks for a drawing with that text in the first line of its title.

Note that two other names exist and thereby confuse the issue. The body template itself has a name which the graphics editor recognizes when you ask to place a copy of that body in a drawing. The file containing the drawing that defines a body has a filename by which the computer operating system recognizes it. Neither of those names has anything to do with the process of finding which drawing to use to expand a macro.

(There are good reasons for that. Keeping the body name separate from the macro name permits multiple bodies to have the same MNAME and thus the same macro definition: for example, consider again a gate which can be either an AND gate which expects its signals to assert high or an OR gate which expects them to assert low. The macro expander can use two different bodies called "2 AND" and "2 ANDO" (the latter looking suspiciously like an OR) to represent the same function. And keeping the drawing filename out of the picture makes SCALD less dependent on the operating system.)

If the macro expander finds more than one file with the same name in the first line of the title, it then goes to the second line of the title in each file and evaluates it as a *selection equation*. It uses the drawing for which the selection equation evaluates to "TRUE".

This is handy because in many cases you will want to implement a function differently depending on some parameter such as (for example) the size. If the number of bits you're generating parity for is 12 or less, for example, you may want to use one circuit but if it's greater than 12 you'll want to use another. By putting a selection equation like "SIZE<=12" on the second title line of one drawing and an equation like "SIZE>12" on that of another, you can accomplish this.

Of course, you must invent these selection equations so that for each value of SIZE you expect to use, the equation inone and only one drawing evaluates to TRUE.

A typical selection equation is a function of one of the macro body parameters--SIZE, TIMES, or VAR--but can in general be any expression. If you provide only one drawing to define a given macro, leave its second title line blank and the expander will always select that drawing.

4.5 Inventing Signal Names

Not every signal need have a name. If, for example, a signal originates at one body and terminates at another within the same drawing, the macro expander can infer from the characteristics of the output and input pins everything it needs to know about the signal: its width in bits, its assertion, and so on.

But attaching a name to a signal--which, within the graphics editor, merely requires attaching text to a point along the line that represents the signal--can provide additional information: wire delay, clock skew, and so on.

And sometimes the macro expander requires a name for a signal--when, for example, the signal is an input or output parameter; or when the signal should be made global so other drawings can refer to it.

Concatenation--To combine several different signals into one multiple-bit signal, use ":" between their names to indicate concatenation. The signal whose name is leftmost provides the most significant bits:

```
HIGHBYTE : MIDDLEBYTE : LOWBYTE
```

Conditional signals--To make a signal name depend on an expression, use an IF/THEN/ELSE construct. If the expression is true, the macro expander uses the name following the word THEN, but otherwise it uses the name following the word ELSE. The quotation marks shown in the following example are required:

```
"IF" SIZE<8 "THEN" FIRSTBYTE "ELSE" -FIRSTBYTE
```

Do not omit the ELSE part, and do not nest the IF/THEN/ELSE construct.

Comments--Everything following a ";" in a signal name becomes a comment. Thus, when you use concatenation or IF/THEN/ELSE, you're allowed only one comment:

RIGHT

```
CA : OR : WA ; Pacific states
"IF" FLAKY "THEN" CA "ELSE" WA : OR ; A pointed comment
```

WRONG

```
CA ; Far out : WA ; Far up : OR ; Far gone
```

4.6 Putting together a signal name

Wholly apart from concatenation or IF/THEN/ELSE, an individual signal name consists of a series of individual pieces strung together, each of them describing some aspect of the signal.

The syntax of a signal name is:

```
<Name> ::=
     <Negation>
     <Signal Class>
     <Simple Name>
     <Timing Assertion>
     <Assert Low>
     <Bit Subscript>
     <Wire Delay>
     <Timing Evaluation Directive>
     <Scope>
     <Multiplier>
     <Version>
```

Not all of the information is meaningful to the macro expander; the <Timing Assertion>, <Wire Delay>, and <Timing Evaluation Directive>, for example, are included for the benefit of the timing verifier. All the components except the <Simple name> are optional, and the last five may appear in any order. Thus, a rather elaborate example of a name is:

```
-SHAKESPEARE$HAMLET.C1-2,3-4 L <0:6:2,4:8> [2.5,3.7]&A /M *(SIZE) /18
```

and a simple example of a name is:

```
CLOCK
```

In order to persuade the macro expander that two signals are the same, the <Signal Class>, <Simple Name>, and <Timing Assertion> pieces must be identical, character for character. Other pieces of the name may or may not appear in various places in a drawing. We'll proceed to talk about the various pieces of syntax, one by one.

4.6.1 <Negation>

To invert a signal *without* indicating that it asserts low, put a minus sign at the front of the subname:

```
DECODE I
```

```
-DECODE I
```

Contrast this with the use of <Assert Low>, described later in this chapter, which requires that the pin receiving the signal have a "bubble" or "diamond" indicating that it expects a signal that asserts low.

If a signal is generated by a gate with the complementary outputs, then the macro expander recognizes that the inverse of the signal is available, too, and will allow you to use that inverse in the drawing without explicitly connecting a line to the inverse output of the gate.

The absence of "-" implicitly indicates the uninverted form of the signal. Note that putting a "+" in front of a signal name creates an entirely different signal name; a plus sign is *not* a superfluous symbol.

A "-" inverts each individual bit of a multiple-bit signal.

4.6.2 <Class Name><Simple Name>

Within <Class Name> and <Simple Name>, you may use alphabetic characters, digits, "+", "-", "(", and ")" as you wish, though you should be prudent about it: if you put a "-" at the front of the name, for example, the macro expander will think it's a <Negation> rather than an innocent character in the name.

Similarly, if a <Simple Name> ends in " L", and there is no <Timing Assertion>, the macro expander interprets the "L" to mean "<Assert Low>" as described later in this chapter.

And if there's no <Class Name> and the <Simple Name> consists of nothing but digits "0" and "1" then the signal is a binary constant as described in Section 4.6.11.

<Class Name> is a sort of prefix, consisting of a string of the characters just mentioned, followed by "$", which you can attach to each member of a family of signal names, making it easy to pick them all out of the crowd. If signals "DECODE INST", "SHIFT LEFT", and "SKIP" are all part of a functional block called "ARITH BOX", for example, you might want to make that clear by using "ARITH BOX" as a signal class:

```
 ARITH BOX$DECODE INST
-ARITH BOX$SHIFT LEFT
   ARITH BOX$SKIP
```

<Simple Name> is just a name made up of the legal characters listed a few paragraphs ago:

```
      W
      8
     2BY4
MANY MANY MANY WORDS
```

4.6.3 <Timing Assertion>

This specifies the time varying behavior of the signal. It's useful for documenting the expected behavior of signals entering and leaving a functional block. In addition, it lets the timing verifier check a subsection of the design even if the entire design is not complete; the verifier can use the <timing assertion> on an input instead of evaluating the unfinished circuitry that will eventually feed that input.

When the macro expander parses a signal name, it does not actually regard the <timing assertion> as separate from the <simple name>. Only the timing verifier recognizes the <timing assertion> as anything more than a few additional characters in the name. The exact syntax of a <timing assertion> appears in Section 6.3.2.

4.6.4 <Assert Low>

To indicate that a signal asserts low, place " L" after the <timing assertion>. Note that in the absence of the timing assertion, the " L" will follow the signal name, reducing to the conventional notation:

```
MULTIPLICAND READY L
  CLK .C1-2, 3-4 L
```

Note also that if a gate has complementary outputs, the macro expander recognizes that the output and its inverse both exist, and will allow you to refer to both even if you connect a line and invent a signal name for only one.

The distinction between the "-" preceding a signal name and the "L" following it is important. Each indicates inversion, but only "L" indicates that the signal asserts low. Thus, a signal with "L" implies that the pin receiving the signal *must* have a "bubble" or "diamond", as Figure 4-5 shows.

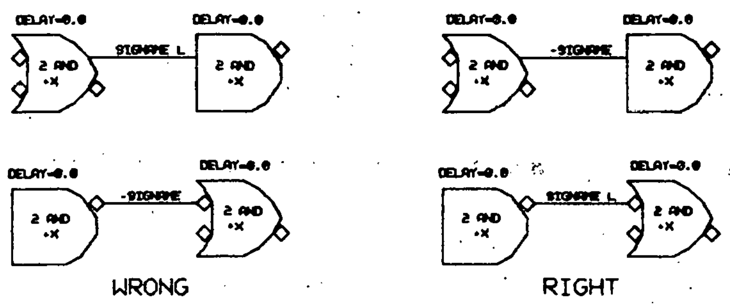


Figure 4-5
Assertion checking

Two special techniques tell the system precisely how to check assertions. First, to regard a signal as asserting low without actually inverting it, use both "-" and "L". The effect is that of using neither "-" nor "L", except that the system will check to make sure all receiving pins have bubbles (Figure 4-6).

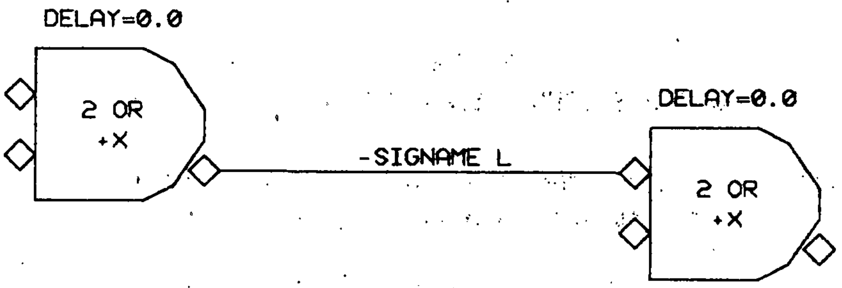


Figure 4-6
Assertion checking without inversion

Second, to flout the convention that a signal originating at a pin with a bubble asserts low and a signal originating at a pin without a bubble asserts high, use a fictitious body called a "NOT"; see Section 4.8.

4.6.5 <Bit Subscripts>

Bit subscripts tell the macro expander that a signal consists of one or more bits of a multiple-bit signal—that is, a bus.

The macro expander accepts either one- or two-dimensional signals. The following examples specify single bits out of multiple-bit signals:

```
ONE DIMENSION<6>
TWO DIMENSIONS<6,3>
```

Either or both subscripts can specify a range of bits, which the macro expander processes in row major order. Thus, the following two examples specify identical three-bit signals:

```
MULTI BIT<0:2>
MULTI BIT<0>:MULTI BIT<1>:MULTI BIT<2>
```

And likewise, the following two examples specify identical four-bit signals:

```
TWO D<1:2,5:6>
TWO D<1,5>:TWO D<1,6>:TWO D<2,5>:TWO D<2,6>
```

Either or both subscripts may also specify a step-size for the range of bits. Thus, the following example includes bits from the fourth through the tenth, incrementing by 3:

```
ONE D<4:10:3>
ONE D<4>:ONE D<7>:ONE D<10>
```

A single bit signal name--one without any bit subscripts--is by default the same as a signal name with both subscripts zero. Thus, the following three examples are identical:

```
LONE BIT
LONE BIT<0>
LONE BIT<0,0>
```

4.6.6 <Wire Delay>

For any input signal to a body, you may specify a wire delay or range of delays within square brackets:

```
POP L[6.8]
CK[2.5:3.7]
```

See Section 6.3.1 for details.

4.6.7 <Timing Evaluation Directive>

When several signals feed into a gate, the timing verifier ordinarily uses the individual timing assertions for each of the inputs in determining the output. But sometimes you may want it to ignore certain aspects of the other inputs when propagating one of them through the circuitry. For details on the syntax for doing this, see Section 6.3.3.

4.6.8 <Scope>

The scope of a signal is the environment within which the macro expander recognizes that particular signal by name. Within a macro name you can specify either of two scopes: "/P" for parameters or "/M" for module-specific signals.

Parameters--If a signal is a formal parameter, then its name is really just a stand-in for the name of whatever signal is used as the actual parameter when the macro is invoked. Thus, it's "hidden" from the world outside the macro; you can use the same name inside another macro and no conflict will result. After all, signal parameters and pin labels are the same thing, and for example it's understood that when a counter and a shift register both have a pin labelled "CLK", the two pins are nevertheless distinct from each other.

By convention, when a parameter is "common"—that is, when it requires only a one-bit signal even when its body gets replicated due to the SIZE parameter—we give its signal a name ending in "C": "CLRC", "INHIBC", and so on.

To provide extra error-checking, the macro expander requires that each drawing that defines a macro contain a PARAMETER list giving the name of each parameter.

To create a PARAMETER list, add to the drawing a fictitious body (typically one keeps a template for it in a graphics editor library) called PAR, which has no visible lines or pins, but whose MNAME is PAR and whose main purpose in life is to support property name/text pairs. For each parameter signal name, create a body property name/text pair. By convention, we name the properties "1", "2", "3", and so on (but it doesn't matter). Each property text should contain the <Signal Class>, <Simple Name>, <Timing Assertion>, and <Bit Subscripts>, if any, from the signal name. Omit all other pieces of the name. List either a signal or its inverse, but not both; including either automatically declares both. (For ECL, the timing verifier will benefit if <Assert low> is included in the parameter list where appropriate, though the macro expander does not require this.)

If a signal parameter is an output, append "/V" to its name. Do this only in the parameter list, not in the drawing as a whole. (This tells the macro expander which signals to tie together when the TIMES body parameter, described in Section 4.3.1 causes it to duplicate a macro).

Module-specific signal--Like a parameter, a module-specific signal is "hidden" from the outside world; the macro expander regards it as distinct from any other signals with the same name in other macros. In addition, the macro expander creates a different incarnation of it every time the macro is invoked.

As a safeguard against accidents, a module-specific signal name must not duplicate that of any signal that is global to the macro. You cannot by creating a module-specific signal name dethrone a global name that your macro would otherwise recognize. To see whether a particular name—ALPHA, for example--is allowed to be module-specific inside a particular macro, simply ask, "If the signal had some other name, would ALPHA always be meaningless and undefined, every time the macro gets called?" If the answer is "yes", then it's safe to use the name ALPHA.

Global signals--Any signal which lacks "/P" or "/M" is global. Ordinarily, a global signal is visible from within every macro; throughout the entire design, every reference to its name means the same signal.

Thus if you call a macro several times, the corresponding global signals in every invocation of the macro all get connected together. And if you use the same global signal name in two different macros, both those signals get connected together.

A feature called DECLARE creates nested scopes similar to those found in many high level programming languages. If you place a global signal name in a DECLARE list inside a macro, the macro expander creates a different incarnation of that signal every time the macro is invoked. In addition, while that incarnation is "hidden" from other macros in general, it's visible throughout the subtree of this macro--that is, within this macro call and also throughout all the circuitry resulting from the macros which it calls in turn.

A signal declared in a macro is local and hidden as far as the caller of that macro is concerned, but global and visible as far as all of the callees of the macro are concerned.

This is similar to the dynamic nesting of scope found in languages like LISP.

To create the DECLARE list, add to the drawing a fictitious body (available from a graphics editor library) called DECL. This works just as the PARAMETER list does: create a body property name/text pair for each signal name in the list; name the properties "1", "2", "3", and so on; and put the <Class Name>, <Simple Name>, and <Timing Assertion> portion of each signal name in a property text. List either a signal or its inverse, but not both. The "/V" rule doesn't apply to the DECLARE list.

4.6.9 <Multiplier>

To guard against errors and to help enforce design rules concerning fanout capability, the macro expander requires the designer to specify very explicitly how to interconnect bodies.

To feed the output of a gate into three different inputs, a conventional schematic would show something like Figure 4-7.

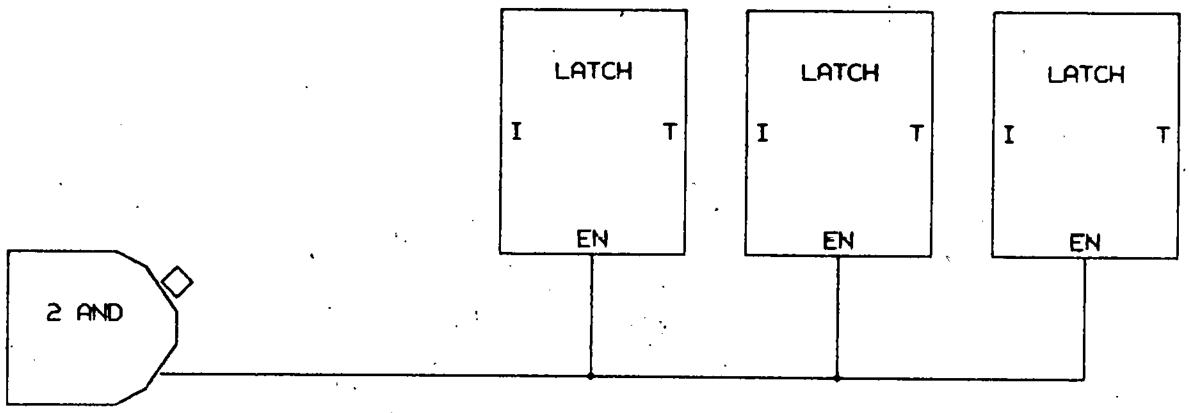


Figure 4-7
Conventional schematic

But the macro expander makes it easy to replicate a logic element without drawing multiple copies of it, simply by setting the SIZE parameter of the corresponding body to the desired number of copies. Thus, on first thought, one might try to draw something like Figure 4-8.

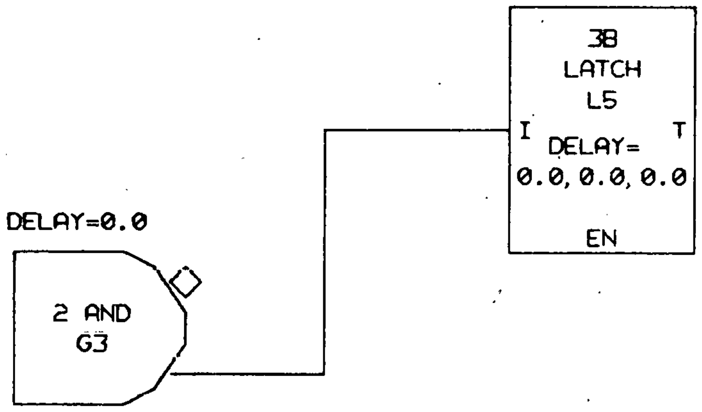


Figure 4-8
Wrong way

However, that would generate an error because the macro expander knows the gate generates a one-bit signal but, because of its definition, the "I" input of a three bit latch requires a three bit signal. The solution is to eliminate the line connecting the gate to the latch. Instead, give the output of the latch a name—"A", for example--and use that name along with a "*3" multiplier to feed the input of the latch (Figure 4-9).

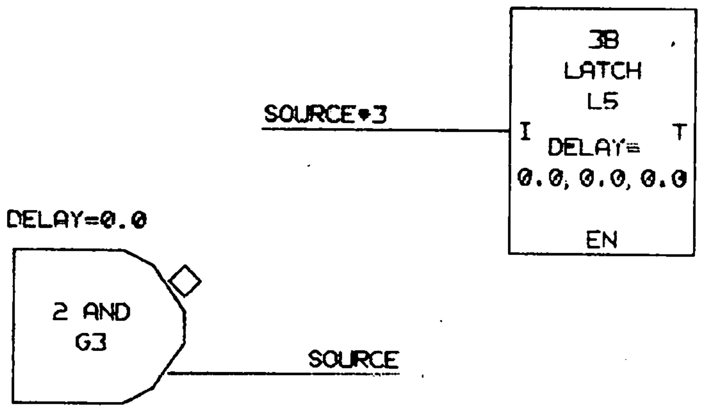


Figure 4-9
Correct way

The multiplier effectively concatenates the signal to itself the specified number of times. Thus, "A*3" is equivalent to "A:A:A". A multiplier will concatenate multiple bit signals, too, so the following examples are equivalent:

```
B<1:5>*3
B<1:5>:B<1:5>*B<1:5>
```

The value for <Multiplier> may be any expression, but it must begin with "*". If the value is 0, the macro expander creates a zero-width signal by concatenating no bits.

Note that using a multiplier does not increase the number of gates that drive a signal; to automate the entire issue, make the TIMES parameter on the gate a function of the SIZE parameters on all logic elements driven by the gate.

4.6.10 <Version>

Using the TIMES parameter to duplicate a logic element as explained earlier in this chapter automatically generates multiple physical output signals for each logical output signal shown and labeled on the drawing. The ability to gather a handful of similar physical signals and deal with them as one logical signal is a particular advantage of the macro expander. The macro expander itself, however, must deal individually with the physical signals, and thus needs a unique name for each one for use in preparing wirelists and so on. It derives these names by placing a slash and a version number after the name you invent.

Ordinarily, these versions need not concern you until it actually comes time to build the prototype. If you ever do need to specify a particular version during the design phase, however, you can do so using the same syntax:

```
MULTIVERSION SIGNAL /0
MULTIVERSION SIGNAL /1
MULTIVERSION SIGNAL /2
```

Note that although the macro expander uses numbers, you may place any alphanumeric string after the slash provided it cannot be confused with the <scope> portion of the name.

You can, in fact, use the <Version> option as a general-purpose qualifier on a signal name, for whatever purpose you wish.

A signal name may accumulate multiple version numbers as one macro calls another. The macro expander concatenates the version numbers, separated by dots, with the version contributed by the highest level macro at the right:

```
A SIGNAL /2.3.1
```

4.6.11 Constants as Signal Names

Naming a signal "1" or "0"indicates that it is permanently TRUE or FALSE—a binary constant rather than a variable. Putting a row of "1"s and "0"s together concatenates them just as the ":" operator would. Thus, the following examples are equivalent:

```
10110
1:0:1:1:0
```

Essentially the only valid options to apply to a binary constant are <Negation>, <Assert Low>, and <Multiplier>. A "–" preceding such a constant or an " L" following it applies to all the bits, inverting each one. Note that the <Multiplier> doesn't multiply the numbers, but rather concatenates each individual bit with itself the specified number of times. Thus "101 *4" is the same as:

```
1:1:1:1:0:0:0:0:1:1:1:1
```

not "101:101:101:101" or "10110100".

4.6.12 Text Substitutions

Within a drawing, you can provide a list of text substitutions or abbreviations to be used throughout that drawing.

Effectively, these are "text macros", but we'll call them "substitutions" to avoid confusion with drawing macros. Each substitution rule should look like "A=B", where A is the abbreviation and B is its meaning. The abbreviation must be a single word (that is, embedded spaces are forbidden) but its meaning may be any string of characters including leading or embedded blanks.

To define text substitutions, you use a DEFINITION list similar to the PARAMETER and DECLARE lists described in Section 4.6.8. Add to your drawing a fictitious body called DEF (usually one keeps a template for it in a graphics editor library) and give it a property name/value pair for each substitution rule. The properties are unimportant (by convention one uses "1" as the first name, "2" as the second, and so on) but each value should give a substitution rule in the form

"A=B":

```
DEFINE
LOWBYTE=8:15
HIGHBYTE=0:7
```

Except inside the <Bit Subscript> portion of a signal name, you must surround an abbreviation with "\" characters. This is a safety feature to prevent destruction of a signal name that happens to contain one of these abbreviations within an otherwise innocent word.

(A more precise explanation is that the entity within the macro expander which scans signal names operates in two states, either looking for abbreviations to expand or not looking for them. When it gets to the beginning of the signal name, it enters the not-looking state. Each "\" toggles the state. Thus, you could place a single "\" before the first abbreviation in a name and leave the scanner in the same state until the end. But it's safer and prettier to turn it off again with another "\" immediately after the abbreviation.

When the scanner reaches the beginning of the bit subscript part, it enables text substitutions. If it encounters a "\" within the bit subscript, it will actually turn substitutions off.)

Substitutions are illegal in the first line of the title of a drawing.

The text substitutions defined in the previous example would cause the macro expander to expand the following signal names as shown:

```
TOP OF STACK<HIGHBYTE> ==> TOP OF STACK<0:7>
TOP OF STACK<LOWBYTE> ==> TOP OF STACK<8:15>
```

The scope of an abbreviation–The definition of an abbreviation takes effect throughout the drawing containing the definition, and throughout any macros called from that drawing, unless those macros themselves override it by redefining the same abbreviation. Thus, the DEFINE list implements dynamic scope similar to that of the LISP language and the macro expander's own DECLARE feature.

Special variables–As mentioned earlier, when you call a macro with SIZE set to a value other than 1, the macro expander executes an implicit loop from X FIRST to SIZE-1 by X STEP. The variables X FIRST and X STEP default to 0 and 1 respectively, but a macro can use the DEFINE list to set them to any desired values during its own evaluation. This has no effect, however, on their value during evaluation of any macros which it calls in turn.

4.6.13 Sundry Details About Naming Signals

The material in this section will interest the serious user of the macro expander, but the casual reader may skim it without loss.

The PATH mechanism--When you use "/M" to make a signal module-specific, or when you use the DEFINE list to create a local scope for a signal, the macro expander must find a way to generate multiple, distinct signal names from each of the names you invent. It accomplishes this by prefixing the signal name with various *paths*.

A path is a route from the root of a tree to a particular node you're interested in. You can visualize the process of expanding macros as a tree, where each node represents the invocation of a macro, and the father of a node is whoever called the macro. You can derive a unique name for a path—and thereby for a particular node--by tracing the chain of macro calls from the root (topmost macro in, your design) to the node you're interested in, making a list of the LOC body parameters of all the macro calls in the chain.

Thus, when the macro expander wants to make a unique incarnation of a "/M" name (or of a global name mentioned in a DECLARE list) for a particular invocation of a macro, it simply prefixes the name with the path (in parentheses) to the node that represents that invocation.

Suppose a module called GAMMA has a module-specific signal called "MINE /M". In Figure 4-10, the invocation of GAMMA at the lowest level of the tree will have a signal named "(A1 B1 G1)MINE /M" while the other invocation of GAMMA will have a signal named "(A1 G1)MINE /M".

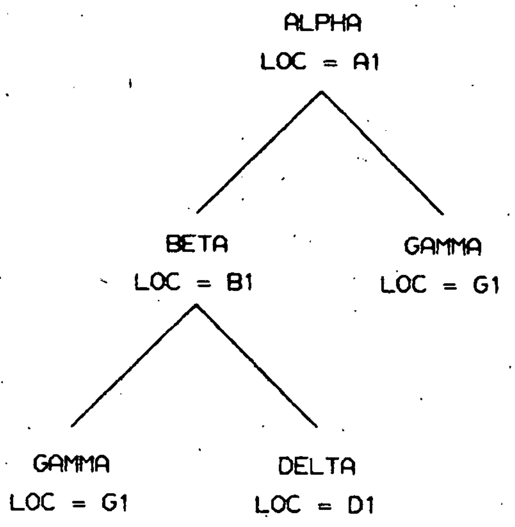


Figure 4-10
Macro expansion tree

When the SIZE parameter causes the macro expander to replicate a macro, the special variable X is different for each copy. Since the macro appears only once in the drawing, there's only one value of LOC for all the copies. To distinguish them, the macro expander appends a "*", followed by the value of X, to the LOC value of each copy for which X is not 0. Thus, if X FIRST=0 and X STEP=1 and SIZE=2, then a body at location S1 in the drawing will generate three bodies as locations S1, S1*1, and S1*2.

(Incidentally, the macro expander follows a similar convention to distinguish multiple copies resulting from the TIMES body parameter. It appends a "+" followed by a number to each copy after the first, so that a body at location S2 with TIMES set to 3 generates three bodies with locations S2, S2+1, and S2+2. If a location has both a TIMES and a SIZE appendage, the TIMES appendage comes first.)

The PATH text abbreviation--Ordinarily, such path construction is transparent. But the macro expander makes the feature available to the designer, too, by providing a special predefined text abbreviation "PATH". When you use "PATH" (inside most of the signal name, of course, you'll have to write \PATH\ as you would with any abbreviation) in a signal name, the macro expander will automatically substitute a parenthesized path for it during macro expansion.

```
DOWN THE PRIMROSE \PATH\
```

becomes

```
DOWN THE PRIMROSE  (G3 M1 A7)
```

Obviously, it's a good idea to place paths in the middle or at the end of signal names so there's no confusion between them and the ones the system adds for module-specific signals.

How is the PATH feature useful? Suppose you want to be able to access any register in your design from a special piece of diagnostic hardware to aid in field servicing. You can hide this extra complexity by providing a special macro called REGISTER which automatically generates the proper circuitry. Each incarnation of this macro will need a pair of signals connecting it to the diagnostic hardware. Giving the macro extra input and output parameters for the diagnostic signals makes it more complicated to draw. But using globals poses a problem because each incarnation of the macro needs a set of globals with unique names.

The solution is to use global signals but include PATH in each global name so that all incarnations of the macro will use the same global name--with a unique path name attached to the end of it. When the design is finished, these global signals will produce error messages because no hardware exists to generate them, and you can then design the diagnostic hardware specifically to generate the signal names that appear in these error messages.

Naming Unnamed Signals--When the designer does not name a signal, the macro expander always derives an internal name for the signal. For each pin the signal connects to, the macro expander constructs a possible name by taking the LOC parameter of the pin's body, then appending a "%", next appending the signal parameter name from the property text field, and finally tacking on " L" if there's a "*" in the property name for that pin:

```
<body LOC>%<signal parameter name>
```

Then it alphabetizes this list of possible names and uses the one that appears first.

When an unnamed signal connects to a body which has no LOC parameter, the macro expander doesn't construct a possible name for that connection. If that leaves the macro expander with no possible names to choose from, then it will reluctantly construct a name of the form:

```
%<number>%<signal parameter name>
```

followed by " L" if the signal invokes low. Then it alphabetizes these and selects the first.

Synonyms--Putting two names on the same signal is perfectly legitimate so long as the names don't conflict on matters like timing or the number of bits the signal represents. The macro expander considers such names to be *synonyms*.

When the signal in question has multiple bits, the macro expander matches the individual bits of one name with those of the other in row major order. Thus if signal A:B:C is synonymous with signal Z<0:2>, then A is the same as B<0>, B the same as Z<1>, and C the same as Z<2>. If signal X<0:3> is synonymous with Y<0:1,0:1>, then X<0> is the same as Y<0,0>, X<1> the same as Y<0,1>, X<2> the same as Y<1,0>, and X<3> the same as Y<1,1>.

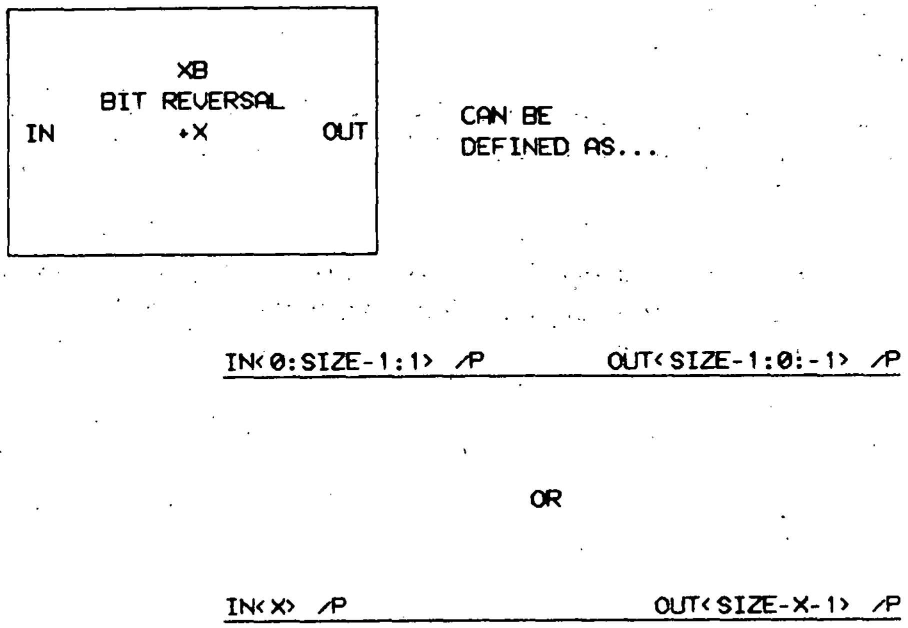


Figure 4-11
Bit reversal

Figure 4-11 shows two different ways to reverse the bits of a bus using synonyms. The first uses a step size of -1 in the bit specification, while the second uses the special variable X which the macro expander increments from X FIRST through SIZE-1 as it expands any macro.

4.7 Matching Signals with Bodies

Just as programming languages demand that actual parameters match formal parameters in terms of data type and number, so the macro expander demands that signals match body pins in terms of assertion level and number of bits.

As mentioned earlier, if a signal has the " L" (<Assert Low>) option, you may connect it only to pins which have diamonds. If a signal asserts high, then you may connect it only to pins which don't have diamonds.

Similarly, the number of bits that a signal name represents must match the number of bits that a body pin represents. It's not always apparent from the outside of a body which pins represent multiple bits, as the Figure 4-12 shows.

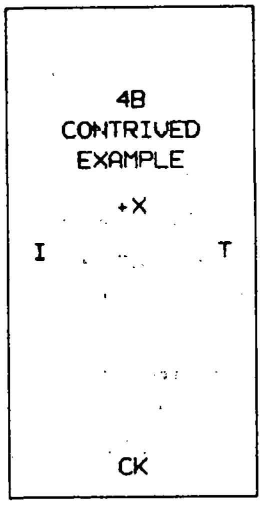


Figure 4-12
Common pins

The CK input of this register is a one-bit signal or *common pin,* since a single clock suffices for multiple cells of the register; the I input and T output are four-bit signals, since they are a function of size. The parameter list inside the drawing that defines the macro determines this by specifying CK independent of SIZE and by dimensioning I and T to have the number of bits specified by SIZE:

```
CK
I<0:SIZE-1>
T<0:SIZE-1>
```

Reconciling bits—the macro expander deals with vectors and arrays of bits in row major order. That is, whenever it processes the bits it travels through an array from the 0th bit to the highest order bit, varying the rightmost subscript most rapidly.

When a multiple-bit signal connects to a multiple-bit input on a body, the only requirement is that the number of bits in the signal matches the number of bits in the input. The means of arranging the multiple bits or multiple inputs into a vector or array doesn't matter; in effect, the system converts the pins into a vector (a one-dimensional array) using row-major order, converts the signal bits into a vector using row-major order, and connects the two vectors bit by bit.

Obviously, this works most neatly when the pins and signals comprise arrays with precisely the same dimensions. But using arrays with like numbers of elements but different dimensions (connecting a 2*6 array of signal bits to a 3*4 array of pins, for example) is permitted also.

Similarly, concatenation always decomposes various signals into vectors (that is, one dimensional arrays) of bits if necessary before "glueing" them together. The decomposition takes place in row major order as just explained.

4.8 Fictitious Bodies

Along with all the bodies that eventually result in semiconductors and wiring, macro expander drawings also use bodies that denote no hardware whatsoever, but convey information to the macro expander and router.

Not--As mentioned earlier, a NOT body allows you to flout assertion-checking conventions for a particular signal. On one side of the NOT body, the signal is considered to assert high and on the other to assert low, as Figure 4-13 shows.

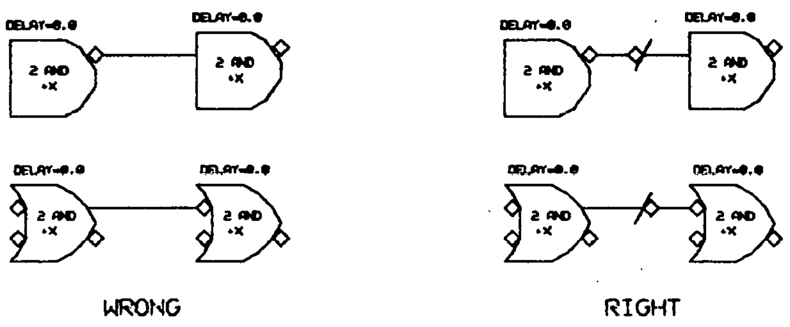


Figure 4-13
Using NOT bodies

Slash--A vertical or horizontal slash has no effect, but makes clear the number of bits on a bus (Figure 4-14).

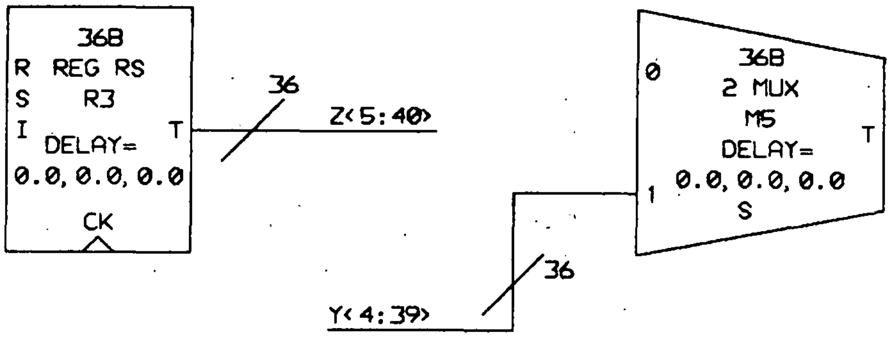


Figure 4-14
Slash body

Merge--Using the ":" operator to combine individual signal names as explained earlier in this chapter can leave a lot of ugly unconnected lines. A prettier means to the same end is to use a fictitious body called a "MERGE", which joins the lines representing two or more signals. The signal on the branch marked "H" supplies the high order bits, the signal on the branch marked "M" (if any) supplies the middle order bits, and the signal on the branch marked "L" supplies the low order bits (Figure 4-15).

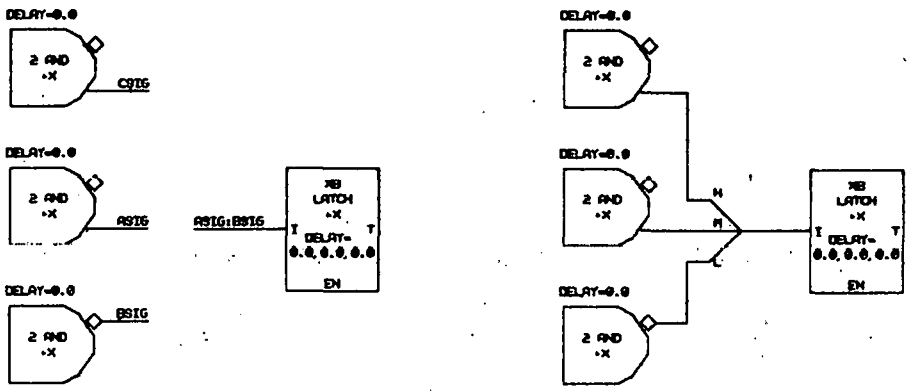


Figure 4-15
Three way merge

You can similarly use a reverse MERGE to split a multiple-bit signal and feed pieces of it to various destinations (Figure 4-16).

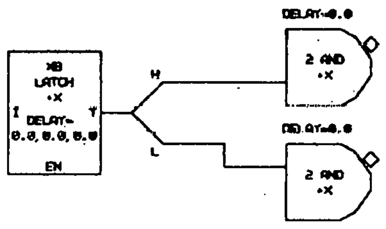


Figure 4-16
Reverse merge

Sign Extension–To widen a signal by replicating the sign bit, use the body shown in Figure 4-17. That example would convert SIGNAL<0:3> into SIGNAL<0>*5:SIGNAL<1:3> and call the result WIDESIGNAL<0:7>. MNAME is the property that provides the text "4 TO 8".

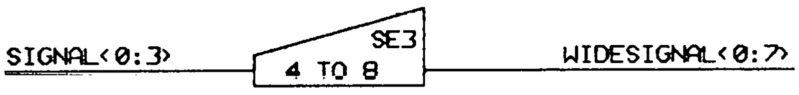


Figure 4-17
Sign extension

Wire-Or–As mentioned earlier, the macro expander does its level best to prevent you from connecting outputs together accidentally. If you truly intend to connect two signals, you must use a WIRE-OR body to do so. Unlike an actual OR, the WIRE-OR implies no hardware, but simply connects the signals. Bodies exist for positive and negative assertion (Figure 4-18).

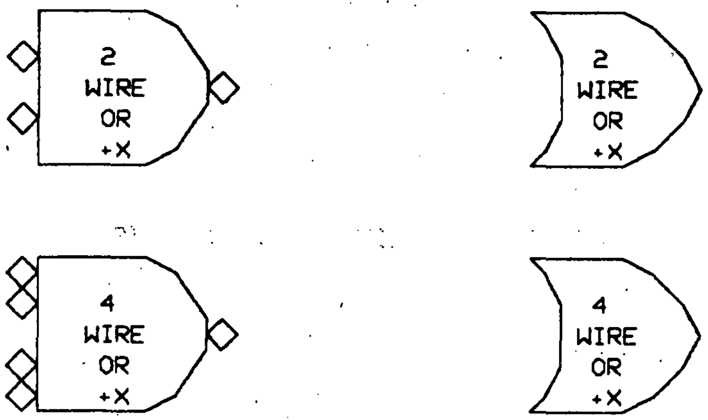


Figure 4-18
Wire-Or bodies

Comment, Par, Def, Decl--The PAR, DEF, and DECL bodies were mentioned in sections 4.6.8 and 4.6.12. The COMMENT body works similarly, allowing you to attach to it body text which appears on your drawing as a comment and has no effect on the macro expander.

4.9 How to construct the Terminal File

As mentioned earlier in the chapter, there are two kinds of terminal files: an actual IC terminal file for use when generating wirelists for the router, and a primitive terminal file for generating input to the simulator. When you run the macro expander program, you tell it which file to use.

The format of the terminal file is the same in either case: a series of entries listing the inputs and outputs of the macro.

On any particular line of the file, anything to the right of a "?" is a comment. Otherwise, the entries appear in free format, each ending with a semicolon.

Here's a typical entry:

```
PARITY GEN 100160 [SIZE=9] (100160,1) IA<0:8>, IB<0:8>, ZA /V,
                                      ZB /V, C L /V;
```

The individual items in the entry are:

- The name of the macro representing the actual IC (which should exactly match the name of the drawing that defines that IC and the macro body that represents it). In the preceding example, the name is "PARITY GEN 100160".

- An optional expression like "SIZE=9" which serves as a selection equation similar to that of the second line of a drawing title. You can make multiple entries in the terminal file for a given IC, and the macro expander will choose from them the entry whose selection equation evaluates to TRUE. (In the terminal file, it's permissable for none of the equations to evaluate to TRUE, in which case the macro expander will decide that it hasn't reached the definition of an actual IC after all, and will go search for a drawing to expand. This is handy, for example, when you want to use an actual IC when SIZE is small enough but cascade several ICs when SIZE is too large.)

  If it appears, this optional expression must be enclosed in square brackets.

- A pair of items inside parentheses: the chip name followed by the number of copies of the same function within each IC. Typically, the chip name is simply the manufacturer's part number. SCALD allows you to use an elaborate macro name during the logical design phase, translating it into the actual part number for use during the physical layout and prototyping.

- The names of the input and output signal parameters for the macro, separated by commas. Each name should include the desired <Class>, <Simple Name>, <Timing Assertion>, and <Bit Subscript> parts. List either the uninverted or the inverted form of a signal, but not both. If a signal is an output, append a "/V" to the name.

For a logical primitive, each parameter name must match the corresponding one built into the SCALD logic simulator. For either an actual IC or a logical primitive, each parameter name must match that in the property text field for the corresponding pin of the body definition, except that the property text field will lack the <Assert Low> and <Bit Subscript> portions of the name.

Note again that the entry for each macro ends with a semicolon.

4.10 Running the Macro Expander

Once all the drawings are prepared, using the macro expander is simply a matter of running the program and specifying all the files it needs.

First you must run the WDPR program to translate binary files from the graphics editor into a form that the macro expander can read. It will ask you for the name of a "WD input file" (the one you want to convert) and a "WDP output file" (the one it will put the conversion into.) When it's finished, it will print "Done." and ask you for a new pair of names, again and again till you abort it.

To run MAC, the macro expander itself, first prepare a file containing nothing but the word "END", which you'll use to terminate input to the expander.

Then run the program. It will ask for the following files:

MACLST--The file into which the program writes its listing.

MACEXP--The file into which the program writes the list of bodies and connections resulting from the macro expansion, which will serve as the input to the next phase of SCALD.

TERM-The TERM file from which the program reads parameters of terminal bodies.

WDP--Tell the program the names of the files produced by the WDPR program for the individual drawings. The macro expander will ask again and again for FILE0 until you give it the file containing only "END".

4.11 The Macro Expander Listing

The macro expander produces an extensive listing. We'll discuss the individual phases of the listing in order of appearance, showing an example of each phase. The examples are occasionally distorted somewhat to squeeze within the width of the page, or to illustrate additional features of the program.

Progress report--This gives a dutiful account of the program's activities:

```
READING TERMINAL BODY DEFINITIONS
FINISHED READING TERMINAL BODY DEFINITIONS
READING MACRO DEFINITIONS STARTED
FINISHED READING MACRO DEFINITIONS
```

Undefined macros--A list of each macro you failed to define, along with the name of the macro which attempted to call it. Despite these errors, the program expands the design as far as it can; connections to undefined macros are simply missing from the output file.

To help you find macro definitions in the listing, the program assigns a number to each macro that *does* have a definition, according to the order in which you supplied the macro names to the program. Thus in the following example, "93" is the number assigned to macro "FORMAT CONTROL".

```
UNDEFINED MACRO(S):

"DUAL 1 OF 4 DECODE 100170L     " CALLED FROM "FORMAT CONTROL #93"
"INT FP SUBTRACT                " CALLED FROM "INT FP A-B #103"
"INT FP ADDER                   " CALLED FROM "RND NORM #111"
"100170                         " CALLED FROM "1 OF 8 DECODE 100170 #143"
"100170                         " CALLED FROM "1 OF 8 DECODE 100170L #144"
"MOBY MUX                       " CALLED FROM "ABOX #149"
"MOBY MUX                       " CALLED FROM "ABOX #149"
```

Alphabetic macro list--The names—as given by the MNAME property and by title line 1—of all the nonterminal macros that are defined, in alphabetic order. Again, each macro's number follows its name.

The column marked "CALLS" gives the number of *static* calls on the macro: the number of times it appears in somebody else's definition, rather than the number of times it actually occurs once the expansion takes place. For example, suppose a macro called POPULAR appears once in the definition for macro ALPHA and twice in the definition for macro BETA. Static calls on POPULAR total 3 even if ALPHA gets used 36 times and BETA never gets used at all.

The column marked "FILE" gives the project name (see Section 2.2.1 for details), if any, followed by a dot and the name of the file holding the drawing for this macro. Sometimes several different drawings may define a given macro; in that case, the "macro portrait" phase of the listing (described later on) gives all of their names, but only the first of them appears here.

```
SEQ CALLS FILE                         MACRO DEFINITION(ALPHABETICALLY ORDERED)
  1     8 LOW.1708[MK2,S1]             1 OF 8 DECODE 100170 #143
  2     2 LOW.17080[MK2,S1]            1 OF 8 DECODE 100170L #144
  3     2 LOW.107CMP[MK2,S1]           100107 CMP #142
  4    22 100112.100112[MK2,S1]        100112V #81
  5     3 LOW.100ADD[MK2,S1]           10 BIT ADDER 100K #85
  6     4 RCMPY.23X10R[MK2,S1]         10X10 RECODE MPY #79
```

Readin-ordered macro list--Identical with the preceding list, but with the macros appearing in the order in which you supplied their names to the program.

Terminal body list--A list of terminal macros--that is, the macros at the very end of the chain of expansions, which cannot themselves be expanded. If you are going to use the logic simulator, these will be logical primitives; otherwise, they'll be macros representing actual integrated circuits.

For each macro, the CALLS column gives the number of static calls upon that macro, the TERMINAL BODY column gives the macro name corresponding to the MNAME property or to title line 1, and the CHIP NAME column gives the chip name that will be used by subsequent programs in the SCALD family. (The terminal file tells SCALD which chip name to use for each terminal macro; see Section 4.9 for details).

```
SEQ CALLS  CHIP NAME   TERMINAL BODY
  1    57  100101      100101
  2    74  100102      100102
  3     1  MB7071H     256W RAM MB7071H
  4     3  100183      2X8 BIT RECODE MPY 100183
  5     7  100179      CARRY LOOK-AHEAD 100179
```

Terminal portrait--For each terminal macro, the program provides a detailed description. First the program prints the MNAME of the macro and the number assigned to the macro. Then it prints the number of static calls upon the macro, and the filenames and MNAMEs of all the other macros which called it.

After the MNAME of each caller, the program prints the caller's number, and the values of parameters LOC and SIZE which the caller used in invoking this terminal macro. (The program omits the SIZE parameter here if SIZE=1.)

Following the list of callers, the portrait shows the formal parameters for the terminal macro, just as they appear in the terminal file.

In the example below, for instance, macro "CARRY LOOK-AHEAD 100179", whose identifying number is 69, was called four times by macro "72 BIT CARRY OUT ADDER", whose number is 118. The drawing defining macro 118 is in file COA[MK2,S1] which belongs to project COA.

Since the program didn't print the SIZE parameter, COA must have invoked this terminal macro with SIZE=1. It did so at locations C1, C2, C3, and C4.

The macro's parameters are "CO+2468 L<0:3> /V", "PG L<0:7,0:1>", and "CI L".

```
--------------------------------------------------------------------------------

TERMINAL:   CARRY LOOK-AHEAD 100179                                 NUMBER   69

CALLED     7 TIMES FROM:  COA.COA[MK2,S1]      72 BIT CARRY OUT ADDER #118(LOC=C2)
                          COA.COA[MK2,S1]      72 BIT CARRY OUT ADDER #118(LOC=C1)
                          COA.COA[MK2,S1]      72 BIT CARRY OUT ADDER #118(LOC=C4)
                          COA.COA[MK2,S1]      72 BIT CARRY OUT ADDER #118(LOC=C3)
                          LOW.18BADD[MK2,S1]   18 BIT ADDER 100K #85(LOC=CLA)
                          LOW.36BADD[MK2,S1]   36 BIT ADDER 100K #84(LOC=CLA1)
                          LOW.36BADD[MK2,S1]   36 BIT ADDER 100K #84(LOC=CLA2)

PARAMETER  CO+2468 L<0:3> /V,  PG L<0:7,0:1>,  CI L

--------------------------------------------------------------------------------
```

Macro portrait--A detailed description of an individual non-terminal macro.

The program first prints the macro MNAME and identifying number. In the following example, the name is HW MPY RND NORM and the number is 106.

Then it lists *all* files which define the macro; elsewhere in the listing, only the first of these filenames will appear. (Typically one would encounter multiple files because each file has a different selection equation as explained in Section 4.4, not because a macro definition won't fit in a single file.) For each file, the program prints the project name (explained in Section 2.2.1) followed by a dot and the actual name of the file. In the example, a single file called HWRN1[MK2,S1] in project HWMRN defines the macro.

Next the program gives the number of static calls on this macro and lists each of the callers; for details, see the preceding explanation of the terminal portrait. In the example, macro 106 is called 4 times by macro "HW ADDER ROUNDER".

Next it lists the formal parameters of the macro, just as they appear in the PAR body inside the macro's definition.

Next the program lists each of the synonyms—signals which have more than one name--within the macro's definition. In the example, for instance, signal %1% is the same as MUX0%0:MUX0%1:MUX0%2. (Evidently, this resulted from using a MERGE to concatenate three unnamed signals into one.)

Next the program lists all of the text substitutions given in the DEFINE body inside the macro's definition. In the example, X STEP is defined to be "1". Next the program prints a list of the synonyms that result when a fictitious SLASH body splits one signal into two identical pieces within the drawing that defines the macro. In the example, for instance, a 9 bit wide signal from the body at LOC=TG apparently got interrupted, forcing the macro expander to invent two synonymous names, %2% and TG%T, instead of just one.

Finally, the program prints one entry for each of the macros called by this macro. At the left of the entry is the LOC parameter for the call. To the right of that is a paragraph beginning with the macro MNAME and number. If the SIZE parameter for the call is not 1, it appears in parentheses following the macro number. Then the program prints a list of formal/actual parameter pairs, also in parentheses.

If the macro being called is a terminal, a "*" appears in front of its name; if it's undefined, a "?" appears in front of its name.

If, from the caller's viewpoint, a formal parameter asserts low (in other words, the pin corresponding to that parameter has a diamond), a "*" appears after its name in the parenthesized list. If the caller did not supply a signal for a particular formal parameter (in other words, left a pin unconnected in the drawing), then the formal/actual parameter pair will look like

```
FORMAL = ,
```

instead of

```
FORMAL = ACTUAL,
```

In the following example, for instance, the macro calls another macro "HW EXP" with LOC=EXP and SIZE=1 (we know that because no size appears on the listing). The macro expander connects signal EXP%EXP to formal parameter EXP, signal EXP%OVFL UNFL L to formal parameter OVFL UNFL L*, and so on. Formal parameter OVFL UNFL L* asserts low, as shown by the "*" following its name. The macro also calls "100171", which is a terminal macro as shown by the "*" preceding its name, with LOC=MUX0 and SIZE=6B, and so on.

When a macro lacks one of the features just described, the program omits the corresponding portion of the profile. Thus, if a macro contains no text substitutions or synonyms, the DEFINE and SYNONYM portions of the profile would not appear at all.

```
-----------------------------------------------------------------------------

MACRO:     HW MPY RND NORM                                    NUMBER  186
FILES: 1/1  HWMRN.HWRN1[MK2,S1]

CALLED    4 TIMES FROM:  HWADDRND.HWADDR[MK2,S1]  HW ADDER ROUNDER #112(LOC=RN3)
```

```
                          HWADDRND.HWADDR[MK2,S1]  HW ADDER ROUNDER #112(LOC=RN2)
                          HWADDRND.HWADDR[MK2,S1]  HW ADDER ROUNDER #112(LOC=RN1)
                          HWADDRND.HWADDR[MK2,S1]  HW ADDER ROUNDER #112(LOC=RN0)

PARAMETER   T<0:26>/V,  CTRL,  I<HWBUS>

SYNONYM     %1% = MUX0%0:MUX0%1:MUX0%2
            %2% = MUX1%0:MUX1%1:MUX1%2

SLASH       (SIZE=9)%2% = TG%T
            (SIZE=18)%1% = EXP%EXP
            (SIZE=1)TG%INT OVFL L = RN%OVFL L
            (SIZE=6)EXP%OVFL UNFL L = TG%FP OVFL UNFL L

DEFINE      X STEP= 1

MACROS CALLED

EXP         HW EXP #107 ( EXP = EXP%EXP, OVFL UNFL L* = EXP%OVFL UNFL L,
            I = I<HWEXP>/P ;HERE AT START OF A2)

MUX0        *100171 #68(SIZE=6B) ( T L* = ,  0 = MUX0%0,  1 = MUX0%1,
            2 = MUX0%2, 3 = 0*6, S = EXPONENT OFFSET<0:1>/M,
            T = T<0:5>/P,  OE L* = MPY HW SEL ENA A3.C3 L)

MUX1        *100171 #68(SIZE=3B) ( T L* = ,  0 = MUX1%0,  1 = MUX1%1,
            2 = MUX1%2,  3 = 0*3,  S = EXPONENT OFFSET<0:1>/M,
            T = T<24:26>/P,  OE L* = MPY HW SEL ENA A3.C3 L)

RN          RND NORM #111 ( EXP = EXPONENT OFFSET<0:1>/M,  OVFL L* = RN%OVFL L,
            T = T<0:23>/P,  I = I<HWFRAC>/P ;HERE AT START OF A3,
            CTRL = CTRL/P)

TG          HW TAG MODIFIER #109 ( T = TG%T,  INT OVFL L* = TG%INT OVFL L,
            FP OVFL UNFL L* = TG%FP OVFL UNFL L,
            I = I<HWTAG>/P ;HERE AT START OF A3)
```

Expansion trace--As it processes your design, the program prints a line of text every time it expands a macro. Up to now, the listing has dealt with the static structure of the macro hierarchy; this part of the listing traces all the dynamic calls. The left half of each line describes the path (defined in Section 4.6.13) leading to that particular macro call, and the right describes the call.

To describe the call, the macro expander prints the level of the call—that is, how deeply the call is nested--the name of the macro it's expanding, the value of the special variable X (as described in Section 4.3.1, the macro expander uses X to count from X FIRST to SIZE in increments of X STEP, determining how many times it will replicate a given macro), and the value of the SIZE parameter used in this call.

In the following example, the first line represents the expansion of the body at location ABOX. This occurs at level 1, and involves a macro called "ABOX" whose number is 149. X is 0 during this call, so the expansion produces a single copy of ABOX. The parameter SIZE is 1.

The second line shows that the program expanding the body at location 1AM1 within the macro at location ABOX. This one is a terminal called "100171" whose number is 68. Again, X=0 and SIZE=1.

The third line shows the program expanding the body at location 1AM2 within the macro at location ABOX, and so on.

The next interesting expansion occurs several lines later in the original listing, so our example skips the boring part. Look at the line immediately following the skip. Here you can see what happens when the loop from X FIRST to SIZE produces more than one copy of a macro during a single expansion. The fourteen lines following the skip alternate between macro 129 at level 6 and terminal macro 58 at level 7. The first instance of macro 129 occurs with X=0, the second with X=8, the third with X=16, and so on; we can infer that X runs from X FIRST=0 to SIZE=48 with an increment of X STEP=8.

```
MACRO EXPANSION PASS 1
(ABOX)                          LEVEL:  1 MACRO:    ABOX #149 (X=0,SIZE=1)
(ABOX 1AM1)                     LEVEL:  2 TERMINAL: 100171 #68 (X=0,SIZE=102)
(ABOX 1AM2)                     LEVEL:  2 TERMINAL: 100171 #68 (X=0,SIZE=102)
(ABOX 1MM1)                     LEVEL:  2 TERMINAL: 100171 #68 (X=0,SIZE=102)
(ABOX 1MM2)                     LEVEL:  2 TERMINAL: 100171 #68 (X=0,SIZE=102)
(ABOX 1MULTFU)                  LEVEL:  2 MACRO:    MULTIPLIER FCN UNIT #150 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR)             LEVEL:  3 MACRO:    HW ADDER ROUNDER #112 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR AMB)         LEVEL:  4 MACRO:    INT FP A-B #103 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR AMB MOD0)    LEVEL:  5 MACRO:    A+-B TAG MODIFIER #114 (X=0,SIZE=1)
  <here we skip a few lines to avoid monotony>
(ABOX 1MULTFU HWAR AMB V0 R)    LEVEL:  6 MACRO:    SHIFT REG CLR 100141 #129 (X=0,SIZE=54)
(ABOX 1MULTFU HWAR AMB V0 R R1) LEVEL:  7 TERMINAL: SHIFT REG 100141 #58 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR AMB V0 R#8)  LEVEL:  6 MACRO:    SHIFT REG CLR 100141 #129 (X=8,SIZE=54)
(ABOX 1MULTFU HWAR AMB V0 R#8 R1) LEVEL:  7 TERMINAL: SHIFT REG 100141 #58 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR AMB V0 R#16) LEVEL:  6 MACRO:    SHIFT REG CLR 100141 #129 (X=16,SIZE=54)
(ABOX 1MULTFU HWAR AMB V0 R#16 R1) LEVEL:  7 TERMINAL: SHIFT REG 100141 #58 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR AMB V0 R#24) LEVEL:  6 MACRO:    SHIFT REG CLR 100141 #129 (X=24,SIZE=54)
(ABOX 1MULTFU HWAR AMB V0 R#24 R1) LEVEL:  7 TERMINAL: SHIFT REG 100141 #58 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR AMB V0 R#32) LEVEL:  6 MACRO:    SHIFT REG CLR 100141 #129 (X=32,SIZE=54)
```

```
(ABOX 1MULTFU HWAR AMB V0 R#32 R1) LEVEL:  7 TERMINAL: SHIFT REG 100141 #58 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR AMB V0 R#40)    LEVEL:  6 MACRO:    SHIFT REG CLR 100141 #129 (X=40,SIZE=54)
(ABOX 1MULTFU HWAR AMB V0 R#40 R1) LEVEL:  7 TERMINAL: SHIFT REG 100141 #58 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR AMB V0 R#48)    LEVEL:  6 MACRO:    SHIFT REG CLR 100141 #129 (X=48,SIZE=54)
(ABOX 1MULTFU HWAR AMB V0 R#48 R1) LEVEL:  7 TERMINAL: SHIFT REG 100141 #58 (X=0,SIZE=1)
(ABOX 1MULTFU HWAR APB)            LEVEL:  4 MACRO:    INT FP A+B #104 (X=0,SIZE=1)
```

Chip counts--For each nonterminal macro, the program tells you the names and numbers of all the terminals that it requires. This includes terminals it uses indirectly--that is, by calling other nonterminals which in turn use the terminals--as well as the terminals that appear directly in this macro's definition. And the totals for a particular macro represent all calls on that macro throughout the system, not just a single invocation.

This is useful for estimating the cost of any particular part of the design. And the listing for the highest-level macro in your design will, by definition, give totals for all chips used throughout the design.

Note that these totals count each invocation of a terminal macro, even when one actual IC containing multiple sections can provide several copies of a macro.

```
SUMMARY OF TOTAL CHIPS USED BY EACH MACRO

MACRO:     18X18 RECODE MPY                     NUMBER 79

  CHIPS  TYPE
     36  100117
    108  100183
  -----
    144

MACRO:     PARTIAL PRODUCT SHIFTER              NUMBER 80

  CHIPS  TYPE
      4  100102
      4  100180
  -----
      8
```

Error summary--At the end of the listing, the macro expander prints the number of errors found throughout all passes of the program. The actual messages are printed in various phases of the listing.

5 A Guided Tour of a SCALD Macro

If you followed the guided tour of the graphics editor in Section 3, you are now no doubt sick of the drawing that defines macro 10016. But you are also no doubt very familiar with it, so it seems like the best locale for a guided tour of the aspects of 10016 that relate to the macro expander language itself.

This time, we won't try to take you step by step through the thought process of the guy who made the drawing originally but instead will let you wander around enjoying the sights while we offer random comments.

Figures 5-1 and 5-2 give a reprise of the drawing and the body respectively.

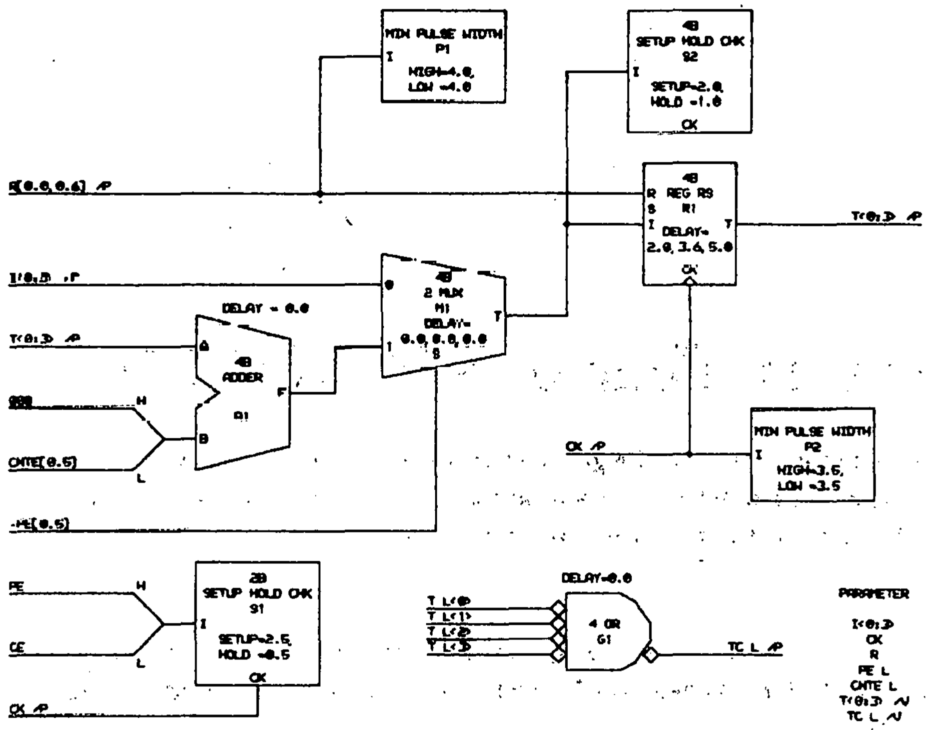


Figure 5-1
The drawing

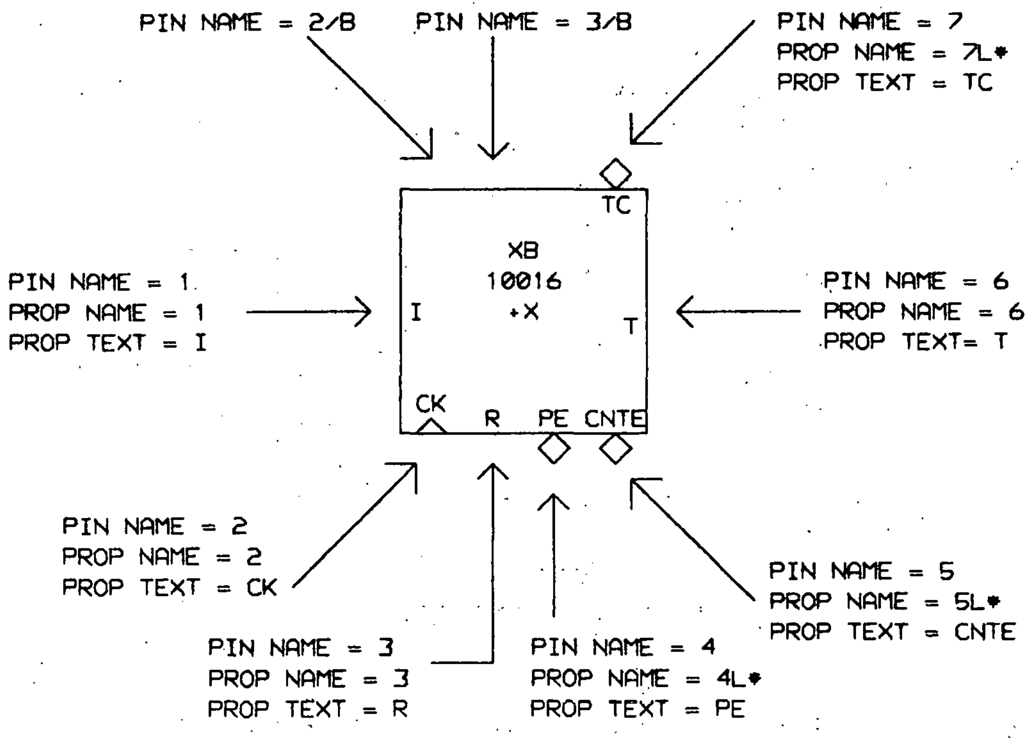


Figure 5-2
The body

The drawing's mission in life is to describe a four bit binary counter in terms that the SCALD simulator can understand. The counter has a clock input CK and a four bit parallel output T, as you would expect. It also has a reset input R, a count enable input CNTE, a four bit parallel input I, a parallel load enable PE, and a terminal count TC which goes low when T reaches 15 decimal.

The various inputs and outputs work together like this:

CNTE L	PE L	R	CK	Function
X	L	L	↑	Load parallel
L	H	L	↑	Count
H	H	L	↑	Hold
X	X	H	X	Reset

Now one way to represent such a counter is to cascade four master-slave flip flops and a bunch of gates. But that way madness lies, because while the manufacturer provides such a representation on the data sheet, the data sheet parameters don't deal with it on such a microscopic basis. Rather, they simply describe the setup and hold times for the various clock and enable signals, plus the delay from the time the device is clocked until all outputs have responded.

An easier representation of a counter is an adder which adds one to its outputs every time a clock pulse occurs (Figure 5-3), which is basically the approach that the drawing takes.

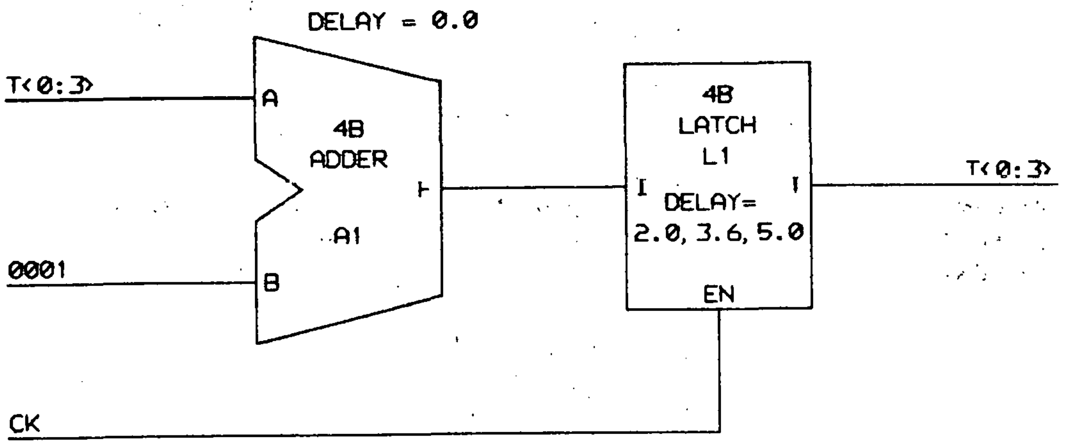


Figure 5-3
A simple view of a counter

The extra bodies and signals in the actual drawing serve either to represent extra functions like TC and PE, or to specify setup, hold, and delay times.

For example, the data sheet specifies that you must set up PE or CE 2.5 ns before CK and hold them 0.5 ns after CK; the body in the lower left corner informs the simulator of this.

The data sheet specifies that the delay between CK and T is 2.0 ns minimum, 3.6 ns typical, and 5.0 ns maximum; the VAR parameter on register R1 expresses this. The delay from R to T is 4.0 ns typical, so a delay of "[0.0,0.6]" on the R signal itself added to the 3.6 ns typical delay on register R1 achieves this.

The drawing illustrates a few intimate details of SCALD syntax, too. For example, we want the adder to add the CNTE signal to the outputs so that the counter counts when CNTE is high and holds when CNTE is not. That's fine, but the macro expander will not let you apply a one bit signal like CNTE to a four bit body like the adder, so you must use a MERGE body to concatenate the three bit binary constant signal "000" to CNTE.

When the outside world asserts PE, the signal goes low, so we want the low state of PE to select the 0 input of our multiplexer and thereby choose the parallel inputs I. But using PE L would cause problems because the multiplexer's S input has no diamond, so we use -PE instead: same signal as the PE L which the outside world sees, but no diamond required.

Note that you must use a bit of care in a case like this: the macro expander will permit you to use PE rather than -PE or PE L in your drawing even though the outside world gives you only PE L, but if you carelessly omit the "-" or "L", the macro expander will invert the signal for you for free, either by finding an inverted form in the outside world or by permuting inputs to the multiplexer.

The first line of the title indicates that this drawing is a candidate to define any body whose MNAME parameter is "10016". The second line, however, makes it a successful candidate only if the body's SIZE parameter is 4. If you use this body with SIZE set to some other value, you'll presumably have another drawing with "10016" as the first line of its title, which cascades enough four bit units to make up the required size.

6 How to use the timing verifier

The timing verifier reads the output of the macro expander and checks for timing errors using knowledge of the minimum and maximum propagation delays of the circuit components, their set-up and hold times, minimum pulse width constraints, and wire delays.

An important feature permits verification of individual modules instead of the entire design. This permits the program to execute on computers with limited memory size, allows errors to be discovered daily, before they can propagate through the design, and helps estimate a machine's cycle time before the design is complete.

The verifier gets information about the design from several different sources:

- For each terminal body--that is, each actual IC function--used in the design, the designer must provide a macro definition in terms of logical primitives. These primitives describe the timing constraints of that terminal body.
- Within the logical design of the machine, the designer may estimate wire delays for certain critical signals as part of the signal names.
- The designer may optionally make assertions about the timing of a particular signal, incorporating them in the signal name.
- The designer may specify how to evaluate the timing of certain gates by incorporating directives in signal names.
- After the physical design system lays out the parts and routes wires, it provides wire delay information, based on chip electrical characteristics and actual wire lengths, for all signals.

This section will first explain the theory behind the verifier, then explain how to define chips in terms of logical primitives, and finally explain how to use wire lengths, assertions, and evaluation directives in a design.

6.1 Theory of operation

Within synchronous sequential circuits, most signals can change only during particular parts of the clock period. For example, it may be possible for a particular signal to change only during the second half of the clock cycle, provided all of the components making up the system are within their timing specifications.

Consider a register that can be clocked only at a particular time within the clock period. The output of the register can change only during a short time after it is clocked, so it is guaranteed to be stable for the entire clock period except around the point at which it is clocked. The output of a gate driven from this register can then be changing only during a period of time determined by its propagation delay and when the output of the register is changing.

Determining when within the clock period a given signal may be changing and when it is stable is the key step for the timing verifier. Once this has been done, it is relatively easy to check all of the timing constraints placed on the circuit. For instance, to check the set-up and hold times on a register, the timing verifier need only determine whether its input could be changing at a time when it might be clocked.

If the timing of the circuit never depended on the values of signals, but merely on when they were changing or stable, the timing verifier would be very simple. Clock signals have a value which is periodic, and have the same value every cycle, so they are easy to handle. The signals which are difficult to treat are those whose values affect the circuit timing, and which have different values during different cycles of the circuit. For example, a control signal which determines whether a register is clocked during a given cycle affects whether the output of the register might change that cycle. If the circuit relies on the register not changing every cycle, then the timing verifier must do *case analysis* to keep from generating false error messages. This requires the timing verifier to check the type of cycle when the control signal is true, and to check the type of cycle when it is false. This could be a time-consuming process, but in practice is not, because most signals have a "worst-case" state. For example, the worst case for most registers is to assume that they are clocked every cycle. Only in those situations where both the clocked and unclocked cases need to be checked separately does the timing verifier have to compute both of them. In those cases, the timing verifier remembers the values of all the signals which are not affected by the signal which is subject to case analysis, and thus has to recompute only the signals which change with the signal being analyzed.

The designer must specify which signals require case analysis and list the cases; most circuits have proven to contain fairly few such signals.

Basically, the timing verifier then takes the first case, calculates when each signal in the system could be changing, and checks for violation of timing constraints for that case. It then goes on to the next case, recomputing only the signals which are different from those in the first case, and checking for any possible timing errors. It repeats this process for all of the cases.

6.1.1 Circuit Period

The circuit being verified must contain one basic clock, whose period is specified to the timing verifier. If different parts of the circuit run at different clock rates, then the period specified to the timing verifier is the least common multiple of the clock periods. For example, for a processor whose instruction unit has a period of 30 nsec and whose execution unit has a period of 15 nsec the designer would specify a 30 nsec period to the timing verifier. Within the circuit, clock signals may occur at any phase within the basic period.

6.1.2 Value system for signals

At any instant, each and every signal has one of seven values:

Value	Meaning
0	false, or 0
1	true, or 1
S or STABLE	signal is stable, not changing
C or CHANGE	signal may be changing
R or RISE	signal is going from zero to one
F or FALL	signal is going from one to zero
U or UNKNOWN	initial value used for all signals

The value of a signal over the clock period is represented by a linked list, each node of which specifies a value and the duration of that value. The sum of the durations of all the nodes in the list must equal the period of the circuit being analyzed.

When a signal propagates through a gate or wire where it is delayed by a variable amount of time, then skew is added to the signal, representing the uncertainty in when the signal will subsequently change. This skew is maintained separately in the representation of the signal to preserve information about the width of pulses in the signal, in order to avoid bogus timing errors asserting that minimum pulse width requirements have not been met. If two or more changing signals are combined, the skew then cannot be simply represented separately. It is therefore incorporated into the signal representation by using the CHANGE, RISE, and FALL values.

6.1.3 Combinational function

The following tables define the INCLUSIVE-OR (OR), EXCLUSIVE-OR (XOR), AND, CHANGE (CHG), and NOT functions for the seven-value logic system used in the timing verifier.

A OR B

A↓ B→	0	1	S	C	R	F	U
0	0	1	S	C	R	F	U
1	1	1	1	1	1	1	1
S	S	1	S	C	R	F	U
C	C	1	C	C	C	C	U
R	R	1	R	C	R	C	U
F	F	1	F	C	C	F	U
U	U	1	U	U	U	U	U

A AND B

A↓ B→	0	1	S	C	R	F	U
0	0	0	0	0	0	0	0
1	0	1	S	C	R	F	U
S	0	S	S	C	R	F	U
C	0	C	C	C	C	C	U
R	0	R	R	C	R	C	U
F	0	F	F	C	C	F	U
U	0	U	U	U	U	U	U

A XOR B

A↓ B→	0	1	S	C	R	F	U
0	0	1	S	C	R	F	U
1	1	0	S	C	F	R	U
S	S	S	S	C	C	C	U
C	C	C	C	C	C	C	U
R	R	F	C	C	C	C	U
F	F	R	C	C	C	C	U
U	U	U	U	U	U	U	U

A CHG B

A↓ \ B→	0	1	S	C	R	F	U
0	S	S	S	C	C	C	U
1	S	S	S	C	C	C	U
S	S	S	S	C	C	C	U
C	C	C	C	C	C	C	U
R	C	C	C	C	C	C	U
F	C	C	C	C	C	C	U
U	U	U	U	U	U	U	U

NOT A

A↓	
0	1
1	0
S	S
C	C
R	F
F	R
U	U

The output of the CHANGE function has the value CHANGE if any of its inputs are changing; otherwise it has the value STABLE. It is a useful function in modeling complex combinational logic, where the actual function being performed is not important to the verification process. Common examples are in the modeling of parity trees and adders, for which the timing verifier cares only when the outputs of these circuits are changing, not for their actual value.

6.2 Defining chips

As Section 4 explained, the macro expander operates in either of two modes, depending on which TERM file it uses. It can expand the design into a network consisting of macros which represent chiptypes, or it can expand one level further, replacing each chiptype with a network of logical primitives which describes the function and timing of the chiptype.

To produce input for the timing verifier, the macro expander must run in the latter mode, and therefore the designer must provide a drawing for each chiptype, defining that chiptype in terms of the logical primitives shown in Figure 6-1. (This is actually a subset of the primitives; logic simulation can use additional kinds not shown here.)

Delay--Most of the bodies have associated with them a string beginning "DELAY=0.0". This is the body parameter called VAR (represented in the graphics editor as a property name/text pair whose name is VAR and whose text is that string). When using these bodies in a drawing, the designer usually replaces the "0.0" with a delay expressed in nanoseconds, or a pair of delays (minimum and maximum) separated by a comma, or a trio of delays (minimum, typical, and maximum) separated by commas. Regardless of the number of zeros in the initial DELAY string for a particular body template, any body can accept one, two, or three delay paramters.

Minimum pulse width--The body called MIN PULSE WIDTH accepts a single bit input and checks that the pulses at that input exceed specified widths. The VAR parameter for this body is a string, initially set to "HIGH=0.0, LOW =0.0", which specifies in nanoseconds how long the input must remain high and how long it must remain low to avoid error.

Setup and hold check--The body whose body name is SETUP HOLD and whose macro name is SETUP HOLD CHK accepts an input (whose width is dictated by the SIZE parameter) at pin "I" and a common (one-bit) clock signal at "CK". Its VAR parameter specifies in nanoseconds the minimum setup and hold times for those inputs with respect to the rising edge of that clock.

The body whose body name is SETUP RISE HOLD FALL and whose macro name is SETUP RISE HOLD FALL CHK works in similar fashion, but checks the setup time with respect to the rising edge of the clock and the hold time with respect to the falling edge.

CHANGE gates--The gates whose names include "CHG" strip away information about the actual values—high or low—of their inputs. Their outputs have instead the two states STABLE and CHANGING. This simplifies the definition of the timing of complex functions for which knowledge of the exact logical operation is unnecessary.

AND, OR, and XOR gates--These operate in obvious fashion.

Latches and registers--Each of these accepts an input at pin "I" and an output at "T"; the SIZE parameter dictates the width of those inputs and outputs. The "CK" (clock), "EN" (enable), "R" (reset), and "S" (set) pins are common (each accepts a one-bit signal).

The first kind of register has only "CK" and "I" inputs, and changes its output on the rising edge of

"CK". The output of the register will be set to the "CHANGE" state between the time determined by the minimum and maximum delays of the register following the rising edge of "CK". Unless the "I" input is a true or false during the rising-edge of the "CK" input, the output will be set to the "STABLE" value for the rest of the cycle; otherwise, it will be set to the value of the "I" input.

The second kind of register has asynchronous "S" and "R" inputs in addition to the "I" and "CK" inputs. If the "S" (or "R") input is one, then it sets (or resets) the output of the register after the specified propagation delay.

The output of the first latch merely follows the "I" input when the "EN" input is high, and is stable for the remainder of the cycle. The second latch has additional asynchronous "S" and "R" inputs, which set or clear the latch when the "EN" input is low, after the specified propagation delay.

Multiplexers--Each of these bodies accepts an input at each of the numbered pins and an output at "T"; the SIZE parameter dictates the width of those inputs and outputs. The "S" input is common--one bit wide for the 2 MUX, two bits wide for the 4 MUX, and three bits wide for the 8 MUX.

If the select lines are changing, the output of a multiplexer is changing. If the select lines are stable but their value is not known, the output is the worst case of all the inputs. If the select lines have a known value, the output reflects the appropriate input. A change in the select lines or the input propagates to the output with the specified delay.

Note that these primitives are deliberately idealized, so it may take more than just a primitive latch to model accurately a real latch, and more than just a primitive multiplexer to model a real multiplexer, and so on. In particular, the primitives provide the same delay from each input to the output. If the real part exhibits different delays--if, for example, the "SET" input of a latch propagates to the output more rapidly than does the data input--then the definition must use a buffer at the slower input to increase its delay.

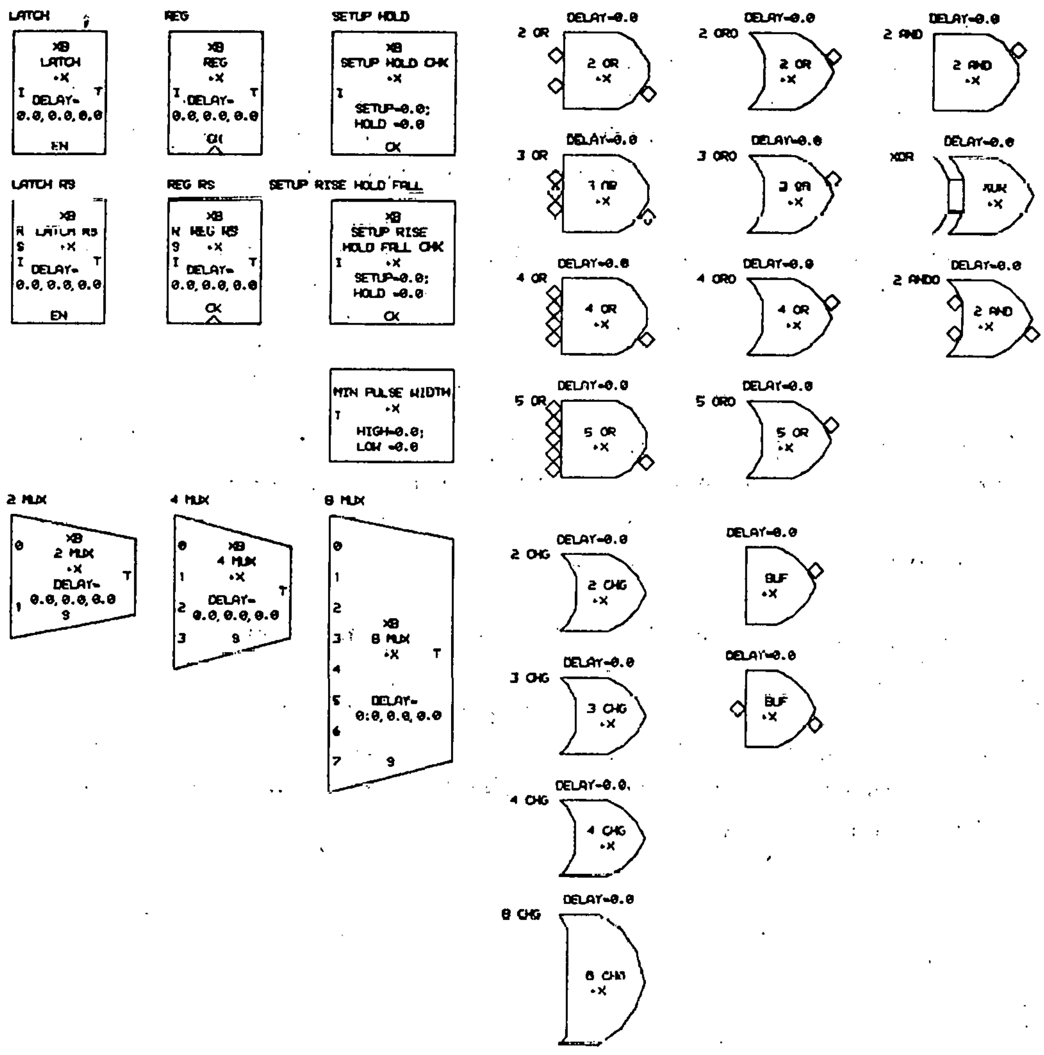


Figure 6-1
Timing verifier logical primitives

Figures 6-2 and 6-3 provide two examples showing how to define the timing of a chip using these primitives.

Figure 6-2 shows the definition of a 10145A, a 16-word RAM. Figure 6-3 shows the definition of a 10158, a 2-input multiplexer. The 10145A example models only timing, not logic function, thanks to the CHANGE gates, which strip away information about logic state. The model for the 10158, on the other hand, is an accurate model, which could be used to do full logic simulation. For the 10158, the model of its complete logical operation is necessary to verify timing constraints in many circuits.

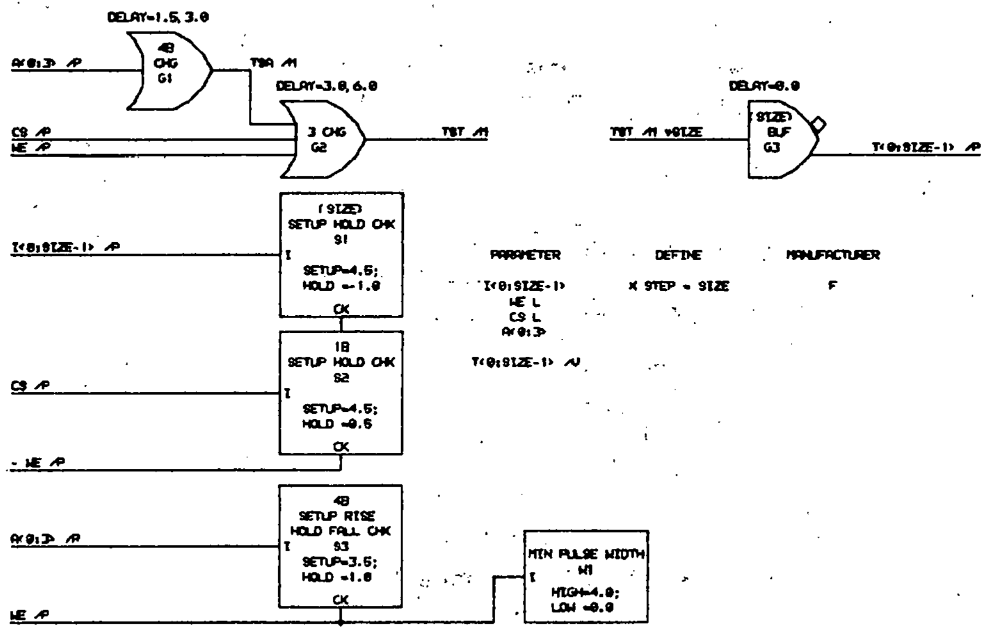


Figure 6-2
A 10145A 16-word RAM

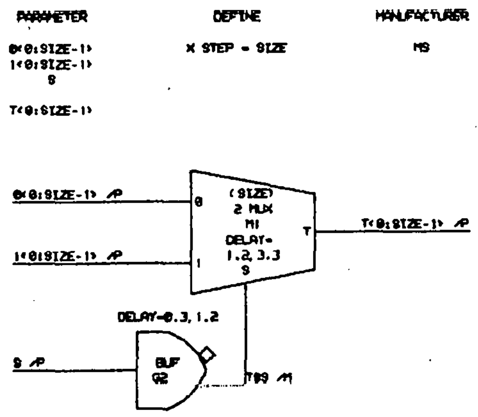


Figure 6-3
A 10158 2-input multiplexer

6.3 Preparing input for the verifier

Assertions, estimated wire delays, and evaluation directives are all incorporated in the names of signals in the drawings that make up the design (not in the drawings that define individual chips).

6.3.1 Wire delays

To specify a wire delay for a particular input signal, the designer must name the signal and include a <wire delay> after the <bit subscript> part (if any), as described in Section 4.5. Expressed in nanoseconds, this consists of either a single value or a pair of values (minimum and maximum, respectively) separated by a colon. In either case, enclose the delay in square brackets:

```
SHORT WIRE L<0:7>[1.0]
LONG WIRE[49.7]
INDETERMINATE WIRE<0:35>[1.0:49.7]
```

If the timing verifier is using wire delay information from the router based on actual wire lengths, it ignores these specifications.

In traversing the macro expansion tree, the timing verifier associates each predicted wire delay with the input which the signal feeds but not with the output that generates the signal, thus assuring that a single delay does not affect the network twice. As a result, placing a wire delay specification on a signal which is an output parameter of a macro definition has no effect.

6.3.2 Assertions on Signals

Assertions serve two purposes. Before a design is complete, the designer can isolate one module and place timing assertions on all the inputs and outputs of that module; the verifier will then use those assertions to take the place of the timing information it would otherwise obtain from the circuitry surrounding that module. Within a module or complete machine, the designer can place timing assertions on any signal for documentation purposes, and to convey to the timing verifier additional requirements that he or she wishes to impose; the verifier will then issue warnings if the assertions are not at least as generous as the actual timing—even if the actual timing is not strictly erroneous.

When it comes time to integrate separately verified modules into a complete machine, the macro expander automatically checks to see that the assertions on the outputs of one module match the assertions on the corresponding inputs of another. It considers an assertion to be part of a signal name, so two otherwise identical names with different assertions represent two different—and incompatible—signals.

If the timing of a signal is not defined by preceding circuitry or by an assertion, then the verifier assumes the signal is always stable; thus, one need not place assertions on input signals whose timing is not of interest.

The <timing assertion> part of a signal name appears after the <simple name> as mentioned in Section 4.5 and consists of a string beginning with a period. The syntax is:

```
<timing assertion>        ::= <clock assertion type>
                              <value specification> <skew specification> |
                              <clock assertion type> <value specification> |
                              <stable assertion type> <value specification>

<clock assertion type>  ::= C | P

<stable assertion type> ::= S

<value specification>     ::= <time range> |
                              <time range> , <value specification>

<time range>              ::= <time> | <time> - <time>

<time>                    ::= <real number>

<skew specification>      ::= ( <minus skew> , <plus skew> )

<minus skew>              ::= <negative real or zero>

<plus skew>               ::= <positive real or zero>
```

For a clock signal, a typical <timing assertion> is:

XYZ .C4-6 L

which says that the signal goes from high to low at time 4, and from low to high at time 6. (Each time unit represents a fraction of the cycle time; when you run the verifier program, you specify the number of units in a cycle. This convention keeps the assertions independent of the duration of the cycle time.) The signal:

XYZ .P2-3,5-6

is high from 2 to 3 and from 5 to 6, and is low for the rest of the clock cycle. If a single time is given instead of a range, then a time interval of one clock unit is assumed. For example,

```
XYZ .P2,5
```

is equivalent to the previous signal.

For clock signals, the "C" and "P" assertions are both useful, the only difference being the default skew used when none is explicitly given. Skew is generated by variations in the delay from the clock generator to different parts of a large digital system, due to varying wire lengths and buffer propagation delays. In a large digital system, these variations can become large enough to degrade performance unacceptably. To reduce this skew, the shorter clock paths can have additional delay deliberately inserted into them. Because the delays in a clock distribution system may vary between successive implementations of a design, in many cases it must be adjusted by hand, by using some type of adjustable delay for each of the clock lines. Using this technique, the skew can be reduced below some designer-specified amount. A "P" assertion assures the verifier it can rely upon such adjustments; a "C" assertion does not.

For a control or data signal, use the "S" assertion, which specifies whether the signal is stable or changing, but not its actual value. For example, the name:

```
XYZ .S4-8
```

says that the signal is stable from time 4 to time 8, and may be changing during the rest of the cycle. Note that an "S" assertion never specifies a skew.

6.3.3 Evaluation Directives

Evaluation directives tell the timing verifier how to evaluate certain gates. They can also specify the exact point in a circuit at which a precision clock is adjusted to reduce skew.

As mentioned in Section 4.5, an <evaluation directive> follows the <wire delay> in a signal name. It consists of "&" immediately followed by a string of letters. The first letter in the string refers to the logical primitive (ordinarily a gate) immediately following the signal, the second refers to the second level of gating following the signal, and so on.

The following letters are permitted:

W	Zero the wire delay going into the gate that this evaluation directive refers to.
Z	Zero the wire delay going into the gate and the delay of the gate itself.
A	When this signal is asserted, make sure all other inputs to the gate are stable. If so, operate as if the directive were "I": ignore the other inputs and base the timing of the gate's output solely on that of this signal. If not, issue an error

message.

I — Ignore the other inputs of the gate and base the timing of its output solely on that of this signal.

H — This directive is equivalent to applying the "A" and "Z" directives together at a single level.

6.3.4 Correlations

When the operation of a network relies on known correlations between clock signals, the timing verifier must be told the correlations or it will generate spurious errors. Consider the two examples in Figure 6-4, each driven by a clock exhibiting plus or minus 2 nsec skew. The first example represents an authentic timing error because if the clock arrives at register R1 2 nsec before it arrives at register R2, the input of R2 will be changing as the clock rises. The timing verifier sees the second case no differently, but in reality no error can occur because the input and output of the latch are governed by exactly the same clock. No matter how great the skew, the changing output cannot propagate back to the input to conflict with the rise of the same clock pulse that caused the changing output.

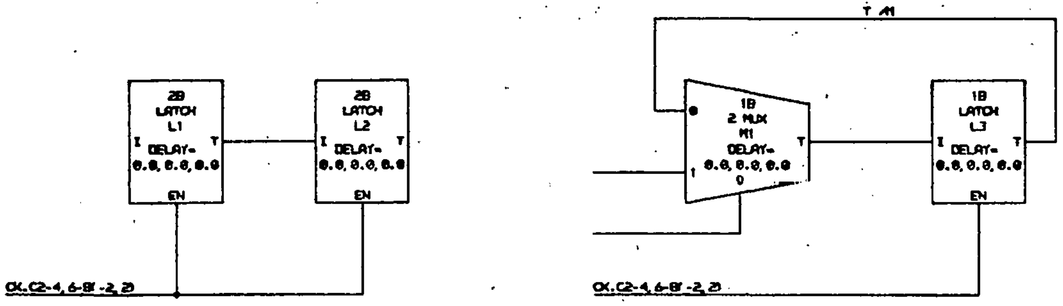


Figure 6-4
Uncorrelated and correlated clocks

To solve the problem, add to the signal called "T /M" a wire delay sufficient to eclipse the clock skew. To make clear that this delay is meant to convey a correlation rather than to suggest a lengthy wire, it is customary to define a text substitution called CORR and use it as the delay:

```
T /M[\CORR\]
```

6.4 Input and output files for the timing verifier

The verifier accepts the following input files:

MACEXP This is the output data from the macro expander.

OPTION This file contains a set of real-number equates specifying various options and parameters. A typical OPTION file might look like:

```
CycleTime=50.0;
ClockUnits=6.25;
ClockSkew=5.0;
PrecClockSkew=1.0;
MaxWDelay=2.0;
MinWDelay=0.0;
```

CycleTime is the length in nsec of the least common multiple of all clock periods in the network.

ClockUnits is the length in nsec of one of the time units used in the <timing assertion> syntax. Usually CycleTime is evenly divisible by ClockUnits, though this is not a requirement.

ClockSkew is the default skew used when a clock signal bears no timing assertion or bears a ".C" assertion with no skew specified. In the preceding example, the default skew is -5 to +5 nsec.

PrecClockSkew is the default skew used when a ".P" timing assertion specifies that a clock is precision adjusted but does not specify the resulting skew. In the preceding example, the default skew for precision clocks is -1 to +1 nsec.

MaxWDelay and *MinWDelay* are the wire delay values in nsec used when a signal name does not specify a wire delay. In the preceding example, defaulting the wire delay would have the same result as specifying "[0:2]".

WIRES Produced by the physical design system router program, this file provides wire delays based on actual wire lengths and chip electrical characteristics. If this information is not yet available, provide a file containing the word "END;" and the verifier will use the wire delay estimates specified within signal names.

The timing verifier produces the following output files:

TIMLST This is a listing of timing errors plus a listing of each signal along with a description of its behavior versus time.

LCROSS	This is a cross reference of local signal names.
GCROSS	This is a cross reference of global signal names.
BCROSS	This is a listing of signals which for various reasons appear to be "dangling". These are not necessarily errors, but might be conscious omissions by the designer.

6.5 A timing verifier example

Figure 6-5 shows a sample SCALD macro consisting of a 16-word by 32-bit RAM, a 32-bit register, a 2-input multiplexer and several gates. It illustrates the use of assertions, evaluation directives, and predicted wire delays in signal names. It in turn calls several more macros, the two most interesting of which appeared earlier as Figures 6-2 and 6-3.

The assertion on the signal "W DATA .S0-6<0:31>" says that it is stable from time 0 to time 6, allowing the verifier to check the timing of this circuit without knowing how the signal is generated. The assertion on the clock signal "CK .P2-3 L" says that it is low between times 2 and 3, and high for the rest of the cycle. The signal "ADR<0:3> [0.0:6.0]" states that the 4 address wires on the RAM can be between 0.0 and 6.0 nsec long.

The clock signal "CK .P2-3 L" is being ANDed with the control signal "WRITE .S0-6 L" to generate a write-enable pulse for the RAM array. If the data is stable every cycle during the period in which the RAM is to be written, then the most efficient way to check for timing errors is just to analyze the case in which the signal "WRITE .S0-6 L" enables a write operation. The "&H" directive shown at the end of the clock signal says to ignore the value of the "WRITE .S0-6 L" signal, allowing the clock signal always to propagate through the gate. In addition, it says the timing specified by the clock signal is to be adjusted so that it refers to the time at which the output, rather than the input, of the gate changes. The "&H" directive also specifies to check that the control signal "WRITE .S0-6 L" is stable while the clock is asserted, to ensure that the write will be either solidly enabled or solidly disabled.

The "&Z" directive on the signal "CK .P0-4" states that the clock timing refers to the time at which the output of the gate changes.

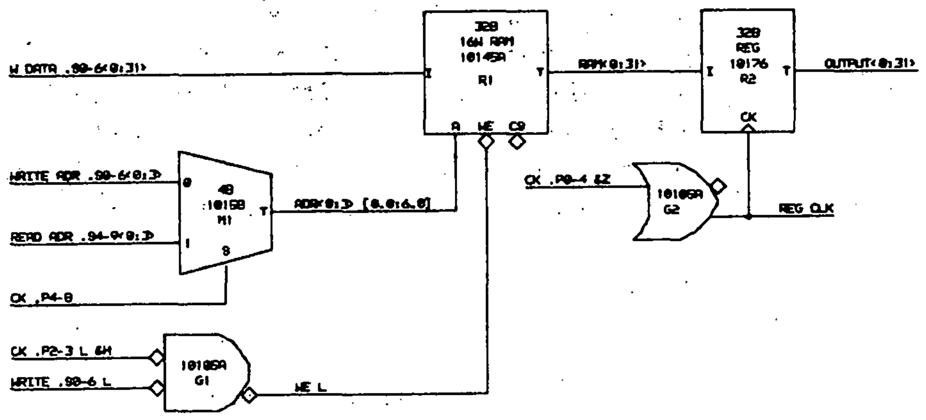


Figure 6–5
Example to be verified

The first step in verifying the timing is to run the macro expander to expand the design into logical primitives. Then run the timing verifier, which processes the MACEXP file generated by the macro expander. It generates a listing (somewhat condensed here to fit the page) which begins with a play-by-play description of its operation:

```
Reading wire list ...
   0 error(s) detected
Doing cross reference listing ...

Initializing signals ...
   0 error(s) detected
Doing timing analysis ...
Circuit evaluation completed
Total number of evaluation passes:     6
Total number of events processed:     28
```

Next the listing shows setup, hold, and pulse width errors:

```
Setup, Hold, and Minimum Pulse Width errors ....

Setup time error; Setup Time = 3.5, Hold Time = 1.0
CK INPUT   = WE                 0:0.0, R:11.5, 1:15.5, F:17.0, 0:21.0
DATA INPUT = ADR                S:0.0, C:0.5, S:11.5, C:25.5, S:36.5

Setup time error; Setup Time = 2.5, Hold Time = 1.5
CK INPUT   = REG CLK            R:0.0, 1:3.0, F:24.0, 0:28.0, R:49.0
DATA INPUT = RAM                S:0.0, C:5.0, S:22.5, C:30.0, S:47.5
```

Because of the long wire specified on the signal "ADR<0:3> [0.0:6.0]", two set-up time errors occur. The first error message shows the address inputs to the RAM becoming stable at 11.5 ns, just as the write enable (WE) signal starts rising. Since the RAM requires a setup time of 3.5 nsec, the wire delay on the address signal must be reduced to 2.5 nsec to eliminate the error. The second error message shows the data output of the RAM becoming stable at 47.5 nsec and the clock starting to rise at 49.0 nsec, giving only 1.5 nsec of setup time instead of the required 2.5 nsec.

Next it prints a list of signal values:

```
Values of all signals

ADR<0:3> .     .    .    .    S:0.0, C:0.5, S:5.5, C:25.5, S:30.5
CK .P0-4 .     .    .    .    R:0.0, 1:1.0, F:24.0, 0:26.0, R:49.0 (constant value)
CK .P2-3 .     .    .    .    0:0.0, R:11.5, 1:13.5, F:17.0, 0:19.0 (constant value)
CK .P4-0 .     .    .    .    F:0.0, 0:1.0, R:24.0, 1:26.0, F:49.0  (constant value)
OUTPUT<0:31>   .    .    .    S:0.0, C:0.5, S:7.5
RAM<0:31>.     .    .    .    S:0.0, C:5.0, S:20.5, C:30.0, S:45.5
READ ADR .S4-9<0:3>.     .    S:0.0, C:0.3, S:25.0
REG CLK  .     .    .    .    R:0.0, 1:1.0, F:24.0, 0:26.0, R:49.0
W DATA .S0-6<0:31> .     .    S:0.0, C:37.5
WE  .    .     .    .    .    0:0.0, R:11.5, 1:13.5, F:17.0, 0:19.0
WRITE .S0-6    .    .    .    S:0.0, C:37.5
WRITE ADR .S0-6<0:3>     .    S:0.0, C:37.5
```

In that listing, "S" stands for "stable", "C" for "changing", "F" for "falling", "R" for "rising", "U" for "unknown", "1" for the high state and "0" for the low state. Consider the first signal in the list, "ADR<0:3>". Because the timing is identical for all four of its bits, the listing describes them all in one line. The signal is stable at time 0 (the beginning of the cycle), changes from 0.5 nsec to 5.5 nsec, remains stable until 25.5 nsec, changes from 25.5 nsec until 30.5 nsec, and finally remains stable from 30.5 nsec until the end of the cycle.

Next it prints a list of signals whose timing failed to fall within the limits set by assertions. (These are signals for which the designer specified assertions even though the verifier could calculate their timing without those assertion. The verifier thus calculates the timing independently and uses the assertions as a check.) This example has no such errors, but a typical one might look like the following example, which gives the signal name including the assertion, followed by the calculated timing:

```
Signals not meeting their stable assertions
DC MODIFIED I0 .S6-12          S:0.0, C:23.0, S:28.6

I-SEQ USING SA .S5-10          S:0.0, C:18.0, S:31.5
```

Finally, the listing shows how much storage the program used. This is useful when running the verifier on computers with limited address spaces, because it helps predict when a design is about to grow so large that it must be split into modules which can be verified individually:

```
All done

Storage summary:
Record Name  Number Used Total Bytes
Value                416        4992
ValueBase            125        2500
ValueHead            120        1440
Signal                17         680
Def                   35        1820
CallLst               13         468
PrimDef               23        3864
ParArr                35         140
CallLstArr           248         992
StringChr            527         527
Str                  111        1332
SortSigArr            14          56
                               18811
```

7 The layout program

Starting with the circuitry established by the macro expander plus a set of instructions from the designer, the SCALD layout program positions chips on circuit boards. The program is semiautomatic: for best results, the designer specifies how to lay out important or complicated macros, but lets the program do the routine part of the job automatically.

Thus the program requires three inputs: a circuit description from the macro expander, two files called CHIPS.LAY and CHPTYP.LAY (derived from the CHIPS file by another part of the SCALD physical design system) which describe the chips themselves, and a file of instructions from the designer. It puts out a listing, a file of runs for use by the SCALD router, a file of unconnected signals for error-checking, and a file describing the position of each chiptype laid out.

To avoid confusion, this chapter will use 'location' to mean the label generated by the graphics editor and used by the macro expander to indicate where within a drawing a particular macro lies. A location is simply an identifying string such as "G1" or "M6". It will use 'position' to indicate where upon a board a particular macro or chip lies. A position is a set of coordinates on a circuit board. In fact, the main task of the layout program is to map a set of locations onto a set of chips at specified positions.

7.1 Preparing instructions for the layout program

To give instructions to the layout program, the designer creates a file containing a sort of program that consists of statements, analogous to the statements of a high level language like PASCAL or FORTRAN.

Context–The layout program starts at the top level of the macro expansion tree and works downward toward the most primitive elements. Similarly it starts at an initial board position and works onward from there. At any time, the program works within a *context*, consisting of the location label for a particular macro call, plus a board position. All of its work takes place relative to that macro call and that position.

Position–Several of the statements use identical syntax for position. A position can specify four elements or 'coordinates': a board, a column of chips on that board, a row of chips on that board, and a section within the chip at that row/column. Most of the statements allow the designer to default one or more of these elements.

Specifying board, row, and column pinpoints exactly one socket; and sockets are all equivalent as far as the layout program is concerned; within the constraints imposed by the instructions from the designer, the program will map chips onto sockets in whatever fashion minimizes wire lengths. (The layout program treats a socket as if it were a point to be placed at a certain coordinate position. Other programs in the physical design system know the true size of each socket, how many pins it has, whether it is interchangeable with sockets at other coordinate positions on the board, and so forth. They worry about checking to make sure sockets don't overlap each other, and so forth.)

The section coordinate, on the other hand, indicates a functional unit within a chip and thus depends on the chiptype, as we will explain later. For now, regard the section as simply another coordinate. Similarly, we will postpone the question of chips that require more than one socket.

Boards, columns, and rows have integer numbers beginning at 1. Sections have names, given to them by the CHIPS.LAY and CHPTYP.LAY files, which consist of an optional alphabetic string followed by a number—'A0', for example, or 'SG12', or just '7'. To identify a board, precede its number with '@B'; for a column, precede its number with '@C'; for a row, '@R'. For a section, precede the alphabetic/numeric name with '@S'. The following specifies board 5 row 16 column 12 section A1:

```
@B5 @R16 @C12 @SA1
```

The following specifies row 12 column 5 within the current board:

```
@R12 @C5
```

More commonly, however, the designer will—by omitting the '@'--specify position relative to the current context. If the program is already working with board 3, for example, then 'B1' indicates board 3 (the current board), 'B2' indicates board 4, 'B3' indicates board 5, and so on. In a section name, the numeric part is taken relative to that of the context while the alphabetic part (if any) is absolute. If the program is already working with section A3, for example, then 'SA0' indicates section A3 (the current one), 'SA1' indicates section A4, and so on.

For example, if the current context is '●B5 ●R16 ●C12 ●SB1', then the following:

```
R4 C6
```

actually indicates board 5, row 19, column 17 section B1.

A simple rule: given the context and a set of context-relative coordinates, determine the position by adding each relative coordinate to the corresponding context coordinate and then subtracting 1 (for board, row, or column) or 0 (for section).

Some layout program statements require a list of positions separated by commas, such as:

```
R16 C12, R16 C13, R16 C14, R16 C15, R16 C16
```

To abbreviate this, one can use an implied loop by specifying the initial value for the loop and the number of times the loop should execute (*not* the final value of the loop as in many programming languages). The following example steps from C12 through C16:

```
C(12,5)
```

and thus is equivalent to:

```
C12,C13,C14,C15,C16
```

An optional step size increments or decrements the loop by any desired integer. The following example steps from C12 through C16, incrementing by 2:

```
C(12,3,2)
```

and thus is equivalent to:

```
C12,C14,C16
```

Ordinarily the order of the board, column, row, and section specifications doesn't matter. When you use more than one of these implied loops, however, the loops nest, with the rightmost loop incrementing most rapidly. The following examples are thus equivalent:

```
R(1,2) C(1,2)
R1 C1, R1 C2, R2 C1, R2 C2
```

but different from the following two, which are likewise equivalent:

```
        C(1,2) R(1,2)
        R1 C1, R2 C1, R1 C2, R2 C2
```

An implied loop for section names looks like this:

```
        SA(0,4,2)
```

which is equivalent to:

```
        SA0, SA2, SA4, SA6
```

Locations--As mentioned before, a location is simply the label designer chose to identify a particular macro call to the macro expander. (originally, the text field of the LOC property name/text pair for the body that calls the macro). The designer needs to keep in mind three additional details.

First, when the SIZE parameter causes multiple invocations of the same macro call, the macro expander appends a '*' and a number to each invocation after the first. Thus, a body for which LOC=G5 and SIZE=3 results in locations called G5, G5*2, and G5*3.

Second, when the TIMES parameter causes multiple invocations of the same macro call, the expander appends a '+' and a number to each invocation after the first. The TIMES suffix precedes the SIZE suffix, if any, so that the last invocation of a body with LOC=G6, SIZE=3, and TIMES=2 would be called 'G6+2*3'. (SCALD customarily deals with these invocations alphabetically, so SIZE varies faster then TIMES as the macro expander steps through the two-dimensional matrix of calls resulting from such a body.)

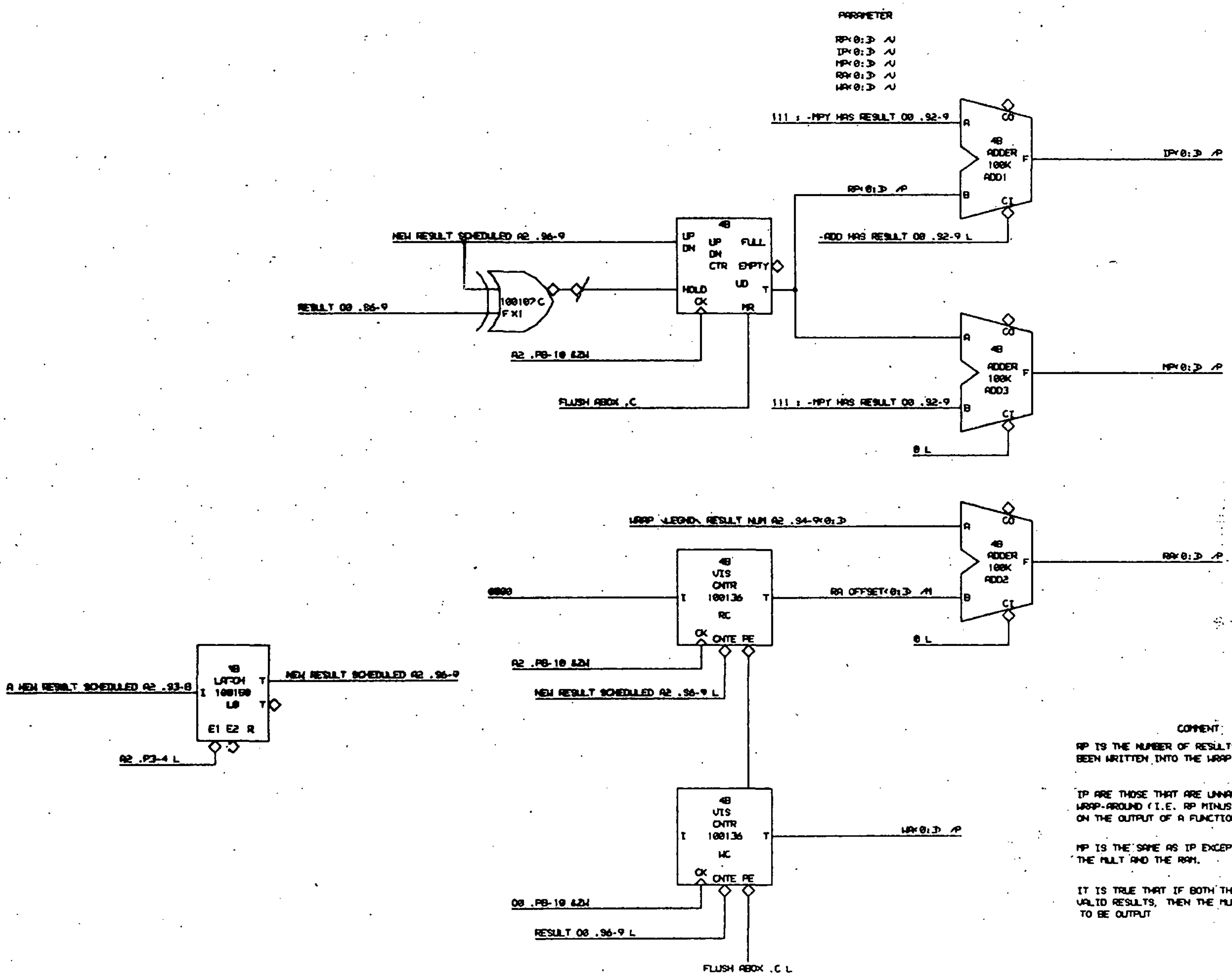
Third, when more than one drawing defines a macro, the expander prefaces each location with the page number of the drawing. Thus, a body for which LOC=G5 would result in location 1G5 if it lay on page 1 of the drawing, location 2G5 if it lay on page 2, and so on.

Chiptypes--Many ICs contain several functions inside one package. The layout program recognizes an entity called a *chiptype* which embodies that concept.

Defined by the CHIPS.LAY and CHPTYP.LAY files, each chiptype is a collection of one or more terminal macros within a single unit which the layout program can place on a board. Such a chiptype contains one or more *sections*, each corresponding to one terminal macro.

An ECL 10105 chip, for example, contains two 2-input OR/NOR gates and one 3-input OR/NOR gate. The macro expander need know nothing about this. Simply use two different macros to represent the two different kinds of gate: for example, one macro called 10105A to represent a 2-input OR/NOR gate and another called 10105B to represent a 3-input OR/NOR gate. The CHIPS.LAY and CHPTYP.LAY files must then tell the layout program that a chiptype called 10105 will provide two 10105A's (in sections 'A0' and 'A1') and one 10105B (in a section called 'B0').

Some chiptypes are a good deal simpler, of course. The chiptype 10016 corresponds to exactly one



COMMENT:

RP IS THE NUMBER OF RESULTS THAT HAVE NOT
BEEN WRITTEN INTO THE WRAP RAM

IP ARE THOSE THAT ARE UNAVAILABLE FOR
WRAP-AROUND (I.E. RP MINUS THOSE THAT ARE CURRENTLY
ON THE OUTPUT OF A FUNCTIONAL UNIT)

MP IS THE SAME AS IP EXCEPT THAT IS ONLY COUNTS
THE MULT AND THE RAM.

IT IS TRUE THAT IF BOTH THE MULT AND THE ADDER HAVE
VALID RESULTS, THEN THE MULT IS ALWAYS FIRST
TO BE OUTPUT

WRAP ADR GEN

7.1.1 The DATE statement

To help document the layout, the designer may include a DATE statement at the beginning of the file of instructions, giving the date and the designer's initials. The layout program will pass this string along to subsequent programs in the SCALD physical design system. (Actually, the statement will in general accept any string of characters not including ";"):

```
DATE 9-Oct-79 JBR;
WITH *;
   WITH C3;
      G5 = R1 C5 S1;
      M4 = R1 C6 S1;
   END;
END;
```

7.1.2 The WITH statement

A WITH statement establishes a context within which other statements work. Every WITH statement must pair with an END statement; the context it establishes applies to every statement between the matching WITH and END.

The simple form of the WITH statement specifies a single location--in other words, specifies a particular macro call-- and causes the statements within its scope to work within the context of that macro. The following example operates on macro calls G5 and M4 within the context of (that is, within the expansion of) macro call C3:

```
WITH C3;
   G5 = R1 C5 S1;
   M4 = R1 C6 S1;
END;
```

Commonly, the instructions for a layout program will nest WITH statements. Each WITH statement descends one level deeper into the macro expansion tree, and concatenating the WITH statement locations one by one creates a *path* that completely and unambiguously describes the context (that is, the particular macro call) within which the innermost statements work. In the following example, the innermost statements deal with the macro calls whose paths are (C1 G2 A1#6 G5) and (C1 G2 A1#7 M3):

```
WITH C1;
   WITH G2;
      WITH A1#6;
         G5 = R1 C5 S1;
      END;
      WITH A1#7;
         M3 = R2 C6 S1;
      END;
   END;
END;
```

Because the topmost level of the macro tree--the macro representing the entire design--never gets called by any other macro, it has no location name. Thus, the first WITH statement in any layout program must use the special symbol '*' to indicate the topmost level of the design. Other statements may precede this initial WITH (provided they do not require a context in which to operate--statements which actually cause the program to place chips on boards always require a surrounding WITH to tell them what part of the design to work on) but its matching END must be the last statement of the program:

```
SET XYZ = 15;
WITH *;
   WITH A1;
      G2 = R2 C6 S1;
```

```
        G7 = R2 C6 S1;
     END;
  END;
```

Specifying position--Any WITH statement may include an AT clause, specifying a position relative to that of the previous WITH context. The program starts at position '@B1 @C1 @R1 @S0', so in the following example the innermost statement operates in a context whose macro call path is (G4 A6) and whose position is board 3, row 17, column 7:

```
WITH * AT R3 C2;
   COMMENT Now we're at board 1 row 3 column 2;
   WITH G4 AT B3 R5 C2;
      COMMENT Now we're at board 3 row 7 column 3;
      WITH A6 AT R11 C4;
         COMMENT Now we're at board 3 row 17 column 6;
         G5 = R1 C2 S0;
      END;
   END;
END;
```

If a WITH statement doesn't include an AT clause, then the context inside that WITH has the same position as the context enclosing it. Note that while intervening layout statements may have altered the position at which the program is working, a WITH statement ignores them, goes back to the previous WITH statement, and alters the context position relative to it. Thus the innermost context is identical in the following two examples:

```
WITH *;                         WITH *;
   WITH G1 AT R16 C7;              WITH G1 AT R16 C7;
      M5 = R2 C1;
      WITH G5 AT R5;                  WITH G5 AT R5;
         M6 = R1 C1;                     M6 = R1 C1;
      END;                            END;
   END;                            END;
END;                            END;
```

Because the AT clause specifies a single position, the notation for an implied loop is illegal within it.

7.1.3 The assignment statement

An assignment statement tells the layout program to place a particular piece of circuitry at a particular position on a board. In its simplest form, it consists of a location name for a terminal macro (that is, a macro which maps directly to a single section of a chiptype), the name of a chiptype, an '=' symbol, and a position. The board, row, and column coordinates of the position are--unless they contain '@'--interpreted relative to the context of the enclosing WITH statement. Any omitted coordinate defaults to that of the enclosing context. Thus, the following example places macro call C7 on the board specified by the context, at the column specified by the context, one row beyond the row specified by the context, and at the third 'B' section of the chiptype:

```
C7 = R2 C1 SB2;
```

Note that the position in an assignment statement is *not* relative to that of a previous assignment statement.

If a terminal macro has its SIZE or TIMES body parameter set to a number larger than one, then that macro will require multiple chiptype sections and thus multiple positions on the board. If, for example, macro C7 had SIZE set to 3, the designer must give a list of positions, one for each invocation of the macro. Either of the following examples would accomplish this:

```
C7 = R2 C1 SB(0,3);
C7 = R2 C1 SB0, SB1, SB2;
```

An entire subtree at once--If the designer specifies the location of a nonterminal macro call, the layout program will automatically expand that macro call to obtain a list of terminal macro calls, and will then lay out each of the terminal macro calls. In other words, the program will traverse the subtree resulting from a nonterminal macro.

In such a case, the assignment statement must give a list of positions to the right of the '=' sign, one position for each terminal macro resulting from the expansion, and taking into account the multiple copies of a terminal that result from SIZE or TIMES parameters that exceed 1. In the following example, the macro at location C6#2 (which is itself the second invocation of macro C6, resulting from the SIZE parameter being greater than one) expands to produce five different terminals, and thus requires a list of five positions:

```
C6#2 = R1 C1, R1 C3, R1 C5, R2 C1, R2 C3;
```

The list of positions must contain at least one position for each terminal resulting from the expansion, but may contain more. The layout program will simply ignore the extra positions rather than using them up or leaving them empty. This makes it easier to use the loop notation to simplify a list. The following assignment statement has precisely the same effect as the preceding example, even though it specifies an extra position:

```
C6#2 = R(1,2) C(1,3,2);
```

When the layout program expands the subtree of a macro for you, it does so predictably. After expanding the subtree into a list of terminal macros, it sorts the list in alphabetic order by path name. (These path names are identical with those that appear in the output listing from the macro expander except that a blank space appears immediately before the closing parenthesis of the path to prevent the parenthesis from spoiling the alphabetizing.)

Inversion--Certain chiptypes allow you to use the complementary form of some of their inputs simply by rearranging connections. With a multiplexer, for example, simply rearranging the data inputs permits use of complemented (assert low) forms of the select lines. This is particularly important with ECL logic families which typically provide both true and complemented forms of gate outputs; using both outputs effectively doubles the fanout capability of the gate.

Provided the CHIPS.LAY file describes the required signal rearrangement, the assignment statement permits the designer to choose whether to use true or complemented connections for any particular instance of a given chip. Simply add a '/' to the end of the assignment statement, followed by a list of 'H' and 'L' letters:

```
CG#2 = R(1,2) C(1,3,2) / H L H H L;
```

Every time it positions a chip on the board, the layout program looks at the next letter in the H/L list. An 'H' tells it to connect the chip according to the drawing that defines that chip, and an 'L' tells it to rearrange the inputs to use complemented inputs.

Two shortcuts make the H/L list easier to use. First, placing a number immediately after a letter is equivalent to repeating the letter that number of times, so the following two examples have the same effect:

```
CG#2 = R(1,2) C(1,3,2) / H L H H H L;
CG#2 = R(1,2) C(1,3,2) / H L H3 L;
```

Second, if the layout program exhausts the H/L list before positioning all the chips, it returns to the beginning and reuses the list as many times as necessary (just as, in FORTRAN, a WRITE statement which exhausts its FORMAT list reuses the list). Thus, if the H/L list repeats a pattern, the designer need write only one cycle of the pattern:

```
CG#2 = R(1,3) C(1,3,2) / H L H H L H H L H;
CG#2 = R(1,3) C(1,3,2) / H L H;
```

If the H/L list is too long, the layout program simply ignores the extra letters.

Versions--As explained in Section 4.3.1, the macro expander provides any macro call with a parameter called TIMES which allows the designer to obtain multiple copies of that macro with their respective inputs tied together and their respective outputs left independent.

Within the drawings, the designer need not distinguish among these independent outputs: a single line and a common name represent all of them. The macro expander does, however, derive a unique name for each actual output by appending to the common name a '/' followed by a number.

When laying out circuitry automatically with the PLACE statement, the designer need not specify which version to connect to which input, because the physical design system router program handles this detail. But with the assignment statement, the designer may either leave the decision to the router or state explicitly which version of a multiversion output to connect to each possible destination.

To specify this, use a *colonstring* which gives a signal name and a list of version numbers, one for each macro call in the subtree. The following example connects version 2 (of whatever signal is appropriate) to input pin 'CNT ENBL' of the first macro in the subtree, version 1 to the corresponding pin of the second macro in the subtree, and version 3 to the corresponding pin of the third macro in the subtree:

```
C5 = R(1,3) C5 S0 :CNT ENBL = 2, 1, 3;
```

Note that the signal name must be an input, never an output. If the statement uses an H/L list, the colon string may either follow or precede it.

As with the H/L list, the program permits shortcuts. If the version list is not long enough, the program rereads it; if the list is too long, it ignores the latter part. To repeat a version, append a '*' followed by the number of repetitions desired. For example, the following two assignments are identical:

```
C6 = R(1,7) C5 S0 : CNT ENBL = 1, 2, 2, 3, 3, 3, 3;
C6 = R(1,7) C5 S0 : CNT ENBL = 1, 2*2, 3*4;
```

Note that the program expects to read from the list one version number for each macro in the expansion, whether or not the macro in question actually has an input pin with the specified name, and whether or not the macro is a terminal. To 'skip over' macros which do not have a particular input, omit the version number but include the appropriate repetition factor. Thus, the following example skips over the first macro and the last three macros in the subtree:

```
C7 = R(1,8) C6 S0 : PRESET = *1, 1*2, 2*2, *3;
```

This ability to skip over certain macros in the expansion is also useful when a subtree contains two different kinds of macro calls with the same input signal name, where one set of calls needs version specifications and the other set does not.

It is possible for the macro expander to produce a signal name with multiple versions, one after another. To cope with this situation in the colon string, specify the versions in the proper order, separated by dots:

```
C8 = R(1,4) C4 S0 :CLEAR = 1.2*2, 1.1*2;
```

If more than one input signal requires version specifications, simply use multiple colon strings in any order:

```
        C9 = R(1,8) C4 S0 :CNT ENBL = 1*4, 2*4 :CLEAR = 2.2*2, 2.1*2;
```

Adding drive capability during layout--By including a TIMES expression in parentheses after the location label, the designer can override the TIMES parameter used for that macro call during macro expansion. This is useful for adding extra drive capability. Like the macro expander TIMES parameter, this feature replicates the macro, ties together the corresponding input signals of the resulting copies of the macro, and leaves the output signals independent, assigning a different version number to each output.

The list of sections in the assignment statement must be large enough to take into account these extra copies of the macro.

The following example produces three copies of the macro at location C7 and assigns them new location labels C7, C7+1, and C7+2:

```
        C7(*3) = R2 C1 SB(2,3)
```

The layout program permits this TIMES expression only with terminal macro calls.

7.1.4 The PLACE statement

Whereas the assignment statement lays out chips manually, the PLACE statement lays them out automatically. While the assignment statement dictates exactly where to put each macro, the PLACE statement allows the program to rearrange the chips (through pairwise interchanges) among the specified positions in an attempt to minimize wirelengths.

To use this automatic layout feature, specify a list of location labels and a list of positions:

```
        PLACE G5, M6, A4, R2 WITHIN B2 R(1,5,2) C(6,12) S(0,2);
```

If the list of positions is larger than it need be, the program will simply allow itself extra freedom in placing chips anywhere within the set of positions.

As with the assignment statement, if the macro call designated by a location label is a nonterminal, the program automatically expands the macro to obtain a list of terminal macros. In other words, it lays out the entire subtree resulting from that macro call.

Combining automatic and manual layout--Using a '*' instead of a list of locations tells the PLACE statement to lay out automatically all macros within the current context for which there are no assignment statements. In the following example, the program expands macro call G6, lays out calls M4 and R1 according to the assignment statements, and lays out the remainder of the subtree resulting from G6 automatically:

```
        WITH G6;
           M4 = R5 C7 S0;
           R1 = R5 C7 S1;
           PLACE * WITHIN R5 C(8,9) S(0,1);
        END;
```

The layout program does not actually perform assigment and PLACE statements in order. Instead, it performs all manual layout throughout the design and then all automatic layout. Thus the automatic layout algorithm can minimize the length of wires between manually-positioned and automatically-positioned chips along with that of the wires within the automatic areas.

As a result, assignment statements may actually occur either before or after the PLACE statement. In addition, the list of locations (or the subtree of a location) given to a PLACE statement may include some macro calls specified in manual layout statements elsewhere; PLACE will process only the macro calls which are not positioned manually.

Chiptypes of varying sizes--Because the layout program will rearrange the chips anyway, the order of positions within the list does not matter. But the 'resolution' of positions does matter: if each chip is two rows tall and one column wide, for example, the positions in the list should be two rows apart. Or, for another example, if each chip is three rows tall and two columns wide, the positions should be three rows apart and two columns apart. Otherwise, because the layout program deals only with the upper left corner position of each chip and not with the actual size of the chip, chips may overlap.

If a particular macro expands to require chiptypes of varying sizes, there are several solutions:

- Specify positions far enough apart to accomodate the largest chiptype, wasting some board space on the smaller chiptypes.
- Use the BIND statement to group macros into 'super-chiptypes' which are then all identical in size and shape.
- Use multiple PLACE statements to force the layout program to segregate the chiptypes by size into different areas.
- Take a chance, hoping that none of the chiptypes overlap, and later replace the PLACE statement with assignment statements if they do.

7.1.5 The BIND statement

The BIND statement aids the PLACE statement by pointing out patterns and symmetries that the automatic layout algorithm might otherwise miss. Ordinarily, the PLACE statement maps terminal macro calls onto chiptype sections in alphabetic order by pathname, without attempting to optimize section assignments or exploit symmetries. The BIND statement, however, allows the designer to group together in one chip the terminal macro calls which logically belong together.

A BIND statement applies to every PLACE statement within the scope of the enclosing WITH statement, even if the BIND actually appears following the PLACE.

The BIND statement takes the form:

```
BIND <location> = <list of sections> # <list of chiptypes>
        & <list of instances>
```

The <location> names a macro call. If the macro is not a terminal, the layout program will expand it as usual in alphabetic order by path name.

The <list of sections> includes a section for each terminal macro resulting from the <location> specified. Its syntax is identical with that described for the list of sections within the assignment statement, and may list more sections than necessary.

The <list of chiptypes> is a series of chiptype names separated by commas.

The <list of instances> is a series of names invented by the designer and separated by commas. Each name must consist of one or more letters, digits, or '+' and '–' characters.

For each terminal macro call in the subtree of <location>, the layout program will read the next section from <list of sections>, the next chiptype from <list of chiptypes> and the next instance name from <list of instances>. It maps that call onto the specified section of the specified chiptype, which will be shared with all other macro calls having the same instance name.

The following example binds the first and third macro calls within the subtree of RA to sections 0 and 1 of one instance (called 'M1') of chiptype 94550, binds the second and fifth to sections 0 and 1 of another instance (called 'M2') of chiptype 94550, and binds the fourth and sixth calls to sections A0 and B0 of an instance (called 'W2') of chiptype 20021:

```
BIND RA = S0,    S0,    S1,    SA0,   S1,    SB0
          #94550, 94550, 94550, 20021, 94550, 20021
          &M1,    M2,    M1,    W2,    M2,    W2;
```

Using various shortcuts makes the statement easier to write but harder to read:

```
BIND RA = S0, S0, S1, SA0, S1, SB0
          #94550*3, 20021, 94550, 20021
```

```
                    &M1, M2, M1, W2, M2, W2;
```

Obviously, it's easy to violate the rules when writing a BIND statement. A particular instance name must not pair with two different chiptypes; each section name must be valid for the corresponding chiptype; and the chiptype and instance lists must not be too short (though they may be too long).

That example suggests that chiptype 20021 may well have additional sections (A1 and B1, perhaps), not mentioned in this BIND statement. Additional BIND statements can, by referencing common instance names, access sections within the same chips that this BIND statement uses. Thus, one would expect to see another BIND statement referencing sections A1 and B1 of instance W2 of chiptype 20021. Two BIND statements which share instances in this manner must be within the scope of one common PLACE statement, but they can be in different WITH contexts beneath it.

If all the BIND statements referencing sections of a particular instance of a chiptype fail to use up all the sections available in that chiptype, the remainder are available for the PLACE statement to use in laying out locations not involved in any BIND statement.

One additional shortcut exists for specifying chiptypes. Omitting the chiptype but including the repetition factor is equivalent to specifying the default chiptype. Thus, if 94550 is the default chiptype for each of the macro calls referencing it in the previous example, then a simpler way to write the BIND statement would be:

```
     BIND RA = S0, S0, S1, SA0, S1, SB0
               #*3, 20021, *1, 20021
               &M1, M2, M1, W2, M2, W2;
```

BIND statements are *not* compatible with manual layout. If a particular macro call appears in a manual layout statement, don't attempt to bind it. If a particular macro call appears in a BIND statement, don't attempt to lay it out manually. The program will flag any such errors.

7.1.6 The CHIP statement

Provided a macro is being laid out manually, the CHIP statement can override the default chiptype for a terminal macro call by specifying a particular chiptype at the board position which that macro will occupy.

For example, if location X4 is a call on a terminal macro, the following pair of statements places it five rows and two columns beyond the current context and forces it to use chiptype 20023:

```
CHIP 20023 = R5 C2;
X4 = R5 C2;
```

Provided the designer knows which macro calls result from a nonterminal macro, the CHIP statement can specify chiptypes for them as well. If location X6 is a call on a nonterminal macro which expands so as to place terminal G7 at the second column from its starting position, then the following pair of statements forces G7 to use chiptype 19711:

```
CHIP 19711 = R5 C4;
X6 = R5 C2;
```

A single CHIP statement can dictate a number of different positions:

```
CHIP X7 = R5 C2, R7 C1, B2 R1 C1;
```

The CHIP statement may precede or follow an assignment statement.

8 References

1. A manual providing complete information on D, the graphics editor, is kept in a file on the SAIL computer system at Stanford University. D is one piece of a package of programs called SUDS (Stanford University Drawing System), so the manual includes information on other programs (such as one for PC board layout, for example) which don't pertain to SCALD.

Since the manual hasn't been published, you must either sign on to the SAIL computer system to read it, or have someone with access to SAIL print a copy for you. The filename is SUDS.RPH[UP,DOC].

2. These papers describe the philosophy behind SCALD. Because they deal with SCALD I, an earlier version, some details may differ from those you've read about here.

McWilliams, T. and Widdoes, L., *SCALD: Structured Computer-Aided Logic Design.* Lawrence Livermore Laboratory Report UCRL-80950, March 1978.

——, *The SCALD Physical Design Subsystem.* Lawrence Livermore Laboratory Report UCRL-80951, March 1978.

9 Implementation information

9.1 Format of the WDP file

Each WDP file gives the macro expander the equivalent of the information in one drawing produced by the graphics editor D—loosely speaking, the definition of a single macro. The file is organized by lines. If column 1 is not blank, the line should contain either the keyword "END" or the keyword "NUL". "END" signifies the end of a list of elements, and "NUL" signifies a null text string on a line. Blank lines are ignored. The format of the file is as follows:

MName	The name of the macro being defined; identical with title line 1 in a drawing generated by the graphics editor D.	
Selection equation	This is title line 2 in a drawing generated by D.	
PageOf	Each drawing bears a page number in the form "Page x of y"; strictly for documentation. The WDP file expresses this as "x/y"	
FileName	The name of this file; strictly for documentation.	
Section	Project name; strictly for documentation.	
	For each macro called from this drawing, include the following:	
	MName	Name of the macro being called.
		For each body parameter, include the following pair of lines:

Body parameter name	The formal parameter name of a body parameter, such as "LOC" or "SIZE". This corresponds to the name portion of a property name/text pair in D.
Body parameter value	The value of the formal parameter just named. This corresponds to the text portion of a property name/text pair in D.

At the end of the list of body parameter names and values, include:

END	This terminates the list of body parameters.

For each signal parameter, include the following pair of lines:

PinName	The formal parameter name for a signal; this corresponds to the "pinname" in D
Signal	The actual signal name for that parameter; this is the name of the signal connected to the pin in question.

At the end of the list of pinnames and signal names, include:

END	This terminates the list of pinnames

END	This terminates the list of called macros
END	This terminates the file

An example of a WDP file is:

```
ADDER 10181
(SIZE < 5)
1/1
ADDER.DRW
ADDERS
10181                   ;CALL MACRO 10181
LOC                     ;PASS IT LOC PROPERTY WITH VALUE A
A
SIZE                    ;PASS IT A SIZE PROPERTY WITH VALUE 4B
4B
```

```
END
 A                        ;PASS PARAMETER "A" THE SIGNAL "A<0:SIZE-1> /P"
 A<0:SIZE-1> /P
 B                        ;PASS ....
 B<0:SIZE-1> /P
 F
 F<0:SIZE-1> /P
END                       ;END OF MACRO CALL
END                       ;END OF MACRO DEFINITION
END                       ;END OF FILE
```

10 Index

U.S.GPO;1980-689-102/8782