PD. Editor

N84-18909

Final Contract Report

Prepared for

George C. Marshall Space Flight Center
Marshall Space Flight Center, Alabama 35812

Intelligent Editor/Printer Enhancements

Contract No. NAS8-34969

Submitted by

Arizona State University
College of Engineering and Applied Sciences
Computer Science Dept

Tempe, Arizona 85287

Principal Investigators: Marvin C. Woodfill
Professor
David C. Pheanis
Assoc Professor

September 15, 1983

CR R 83005

Final Report

Contract NAS8-34969

ENHANCEMENT OF INTELLIGENT EDITOR/PRINTER

This report was prepaired by Arizona State University under contract NAS8-34969 Enhancement of Intelligent Editor/Printer for Marshall Space Fligh Center of the National Aernautics and Space Administration.

The purpose of this contract was to develop for and furnish to the government for it's unrestricted use all microprocessor support hardware, software and cross assemblers relating to the Motorola 6800 and 6809 processor systems. Furthermore, a printer controller and intelligent CRT (similar to the 6800 versions developed under NAS 8-32230) were to be developed using the 6809, third generation microprocessor.

The following software was delivered and installed on a VAX 11-750 system at MSFC 13 May 1983:

1) The source program for the Motorola 6800/6909 assembler.
2) The source program for the Motorola 68000 assembler.
3) The source program of the Motorola Utility Debug (MUDBUG) package for the Motorola 6800.
4) The source program of the Motorola Utility Debug (MUDBUG) package for the Motorola 6809.
5) A documentation file for the MUDBUG system.
6) The executable image for the Motorola 6800 assembler.
7) The executable image for the Motorola 6809 assembler.
8) The executable image for the Motorola 68000 assembler.
9) The executable image for the ASU Compose (word proc) package.
10) The command procedures developed at ASU to support the VMS VAX system.

Appendix A of this report is a copy of the current User's manual for MUDBUG-2, the latest version of the M6800 Debug Package. Appendix B of this report is a copy of the design specifications for the MC6809 version of the intelligent printer controller card. MSFC is currently constructing a printed circuit card to implement this design. The software necessary to use this card as a controller for a Diablo Hy-Type 2 printer is currently under development and will be supplied to the government after it is checked out on the finished version of this card which is to be supplied to ASU by MSFC. Appendix C of this report is the design specification for a 132 Character by 64 Line intelligent CRT display system using a Motorola 6809 MPU. This version also has four pages of refresh memory. (A 68000 version of this design is under development.) Appendix D of this report is a report concerning a feature that is currently being added to MUDBUG which will greatly increase it ease of use, especially for the in-frequent or casual user. This feature consists of a one-line assembler and disassembler. This feature will be added to the Memory change, memory dump, and register display commands. Its use is documented in this appendix. This capability has proved to be a significant aid because it allows an engineer to debug in assembly language rather than machine language.

Conclusions:

The hardware developed for this contract seems to work well in proto-type form. When the finished version of the printer card is supplied, the software will be verified and that software will be supplied to the government at no additional cost. Furthermore, the addition of the one-line assembler and dis-assembler to the debug package represents a significant improvement in the usefulness of the system to any user, but especially to those not intimately familiar with the machine language of the MPU.

Recommendations:

The impact of the one-line assembler/dis-assembler has been absolutely phenomenal. The amount of improvement that the one-line assembler/dis-assembler makes for a person who is developing and debugging assembly-language programs is so great that it far exceeds our initial expectations. Presently, the one-line assembler/dis-assembler for the 6800 microprocessor is available only in preliminary form and is not fully integrated into MUDBUG, but the package is already tremendously useful.

A one-line assembler/dis-assembler package for the 68000 microprocessor is available commercially, but it is rather crude and inconvenient to use. We are confident that we could develop a much better package for the 68000, and we are also confident that we could develop a good one-line assembler/dis-assembler for the 6809 microprocessor.

It is strongly recommended that the government fund a follow-on project to work on the development of one-line assembler/dis-assembler packages for the 6809 and 68000 microprocessors. The integration of such packages into the debuggers for the 6809 and the 68000 would represent a significant advance of the state of the art, and the result would be immediately useful to anyone who wants to develop and debug programs for those two modern microprocessors.

Final Report

Contract NAS8-34969

ENHANCEMENT OF INTELLIGENT EDITOR/PRINTER

This report was prepaired by Arizona State University under contract NAS8-34969 Enhancement of Intelligent Editor/Printer for Marshall Space Fligh Center of the National Aernautics and Space Administration.

A P P E N D I X

A

The User's Manual for the Latest version of MUDBUG-2 the MC6800 Debug Package.

M U D B U G - 2 / D 2

User's Manual

by

David C. Pheanis

Fourth Edition

August, 1983

I have gone to a great deal of effort to make MUDBUG and this manual as accurate and as usable as possible, but I have no doubt overlooked a few shortcomings. In the interest of quality, therefore, I am offering a reward of $1.00 in U.S. funds to the first finder of each error, whether it is technical, typographical, grammatical, or otherwise. I hope that this offer will convince people that I really do want to hear about my mistakes so I can correct them. Students in my class who use MUDBUG and this manual may elect to receive 20 points of class credit in lieu of a cash reward.

I shall appreciate receiving positive suggestions for improving this manual or the MUDBUG system in any way. Some of the features that are already implemented in MUDBUG evolved from discussions with M6800 users, and future suggestions for improvements will be more than welcome.

DCP

First Printing: August, 1983

Table of Contents

Chapter 1

Introduction

MUDBUG is a general-purpose Monitor-Utility-DeBUG software package for a Motorola M6800 microprocessor. The first version of MUDBUG, which was known as MUDBUG-1, was designed and developed at Arizona State University in 1975 by Dr. Marvin C. Woodfill and Ms. Mary L. Dryden to replace the Motorola-supplied MAID and MIKBUG systems, both of which were deficient in several respects. In the summer of 1976 Mr. Don E. Smoker converted MUDBUG from its original hand-written machine-language form into a machine-readable assembly-language file, and then in 1977 the MUDBUG system was almost completely redesigned and rewritten by Dr. David C. Pheanis with the help of some background information from Dr. Woodfill.

Until 1982 MUDBUG existed as a 1-K program capable of running in a 1-K EPROM such as a 2708. Falling memory prices and rising chip capacities eventually modified the marketplace to the extent that 2-K EPROMs such as the 2716 were actually cheaper than 1-K 2708 EPROMs. In 1982, therefore, Dr. Pheanis started using a 2-K 2716 EPROM and modified MUDBUG by adding several enhancements that had not been possible when the program had been restricted to a 1-K EPROM.

Anyone who has any suggestions, comments, or questions regarding MUDBUG should contact:

Dr. David C. Pheanis
6822 S. Butte Avenue
Tempe, Arizona 85283
(602) 839-5229

Also, in the unfortunate event that anyone ever detects any error in MUDBUG, the author would appreciate receiving a detailed report of the suspected error and its symptoms. Finally, the author will appreciate receiving any well-conceived ideas for useful features that could be included in the next release of MUDBUG.

In its present form MUDBUG requires 2-K (i.e., 2,048) words of ROM program space in locations $F800 through $FFFF. (Note: A leading "$" is a standard M6800 notation that indicates a hexadecimal number. A decimal number is normally indicated by the absence of a leading "$", and an octal number is indicated by a leading zero. A binary number is indicated by a leading "%" character.) In the rest of this document we'll use the label ROM to refer symbolically to the first location of the MUDBUG ROM.

Besides requiring a 2-K block of ROM, MUDBUG also requires a 128-word block of RAM for its stack and internal variables. This block of RAM can

start at any convenient location, and its starting location is indicated by the label RAM, which is defined by an equate near the beginning of MUDBUG.

Locations RAM+$40 through RAM+$7F of the MUDBUG RAM area are currently reserved for use by the user's programs, and, in particular, the user's default stack grows downward from location RAM+$7F toward location RAM+$40.

Besides using ROM and RAM, MUDBUG also needs an ACIA for RS-232 communications with the user's terminal. The memory location of the ACIA's control register is indicated by the label ACONT, and the location of the ACIA's data register is indicated by the label ADATA. Both of these labels are defined by equates near the beginning of MUDBUG, so they can be modified easily to fit various hardware configurations.

Chapter 2

Interrupt Vectors

MUDBUG is designed such that the last eight locations of its ROM space function as the interrupt vectors for the microprocessor system, and the interrupt vectors are set up in the MUDBUG ROM to make the M6800 interrupts function as follows:

Hardware Interrupt	(IRQ):	Branch indirectly through locations RAM and RAM+1. (Note: The label RAM is used throughout this document to designate the first location of the MUDBUG RAM area.)
Software Interrupt	(SWI):	Branch indirectly through locations RAM+2 and RAM+3.
Nonmaskable Interrupt	(NMI):	Branch indirectly through locations RAM+4 and RAM+5.
Restart Interrupt	(RSI):	Initialize or restart the MUDBUG system as if a power-up condition had occurred.

Locations RAM and RAM+1, RAM+2 and RAM+3, and RAM+4 and RAM+5 are automatically set up at MUDBUG-initialization time so that a hardware interrupt (IRQ), a software interrupt (SWI), or a nonmaskable interrupt (NMI) invokes a routine that prints the contents of all of the registers and returns control to the top of the MUDBUG command-decoding routine.

For purposes of flexibility, the indirect interrupt-vector pointer values in the first six locations of the MUDBUG RAM can be modified quite easily via the MUDBUG memory-change (i.e., "C") command to allow the user to capture the various interrupts and to service them via his or her own specialized interrupt-handling routines if necessary. MUDBUG's memory-change command is explained in detail later.

Users who capture the interrupts for their own interrupt-handling routines should carefully note the fact that the values of the XR and the CC register are not preserved in the registers upon entry to a user-defined interrupt handler because MUDBUG loads the XR with the appropriate indirect interrupt vector to route control to the user-defined interrupt-handling routine. Loading the XR, of course, modifies some of the bits in the CC register. The original values of all of the registers (including the XR and the CC register) are always preserved on the stack, so they are readily recoverable by any interrupt-handling routine that ever needs them.

Chapter 3

MUDBUG Utilities

Whenever control comes to the top of MUDBUG (i.e., following system initialization, upon the completion of a MUDBUG command, at the termination of a user's program, etc.), the system outputs a prompt (usually a ">" but sometimes a "?") on a new line and waits for an input character. The user can then activate a MUDBUG utility routine by typing a command mnemonic followed by the appropriate parameter(s) (if any) for the selected command. MUDBUG command mnemonics currently consist of only one or two characters each, so the user can type them quickly and easily.

When MUDBUG receives a valid command mnemonic in response to a prompt, it outputs a single blank space to the terminal to show that it recognizes the command. Then it reads the parameter(s) (if any) that are required for the given command. If MUDBUG receives an invalid command mnemonic in response to a prompt, it outputs a backslash ("\") and a bell to reject the bad input, and then the system prompts the user again for a good MUDBUG command mnemonic.

Individual MUDBUG commands require from as few as zero to as many as four parameters, and the parameters are known as START, STOP, KEY, and MASK, respectively. The values of the parameters are input following a command mnemonic as hex-number inputs, which are described below.

A hex-number input may be signed or unsigned, and it is terminated by a comma, a blank space, a tab, an asterisk, a period, a solidus ("/"), or a carriage return. We'll frequently use the notation "<CR>" in this document when we wish to emphasize the presence of a carriage return. At other times, we'll simply assume the presence of a carriage return at the end of an input line.

A solidus termination character has a rather special global meaning in that it immediately aborts the MUDBUG command that is currently being typed. A solidus is therefore useful as an abort character, and a user can easily abort any MUDBUG command line by simply typing a solidus before the input line has been completed.

A carriage-return terminator also has a rather special global meaning in that it always terminates the command that is being typed. If the user types a carriage return before all of the command's parameter(s) have been typed, MUDBUG simply uses default values for the remaining parameter(s).

The period, comma, space, tab, and asterisk terminators don't have any special meanings except as noted for certain individual MUDBUG commands. Most users use the comma or space terminator to terminate each parameter except the last parameter for a command, and they typically use a carriage return to terminate a command's last parameter. For example, a typical

command might be typed as "F 100, 200, 23, FF<CR>" where the <CR> terminator is used to terminate the command's last parameter. Similarly, a user might type "F 100 200 23 FF<CR>" to perform the same operation.

A hex-number input to MUDBUG is always typed without a leading "$" because a leading "$" would be redundant in view of the fact that MUDBUG automatically assumes that all input numbers are hexadecimal by default.

MUDBUG allows the user to type leading blank space(s) and/or leading tab(s) with a hex-number input because many users like to insert white space at the beginning of an input number for improved readability. No leading white space is required, of course, but blanks and tabs are allowed until the first hextet of the input number has been typed. Since MUDBUG allows the user to type leading blanks and/or leading tabs, a user can't use a blank or a tab to terminate an empty (i.e., missing) input number. A user who wants to omit an input number entirely must type one of the other termination characters to indicate the missing number. For example, a user could type "I 100, , 0<CR>" to obtain a default value for the second input parameter, but a user could not use a blank or a tab to terminate the missing second parameter.

Since a blank space or a tab can't be used to terminate a missing input number, we refer to the blank space and the tab as soft termination characters. The other termination characters (asterisk, comma, period, solidus ("\"), and <CR>) are all known as hard termination characters because they can be used to terminate any input number including a missing input number.

A leading minus sign negates a hex-number input value, and a leading plus sign has no effect on the value of a hex-number input. Two minus signs (or any even number of minus signs for that matter) cancel each other, so a user can conveniently change his or her mind after typing a minus sign by mistake. Any odd number of minus signs, of course, results in the negation of the hex-number input value. Leading sign characters are permitted until the first hextet of the input number has been typed, and after that time sign characters are rejected (see below).

A user who makes a mistake when typing an input number can quickly correct the error by simply typing enough hextets to shift the mistake out of the left end of the number. For example, "1241234" is equivalent to "1234" as a four-hextet input number, and "-1202" is equivalent to "-02" or just "-2" as a two-hextet input number. Similarly, "-1000F" is equivalent to "-F" as an input number.

A user can also correct a typing error in a hex-number input by typing a backslash ("\") before the number's termination character has been typed. A backslash deletes all previous input characters for a number, so a user can easily restart an input number from its first character. For example, "---127\-3D75\-\3C51" is equivalent to "3C51" as a hex-number input, and "5ADDD\ - 5ADD" is equivalent to "-5ADD" as a hex-number input. Notice that the backslash correction feature can be used as many times as desired

in a single input number, and also notice that a backslash deletes all previous input characters for the number, including sign characters.

If MUDBUG receives an invalid input character when it thinks it should be reading a hexadecimal number, the system outputs a bell and a question mark to the terminal to alert the operator to the fact that the bad input character is being rejected. The invalid input character is effectively ignored, and MUDBUG reads the next input character as the next character of the hexadecimal input number. Only the following ASCII characters are valid for hex-number inputs:

blank tab + - 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f \ . , * / <CR>

The first two characters in this list (blank and tab) are valid as leading characters before the first hextet of the input number has been typed, and they are also valid as termination characters after the first hextet of the input number has been typed. The next two characters in the list (+ and -) are valid only before the first hextet of the input number has been typed, and they are rejected as invalid inputs if they are ever typed anywhere other than at the beginning of a number. A backslash ("\") can always be used to restart an input number, so a user can go back and type a forgotten sign character by first typing a backslash. Notice that lower-case alphabetic hextets (a-f) are accepted as being equivalent to the corresponding upper-case alphabetic hextets (A-F), so users don't need to worry about pressing or not pressing the shift key when talking to MUDBUG.

MUDBUG always remembers the values of the START, STOP, KEY, and MASK parameters from one command to the next, so a user can conveniently specify the previously-existing value of a parameter by typing only a hard termination character (in this case, a comma, period, asterisk, or <CR>) for that parameter. For example, if the last value that was used for the START parameter was $89AB, then simply typing "C." is equivalent to typing "C 89AB." This feature reduces the number of characters that the user must type, so it helps prevent mistakes that are caused by simple typing errors. Additionally, it alleviates frustration because the user doesn't need to retype the same input number over and over.

Recall that a user can specify default values for the remaining parameter(s) on a command line by typing a carriage return to terminate the command. For example, a user might want to type new values for the first two parameters and then use a carriage return to specify the default values for the command's remaining parameters.

MUDBUG maintains a complete set of pseudo-register values, and the user can easily inspect and/or change these pseudo-register values via the MUDBUG register-change commands, which are explained in detail later. The user can also inspect and/or change individual memory locations, and one can initialize memory in blocks, load object modules into memory, compare object modules to memory, write object modules from memory to a tape, dump memory to the terminal in a readable format, search memory to find any specified values, and/or execute any program. MUDBUG conveniently performs self-relative addressing calculations for the user, and MUDBUG also does

hexadecimal addition and subtraction for the user. Interactive debugging is quick and easy because the user can tell MUDBUG to trap control at any specified point in his or her program. Finally, if the hardware of the user's system supports step-mode operations, the user can tell MUDBUG to execute a single instruction or a specified number of instructions.

The table on the following page, which is intended to be used as a handy reference sheet, briefly summarizes all of the MUDBUG commands, and the pages that follow the command-summary table describe the individual MUDBUG commands in sufficient detail to serve as a programmer's manual. The table provides a brief functional description of each command, and the table also includes a number with each command to tell how many parameters the command requires.

Upper-case command mnemonics (e.g., A, B, and C) are used throughout this document in all descriptions and examples, but MUDBUG also accepts lower-case command mnemonics (e.g., a, b, or c) as being equivalent to the corresponding upper-case command mnemonics. Therefore, users can always type MUDBUG commands with either upper-case or lower-case alphabetic characters, whichever happens to be more convenient.

3.1. MUDBUG Command Summary.

Parameters: START, STOP, KEY, and MASK.

Pseudo Registers: AR, BR, XR, PR, SP, and CC (H I N Z V C).

```
--------  Command Mnemonic
|
|   ---   Number of Parameters
|   |
V   V     Description of Command's Function

*   0     An asterisk introduces a comment line.
A   0     Print the AR value in hex, and accept a new hex AR value.
B   0     Print the BR value in hex, and accept a new hex BR value.
C   1     Change (after printing) the contents of memory location START.
E   0     Establish a new CC value after printing the existing CC value.
F   4     Find KEY under the bits of MASK in locations START through STOP.
G   1     Go to location START to execute the user's program.
H   0     Halt return.  Go to the location that is addressed by the PR.
I   3     Initialize locations START through STOP to the value KEY.
K   2     Kalculator.  Set START <-- START + STOP, and print START.
L   1     Load (.) or compare (,) an object module to memory; Disp = START.
M   3     Memory dump of locations START through STOP to the terminal.
N   1   & N step.  Execute START instructions beginning at location PR.
O   1   & One step.  Execute one instruction at START.  Default START = PR.
PE  1     Peek at memory.  Like C except that it never writes to memory.
PO  1     Poke into memory.  Like C except that it never reads from memory.
PR  0     Print the PR value in hex, and accept a new hex PR value.
Q   0     Query the registers.  Print AR, BR, XR, PR, SP, HINZVC.
R   1     Relative address:  Print and/or set the destination of a branch.
S   0     Print the SP value in hex, and accept a new hex SP value.
T   4   # Trap.  Go to START; trap control when it reaches STOP KEY times.
V   4   & Verify ROM Program.  Like "T" except that it can be used in ROM.
W   2     Write locations START through STOP to tape in object format.
X   0     Print the XR value in hex, and accept a new hex XR value.
Z   0     Zero the AR, BR, XR, PR, and CC, and initialize the SP.
```

& Commands that are marked with an "&" are available only if the system on which MUDBUG is running has the necessary hardware to support step-mode operations.

Only the first two parameters of the "T" command are available if the system on which MUDBUG is running doesn't have the necessary hardware to support step-mode operations.

3.2. MUDBUG Command Descriptions.

This section contains user-oriented descriptions of all of the MUDBUG commands. The descriptions are alphabetized by the command mnemonics for the convenience of the reader.

"*" Command; No Parameters; Comment.

The "*" command, which has no parameters, is used to introduce a comment line into the transcript of the user's session at the terminal. If the user inputs an asterisk in response to the prompt (">" or "?") for a MUDBUG command mnemonic, MUDBUG reads and echoes (but otherwise ignores) all subsequent input characters (including solidus characters) until a carriage return is input. Users can therefore annotate and document the listings of their MUDBUG transactions by typing any desired comments on lines that begin with asterisks. Also, the comment facility effectively turns the user's MUDBUG transcript into a handy scratchpad for recording ideas and problems as they occur (before they are forgotten).

The use of an asterisk to introduce a comment line in MUDBUG is compatible with the definition of a comment line in the M6800 assembly language, so MUDBUG comment lines should seem quite natural to most users.

"A" Command; No Parameters; Change AR.

The "A" command, which has no parameters, displays the current hex value of the pseudo AR, and then it accepts a new hex value for the AR. If a user types a hard termination character (/ * , . or <CR>) alone without a number for a new AR value, the pseudo AR retains its present value.

"B" Command; No Parameters; Change BR.

The "B" command, which has no parameters, displays the current hex value of the pseudo BR, and then it accepts a new hex value for the BR. If a user types a hard termination character (/ * , . or <CR>) alone without a number for a new BR value, the pseudo BR retains its present value.

"C" Command; 1 Parameter; Change Memory.

The "C" command, which has one parameter, displays the address and the current hex value of the contents of memory location START, and then it accepts a new hex value to be put into that location.

If the new hexadecimal value for location START is terminated by a carriage return, MUDBUG automatically continues the "C" command by setting START <-- START + 1 and performing the "C" command's function for the next location in memory. If an asterisk, blank, or tab termination character is used to terminate the new hexadecimal value for location START, MUDBUG sets START <-- START - 1 and automatically performs the "C" command with the preceding memory location. If a comma terminator is used to terminate the new hex value for location START, MUDBUG performs the "C" command's function again with the same location. If the new hex value is terminated by a period, the "C" command returns control to the top of MUDBUG after it installs the new value into memory. Finally, if a solidus ("/") terminator is used, the "C" command is aborted and control returns at once to the top of MUDBUG with no change to the contents of location START.

If a user types a hard termination character (/ * , . or <CR>) alone without a number for a new memory value, the contents of the memory location remain unchanged, but the function of the termination character regarding the continuation or termination of the "C" command is still effective. Users can therefore examine several consecutive memory locations rather conveniently by using carriage-return or asterisk terminators, and they need to type new values only when new values are actually desired.

The "C" command reads back and verfies the value that it stores into memory, so any attempt to use the "C" command to alter a value in ROM or in nonexistent memory is automatically aborted with a backslash and a bell. This feature is included so the user won't think a value is changed when it is, in fact, not changed.

The "C" command is not intended for use with read-only or write-only registers such as the status and control registers of an ACIA. A user who wishes to examine a read-only register should use the "PE" (peek) command, and a user who wishes to put a value into a write-only register should use the "PO" (poke) command.

"E" Command; No Parameters; Establish a CC Value.

The "E" command, which has no parameters, is used to establish a value for the pseudo CC (condition-code) register. The "E" command displays the current eight-bit hexadecimal value of the condition-code register, and then it accepts a new hexadecimal value for the pseudo CC register.

Notice carefully that the pseudo CC register contains eight bits. The top (i.e., leftmost) two bits are essentially meaningless and will always be set following the execution of any portion of any user's program because the M6800 hardware always sets these two condition-code bits. We refer to the other six condition-code bits (from left to right) as H, I, N, Z, V, and C, and they are generally quite important for purposes of debugging programs. The condition codes are interpreted as follows:

H = half carry	I = interrupt mask	N = negative
Z = zero	V = overflow	C = carry/borrow

If a user types a hard termination character (/ * , . or <CR>) alone without a new value for the CC register, the pseudo CC register retains its present value.

"F" Command; 4 Parameters; Find Memory Values.

The "F" command requires all four parameters, and it is used for finding specified kinds of values in a specified area of memory.

If the MASK parameter is nonzero, the "F" command searches memory locations START through STOP to find any words that contain the value KEY when considering only the bit(s) that are set in MASK. MUDBUG displays both the memory address and the contents of each word that is a match. When the MASK parameter is nonzero, therefore, the MUDBUG "F" command finds and lists all words from location START through location STOP such that [(WORD) .and. MASK] = [KEY .and. MASK]. To find words that exactly equal KEY, of course, a user should specify a MASK value of $FF (i.e., -1).

If the MASK parameter is zero, the "F" command searches memory locations START through STOP for any words that do not contain exactly the value KEY. MUDBUG displays both the memory address and the contents of each such word that is found.

"G" Command; 1 Parameter; Go to START.

The "G" command, which has one parameter, is used to transfer control to the user's program. The "G" command sets the pseudo PR equal to START. Then it loads the M6800 hardware registers from the software-defined pseudo registers and transfers control to location START to execute the user's program.

MUDBUG temporarily treats the "G" command as an illegal command when the pseudo stack pointer has been set to address ROM or nonexistent memory. This feature protects the user from inadvertently transferring control to his or her program with an invalid value in the stack pointer. Please refer to the discussion of the "S" command for details.

"H" Command; No Parameters; Halt Return.

The "H" command, which is known as the halt-return command, has no parameters. It loads the M6800 hardware registers from the pseudo registers, and then it transfers control to the location that is designated by the pseudo PR.

The pseudo PR normally points to the memory location immediately following the location from which the last halt (i.e., SWI) was executed, but the user can, of course, change the pseudo PR by using the "PR" command, which is described elsewhere in this chapter.

The "H" command is convenient for executing a user's program in segments where the end of each segment is marked by the occurrence of an SWI instruction. The "H" command is also useful for returning control to a program that has been interrupted by a nonmaskable interrupt (NMI). A user who suspects that his or her program is lost in an infinite loop can generate a nonmaskable interrupt (usually by pressing a button marked NMI), and this action normally halts the program and dumps the registers to the terminal just as if an SWI instruction had been executed. If the user then decides to continue the execution of his or her program, an "H" command conveniently and automatically returns control to the precise point of the interruption just as if no interruption had occurred.

The NMI interrupt and MUDBUG's "H" command can be used together as described above to monitor the progress of a program that has a very long execution time.

MUDBUG temporarily treats the "H" command as an illegal command when the pseudo stack pointer has been set to address ROM or nonexistent memory. This feature protects the user from inadvertently transferring control to his or her program with an invalid value in the stack pointer. Please refer to the discussion of the "S" command for details.

"I" Command; 3 Parameters; Initialize Memory.

The "I" command requires three parameters, and it initializes memory locations START through STOP (inclusive) to contain the value KEY. MUDBUG automatically aborts any attempt to use the "I" command to alter any ROM or nonexistent memory locations, and the system notifies the user of the error by outputting a backslash and a bell to the terminal. All RAM locations that were initialized before the ROM or nonexistent memory location was encountered remain initialized just as requested, so part of the command's function may be performed successfully even in the event of an error.

The "I" command is useful for backgrounding memory areas with desired values, and it can also be used to initialize tables and data areas.

"K" Command; 2 Parameters; "Kalculator" Utility.

The "K" command, which requires two parameters, is convenient for performing 16-bit two's-complement hexadecimal additions and subtractions. The "K" command sets START <-- START + STOP, and then it prints the updated 16-bit value of START in hexadecimal. Notice that MUDBUG puts the sum into the START parameter to facilitate subtotaling operations via several consecutive executions of the "K" command. For example, the following "K" commands could be used to compute the sum of $2A12, $21, -$B4C, $591A, and -$1F2E:

```
K 2A12, 21.
K , -B4C.
K , 591A.
K , -1F2E.
```

The "K" command is also useful for finding the 16-bit two's complement of a hexadecimal number. To find the two's complement of a value, one can simply use the "K" command to subtract the value from zero. For example, to find the two's complement of $B7C, a user could type either "K 0, -B7C" or "K -B7C, 0." Similarly, to find the two's-complement representation of -$1A8, one could type "K -1A8, 0" or "K 0, -1A8."

In another application, the "K" command can be used to find the 16-bit one's complement of a hexadecimal number. The one's complement of a number is just its two's complement minus one, so subtracting a number from minus one yields the one's complement of the number. For example, a user could find the one's complement of $DA9 by typing "K -1, -DA9" or "K -DA9, -1." Similarly, one could type "K -1, -B5C" or "K -B5C, -1" to find the one's-complement representation of -$B5C.

"L" Command; 1 Parameter; Load/Compare.

The "L" command, which requires one parameter, loads or compares an M6800 object tape to memory. MUDBUG takes the value of the START parameter as a 16-bit positive or negative signed relocation displacement, and the parameter's termination character specifies the function that is to be performed. A carriage return or a period termination character tells MUDBUG to load the values from the object tape into memory, and a comma, space, tab, or asterisk termination character tells MUDBUG to compare the object tape to memory.

The object tape that is loaded or compared to memory is offset from its normal memory locations by a displacement that is equal to the START parameter. For example, if START = -$1000, an object module with an origin of $4000 would be loaded into (or compared to) memory starting at location $3000. Similarly, if a user specifies START = +$2000, an object module with an origin of $5000 would be loaded into (or compared to) memory starting at location $7000. For normal lab work, most users employ the "L" command with START = 0 to load or compare programs with no displacement,

but nonzero displacement values are frequently necessary when MUDBUG is used in special applications such as M6800-based PROM programmers.

When MUDBUG performs a compare function, the MUDBUG Load/Compare routine prints three hex values for each object-tape word that fails to match the corresponding word in memory. The three values that are printed are as follows:

Memory Address Memory Value Tape Value

When MUDBUG performs a load function, the loader generates a compare printout as above for any word that can't be loaded, so attempts to load into ROM or nonexistent memory are flagged for the user.

If an invalid object tape is input to the loader or if a bad read occurs or if a checksum error is detected, the Load/Compare routine aborts execution with a backslash and a bell.

"M" Command; 3 Parameters; Memory Dump.

The "M" command requires three parameters known as START, MSTOP, and SCREEN. If MSTOP >= START and MSTOP is not typed with a leading plus sign, the "M" command nominally dumps locations START through MSTOP in both hexadecimal and ASCII format to the user's terminal. If MSTOP < START on the other hand, (or if MSTOP is typed with a leading plus sign), the "M" command uses MSTOP as a count and nominally dumps MSTOP locations beginning with location START. The third parameter for the "M" command, which is known as the SCREEN parameter, specifies the maximum number of lines that MUDBUG will dump before pausing. The SCREEN parameter allows the user to control the dump interactively, so the user can prevent information from scrolling off the top of a CRT screen too quickly.

MUDBUG prints the memory dump with 16 values per line, and the dump is formatted for ease of readability. Each line of the dump begins with the hexadecimal memory address of the first value on the line. Then the line contains 16 hexadecimal values corresponding to the values of the 16 memory locations that are being dumped. At the end of each line MUDBUG prints 16 ASCII characters corresponding to the ASCII character codes (if any) that reside in the 16 memory locations that are being dumped. If a location doesn't contain a printable ASCII character, MUDBUG prints an underscore for that location.

The memory dump actually starts with the location whose address is FLOOR[START/16]*16, so the hexadecimal memory address of the first word of the dump always ends with zero. The default value for START is the current value of the MUDBUG START parameter.

The "M" command uses its own private MSTOP parameter instead of using MUDBUG's global STOP parameter, and the MSTOP parameter tells MUDBUG where to stop dumping. If MSTOP >= START and the MSTOP parameter is not typed

with a leading plus sign, the memory dump ends with the location whose address is FLOOR[STOP/16]*16 + 15. Notice that the memory dump always starts with the line that contains location START, and the memory dump ends with the line that contains location MSTOP.

If MSTOP < START, MUDBUG conveniently interprets the MSTOP value as a count of the number of locations to dump. For example, if START = $2000 and MSTOP = $100, MUDBUG dumps $100 locations starting at location $2000. Since MUDBUG always prints values for 16 memory locations on each line of the dump, MUDBUG sometimes dumps a few more locations than the actual number that were specified by the MSTOP parameter. For example, if START = $2005 and MSTOP = $3, MUDBUG actually dumps locations $2000 through $200F.

If the user types a plus sign as a leading sign character with the MSTOP parameter, MUDBUG interprets the MSTOP value as a count of the number of locations to dump even if MSTOP >= START. This feature allows a user to use the count option even when the starting address for the memory dump is a small number. For example, a user could type "M 10, +20" to dump $20 locations beginning at location $0010.

The MSTOP parameter doesn't have a default value. If the user omits the MSTOP parameter, MUDBUG potentially dumps forever and never stops the dump at any particular address. Instead, MUDBUG stops the dump only when the user interactively chooses to stop it. The user can interactively tell MUDBUG to stop the dump by typing a termination character during the pause at the end of a screen. This feature is extremely convenient because it allows the user to dump interactively and to terminate the dump only when desired.

The SCREEN parameter tells MUDBUG how many lines it should dump to the terminal before pausing to let the user control the dump interactively. The default value for the SCREEN parameter is $10 = 16, so MUDBUG normally dumps 16 lines containing $100 locations per screen. MUDBUG accepts SCREEN parameters in the range from $00 through $FF (255), and MUDBUG interprets any value larger than $FF by taking its value modulo 256. Notice that the SCREEN parameter, like all MUDBUG parameters, is typed as a hexadecimal number, not as a decimal number.

Zero is a special value for the SCREEN parameter, and it means that MUDBUG should never pause. Zero is therefore equivalent to an infinite value for the SCREEN parameter, and a zero SCREEN parameter is useful for a person who is using a hardcopy terminal. Notice that a user can put MUDBUG into an infinite dump by specifying zero for a SCREEN parameter and also omitting the MSTOP parameter. This somewhat dubious feature allows a user to make MUDBUG run continuously over a weekend to verify that a particular microcomputer can run that long without any glitches.

When MUDBUG reaches the end of a screen of the memory dump, the system pauses with the cursor at the beginning of the next line on the terminal. MUDBUG waits at this point (forever if necessary) until the user types an input character to tell MUDBUG what to do next, so the user can control the

dump interactively. The choices are similar to the choices that a user has with the memory-change command.

If the user types a carriage return (<CR>), MUDBUG displays the next screen of data. The <CR> therefore allows a user to scroll through memory until all relevant values have been displayed. MUDBUG considers memory to be circular, so location $0000 logically follows location $FFFF in memory.

If the user types a comma, MUDBUG re-displays the current screen of data. A comma therefore allows a user to re-examine a particular area of memory as often as desired. A user might want to use commas to keep examining some locations that correspond to the registers of an input device that is receiving inputs from an external source.

If the user types an asterisk, a blank space, or a tab, MUDBUG displays the previous screen of data. The previous screen of data is the screen of data that corresponds to the memory locations that immediately precede the locations of the current screen in memory. The asterisk, blank, and tab allow a user to scroll back through memory until all relevant values have been displayed. A user can use a <CR> after one screen and an asterisk after the next screen to bounce back and forth between two screens of data.

If the user types a period, a solidus, or any other character that hasn't been mentioned above, MUDBUG terminates the dump and returns control to the top of MUDBUG to wait for the next command.

The interactive dumping feature is much like the memory-change command, and it is extremely convenient to use. For example, a user can interactively dump memory starting at location $2000 by simply typing "M 2000<CR>" to tell MUDBUG to start the dump in the interactive mode. On the other hand, a user who wants to dump $80 locations beginning at location $2000 can perform the dump non-interactively by typing either "M 2000, 207F<CR>" or "M 2000, 80<CR>" to tell MUDBUG to dump the desired area of memory.

"N" Command; 1 Parameter; N-Step Command.

The "N" command, which requires one parameter, is useful for debugging an otherwise intractable section of a program. The "N" command is known as the N-step command, and it tells MUDBUG to execute the next N instructions of the user's program where the value of N is defined by the "N" command's parameter. For example, a user could type "N 5" to tell MUDBUG to execute the next five instructions of his or her program. MUDBUG provides a register dump after executing the next N instructions of the user's program, and then control goes back to the top of MUDBUG to wait for another MUDBUG command from the user.

Suppose that a user types "N 5" as suggested above to execute the next five instructions of his or her program. Suppose further that the user

then wants to execute five more instructions. In this case, the user could simply type "N<CR>" because MUDBUG remembers that the previously-specified parameter value was five. The default value for the "N" command's parameter is always taken from the previously-defined START parameter.

The N-step command tells MUDBUG to execute the next N instructions of the user's program, and the next instruction to be executed in a program is always indicated by the PR: An N-step command therefore causes MUDBUG to execute the next N instructions starting at the location that is indicated by the pseudo PR. Clearly, the pseudo PR must point to the desired program segment before the N-step command can be used. Therefore, the N-step command should normally be used after the user has executed part of a program and has trapped control at some point of interest in the program. Control can be trapped at some desired point in the user's program by including a software-interrupt instruction (SWI) at that point or by using MUDBUG's "T" command. There are other ways to trap control at some specified point, but MUDBUG's "T" command, which is explained elsewhere in this chapter, provides the most convenient method.

A user can also use the "PR" command, which is explained elsewhere in this chapter, to set the pseudo PR to a desired location before using the "N" command. However, this approach is error prone because it allows the user to start somewhere in the middle of a program without executing the first part of the program. The first part of the program may be necessary to set up the conditions that allow the next part of the program to execute properly.

Most users employ the "N" command as follows. First they use the "T" command to trap control at the beginning of a suspected problem area, and then they use the "N" command to execute a few instructions at a time in the problem area to examine the problem in detail.

The user should always remember that the "N" command's parameter, like all of MUDBUG's parameters, is interpreted as a hexadecimal number. Typing "N 10," therefore, tells MUDBUG to execute the next $10 (i.e., 16) instructions of the user's program. For purposes of debugging, most users tend to execute only two to six instructions with each use of the "N" command, and there is no difference between hex and decimal numbers in this range of values.

Although most people use the "N" command with small parameter values, the "N" command accepts any 16-bit unsigned number for its parameter. A parameter value of zero is interpreted to mean 65,536, so a user can execute a maximum of 65,536 instructions with a single use of the "N" command.

Some users employ the "N" command with large parameter values to determine how many instructions are executed in a particular program or subroutine. In this application the "N" command is useful as an aid for making timing measurements (in terms of the number of instructions executed).

Using the "N" command puts MUDBUG into step mode. When MUDBUG is in step mode, the system changes its prompt from the normal ">" to a "?" to let the user know that MUDBUG is in step mode. Step mode is just like normal mode with one important exception. When MUDBUG is in step mode, the user can simply type a carriage return as a MUDBUG command to tell MUDBUG to execute the next instruction of the user's program. Thus a user can conveniently step through a program one instruction at a time by simply typing a carriage return at each step. In this case MUDBUG provides a register dump after each instruction is executed.

When MUDBUG is not in step mode, a carriage return alone is treated as a do-nothing command. A user can therefore type a carriage return as a MUDBUG command in normal mode to advance the cursor to a new line.

MUDBUG automatically returns from step mode to normal mode whenever the user types any non-debugging command (except a comment command) or when the user types an invalid command. The debugging commands are the "N" command, the "O" command, the "T" command, and the "V" command. In other words, any command that tells MUDBUG to execute a portion (but not all) of the user's program is a debugging command. Debugging commands normally put MUDBUG into step mode, and other commands (except the comment command) take MUDBUG out of step mode. Step mode is designed to provide a convenient single-step mechanism for the user who is debugging a program.

MUDBUG temporarily treats the "N" command as an illegal command when the pseudo stack pointer has been set to address ROM or nonexistent memory. This feature protects the user from inadvertently transferring control to his or her program with an invalid value in the stack pointer. Please refer to the discussion of the "S" command for details.

The "N" command is available only on systems that have the necessary hardware to support step-mode operations. If the system on which MUDBUG is running doesn't have the necessary hardware to support step-mode operations, MUDBUG treats the "N" command as an invalid command.

"O" Command; 1 Parameter; One-Step Command.

The "O" command, which requires one parameter, tells MUDBUG to execute one instruction in the user's program. The "O" command's parameter tells MUDBUG the address of the instruction that is to be executed. MUDBUG executes the specified instruction and provides a register dump after the instruction has been executed. Then MUDBUG automatically goes into step mode, and control returns to the top of MUDBUG to wait for the user's next command.

The default value for the One-step command's parameter is not taken from the previously-specified START parameter. Instead, the default value for the One-step command's parameter is taken from the pseudo PR, and the START parameter is neither used nor affected by an "O" command. By using the pseudo PR as a default value for the "O" command's parameter, MUDBUG

conveniently allows the user to execute the next instruction of his or her program by simply typing "O<CR>" with no parameter value.

The One-step command is a powerful debugging aid. If all else fails, a desperate programmer can always use the "O" command to single step through a small program segment that is causing problems that temporarily seem to be insoluble.

Most users employ the "O" command as follows. First they use the "T" command to trap control at the beginning of an intractable problem area, and then they use the "O" command with its default parameter value to execute one instruction at a time in the problem area until they've discovered the exact cause of the problem.

A successful use of the "O" command always puts MUDBUG into step mode. MUDBUG changes its prompt from the normal ">" to a "?" to let the user know that the system is in step mode, and a carriage return as a response to a "?" prompt tells MUDBUG to execute the next instruction in the user's program. Thus a user doesn't even need to type an "O" to request another "O" command; a carriage return alone is sufficient when the system is in step mode. MUDBUG provides a register dump after executing an instruction in step mode, and the user can conveniently step through a program one instruction at a time by simply typing a carriage return at each step.

When MUDBUG is not in step mode, a carriage return alone is treated as a do-nothing command. A user can therefore type a carriage return as a MUDBUG command in normal mode to advance the cursor to a new line.

MUDBUG automatically returns from step mode to normal mode whenever the user types any non-debugging command (except a comment command) or when the user types an invalid command. The debugging commands are the "N" command, the "O" command, the "T" command, and the "V" command, so any command that tells MUDBUG to execute a portion (but not all) of the user's program is a debugging command. Any debugging command that is executed successfully puts MUDBUG into step mode, and other commands (except the comment command) take MUDBUG out of step mode. Step mode is designed to provide a convenient single-step mechanism for the user who is debugging a program.

MUDBUG temporarily treats the "O" command as an illegal command when the pseudo stack pointer has been set to address ROM or nonexistent memory. This feature protects the user from inadvertently transferring control to his or her program with an invalid value in the stack pointer. Please refer to the discussion of the "S" command for details.

The "O" command is available only on systems that have the necessary hardware to support step-mode operations. If the system on which MUDBUG is running doesn't have the necessary hardware to support step-mode operations, MUDBUG treats the "O" command as an invalid command.

"PE" Command; 1 Parameter; Peek at Memory.

The "PE" command allows a user to peek at a memory location with the assurance that MUDBUG will not write anything to the memory location. The "PE" command, which has one parameter, displays the address and the contents of memory location START, and then it waits for the user to type a termination character. Actually, the "PE" command waits after displaying the address and value of location START for the user to type a hex input number, but the command ignores the value of the input number and uses only the number's termination character.

The termination character allows a user to continue the "PE" command the same way a user can continue the "C" command, and the termination characters that are used with the "PE" command have the same meanings that they have with the "C" command. The "PE" command is therefore exactly like the "C" command except that MUDBUG never writes anything to the specified memory location when the user uses the "PE" command.

The "PE" command is extremely useful when a user wants to examine a read-only register such as the status register of an ACIA. If a user tried to use the "C" command to examine the status register of an ACIA, MUDBUG might try to write a new value back into the location of the status register, but the write operation would fail because the status register of an ACIA is a read-only register. Furthermore, the write operation in this case would have bad side effects because the read-only status register of an ACIA shares its memory address with the ACIA's write-only control register, and the status register's value would actually be written into the ACIA's write-only control register.

The "PE" command is generally intended for users who are dealing with special hardware devices that have read-only registers, but other users can also use the "PE" command to examine ordinary memory locations.

"PO" Command; 1 Parameter; Poke a Value into Memory.

The "PO" command allows a user to poke a value into a memory location with the assurance that MUDBUG will not try to read from the specified location. The "PO" command, which has one parameter, displays the address (but not the contents) of memory location START, and then it waits for the user to type a value that is to be put into location START.

The termination character of the new value allows a user to continue the "PO" command the same way a user can continue the "C" command, and the termination characters that are used with the "PO" command have the same meanings that they have with the "C" command. The "PO" command is therefore exactly like the "C" command except that MUDBUG never reads from the specified memory location when the user uses the "PO" command.

If a user types a hard termination character (/ *, . or <CR>) alone without a number for an input value, MUDBUG doesn't poke anything into the

specified location. The function of the termination character regarding the continuation or termination of the "PO" command is still effective, though, so a user can conveniently skip over a location with the "PO" command.

The "PO" command is extremely useful when a user wants to put a value into a write-only register such as the output data register of an ACIA. If a user tried to use the "C" command to put a value into the output data register of an ACIA, MUDBUG would try to read back the value to verify that it was stored correctly. The verification would fail, though, because the output data register of an ACIA is a write-only register. Furthermore, the attempted verification would have bad side effects in this case because the write-only output data register of an ACIA shares its memory address with the ACIA's read-only input data register. The attempted verification would cause MUDBUG to read from the ACIA's read-only input data register, and a read operation that accesses an ACIA's input data register causes the ACIA to change some of its internal status flags.

The "PO" command is generally intended for users who are dealing with special hardware devices that have write-only registers or registers that are sensitive to read accesses, but other users can also use the "PO" command to poke values into ordinary memory locations.

"PR" Command; No Parameters; Change PR.

The "PR" command, which doesn't require any parameters, displays the current hexadecimal value of the pseudo PR, and then it accepts a new hex value for the PR. If a user types a hard termination character (/ * , . or <CR>) alone without a number for a new PR value, the pseudo PR retains its present value. The "PR" command really isn't necessary for ordinary operations, but some users like to use it to control the value of the PR.

"Q" Command; No Parameters; Query Registers.

The "Q" command doesn't require any parameters, and it prints the values of the pseudo registers just as the execution of a software-interrupt instruction (SWI) normally does in the following format:

```
AR BR  XR   PR   SP  HINZVC
aa bb xxxx pppp ssss cccccc
```

All values are printed in hexadecimal with the exception of the condition-code values, which are printed in binary for ease of interpretation. Since there are only six actual bits in the condition-code register, only six bits are printed. The condition-code bits are interpreted as follows:

H = half carry I = interrupt mask N = negative
Z = zero V = overflow C = carry/borrow

The "Q" command provides a disctinctive output, so the "Q" command is a good command to use if the user wants to see if control is actually in MUDBUG. The "Q" command is also a good command to use if the user wants to change MUDBUG from step mode to normal mode with no side effects.

"R" Command; 1 Parameter; Relative Addressing.

The "R" command, which requires one parameter, is used to determine and/or to set the destination address of a branch-class instruction. The second word of a branch-class instruction in the M6800 always contains a signed, self-relative displacement value, so the destination address of a branch-class instruction isn't immediately obvious to most people by inspection. The "R" command, however, allows the user to determine and/or to set the destination address of a branch-class instruction rather easily.

The START parameter is assumed to be the memory address of the second word of a branch-class instruction, and the "R" command computes and prints the instruction's destination address based on that assumption.

Then the "R" command waits for the user to type a new value for the destination address. If the new destination address that is typed is out of range for the branch-class instruction that is being processed, MUDBUG outputs a backslash and a bell and aborts the "R" command. Otherwise, the "R" command computes and prints the new self-relative displacement value that will give the branch-class instruction the desired new destination address. Additionally, the "R" command puts the new self-relative displacement value into the second word of the branch-class instruction in memory, so the instruction is automatically changed in memory to address the desired new destination. If the second word of the branch-class instruction happens to be in ROM or nonexistent memory, of course, the "R" command aborts with a backslash and a bell instead of changing the self-relative displacement value in memory.

The termination character (blank tab / * , . or <CR>) that is used to terminate the new destination address is interpreted just as the termination character is interpreted by MUDBUG's "C" command: A carriage-return terminator causes MUDBUG to proceed automatically into the "C" command after first setting START <-- START + 1, so the location immediately following the branch-class instruction in memory is opened for changing via the "C" command. An asterisk, blank, or tab termination character causes MUDBUG to proceed into the "C" command after first setting START <-- START - 1, so the first word of the two-word branch-class instruction is opened for changing via the "C" command. A comma terminator causes MUDBUG to proceed into the "C" command with no change to the START parameter, so the second word of the two-word branch-class instruction is opened for changing via the "C" command. A period terminator returns control to the top of MUDBUG after the "R" command has put the new self-relative displacement value into memory, and a solidus terminator aborts the "R" command at once before it can change anything in memory.

If a user types a hard termination character (/ * , . or <CR>) alone without a number for a new destination address, MUDBUG retains the existing destination address and treats it as the desired destination address. The function of the termination character regarding the continuation or termination of processing as discussed above is still effective, so the "R" command is conveniently useful for checking destination addresses without necessarily changing them.

The "R" command alleviates some of the headaches of self-relative addressing by allowing a user to follow the logical flow of his or her program conveniently and easily. In fact, if the second word of a branch-class instruction has just been examined via the "C" command, the user needs only to type "R." or "R<CR>" to invoke the "R" command since the START parameter is already set properly from the previous "C" command.

The "R" command is also extremely convenient for users who write small programs in machine language, and it is equally useful for making machine-language patches to larger programs during debugging. When the user employs the "R" command, MUDBUG takes care of all of the annoying details of self-relative addressing, so the user can conveniently think in terms of direct memory addresses.

"S" Command; No Parameters; Change SP, the Stack Pointer.

The "S" command, which has no parameters, displays the current hex value of the pseudo stack pointer (SP), and then it accepts a new hex value for the pseudo stack pointer. If a user types a hard termination character (/ * , . or <CR>) alone without a new stack-pointer value, the pseudo stack pointer retains its present value.

The user should exercise a certain amount of caution and good judgement when changing the pseudo stack pointer because the stack pointer must always point into a RAM area of memory with at least 7 locations of RAM at and below the stack pointer. In other words, if SP represents the value of the stack pointer, locations SP, SP-1, SP-2, ..., and SP-6 must all be locations in a RAM (read/write) area of memory. At least 7 locations of RAM must be available for the user's stack at all times because 7 locations are required for saving the M6800 registers in the event of an interrupt. Most users would be wise to provide quite a few more than 7 locations for the stack because multiple interrupts can always occur. In fact, users should even be cautious about changing the stack pointer via instructions in their programs because all M6800 programs should (indeed must) be ready to accommodate an interrupt at any time.

If the stack pointer is ever set in such a way that fewer than 7 locations of RAM are available for the stack, MUDBUG temporarily makes the "G," "H," "N," "O," and "T" commands illegal, so the errant user is protected from being able to transfer control into his or her own program with the stack pointer initially pointing into ROM or nonexistent memory.

Most beginning users will never need the "S" command, but the command is made available as a convenience for the relatively sophisticated users who will want and need it. The "S" command can be used to reset the pseudo stack pointer following a program error or a partial execution of a program, so the user can always keep the pseudo stack pointer pointing into a desired RAM area.

"T" Command; 4 Parameters; Trap.

The "T" command, which is used to trap control at any desired point in the user's program, can accept as many as four parameters. Before we consider the case of a four-parameter "T" command, though, let us first discuss a two-parameter "T" command. If the user types a carriage return to terminate the "T" command's second parameter, MUDBUG automatically provides innocuous default values to make the remaining two parameters transparent to the user.

The first two parameters for the "T" command are known as TSTART and STOP. The "T" command starts execution of the user's program at location TSTART, and then it uses a software interrupt to trap control when or if control reaches location STOP. When (i.e., if) control reaches location STOP, the trap interrupt returns control to the top of MUDBUG after first invoking a routine that prints the values of the registers just as if a "Q" command had been input. The user's instruction at location STOP is not executed, so following a successful trap operation the pseudo PR points to location STOP, which contains the first word of the next sequential instruction to be executed in the user's program. This particular value of the pseudo PR conveniently allows the user to continue the execution of his or her program if desired. For example, the user might employ an "H" command, an "N" command, an "O" command, or another "T" command to continue execution of his or her program.

Notice that the first parameter for the "T" command is called TSTART. The "T" command uses its own private TSTART parameter instead of using the system-defined START parameter that is used by most other MUDBUG commands. Therefore, the "T" command can have a unique default value for its TSTART parameter instead of using the standard default value of the general-purpose START parameter. The "T" command uses the value of the pseudo PR as the default value for the TSTART parameter, so the user can conveniently continue trapping through his or her program from one segment to the next without typing a new TSTART parameter at each step. For example, typing "T , 2100" continues execution from the present point in the program and traps control at location $2100. Using the pseudo PR as the default value for the TSTART parameter is particularly convenient because the pseudo PR retains its useful significance even if several operations (e.g., memory dumps, "C" commands, etc.) are performed between successive "T" commands.

MUDBUG's "T" command is probably the single most powerful debugging tool that is available to the system's users. By using the "T" command, a user can execute any selected portion of a program, so a programmer can

interactively trace the flow of a program at execution time and easily isolate an error to a small routine or even to a single instruction. Users who quickly master the efficient use of the "T" command inevitably find that debugging their programs becomes an almost trivial task.

As with other commands that request execution of the user's program, MUDBUG temporarily treats the "T" command as an illegal command when the pseudo stack pointer has been set to address ROM or nonexistent memory. This feature protects the user from inadvertently transferring control to his or her program with an invalid value in the stack pointer. Please refer to the discussion of the "S" command for details.

The "T" command is also illegal if location STOP happens to be in ROM or nonexistent memory. This restriction is necessary because the routine that implements the "T" command's function must temporarily store a software-interrupt instruction (SWI) into location STOP. If the user tries to specify a STOP address in ROM or nonexistent memory, MUDBUG aborts the "T" command with a backslash and a bell.

The "T" command traps control at location STOP via a software interrupt, so users who employ the "T" command should carefully insure that their programs always keep the stack pointer pointing into enough RAM to support an interrupt. If the user's program fails to maintain the stack pointer in such a way that the stack can support an interrupt properly at the trap point (i.e., when control reaches location STOP), the "T" command still traps control successfully at location STOP, but the invalid stack-pointer value generally causes some or all of the software-defined pseudo registers (other than the pseudo stack pointer) to be set erroneously following the trap at location STOP.

Notice carefully that the trap point (i.e., STOP) must be the first word of an instruction. Control cannot be trapped in the middle of an instruction, and any users who try to trap control in the middle of an instruction will experience erroneous results.

A successful use of the "T" command automatically puts MUDBUG into step mode if the hardware of the system supports step-mode operations. The "T" command is considered to be successful if control is successfully trapped at the specified STOP location. If control stops somewhere else, of course, the "T" command is not considered to be successful.

When MUDBUG is in step mode, the MUDBUG prompt is changed from the normal ">" to a "?" to let the user know that the system is in step mode. The user can simply type a carriage return as a response to a "?" prompt to tell MUDBUG to execute the next instruction of the user's program. Thus a user who has just trapped control with the "T" command can conveniently step through the next section of his or her program one instruction at a time by simply typing a carriage return at each step. MUDBUG provides a register dump after each step when instructions are executed one at a time in step mode.

When MUDBUG is not in step mode, a carriage return alone is treated as a do-nothing command. A user can therefore type a carriage return as a MUDBUG command in normal mode to advance the cursor to a new line.

MUDBUG automatically returns from step mode to normal mode whenever the user types any non-debugging command (except a comment command) or when the user types an invalid command. The debugging commands are the "N" command, the "O" command, the "T" command, and the "V" command. In other words, any command that tells MUDBUG to execute a portion (but not all) of the user's program is a debugging command. Debugging commands that are executed successfully put MUDBUG into step mode, and other commands (except the comment command) take MUDBUG out of step mode. Step mode is designed to provide a convenient single-step mechanism for the user who is debugging a program.

Now let us consider the "T" command's third and fourth parameters. The "T" command's third and fourth parameters are available only if the system on which MUDBUG is running has the necessary hardware to support step-mode operations. If the system's hardware doesn't support step-mode operations, MUDBUG allows only the first two parameters for the "T" command.

The "T" command's third parameter is generally known as the stop-count parameter. The stop-count parameter, which is useful for debugging loops, can be used when the user wants control to go through a loop some specified number of times before control is trapped at a specified instruction in the loop. If the value of the stop-count parameter is nine, for example, MUDBUG traps control the ninth time that control reaches location STOP (if, in fact, control reaches location STOP nine times). The instruction at location STOP is executed normally the first eight times it is encountered, but control is trapped and the instruction is not executed the ninth time.

Following a successful trap operation the pseudo PR points to location STOP, which contains the first word of the next sequential instruction to be executed in the user's program. The PR is therefore set properly to allow the user to continue the execution of his or her program via any of several MUDBUG commands. For example, the user might want to use MUDBUG's step-mode feature to execute the next few instructions one step at a time.

The user should always remember that the stop-count parameter, like all of MUDBUG's parameters, is interpreted as a hexadecimal number. Typing "100" for the stop-count parameter therefore tells MUDBUG to trap control the $100th (i.e., 256th) time that control reaches location STOP. MUDBUG treats the stop-count parameter as a 16-bit unsigned value, and MUDBUG interprets the value zero to mean 65,536.

The default value for the stop-count parameter is always one, so a user who wants to trap control the first time control reaches location STOP can simply omit the stop-count parameter. The stop-count feature is therefore conveniently transparent when it is not being used.

Notice that the stop-count parameter is a private parameter that is used only by the "T" command. The stop-count parameter is not related to the system-defined KEY parameter that appears as the third parameter of some other MUDBUG commands.

The "T" command's fourth parameter, which is useful only if the stop-count parameter has been specified, is known as the print-count parameter. The print-count parameter tells MUDBUG how frequently the system should generate a register dump. For example, if the value of the print-count parameter is three, MUDBUG generates a register dump every third time that control reaches location STOP. MUDBUG generates the register dump immediately before the instruction at location STOP is executed.

The print-count parameter provides a method for obtaining dynamic register dumps as the user's program is executing. MUDBUG always generates a register dump when control is actually trapped at location STOP, and the print-count parameter can be used when the user wants to see some register dumps at some intermediate steps as well.

The user can specify any desired value for the print-count parameter if intermediate register dumps are desired. For example, if a user specifies the value one for the print-count parameter, MUDBUG prints a register dump every time control reaches location STOP. Similarly, if a user specifies the value five for the print-count parameter, MUDBUG prints a register dump every fifth time control reaches location STOP.

The default value for the print-count parameter is always taken from the value (if any) that was specified for the stop-count parameter. By default, therefore, MUDBUG generates a register dump only when control is actually trapped. The default value for the print-count parameter conveniently makes the print-count feature transparent when it is not being used.

Like all other MUDBUG parameters, the print-count parameter is always interpreted as a hexadecimal number. MUDBUG treats the print-count parameter as a 16-bit unsigned number, and the value zero is interpreted to mean 65,536. The print-count parameter is a private parameter that is used only by the "T" command, and the print-count is not related to the system-defined MASK parameter.

"V" Command; 4 Parameters; Verify ROM Program.

The "V" command is functionally identical to the "T" command, but the "V" command can be used with a program that resides in ROM whereas the "T" command cannot be used with a program that resides in ROM. The "V" command is available only if the system on which MUDBUG is running has the hardware that is necessary to support step-mode operations. If the system can't support step-mode operations, MUDBUG treats the "V" command as an invalid command.

Although the "V" command can be used with programs that reside in RAM or pseudo ROM, the "T" command is better for those programs. The "V" command executes a program much more slowly than the "T" command does, so the "V" command is clearly unsuitable for any programs that contain any real-time segments.

The "V" command is provided as a special command that is occasionally useful in a ROM environment, but most users seldom have any real need to use the "V" command.

"W" Command; 2 Parameters; Write Object Tape.

The "W" command, which requires two parameters, writes the values of memory locations START through STOP to the terminal in M6800 object format. If a tape-producing device is attached to the terminal, the "W" command produces an object tape that is compatible with any standard M6800 loader.

If the STOP parameter is terminated by a comma, a blank space, a tab, or an asterisk, the "S9" record and the trailer (i.e., nulls) that normally mark the end of an M6800 object tape are not written, so users can conveniently write several disjoint areas of memory onto a single object tape by simply invoking the "W" command several different times with comma terminators.

A carriage-return or period terminator for the STOP parameter, of course, causes the "S9" end-of-tape record and six inches of trailer to be written normally, and the carriage-return terminator is therefore the terminator that most users type for ordinary lab work.

The "W" command is useful for obtaining a reloadable object tape of a program that has been modified in memory during debugging. This object tape can then be reloaded later (via the "L" command) if the user's program is ever accidentally destroyed in memory, so the user doesn't need to waste any time recreating patches and modifications that have already been made once. Since a power interruption or a minor programming error can easily destroy a program that resides in RAM, users can potentially save a great deal of time and effort by using the "W" command before their programs are destroyed.

"X" Command; No Parameters; Change XR.

The "X" command, which doesn't require any parameters, displays the current hexadecimal value of the pseudo XR, and then it accepts a new hex value for the XR. If a user types a hard termination character (/ * , . or <CR>) alone without a number for a new XR value, the pseudo XR retains its present value.

"Z" Command; No Parameters; Zero Registers.

The "Z" command, which doesn't require any parameters, initializes the registers as follows: It clears the pseudo AR, BR, XR, PR, and CC register, and it reinitializes the pseudo stack pointer to point to the user's default stack area in the MUDBUG RAM.

The "Z" command provides a quick way of reinitializing all of the pseudo registers with a single command, so the user doesn't need to type individual commands to initialize each register individually.

Users should carefully note the fact that a restart interrupt (RSI) performs the function of a "Z" command besides restarting MUDBUG as if a power-up condition had occurred.

Chapter 4

Internal Routines and Subroutines

MUDBUG quite naturally includes several internal subroutines, and many of these internal subroutines are potentially useful for general-purpose applications. Some of MUDBUG's internal subroutines have therefore been made available to the system's users, and programmers who wish to invoke any of MUDBUG's internal subroutines can simply call the desired routine(s) from their own programs. MUDBUG users can thus avoid the unnecessary effort of re-inventing and re-coding any routines that have already been implemented in MUDBUG.

There is one caution that users must observe before writing any programs that call any of MUDBUG's internal routines: Future releases of MUDBUG will not necessarily be completely compatible with the current version of MUDBUG with regard to internal subroutines. MUDBUG's internal subroutines are offered to the system's users only as a convenience factor, not as a fully-supported feature, and any users who lock themselves inflexibly into MUDBUG's current set of internal routines may have some difficulty upgrading to subsequent new releases of the MUDBUG system.

The alphabetized list of routines on the next page summarizes the internal MUDBUG routines that are available to the system's users, and interface characteristics such as calling-sequence requirements and return conditions are given for each subroutine in the pages that follow. Notice that the entry points for the available internal routines all occur as consecutive entries in a vector table that starts at the first memory location of the MUDBUG ROM. This vector table is provided as a convenience for the user so that the entry-point addresses for MUDBUG's internal subroutines will not change from one sub-version of MUDBUG to another.

4.1. Summary of Internal Routines and Subroutines.

The list below summarizes the internal MUDBUG routines that are available for direct access by the system's users. The symbolic label "ROM" is used here and throughout this document to represent the memory address of the first word of the MUDBUG ROM.

Name	Entry Point	Function
CRLF	ROM+$00	Output a CRLF to the terminal.
CRLF4H	ROM+$03	Output a CRLF and the four-hextet value of locs XR & XR+1.
ERROR	ROM+$06	Output a backslash and a bell; then return to MUDBUG.
INCHR	ROM+$09	Input a single character from the keyboard to the AR.
MUDBUG	ROM+$0C	Return control directly to the top of MUDBUG.
NEWVAL	ROM+$0F	Read a four-hextet number into locations XR and XR+1.
NUMBI	ROM+$12	Read a four-hextet number into the AR (MSB) and BR (LSB).
OUT2H	ROM+$15	Output the two-hextet hexadecimal value of location XR.
OUT2HB	ROM+$18	Output the two-hextet hexadecimal value of the BR.
OUT2HS	ROM+$1B	Output the two-hextet value of location XR and a space.
OUT4HS	ROM+$1E	Output four-hextet value of locs XR & XR+1 and a space.
OUTCHR	ROM+$21	Output an ASCII character from the AR to the terminal.
OUTS	ROM+$24	Output a single blank space to the terminal.
POWERUP	ROM+$30	Restart the system as if from a cold start.
PRTXM	ROM+$27	Print the four-hextet XR value and the value of loc XR.
READPT	ROM+$2A	Read one character from the tape reader into the AR.
STAAB	ROM+$2D	STAA 0, X; STAB 1, X; RTS.

4.2. Subroutine Descriptions.

This section contains interface information for the internal MUDBUG routines that are available for direct access by the system's users. Each routine is briefly described, and then its calling sequence and return conditions are documented as may be appropriate. The routines are listed in alphabetical order for the convenience of the user.

Subroutine CRLF; Entry Point = ROM+$00.

Subroutine CRLF outputs a carriage return and a line feed (CRLF) to the user's terminal.

Calling Sequence:
```
JSR   CRLF        Output a carriage return and a line
                  feed to the terminal.
```

Return Condition: Part of the CC value is destroyed, and the AR contains a line-feed code ($0A), but the CC.I bit and all of the other register values are preserved.

Subroutine CRLF4H; Entry Point = ROM+$03.

Subroutine CRLF4H outputs a CRLF followed by the four-hextet value of the two words that are addressed by the XR and XR+1, and the subroutine outputs a blank space following the four-hextet value.

Calling Sequence:
```
LDX   =VALUE      This example calling sequence shows
JSR   CRLF4H      how to output the four-hextet value
                  from locations VALUE and VALUE+1.
```

Return Condition: Part of the CC value is destroyed, and the AR contains an ASCII blank ($20), but the CC.I bit and all of the other register values are preserved.

ERROR Routine; Entry Point = ROM+$06.

The ERROR routine outputs a backslash ("\") and a bell to alert the user to some error condition, and then it transfers control to the top of MUDBUG with no updating of the pseudo-register values. Notice that the ERROR routine is not a subroutine. Instead of returning control to the calling program, the ERROR routine transfers control directly to the top of MUDBUG.

Calling Sequence: JMP ERROR N.B.: JMP, not JSR.

Return Condition: There is no return. Control returns directly to the top of MUDBUG.

Subroutine INCHR; Entry Point = ROM+$09.

Subroutine INCHR inputs a single ASCII character from the terminal's keyboard through the ACIA (Asynchronous Communications Interface Adapter; part number MC6850). Subroutine INCHR returns the input character to the calling routine with zero parity in the AR. Besides returning the input character to the calling routine, subroutine INCHR automatically echoes the character to the terminal. If the input character is a carriage return, however, subroutine INCHR does not echo it to the terminal. This feature allows the software to control the position of the cursor on the user's terminal.

If subroutine INCHR receives an XOFF flow-control character, the subroutine waits until it receives a matching XON flow-control character. Then subroutine INCHR gets the next input character and returns that input character to the calling routine. Subroutine INCHR also ignores any extraneous XON characters that it receives, and the subroutine does not echo XON or XOFF characters to the terminal.

Subroutine INCHR doesn't treat XON and XOFF characters as ordinary input characters because those flow-control characters can be transmitted automatically by many terminals. A terminal can transmit an XOFF character to tell MUDBUG to suspend I/O processing, and the terminal can transmit an XON character to tell MUDBUG to resume I/O operations.

Calling Sequence: JSR INCHR Input one character to the AR.

Return Condition: The AR contains the 7-bit, zero-parity value of the input character, and the condition codes are set to reflect a comparison of the input character against a carriage return. The BR, XR, and SP are all preserved, and the CC.I bit is also preserved.

Stack Usage: Subroutine INCHR requires 6 locations of stack memory. These 6 stack locations include the two locations that contain the subroutine's return address.

MUDBUG; Entry Point = ROM+$0C.

Users can return control directly to the top of MUDBUG without changing the old values of the pseudo registers by jumping to location MUDBUG. This entry point to MUDBUG is used when the user wishes to terminate execution without generating a register dump.

Calling Sequence: JMP MUDBUG N.B.: JMP, not JSR.

Return Condition: There is no return. Control remains in MUDBUG, and MUDBUG prompts the user for the next MUDBUG command.

Subroutine NEWVAL; Entry Point = ROM+$0F.

Subroutine NEWVAL reads a 4-hextet hexadecimal number from the user's terminal, and the subroutine stores the value of the input number into the locations that are addressed by the XR and XR+1. If the input number is terminated by a solidus or if the input consists of a termination character with no hextets, subroutine NEWVAL doesn't change the value that is already in memory.

Subroutine NEWVAL uses subroutine NUMBI internally, so the reader should refer to subroutine NUMBI for more details regarding NEWVAL's operation and its return conditions. In particular, the documentation for subroutine NUMBI explains how subroutine NUMBI sets certain variables to provide some detailed information about the nature of the input number and its termination character.

If the input number is terminated by a solidus, subroutine NEWVAL transfers control directly back to the top of MUDBUG. Otherwise, control returns to the calling routine.

Calling Sequence:

```
LDX   =VALUE
JSR   NEWVAL
```

This example calling sequence shows how to read a new value for the double-precision number that is in locations VALUE and VALUE+1.

Return Condition:

1. Normal return. The AR and the BR contain the new (or preserved) value that is in locations XR and XR+1, and the other registers (except the CC register) are preserved. Location TERMCH is set to reflect the termination character that was used.

2. Solidus terminator. If the input number is terminated by a solidus, control goes directly to the top of MUDBUG instead of returning to the calling routine.

Subroutine NUMBI; Entry Point = ROM+$12.

Subroutine NUMBI reads an ASCII-coded hexadecimal number from the terminal, and it returns the 16 least-significant bits of the number's value in the AR (most-significant byte) and the BR (least-significant byte).

The input number must be terminated by one of seven termination characters: a period, a comma, an asterisk, a blank space, a tab, a solidus ("/"), or a carriage return. A solidus termination character both terminates and cancels the input number, so subroutine NUMBI returns the AR and the BR unchanged to the calling routine when the user types a solidus terminator. The other six termination characters merely terminate the input number.

If the user types a carriage return as the termination character, subroutine NUMBI does not echo the carriage return to the user's terminal. This feature allows the calling routine's software to control the position of the terminal's carriage or cursor.

Subroutine NUMBI always leaves a special, easy-to-test termination code in location TERMCH (i.e., RAM+$06) to represent the particular termination character that terminated the input number. The termination codes for the various termination characters are as follows:

blank	= -7	= $F9	= %11111001
tab	= -7	= $F9	= %11111001
"*"	= -5	= $FB	= %11111011
","	= -3	= $FD	= %11111101
"."	= -1	= $FF	= %11111111
"/"	= 0	= $00	= %00000000
<CR>	= +13	= $0D	= %00001101

Besides leaving the termination code in location TERMCH, subroutine NUMBI always returns the condition codes reflecting the value that is in TERMCH, so the user can employ a conditional branch following a call to subroutine NUMBI to determine something about the termination character.

Subroutine NUMBI allows the user to type leading blank space(s) and/or tab(s) with a hex-number input because many users like to insert white space at the beginning of an input number for improved readability. No leading white space is required, of course, but blanks and tabs are allowed until the first hextet of the input number has been typed. Since subroutine NUMBI allows the user to type leading blanks and/or leading tabs, a user can't use a blank or a tab to terminate an empty (i.e., missing) input number. A user who wants to omit an input number entirely must type one of the other termination characters to indicate the missing number.

The input number for subroutine NUMBI can be signed. A leading minus sign causes subroutine NUMBI to return the two's complement of the input number as the input value, and leading plus signs have no effect on the

value of the input number. Two minus signs (or any even number of minus signs) cancel each other, but any odd number of minus signs result in the negation of the input value. Leading sign characters are permitted until the first hextet of the input number has been typed, and after that time sign characters are rejected as invalid input characters.

If a user types a leading plus sign, subroutine NUMBI clears location PLUSFLG (i.e., RAM+$2C). Otherwise, subroutine NUMBI puts a nonzero value into location PLUSFLG, so the calling routine can check location PLUSFLG to determine whether or not a leading plus sign was typed.

The terminal operator can correct any typing errors in an input to subroutine NUMBI by simply typing a backslash ("\") before the number's termination character has been typed. A backslash cancels everything that has already been typed for the input number and restarts the number from its first character.

The operator can alternatively correct simple typing errors by taking advantage of the fact that subroutine NUMBI retains only the 16 least-significant bits of the input number. By simply typing new hextets, the operator effectively shifts old hextets out of the left end of the number.

If no valid hextets (0-9, A-F, or a-f) are input to subroutine NUMBI (i.e., if a termination character other than a blank or a tab is typed without any valid hextets preceding it), the subroutine returns the AR and the BR to the calling routine with their original values unchanged. This feature allows the calling routine to provide a default value for the input number. Additionally, subroutine NUMBI puts a nonnegative value into location HEXTETS (i.e., RAM+$2B) if the subroutine receives an empty number. Otherwise, subroutine NUMBI puts a negative value into location HEXTETS.

If subroutine NUMBI receives an invalid character for an input, it outputs a bell (ASCII code = $07) and a question mark ("?"; ASCII code = $3F). Aside from making these outputs, subroutine NUMBI ignores the invalid input character. Only the following characters are accepted as valid inputs by subroutine NUMBI:

blank tab + - , * . <CR> / 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f \

The first two characters in this list (blank and tab) are valid as leading characters before the first hextet of the input number has been typed, and they are also valid as termination characters after the first hextet of the input number has been typed. The next two characters in the list (+ and -) are valid only before the first hextet of the input number has been typed, and they are rejected as invalid inputs if they are ever typed anywhere other than at the beginning of a number. A backslash ("\") can always be used to restart an input number, so a user can go back and type a forgotten sign character by first typing a backslash.

The calling sequence and return conditions for subroutine NUMBI are as follows:

Calling Sequence: Typically, JSR NUMBI or BSR NUMBI

Return Condition:

1. Normal return. The AR-BR contains the double-precision input value, and locations TERMCH, PLUSFLG, and HEXTETS have all been set as specified above. The CC.I bit is preserved, and the other CC bits reflect the value of the TERMCH code. All other register values are preserved.

2. Solidus terminator. The AR and the BR are both returned containing their original values, and location TERMCH contains the termination code for a solidus (i.e., 0). The CC.I bit is preserved, and the other CC bits reflect the zero value in TERMCH. All other register values are preserved. Location HEXTETS is nonnegative to indicate that no hextets were retained, and PLUSFLG is set to indicate whether or not the user typed a leading plus sign.

3. No number input. If no valid hextets are input ahead of the termination character, the AR and the BR are both returned unchanged, and the special code for the termination character is returned in location TERMCH. The CC.I bit is preserved, and the other CC bits reflect the value of the TERMCH code. All other register values are preserved. Location HEXTETS is nonnegative to indicate that no hextets were typed, and PLUSFLG is set to indicate whether or not the user tyed a leading plus sign.

Subroutine OUT2H; Entry Point = ROM+$15.

Subroutine OUT2H outputs the two-hextet hexadecimal value of the memory word that is addressed by the XR.

Calling Sequence:

```
LDX   =VALUE
JSR   OUT2H
```

This example calling sequence shows how to output the two-hextet value from location VALUE.

Return Condition: The AR and part of the CC value are destroyed, but all other register values and the CC.I bit are preserved.

Subroutine OUT2HB; Entry Point = ROM+$18.

Subroutine OUT2HB outputs the two-hextet hexadecimal value of the BR to the user's terminal.

```
Calling Sequence:  LDAB  VALUE      This example calling sequence shows
                   JSR   OUT2HB     how to output the two-hextet value that
                                    is in location VALUE.
```

Return Condition: The AR and part of the CC value are destroyed, but the other registers and the CC.I bit are preserved.

Subroutine OUT2HS; Entry Point = ROM+$1B.

Subroutine OUT2HS prints the two-hextet hexadecimal value of the memory word that is addressed by the XR, and it prints a blank space following the two-hextet value.

```
Calling Sequence:  LDX   =VALUE     This example calling sequence shows
                   JSR   OUT2HS     how to print the two-hextet value from
                                    location VALUE.
```

Return Condition: Part of the CC value is destroyed, and the AR contains an ASCII blank ($20), but the CC.I bit and all of the other register values are preserved.

Subroutine OUT4HS; Entry point = ROM+$1E.

Subroutine OUT4HS prints the four-hextet hexadecimal value of the two memory words that are addressed by the XR and XR+1, and it prints a blank space following the four-hextet value.

```
Calling Sequence:  LDX   =VALUE     This example calling sequence shows
                   JSR   OUT4HS     how to output the two-byte value from
                                    locations VALUE and VALUE+1.
```

Return Condition: Part of the CC value is destroyed, and the AR contains an ASCII blank ($20), but the CC.I bit and all of the other register values are preserved.

Subroutine OUTCHR; Entry Point = ROM+$21.

Subroutine OUTCHR outputs a single ASCII character from the AR to the ACIA, which is interfaced to the terminal.

Calling Sequence: JSR OUTCHR Output an ASCII character from the AR to the user's terminal.

Entry Conditions: The AR contains the ASCII character that is to be output.

Return Condition: Part of the CC value is destroyed, but the CC.I bit and all other register values are preserved.

Subroutine OUTS; Entry Point = ROM+$24.

Subroutine OUTS merely prints a blank space at the user's terminal.

Calling Sequence: JSR OUTS Output one blank space.

Return Condition: Part of the CC value is destroyed, and the AR contains an ASCII blank ($20), but the CC.I bit and all of the other register values are preserved.

POWERUP Routine; Entry Point = ROM+$30.

Users can transfer control to location POWERUP to restart the MUDBUG system as if it were coming up from a cold start. The POWERUP routine initializes the system's hardware and MUDBUG's pseudo-register values, and this same routine is invoked by a restart interrupt (RSI) at power-up time.

Calling Sequence: JMP POWERUP Restart the system.

Return Condition: There is no return. Control remains in MUDBUG.

Subroutine PRTXM; Entry Point = ROM+$27.

Subroutine PRTXM prints a CRLF and the four-hextet hexadecimal value of the XR followed by two blank spaces. Then it prints the two-hextet value of the contents of the memory location that is addressed by the XR, and finally it prints another blank space. The output format for subroutine PRTXM is therefore as follows:

"XXXX MM ".

Calling Sequence: JSR PRTXM — Print the XR and the value of the location that is addressed by the XR.

Return Condition: Part of the CC value is destroyed, and the AR contains an ASCII blank ($20), but the CC.I bit and all of the other register values are preserved.

Subroutine READPT; Entry Point = ROM+$2A.

Subroutine READPT reads a single character from the terminal's tape reader. The input character is returned in the AR, and it is not echoed to the terminal.

Calling Sequence: JSR READPT — Read one frame of tape.

Return Condition: The AR contains the input character, and part of the CC value is destroyed. All of the other register values and the CC.I bit are preserved.

Subroutine STAAB; Entry Point = ROM+$2D.

Subroutine STAAB uses indexed addressing to store the values from the AR and the BR into the locations that are addressed by the XR and XR+1.

Calling Sequence: JSR STAAB — Store the AR and the BR.

Return Condition: The AR and the BR have been stored, and all registers (except the CC register) are preserved. The CC.I bit is preserved.

Final Report

Contract NAS8-34969

ENHANCEMENT OF INTELLIGENT EDITOR/PRINTER

This report was prepaired by Arizona State University under contract NAS8-34969 Enhancement of Intelligent Editor/Printer for Marshall Space Fligh Center of the National Aernautics and Space Administration.

A P P E N D I X

B

The design specifications for the MC6809 version of the intelligent printer controller card. MSFC is currently constructing a printed circuit card to implement this design. The software necessary to use this card as a controller for a Diablo Hy-Type 2 printer is currently under development and will be supplied to the government after it is checked out on the finished version of this card which is to be supplied to ASU by MSFC.

Printer Controller PWB
(6809)

NOTES:

(1) The distance from the 3.686 MHz crystal to the MC6809 is critical and must be less than 20 mm. Also the two 24 pf caps on each leg of the crystal must be within 20 mm of each other (refer to the Motorola Data book for clarification).

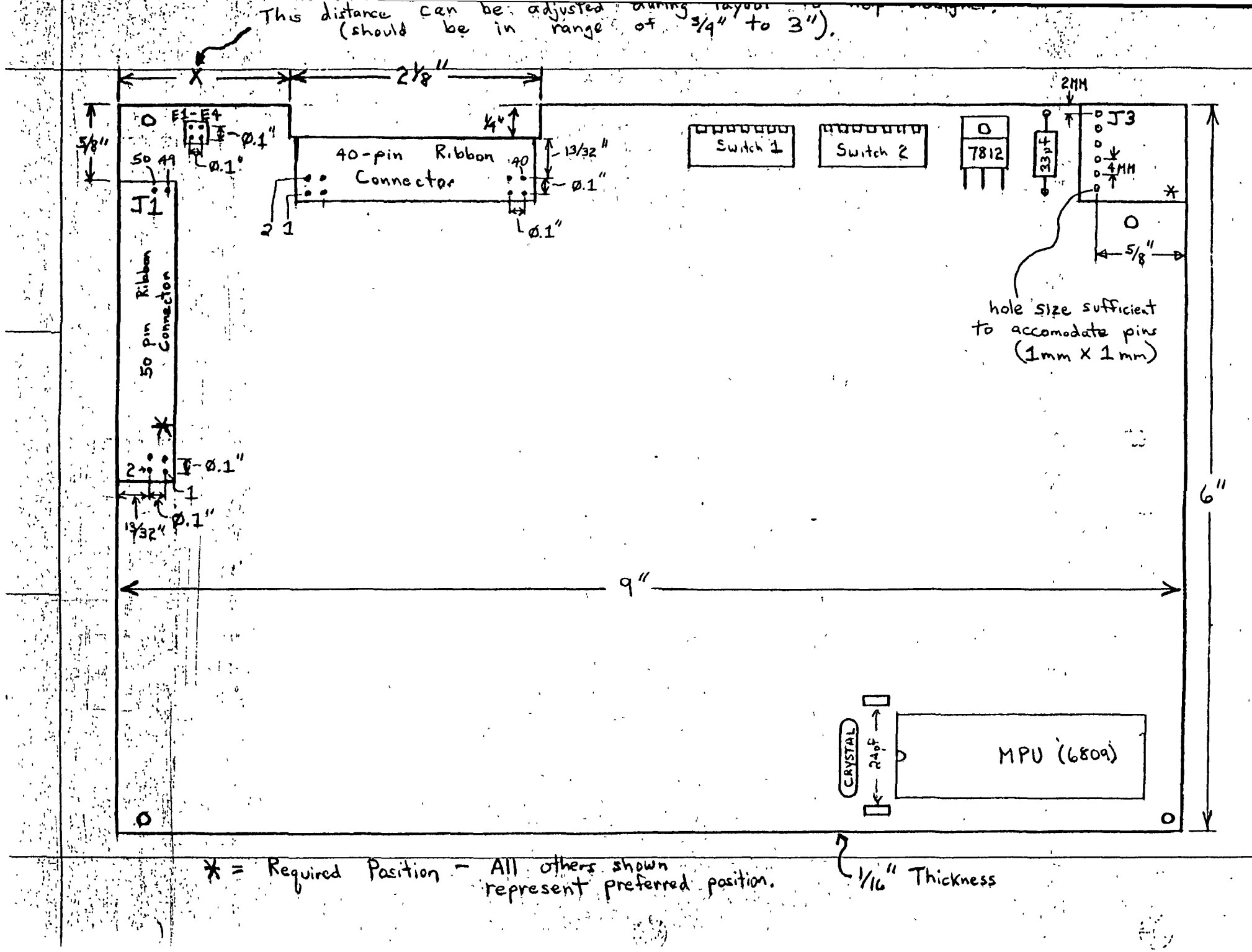
(2) All discrete resistors are 1/4 watt carbon composition.

(3) Decoupling caps should be added between +5 VDC and Ground liberally (1 for every 4 I.C.'s).

(4) IC numbers (Ux) should be reassigned per the layout of the PWB. Suggest U1 be the upper left hand corner, U2 to its right, and so forth.

(5) It would be nice to have four mounting holes (min 3/16" dia.) located as symmetrical as possible and close to the corners.

(6) The connections to hex inverters, AND gates, and other generic IC's can be changed during layout as long as the change is electrically equivalent. Gates required to be Schmidt-Triggered are marked accordingly and should NOT be replaced by a regular totem pole gate. Also ensure that open collector gates are not used in place of totem pole outputs and vice versa.


This distance can be adjusted during layout (should be in range of 3/4" to 3").
X
2 1/8"
E1-E4
Ø.1"
5/8"
50 49
J1
50 pin Ribbon Connector
40-pin Ribbon Connector
40
2 1
1/4"
13/32"
Switch 1
Switch 2
7812
33pf
2MM
J3
4MM
5/8"
hole size sufficient to accomodate pins (1mm x 1mm)
6"
9"
CRYSTAL
24pf
MPU (6809)
* = Required Position - All others shown represent preferred position.
1/16" Thickness

Parts List
Printer Controller — 6809

A. Integrated Circuits (IC's)

Des.	Part	# pins	Page	+5 VDC	Ground
U1	MPU — MC6809	40	1	7	1
U2	PIA0 — MC6821	40	4	20	1
U3	PIA1 — MC6821	40	4	20	1
U4	ACIA1 — MC6850	24	5	10,12	1,23
U5	PROM0 — 2764	28	6	1,26,27,28	14
U6	PROM1 — 2764	28	6	1,26,27,28	14
U7	74LS138	16	2	16	4,5,8
U8	82S129	16	2	16	1,8,13,14
U9	PTM — MC6840	28	2	14	1,2,4,5,7,28
U10	RAM0 — 2016	28	3	26,28	14
U11	RAM1 — 2016	28	3	26,28	14
U12	RAM2 — 2016	28	3	26,28	14
U13	RAM3 — 2016	28	3	26,28	14
U14	PIA2 — MC6821	40	7	20	1
U15	SW1 — Dip Switch	16	2,7	—	1,2,3,4,5,6,7,8
U16	SW2 — Dip Switch	16	7	—	1,2,3,4,5,6,7,8
U17	74LS244	20	7	20	10
U19	74LS244	20	1	20	1,10,19
U20	74LS244	20	1	20	1,10,19
U21	74LS14	14	1,4	14	7
U22	74LS14	14	4,7	14	7
U23	7404	14	1,2	14	7
U24	7410	14	2,4	14	7
U25	7400	14	1	14	7
U26	7432	14	2,7	14	7
U27	Res. Pack 899-1-1k	16	2,4	16	—
U28	Res. Pack 899-1-1k	16	4,7	16	—
U29	Res. Pack 899-1-1k	16	7	16	—
U31	MC1488	14	5	—	7
U32	MC1489A	14	5	14	7
U33	7416	14	4	14	7
U34	7416	14	4	14	7
U35	7416	14	4	14	7
U36	7416	14	4	14	7
U37	ACIA0 — MC6850	24	5	10,12	1,23

B. Discrete Components

Des	Part	Page
R1	10K Resistor	1
R2	3.3K Resistor	1
R3	3.3K Resistor	1
R4	3.3K Resistor	1
R5	3.3K Resistor	1
R6	3.3K Resistor	1
R7	10K Resistor	5
R8	10K Resistor	5
C1	24pf Cap — Ceramic	1
C2	24pf Cap — Ceramic	1
C3	1.0uf Cap — Ceramic	5
C4	1.0uf Cap — Ceramic	5
C5	33uf Cap — Tantalum	5
D1	Diode	1
A1	7812 — +12VDC regulator	5

C6 — Ø.1 µF Cap - Ceramic — 4
C7 — Ø.1 µF Cap - Ceramic — 4
C8 — 1.Ø µF Cap - Ceramic — 1

C. Connectors

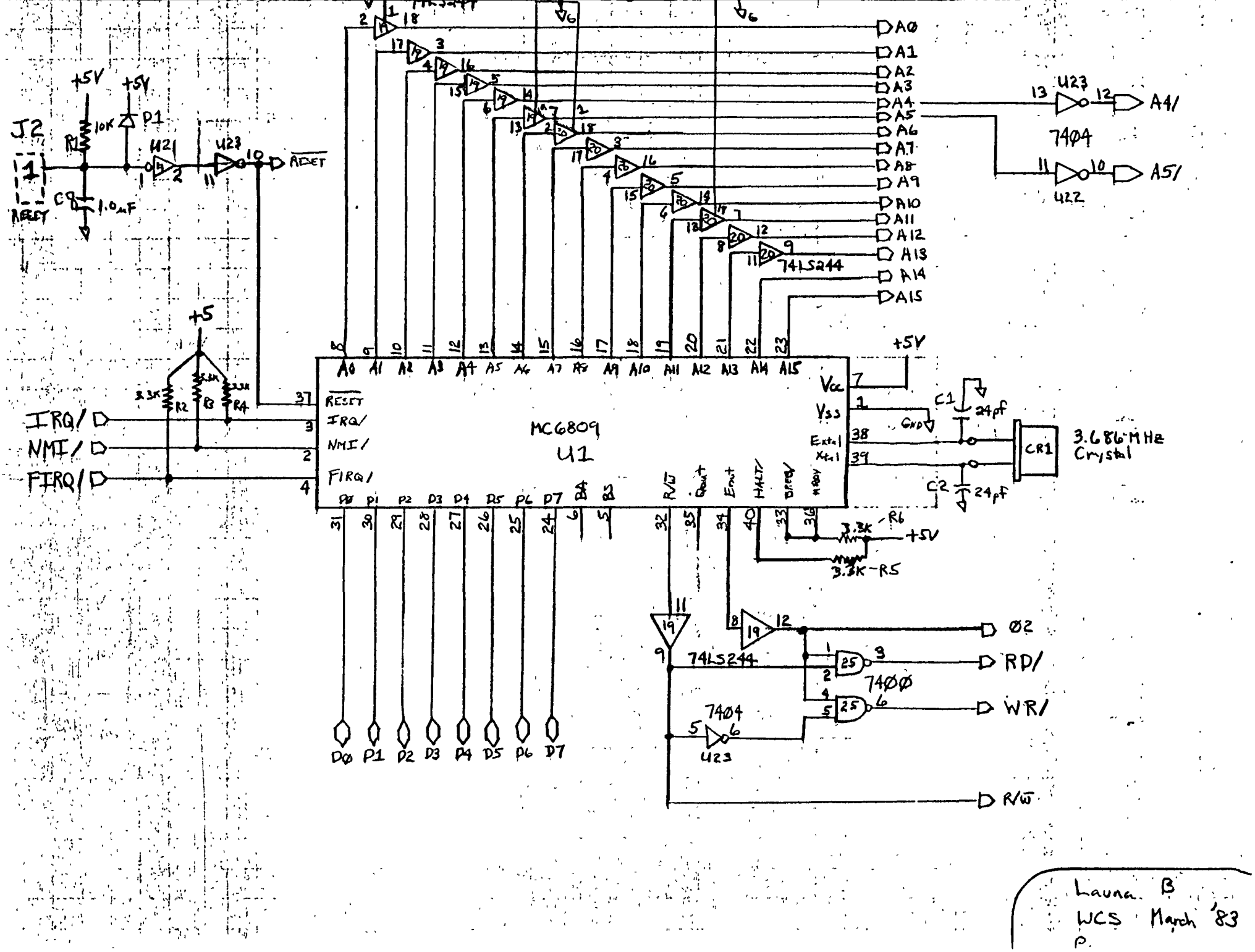
J1 — 50-pin right angle ribbon connector
J2 — 40-pin right angle ribbon connector
J3 — 6-pin right angle power connector

D. Miscellaneous

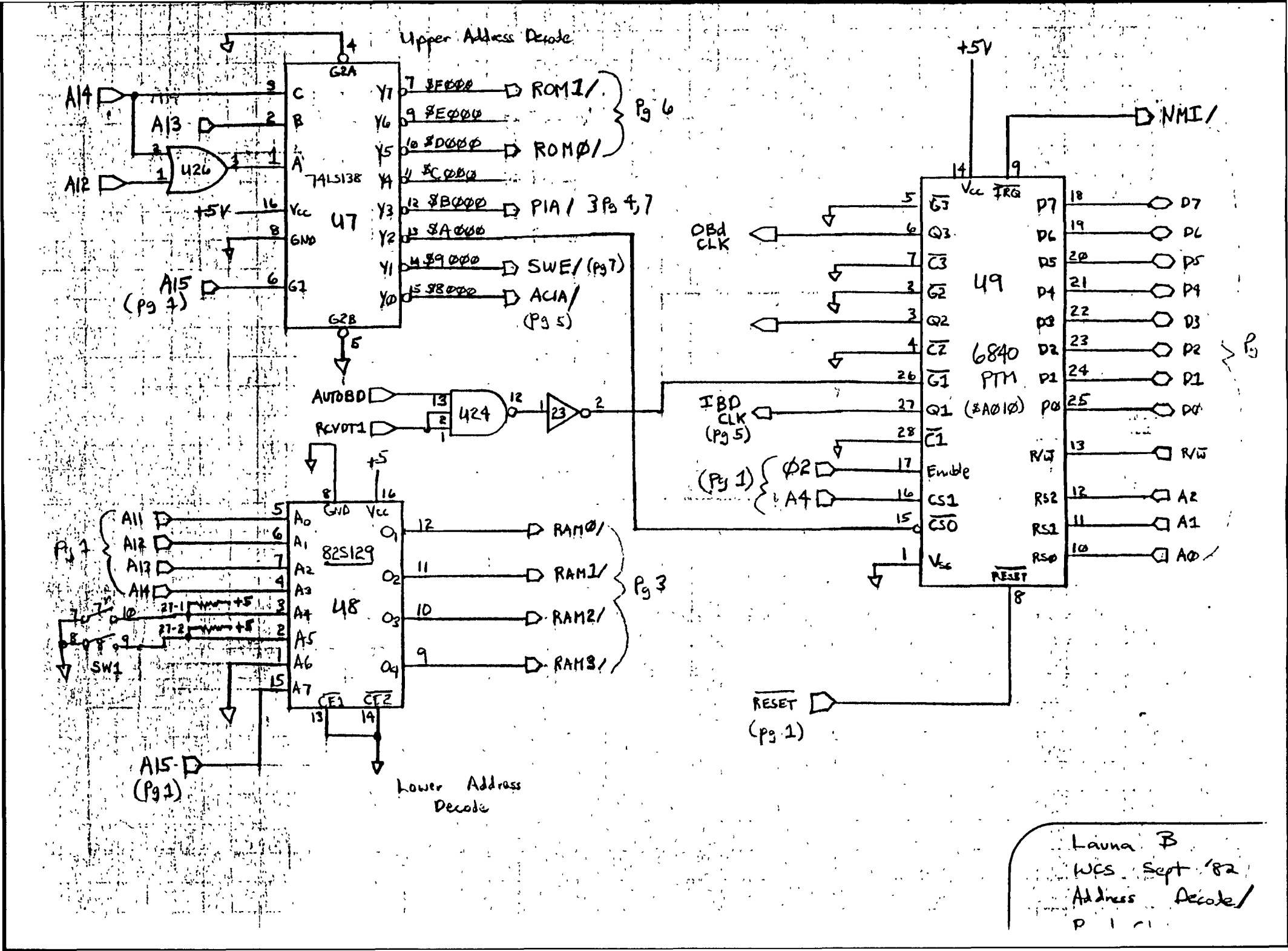
(1) Connector J1 should have the following Ground connections:
J1-2,8,11,14,16,18,20,22,25,30,31,32,35,38,41,44,47

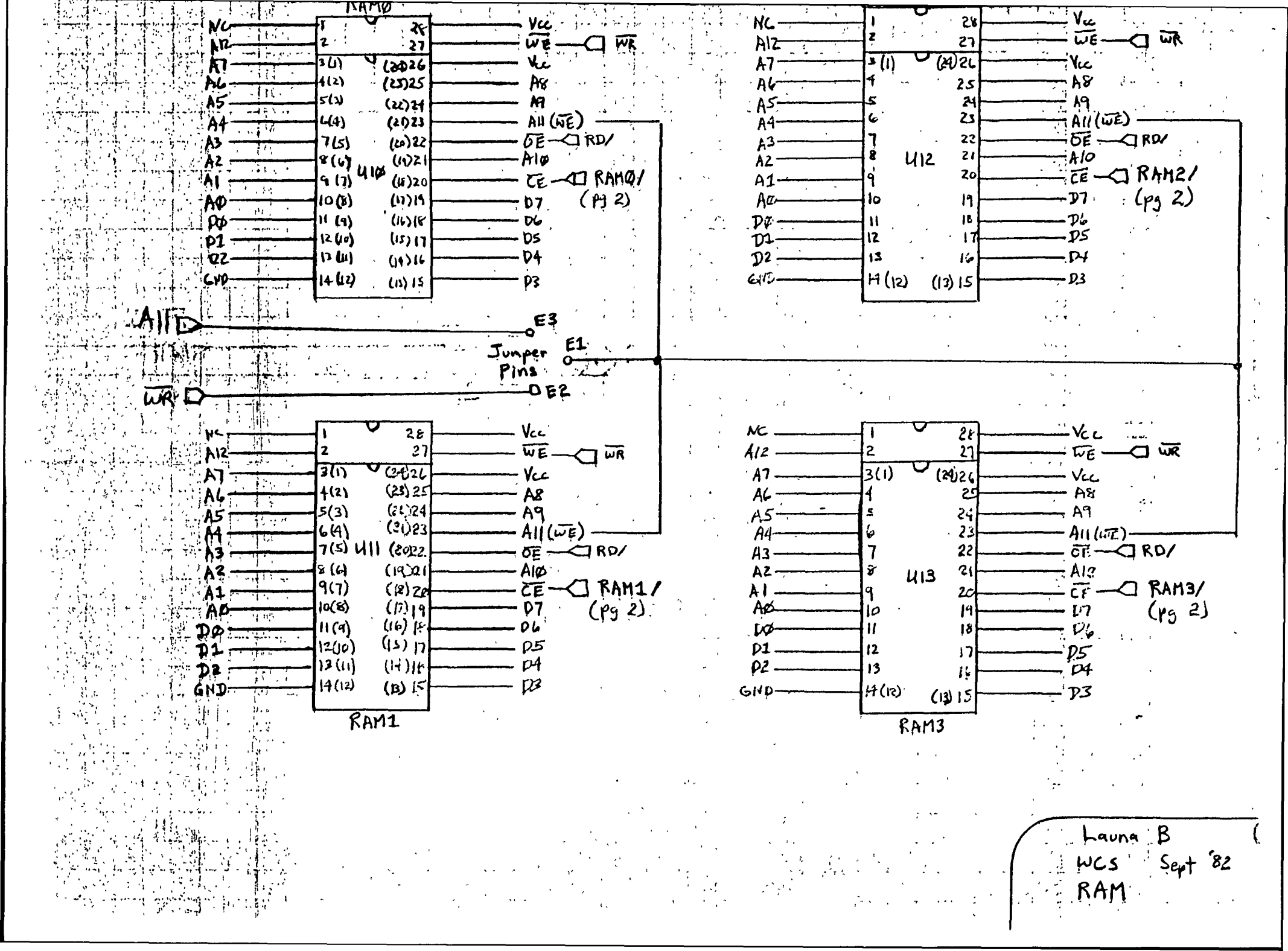
E. Spare Gates

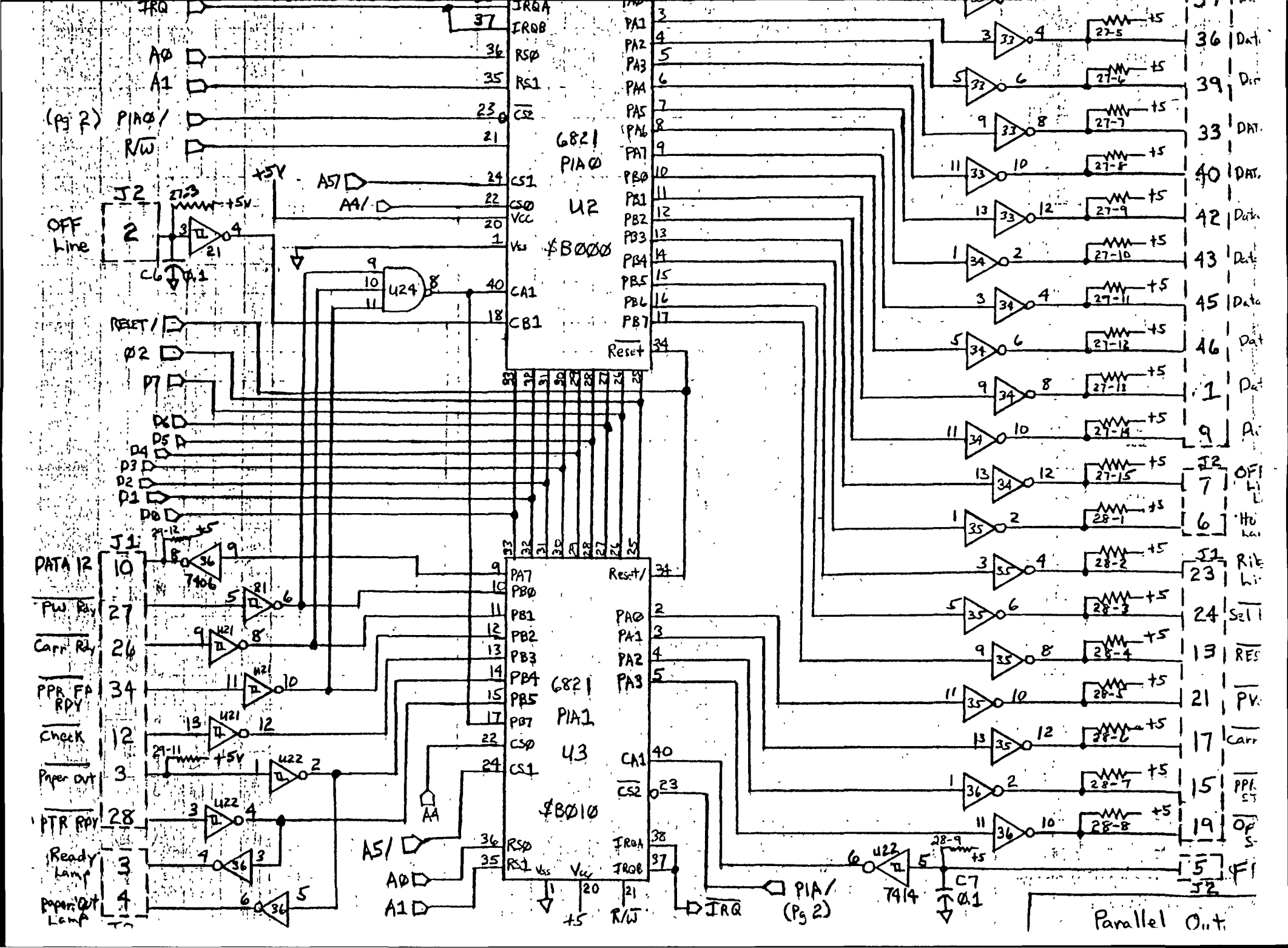
Des	Part	Spares
U22	74LS14	2
U23	7404	1
U24	7410	1
U25	7400	2
U26	7432	2
U29	Resistor Pack	3
U36	7416	1

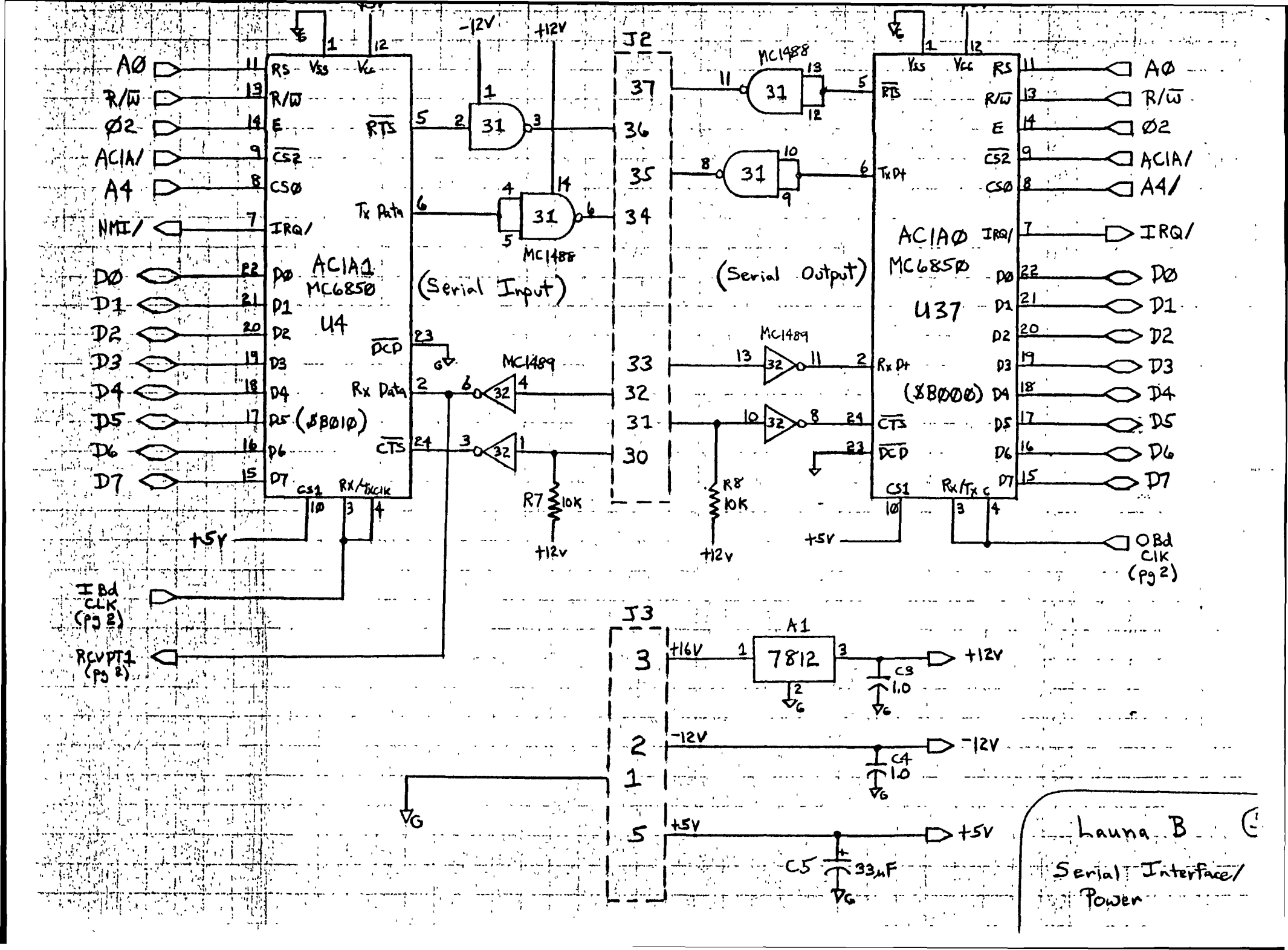


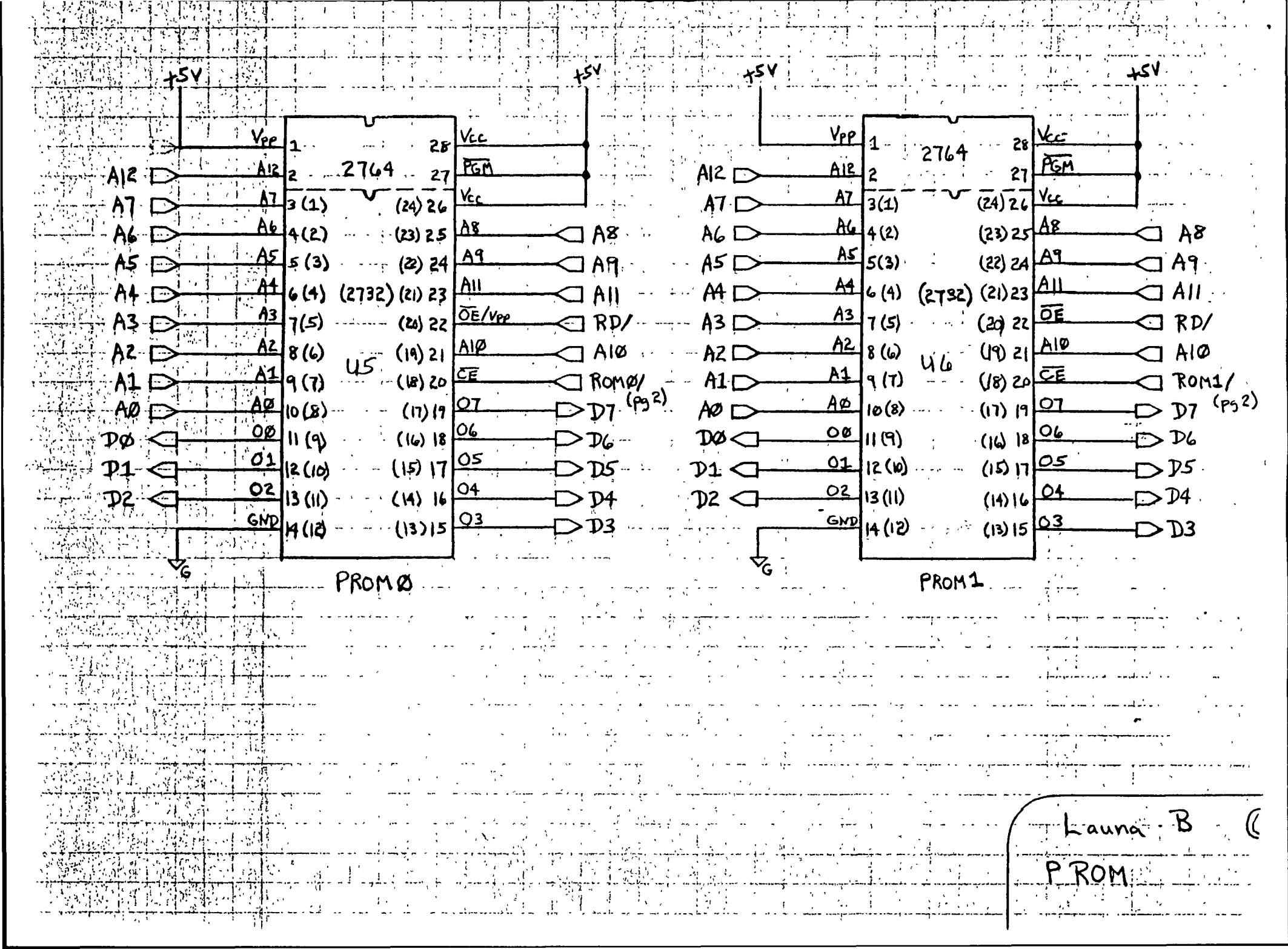
Launa B
WCS March '83
p.

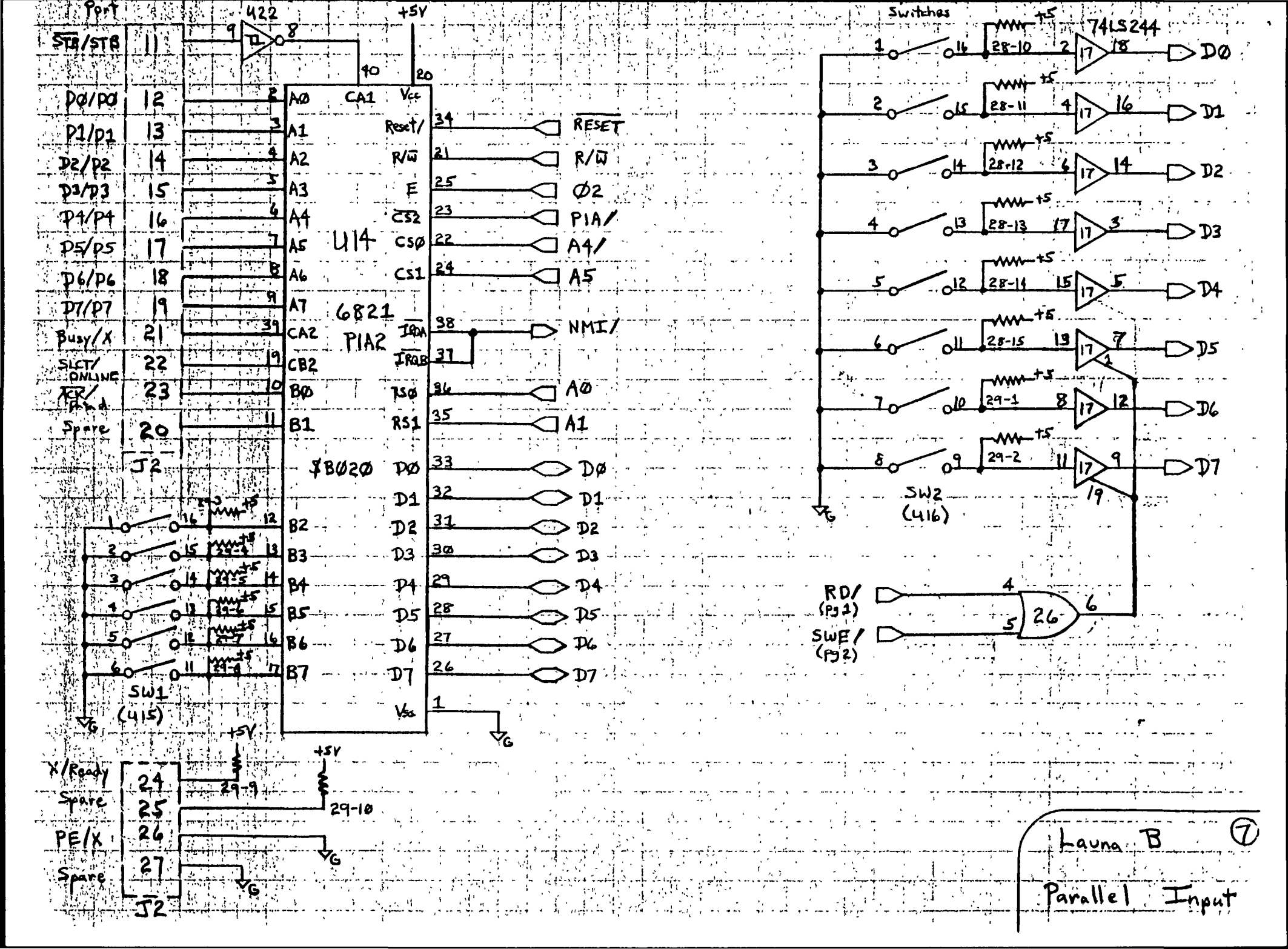

Upper Address Decode
A14
A13
A12
A15
(Pg 1)
426
74LS138
U7
G2A
G2B
C
B
A
Y7
Y6
Y5
Y4
Y3
Y2
Y1
Y0
Vcc
GND
G1
+5V
$E000
$E000
$D000
$C000
$B000
$A000
$9000
$8000
ROM1/
ROM0/
Pg 6
PIA / 3Pg 4,7
SWE/ (Pg7)
ACIA/
(Pg 5)
AUTOBD
RCVDT1
424
23
IBD
CLK
(Pg 5)
OBd
CLK
+5V
NMI/
6840
PTM
($A010)
U9
Vcc
IRQ
G3
Q3
C3
G2
Q2
C2
G1
Q1
C1
Enable
CS1
CS0
Vss
RESET
D7
D6
D5
D4
D3
D2
D1
D0
R/W
RS2
RS1
RS0
Pg 2
(Pg 1)
Ø2
A4
RESET
(Pg 1)
+5
GND
Vcc
82S129
U8
A0
A1
A2
A3
A4
A5
A6
A7
O1
O2
O3
O4
CE1
CE2
RAM0/
RAM1/
RAM2/
RAM3/
Pg 3
A11
A12
A13
A14
Pg 1
SW1
27-1
27-2
A15
(Pg 1)
Lower Address
Decode
Launa B
WCS Sept '82
Address Decode/
P 1 of 1


RAM0
U10
NC
A12
A7
A6
A5
A4
A3
A2
A1
A0
D0
D1
D2
GND
Vcc
WE
WR
A8
A9
A11 (WE)
OE
RD/
A10
CE
RAM0/
(pg 2)
D7
D6
D5
D4
D3
U12
RAM2/
(pg 2)
A11
E3
Jumper
Pins
E1
E2
U11
RAM1/
(pg 2)
RAM1
U13
RAM3/
(pg 2)
RAM3
Launa B
WCS Sept '82
RAM


IRQ
A0
A1
(Pg 2) PIA0/
R/W
A5/
A4/
+5V
J2
OFF Line
C6
Ø1
RESET/
Ø2
P7
P6
P5
P4
P3
P2
P1
P0
J1
DATA 12
PW Rdy
Carr Rdy
PPR FP RDY
Check
Paper out
PTR RDY
Ready Lamp
Paper Out Lamp
7406
U21
U22
U24
6821 PIA0 U2 $B000
6821 PIA1 U3 $B010
IRQA
IRQB
RS0
RS1
CS2
CS1
CS0
VCC
Vss
CA1
CB1
Reset
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
R/W
IRQ
PIA/ (Pg 2)
7414
C7 Ø1
Parallel Out


A0
R/W
Ø2
ACIA/
A4
NMI/
D0
D1
D2
D3
D4
D5
D6
D7
ACIA1
MC6850
U4
RS
R/W
E
CS2
CSØ
IRQ/
Vss
Vcc
RTS
Tx Data
DCD
Rx Data
CTS
CS1
RX/TXCLK
(SBØ1Ø)
+5V
IBd CLK (Pg 2)
RCVPT1 (Pg 2)
-12V
+12V
J2
37
36
35
34
33
32
31
30
MC1488
MC1489
(Serial Input)
(Serial Output)
R7 10K
R8 10K
ACIAØ
MC6850
U37
TxDt
RxDt
(SBØØØ)
OBd CLK (Pg 2)
J3
A1
7812
+16V
C3 1.0
C4 1.0
C5 33µF
Launa B
Serial Interface/
Power


+5V
+5V
Vpp
A12
A7
A6
A5
A4
A3
A2
A1
A0
O0
O1
O2
GND
G
1
2
3 (1)
4 (2)
5 (3)
6 (4)
7 (5)
8 (6)
9 (7)
10 (8)
11 (9)
12 (10)
13 (11)
14 (12)
2764
(2732)
U5
28
27
(24) 26
(23) 25
(22) 24
(21) 23
(20) 22
(19) 21
(18) 20
(17) 19
(16) 18
(15) 17
(14) 16
(13) 15
Vcc
PGM
Vcc
A8
A9
A11
OE/Vpp
A10
CE
O7
O6
O5
O4
O3
A8
A9
A11
RD/
A10
ROM0/
(pg 2)
D7
D6
D5
D4
D3
PROM 0
+5V
+5V
Vpp
A12
A7
A6
A5
A4
A3
A2
A1
A0
O0
O1
O2
GND
G
1
2
3(1)
4(2)
5(3)
6(4)
7(5)
8(6)
9(7)
10(8)
11(9)
12(10)
13(11)
14(12)
2764
(2732)
U6
28
27
(24) 26
(23) 25
(22) 24
(21) 23
(20) 22
(19) 21
(18) 20
(17) 19
(16) 18
(15) 17
(14) 16
(13) 15
Vcc
PGM
Vcc
A8
A9
A11
OE
A10
CE
O7
O6
O5
O4
O3
A8
A9
A11
RD/
A10
ROM1/
(pg 2)
D7
D6
D5
D4
D3
PROM 1
Launa · B
PROM


Port
U22
+5V
STB/STB
11
P0/P0
12
P1/P1
13
P2/P2
14
P3/P3
15
P4/P4
16
P5/P5
17
P6/P6
18
P7/P7
19
Busy/X
21
SLCT/ ONLINE
22
ACK/ Pout
23
Spare
20
J2
U14
6821
PIA2
$B020
A0
A1
A2
A3
A4
A5
A6
A7
CA2
CB2
B0
B1
B2
B3
B4
B5
B6
B7
CA1
Vcc
Reset/
R/W
E
CS2
CS0
CS1
IRQA
IRQB
RS0
RS1
D0
D1
D2
D3
D4
D5
D6
D7
Vss
RESET
R/W
Ø2
PIA/
A4/
A5
NMI/
A0
A1
D0
D1
D2
D3
D4
D5
D6
D7
SW1
(415)
X/Ready
Spare
PE/X
Spare
J2
29-9
29-10
Switches
74LS244
28-10
28-11
28-12
28-13
28-14
28-15
29-1
29-2
SW2
(416)
RD/
(PJ1)
SWE/
(PJ2)
26
Launa B
Parallel Input
7

Final Report

Contract NAS8-34969

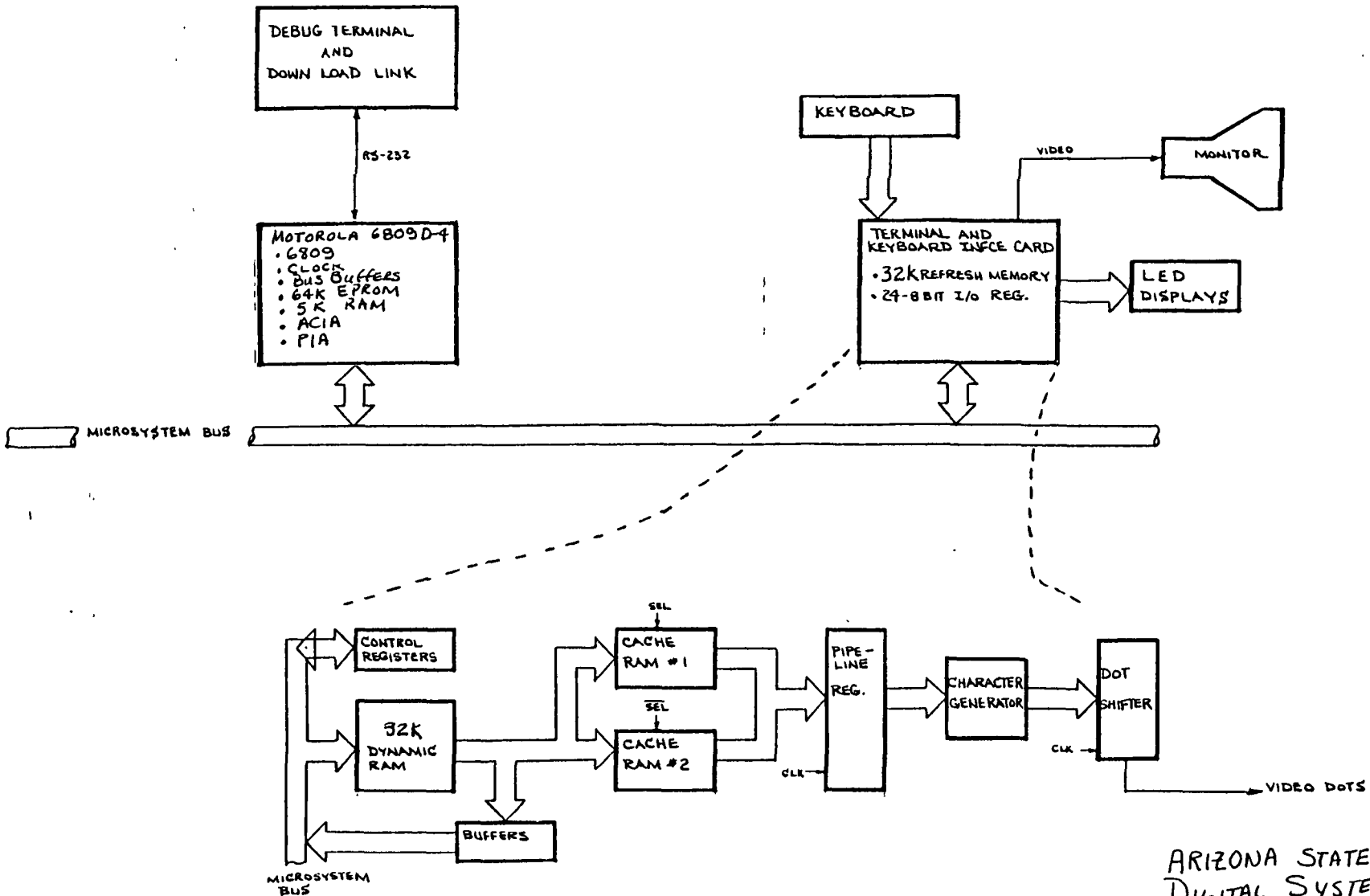
ENHANCEMENT OF INTELLIGENT EDITOR/PRINTER

This report was prepaired by Arizona State University under contract NAS8-34969 Enhancement of Intelligent Editor/Printer for Marshall Space Fligh Center of the National Aernautics and Space Administration.

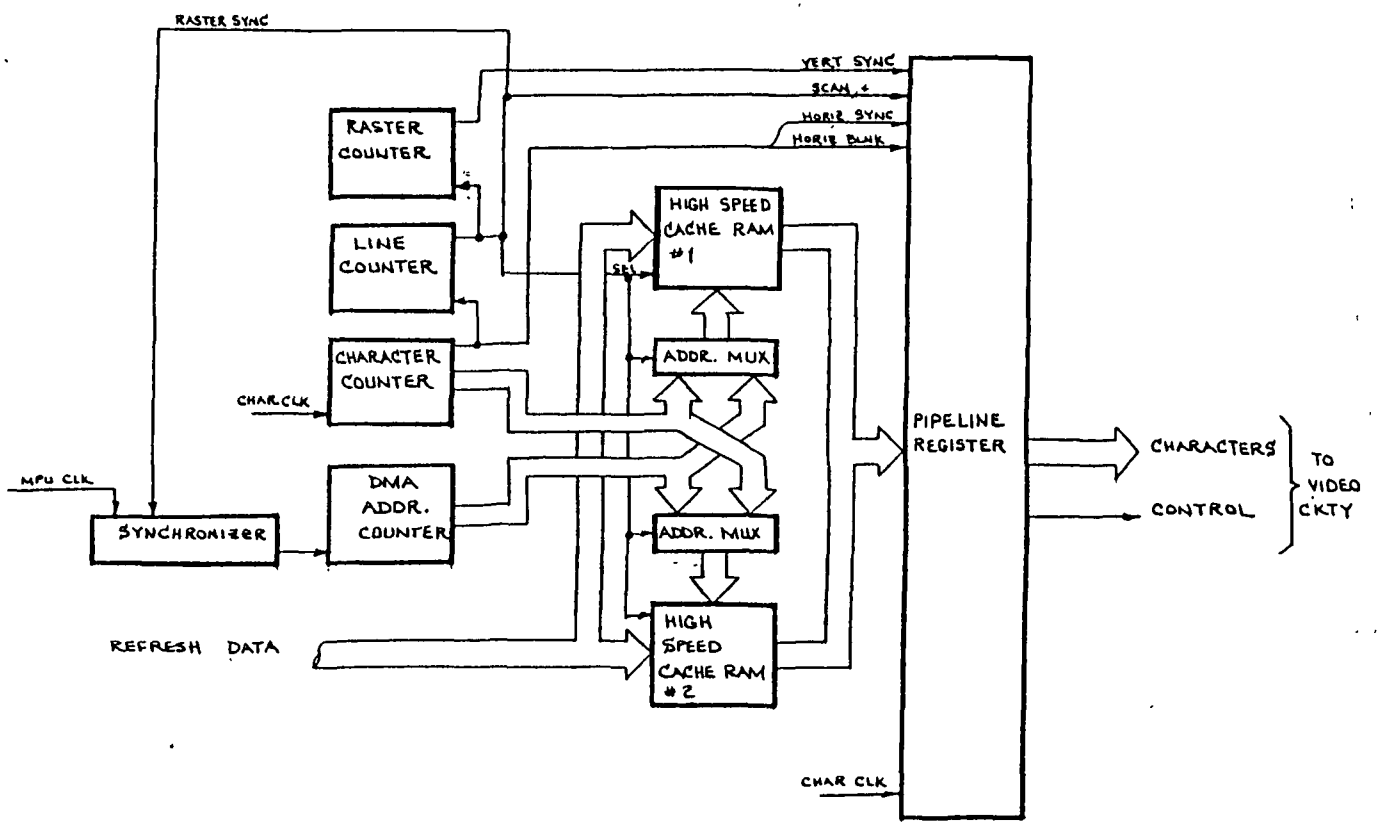
A P P E N D I X

C

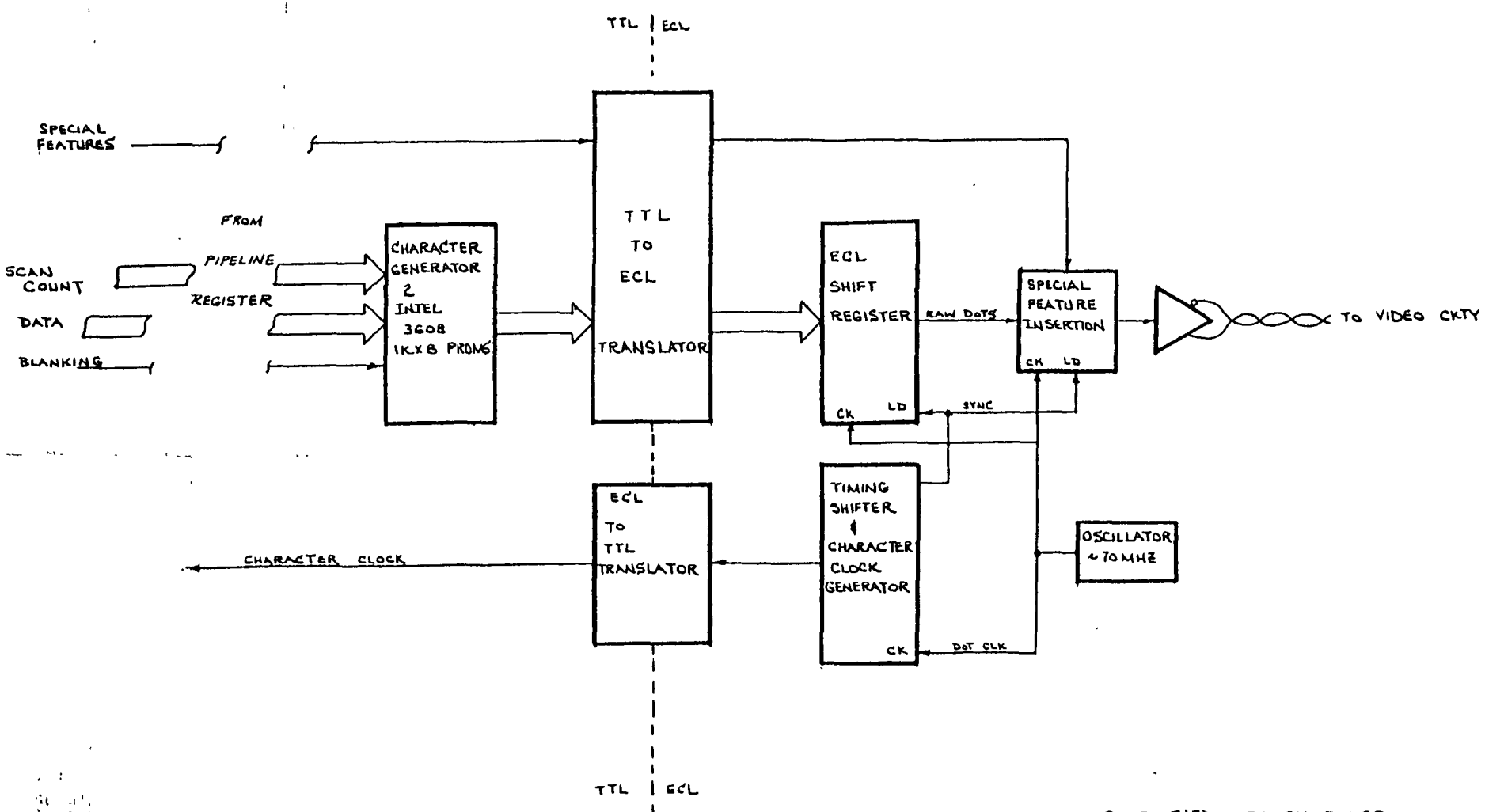
The design specification for a 132 Character by 64 Line intelligent CRT display system using a Motorola 6809 MPU. This version also has four pages of refresh memory.



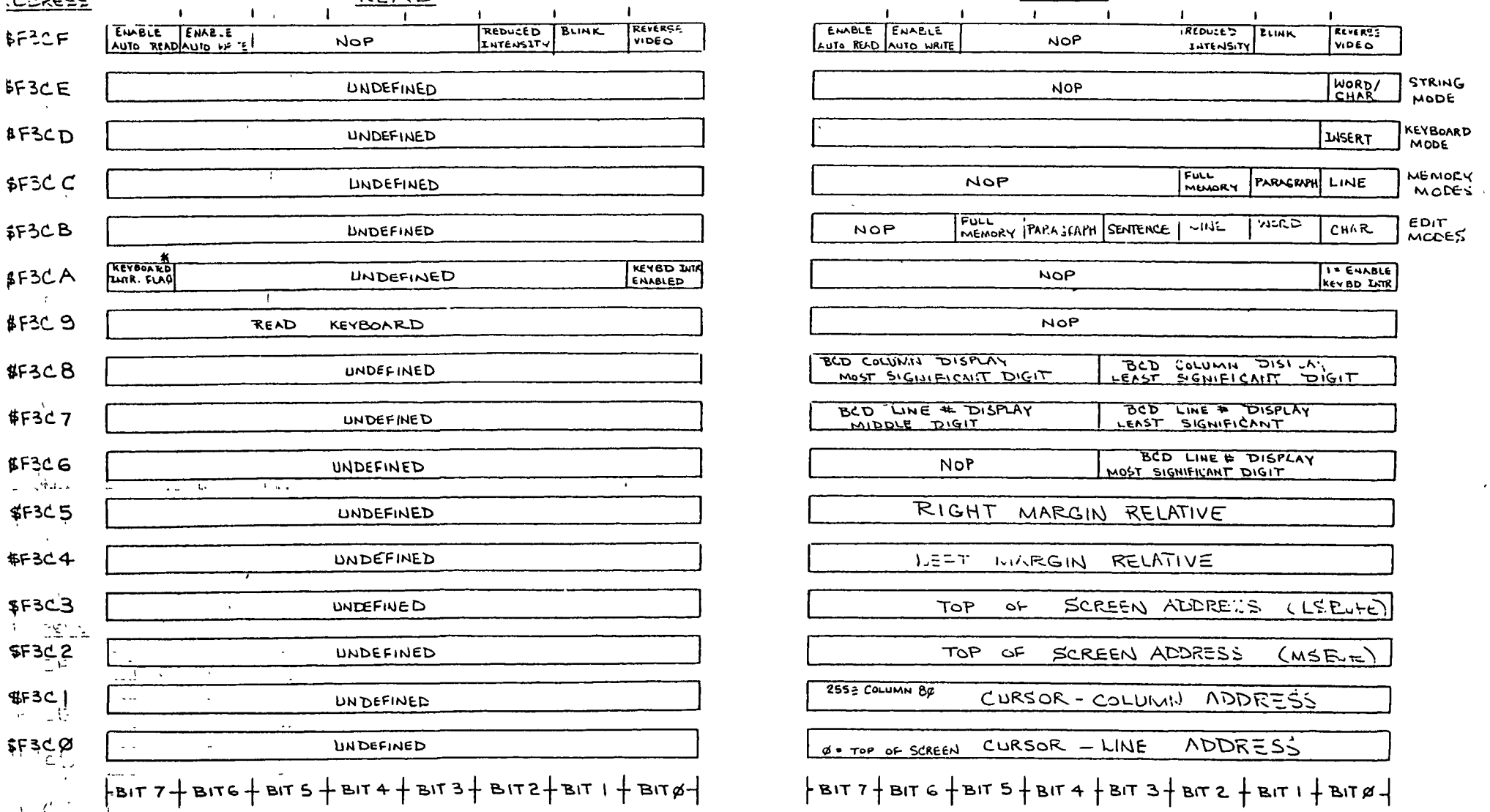
ARIZONA STATE UNIVERSITY
DIGITAL SYSTEMS LAB
132 CHAR X 64 LINE VIDEO DISPLAY SYSTEM USING MOTOROLA 6809 MPU WITH 4 PAGE REFRESH MEMORY



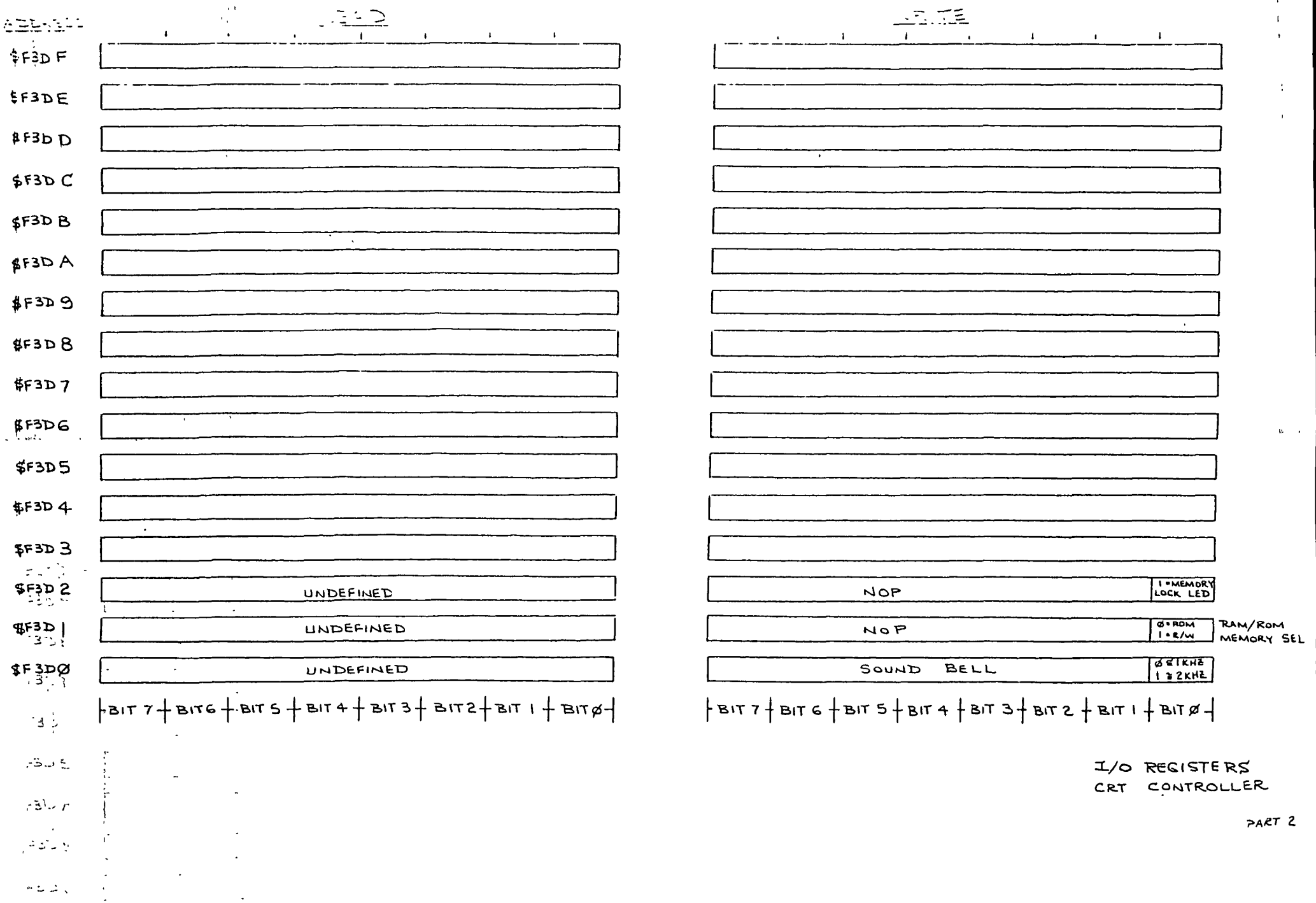
BLOCK DIAGRAM
CACHE MEMORY SYSTEM
CRT CONTROLLER



SIMPLIFIED BLOCK DIAGRAM
CHARACTER GENERATION CIRCUITRY

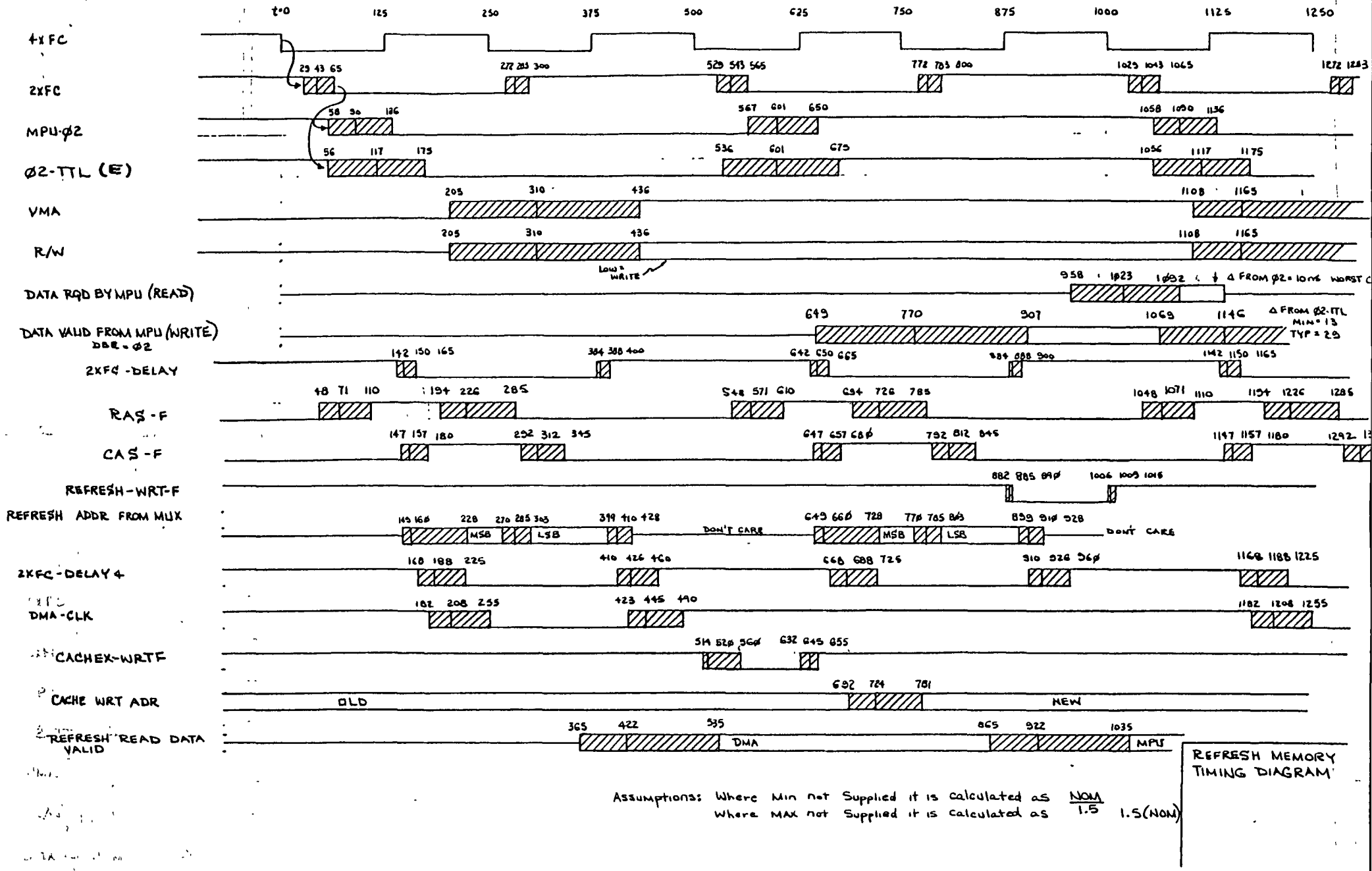


I/O REGISTERS
CRT. CONTROLLER



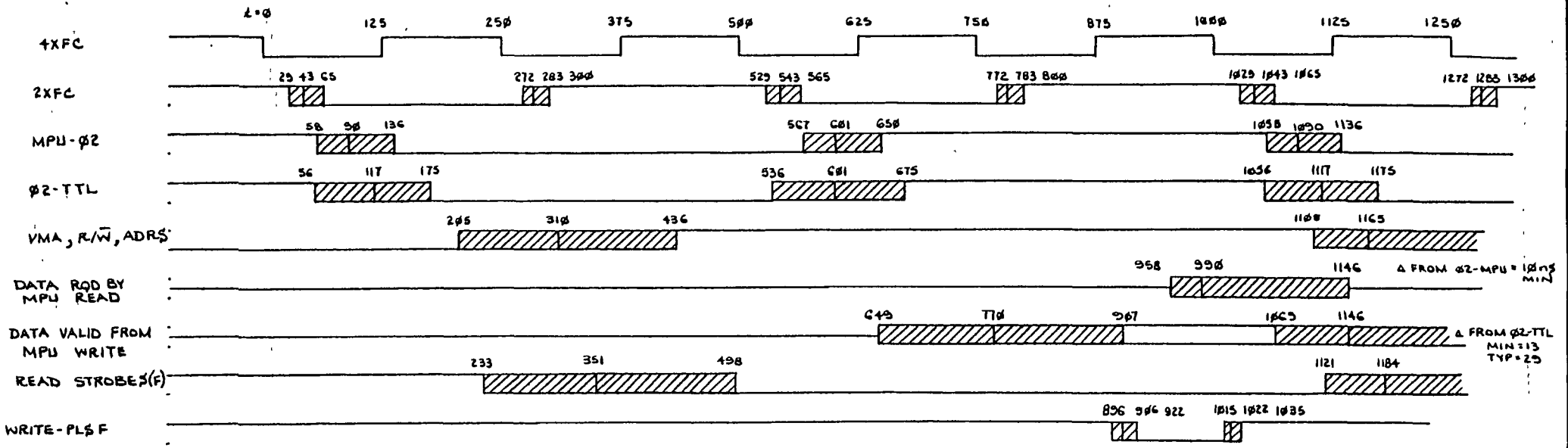
I/O REGISTERS
CRT CONTROLLER

PART 2

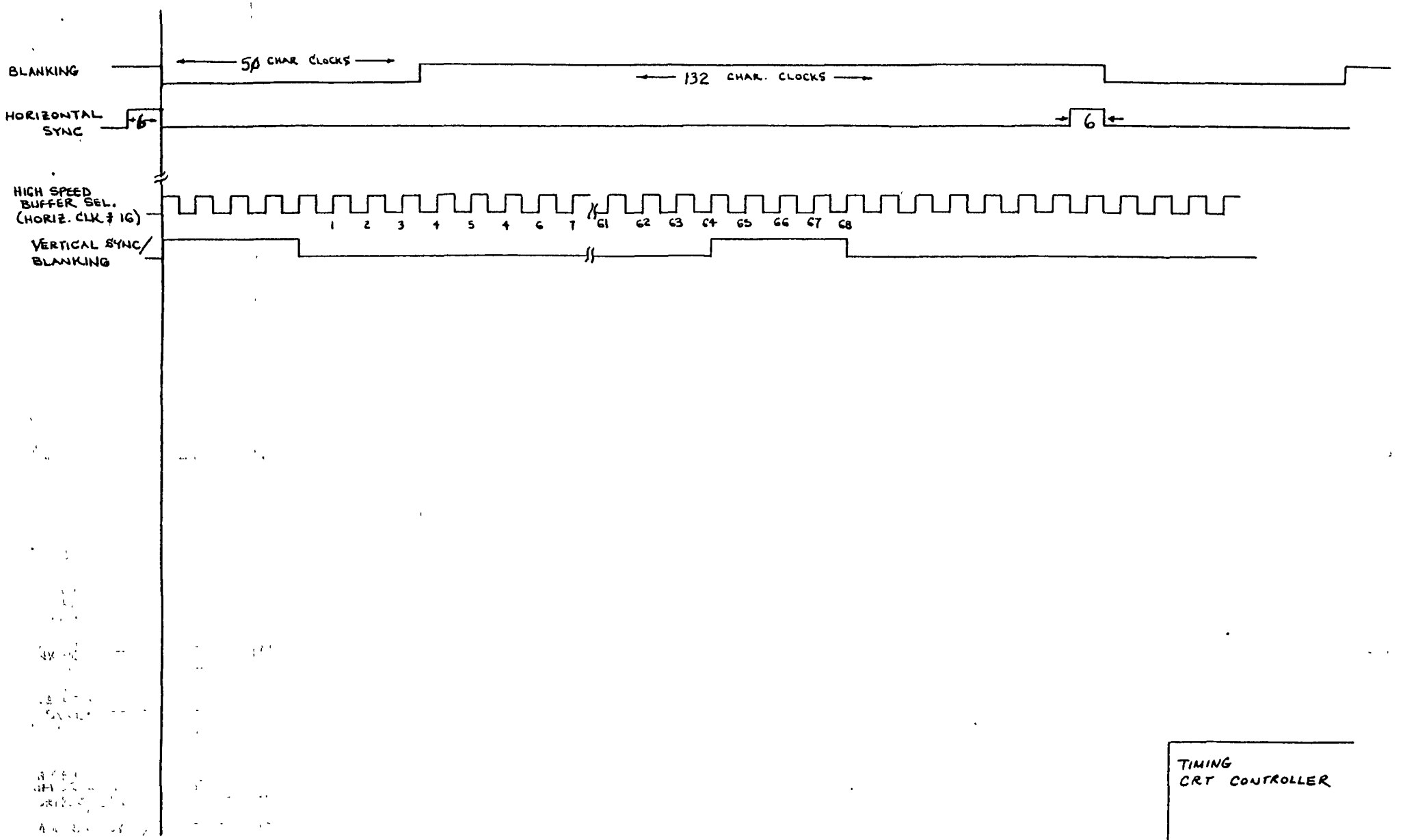


Assumptions: Where Min not Supplied it is calculated as $\frac{NOM}{1.5}$
Where MAX not Supplied it is calculated as 1.5(NOM)

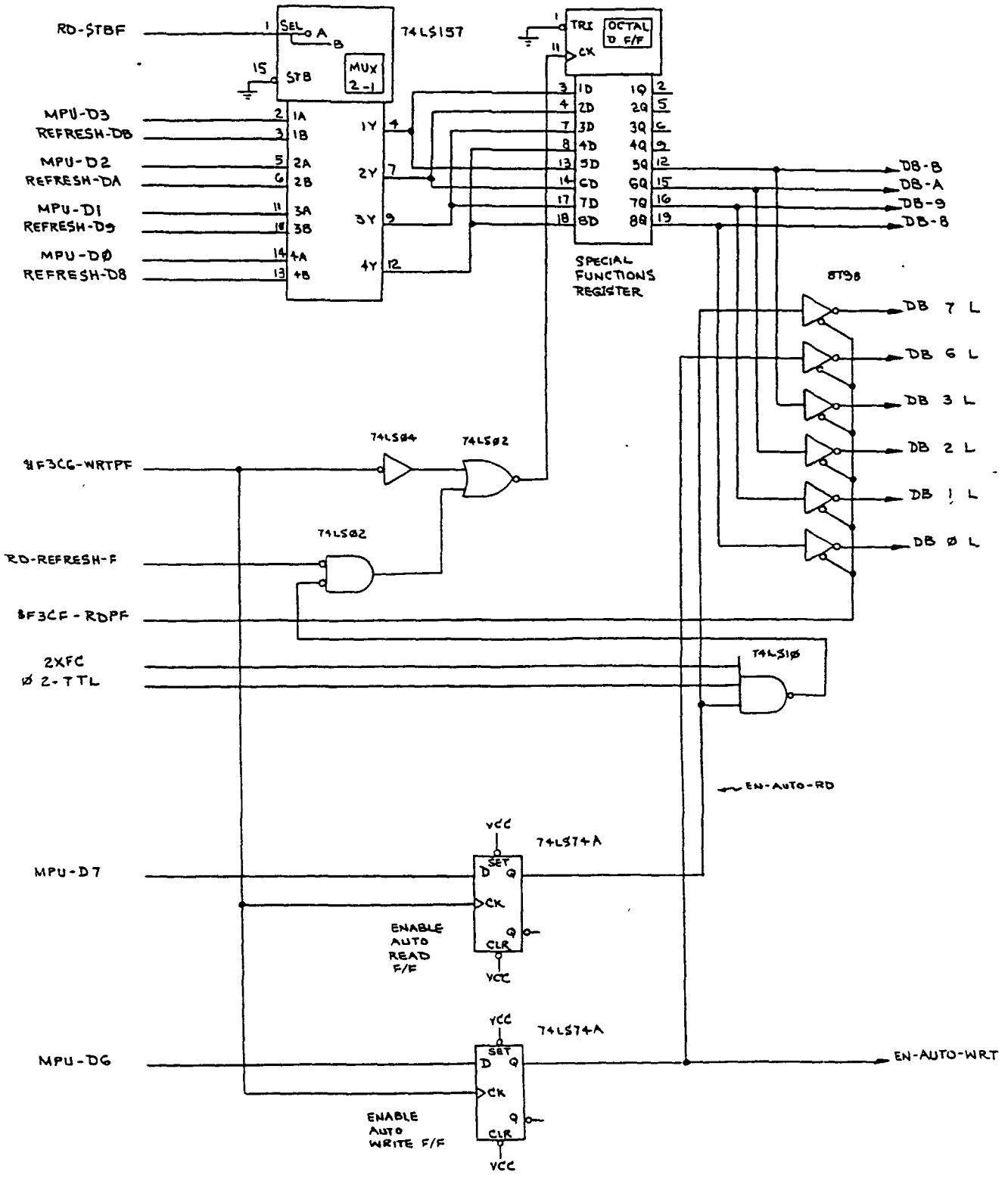
REFRESH MEMORY TIMING DIAGRAM



I/O READ/WRITE TIMING DIAGRAM

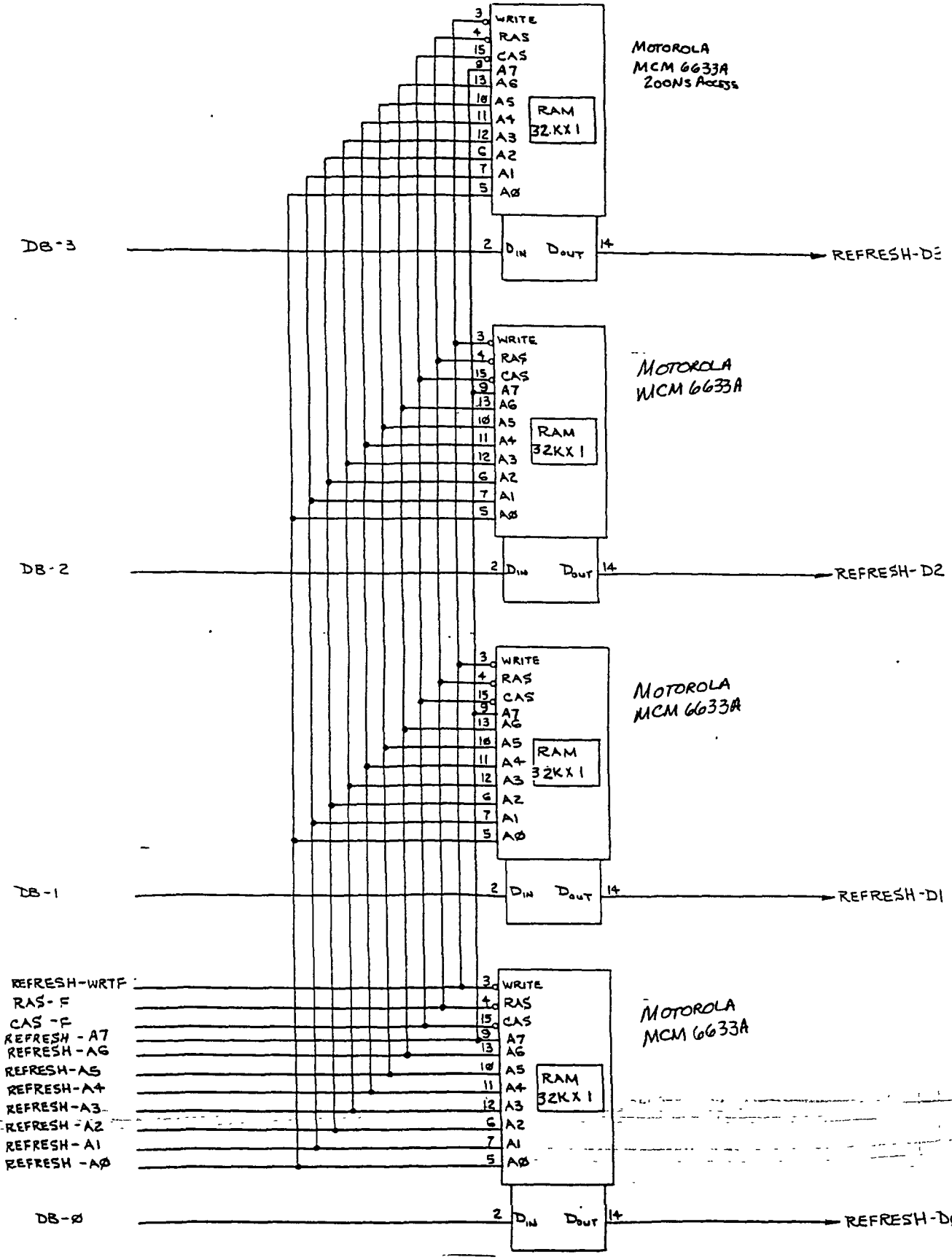


TIMING
CRT CONTROLLER



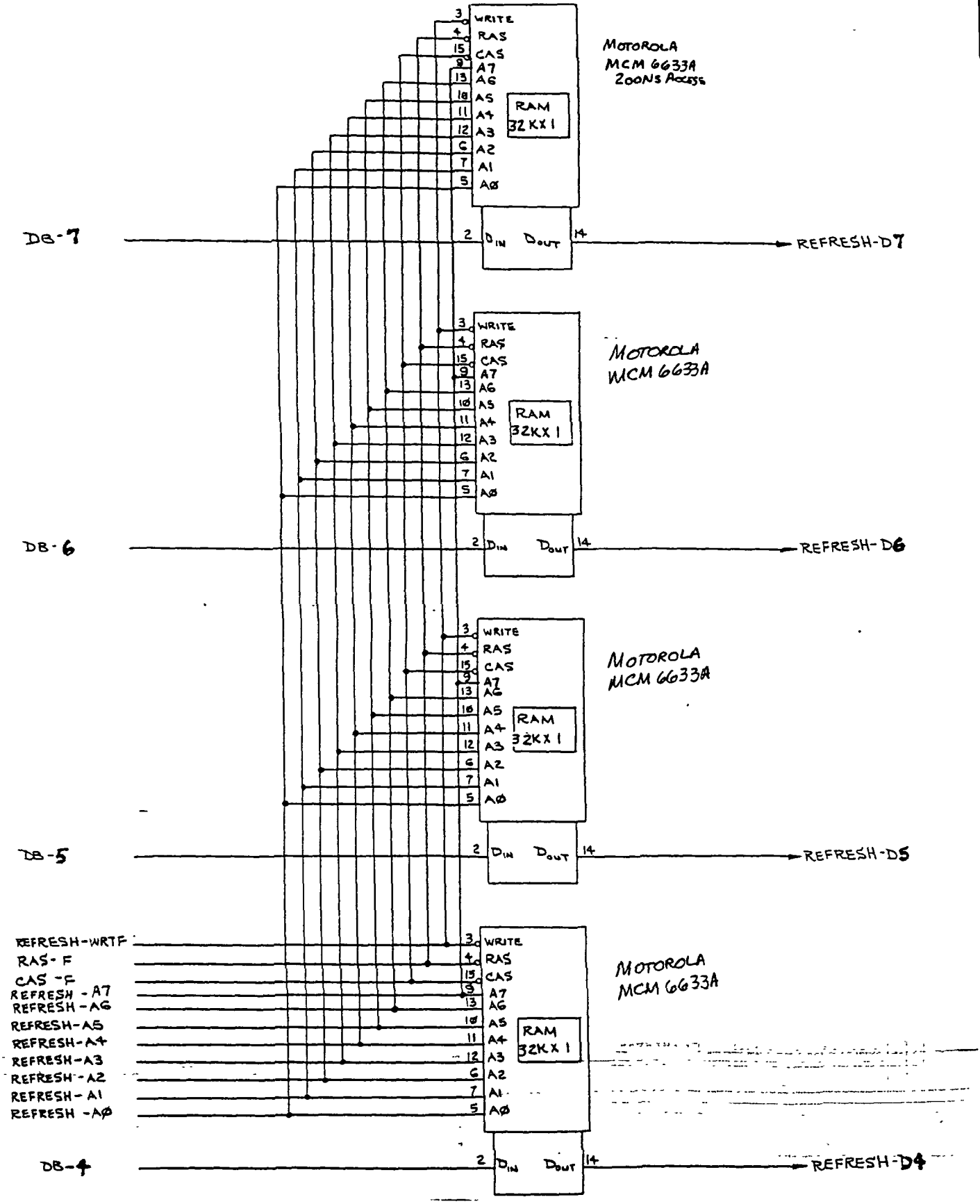
SPECIAL FUNCTION REGISTER

CRT CONTROLLER



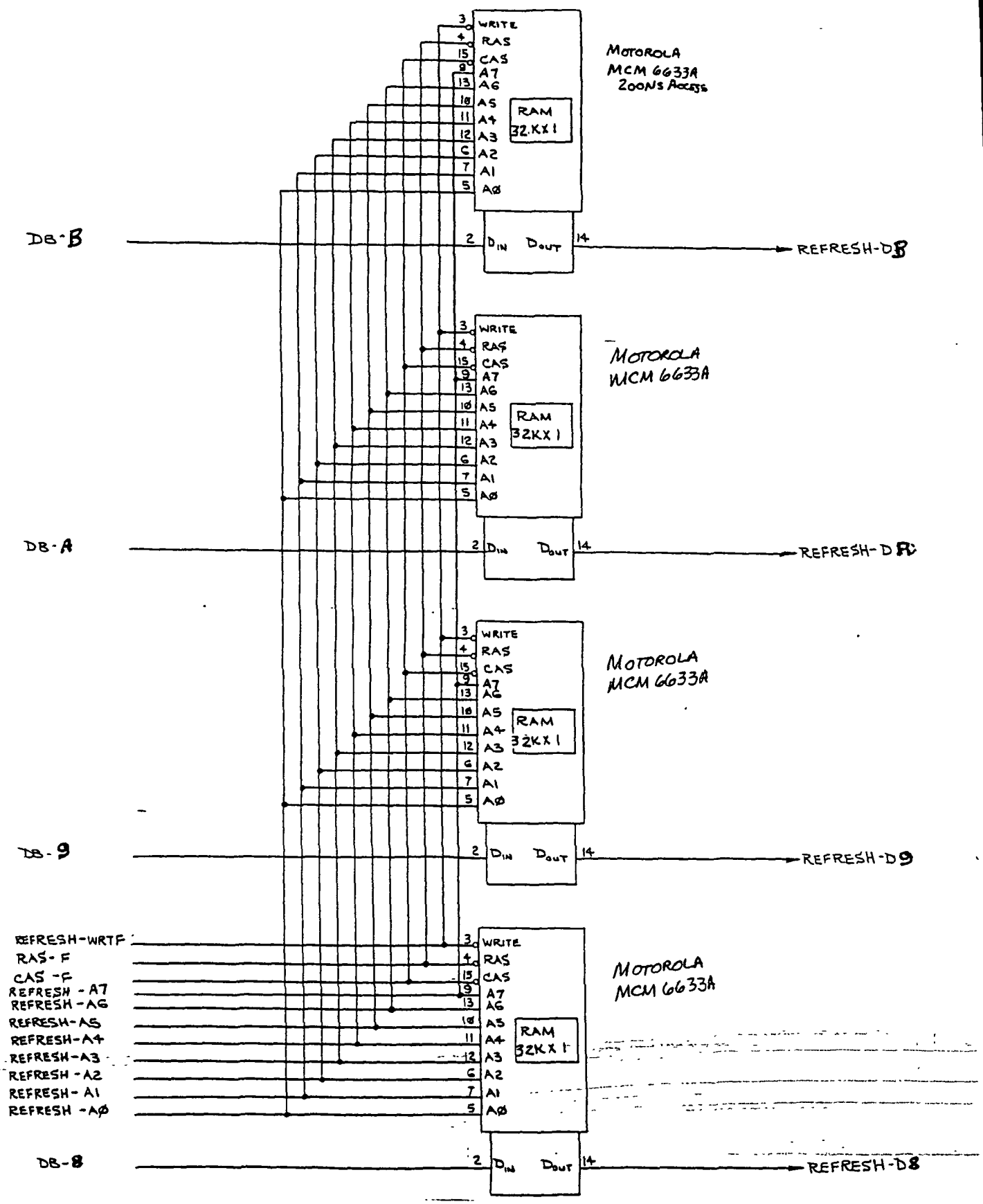
PIN 16 = VSS = GND
PIN 8 = VDD = +5VDC

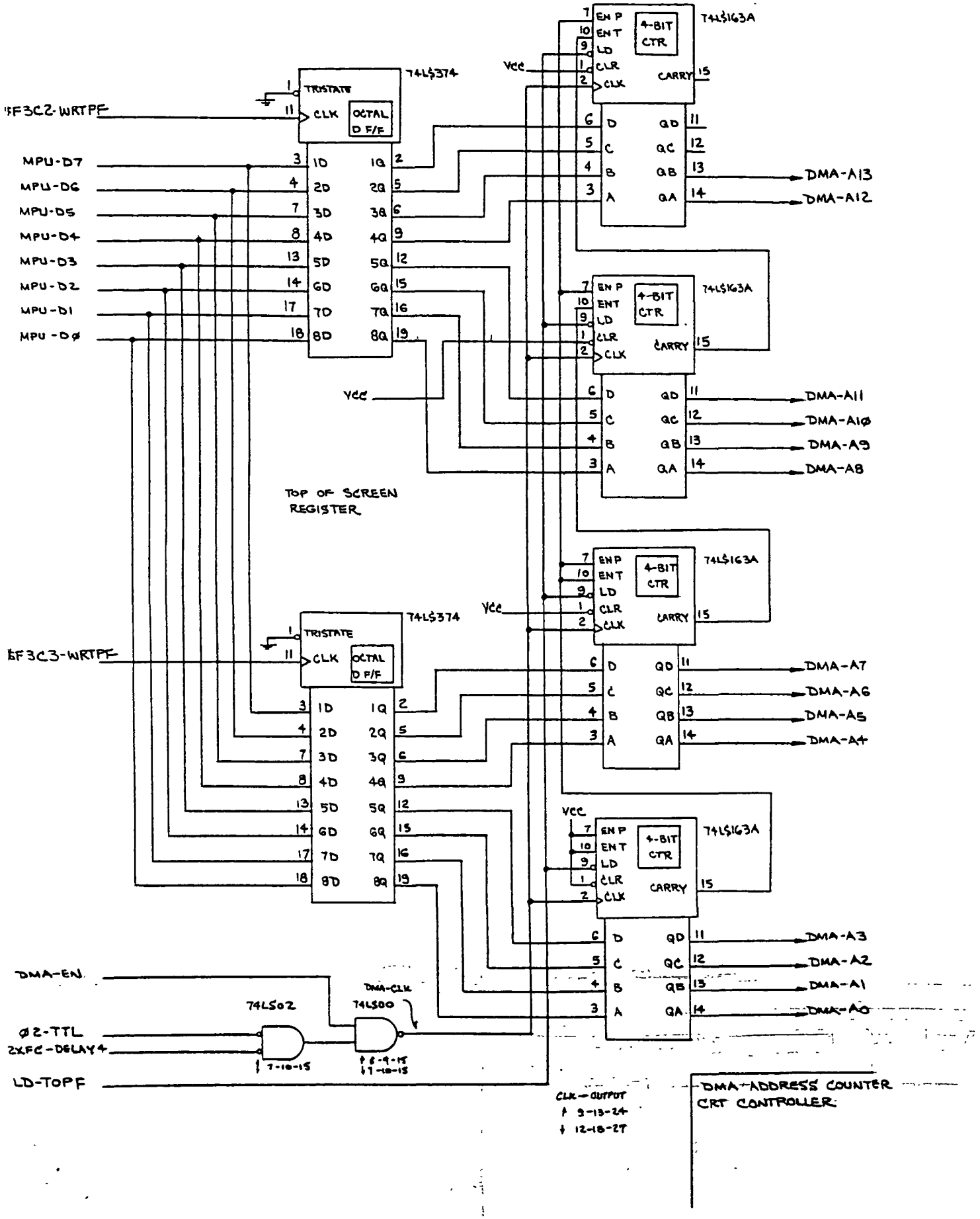
REFRESH MEMORY
CRT CONTROLLER.

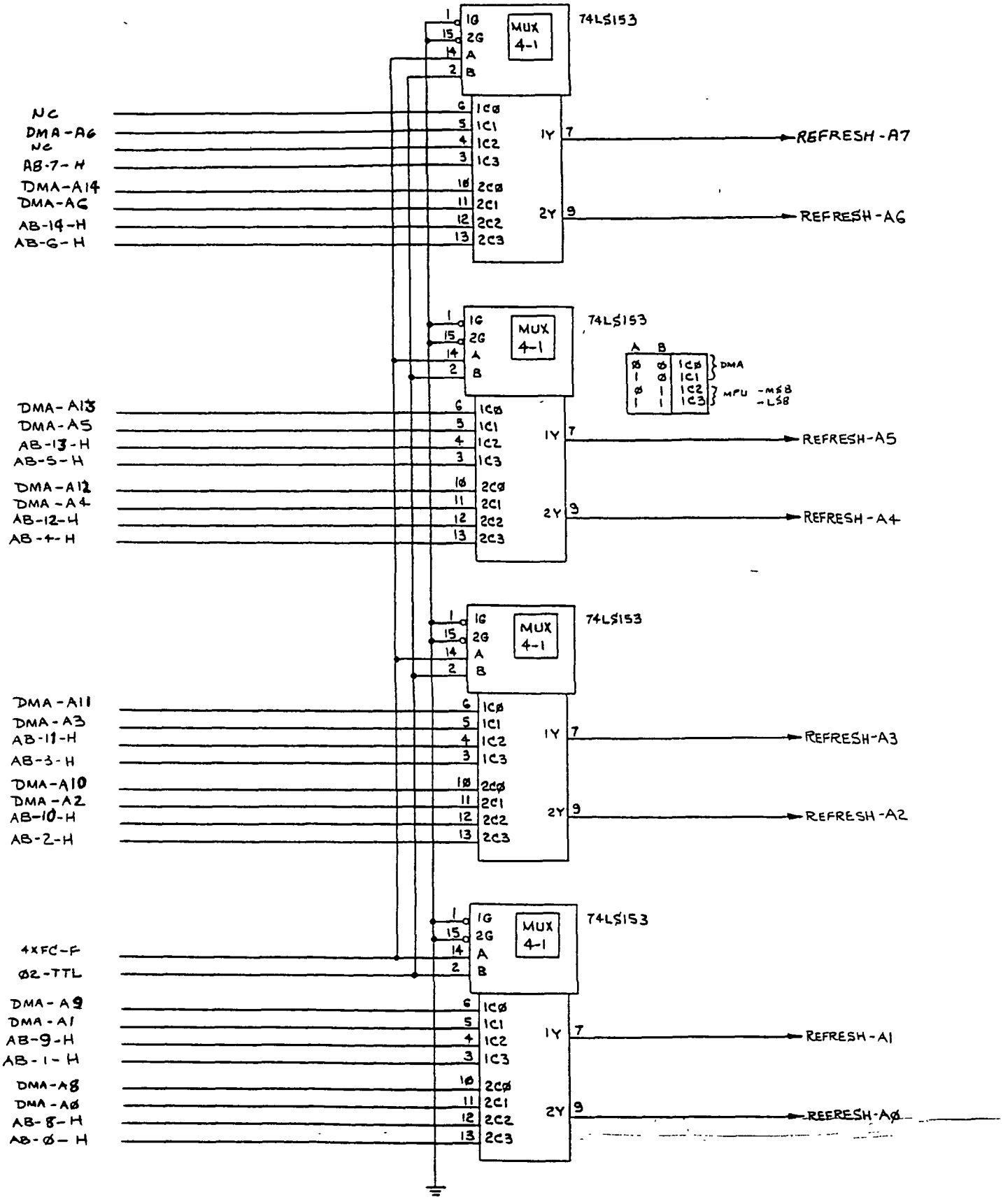


PIN 16 = VSS = GND
PIN 8 = VDD = +5VDC

REFRESH MEMORY
CRT CONTROLLER


WRITE
RAS
CAS
A7
A6
A5
A4
A3
A2
A1
A0
RAM
32KX1
DIN
DOUT
MOTOROLA
MCM 6633A
200NS ACCESS
DB-B
REFRESH-DB
DB-A
REFRESH-DA
DB-9
REFRESH-D9
DB-8
REFRESH-D8
REFRESH-WRTF
RAS-F
CAS-F
REFRESH-A7
REFRESH-A6
REFRESH-A5
REFRESH-A4
REFRESH-A3
REFRESH-A2
REFRESH-A1
REFRESH-A0
PIN 16 = VSS = GND
PIN 8 = VDD = +5VDC
REFRESH MEMORY
CRT CONTROLLER

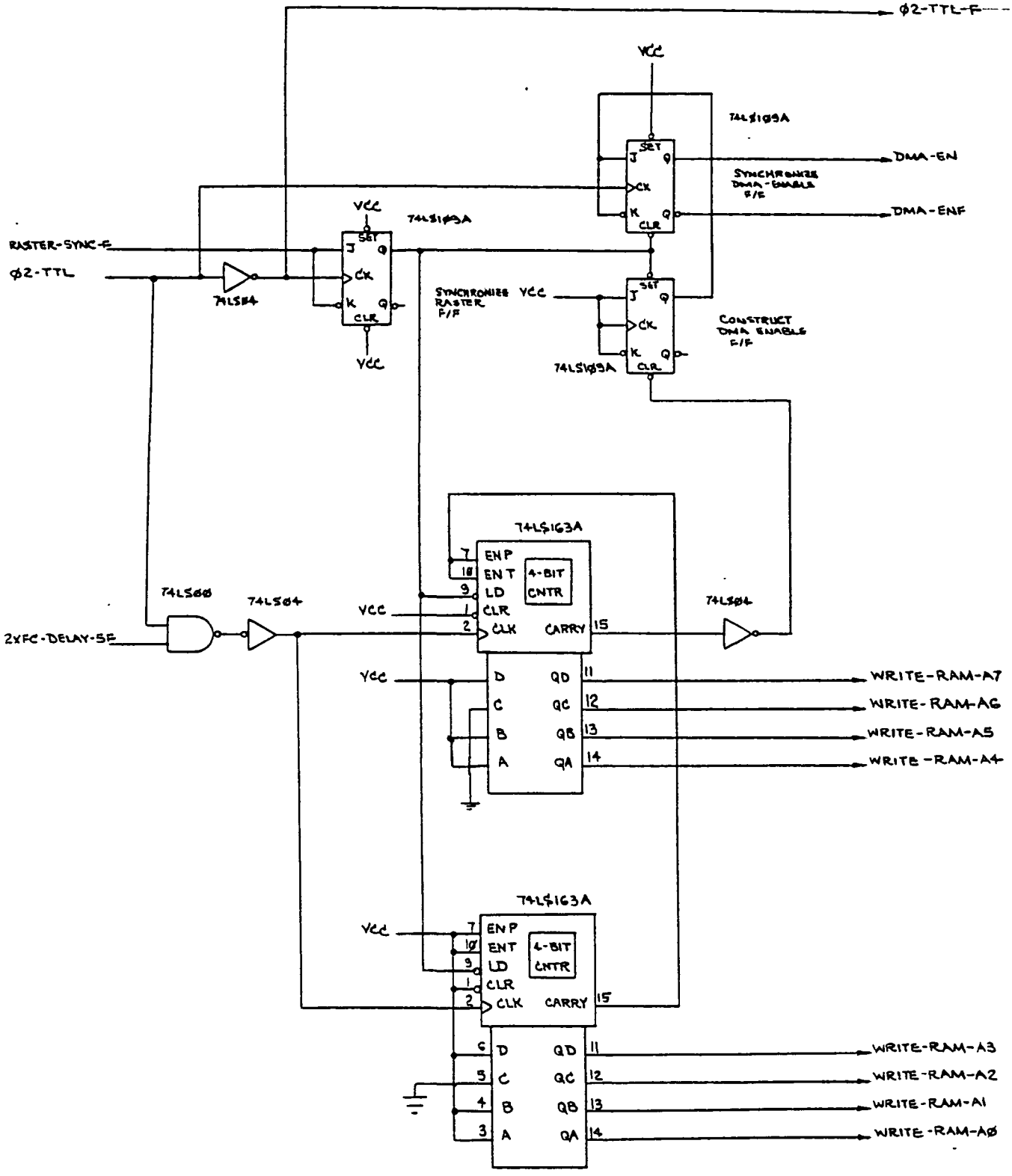

74LS374
TRISTATE
CLK
OCTAL D F/F
1D 2D 3D 4D 5D 6D 7D 8D
1Q 2Q 3Q 4Q 5Q 6Q 7Q 8Q
$FF3C2-WRTPF
MPU-D7
MPU-D6
MPU-D5
MPU-D4
MPU-D3
MPU-D2
MPU-D1
MPU-D0
74LS163A
4-BIT CTR
ENP ENT LD CLR CLK CARRY
A B C D QA QB QC QD
Vcc
DMA-A13
DMA-A12
DMA-A11
DMA-A10
DMA-A9
DMA-A8
TOP OF SCREEN REGISTER
$FF3C3-WRTPF
DMA-A7
DMA-A6
DMA-A5
DMA-A4
DMA-A3
DMA-A2
DMA-A1
DMA-A0
DMA-EN
74LS02
74LS00
DMA-CLK
Ø2-TTL
2XFC-DELAY4
LD-TOPF
CLR-OUTPUT
DMA-ADDRESS COUNTER
CRT CONTROLLER



(MAY REQUIRE S 153 for Capacitive DRIVE)

DATA ↓	11-17-26
DATA ↑	7-10-15
SEL ↑	13-19-29
SEL ↓	17-25-38

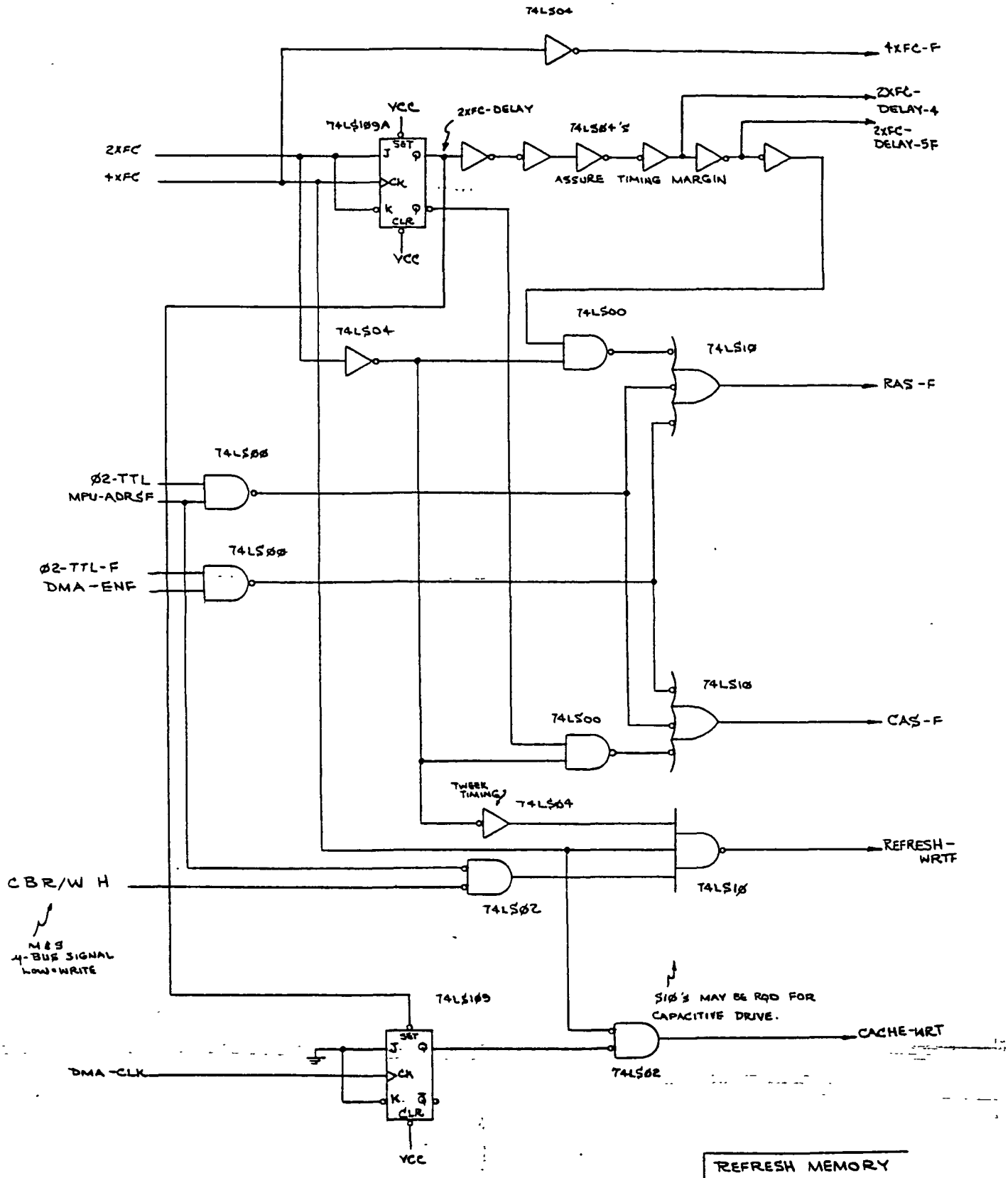
REFRESH MEMORY
ADDRESS MUX
CRT CONTROLLER



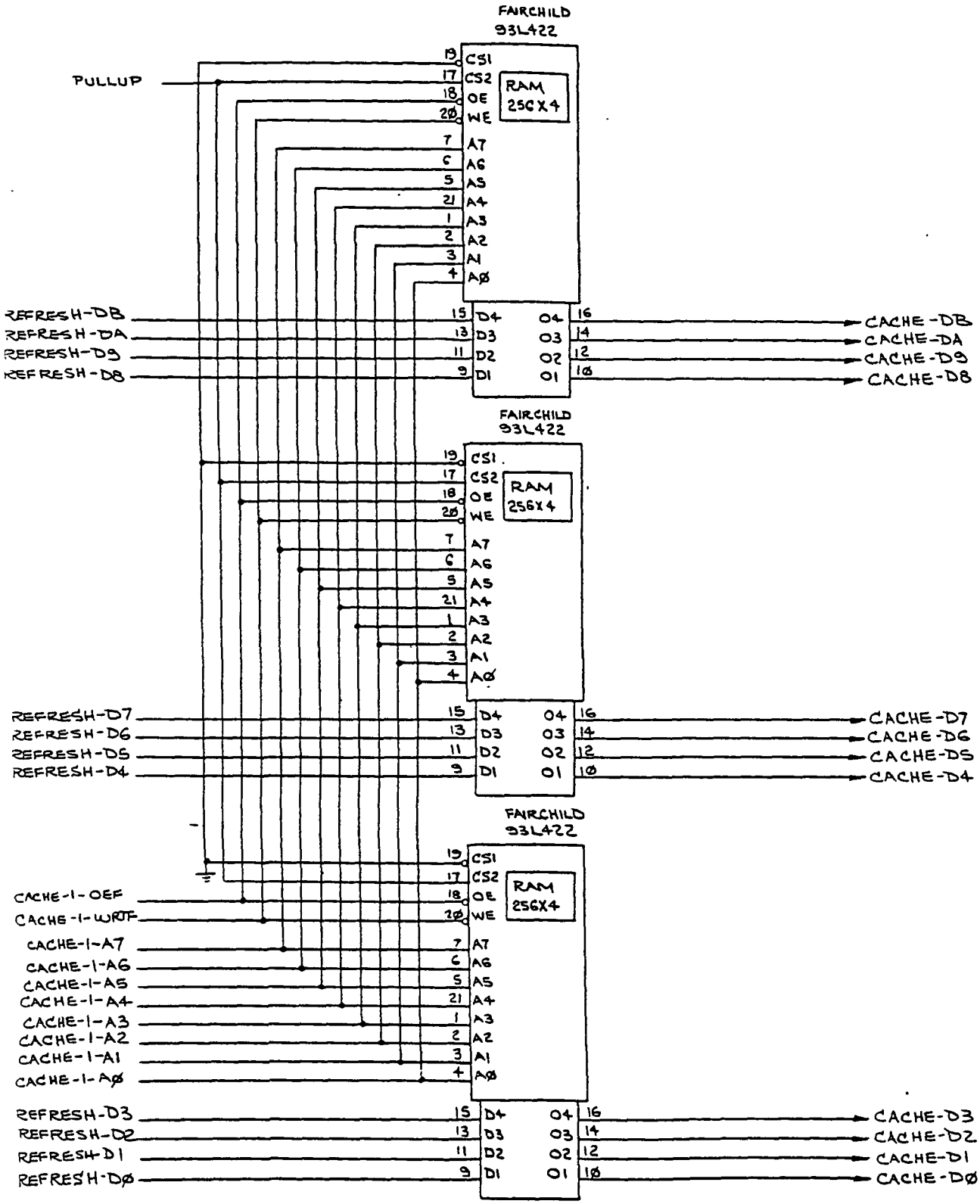
REFRESH MEMORY TO CACHE MEMORY LINE COUNTER

(25C-132) -1 = 123

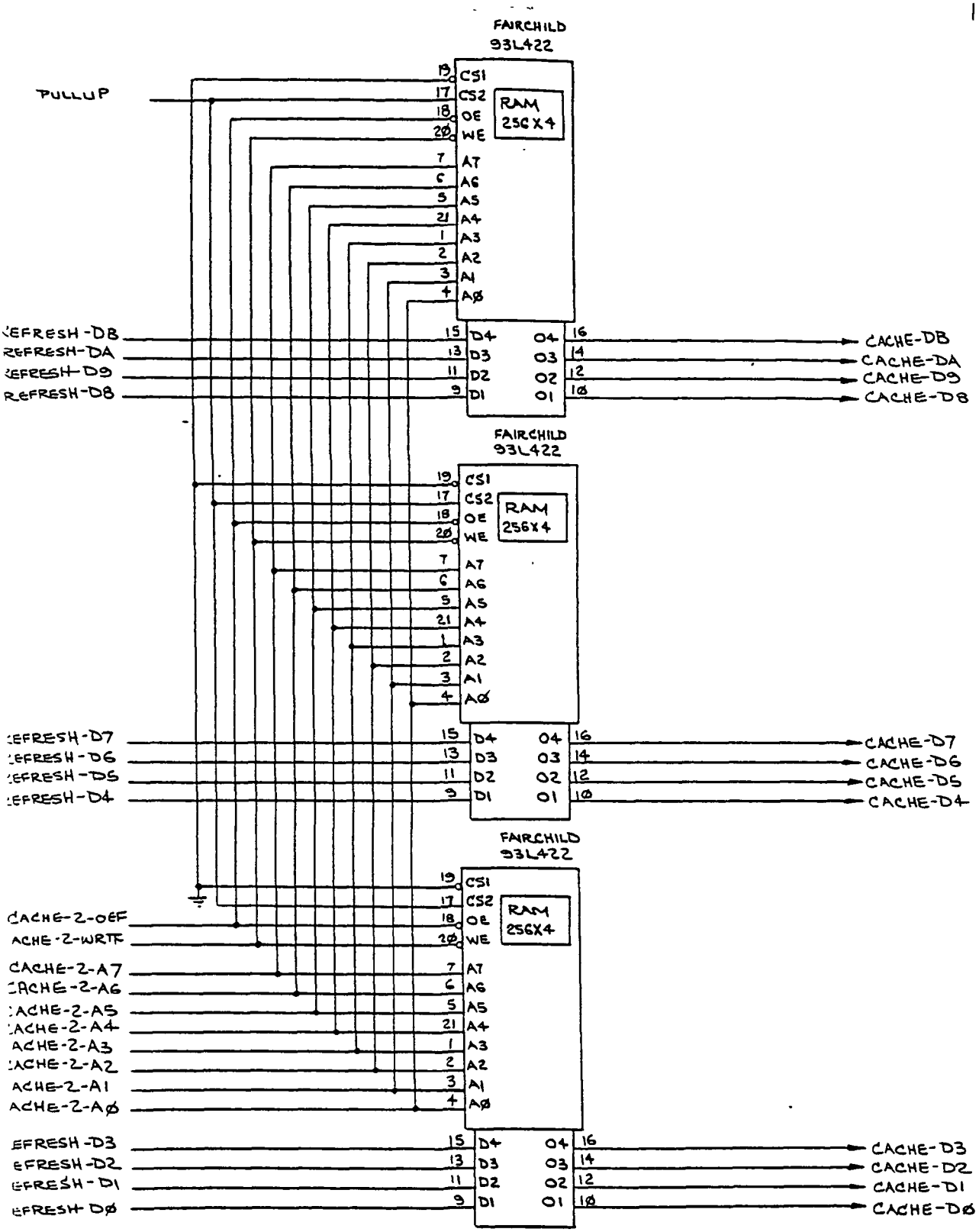
GETS CLOCKED BEFORE DMA-EN



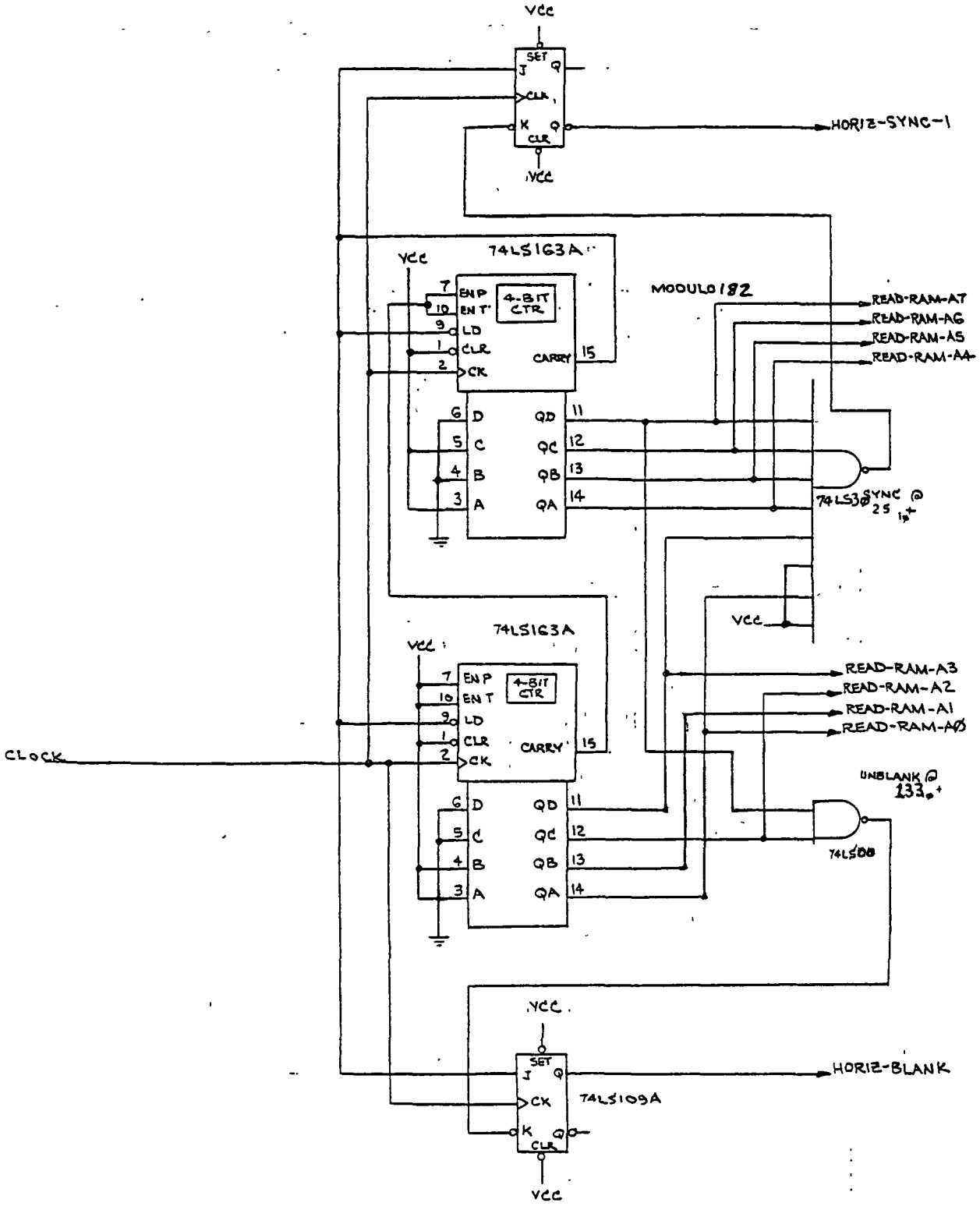
REFRESH MEMORY TIMING LOGIC



CACHE HI SPEED RAM
CRT CONTROLLER
#1

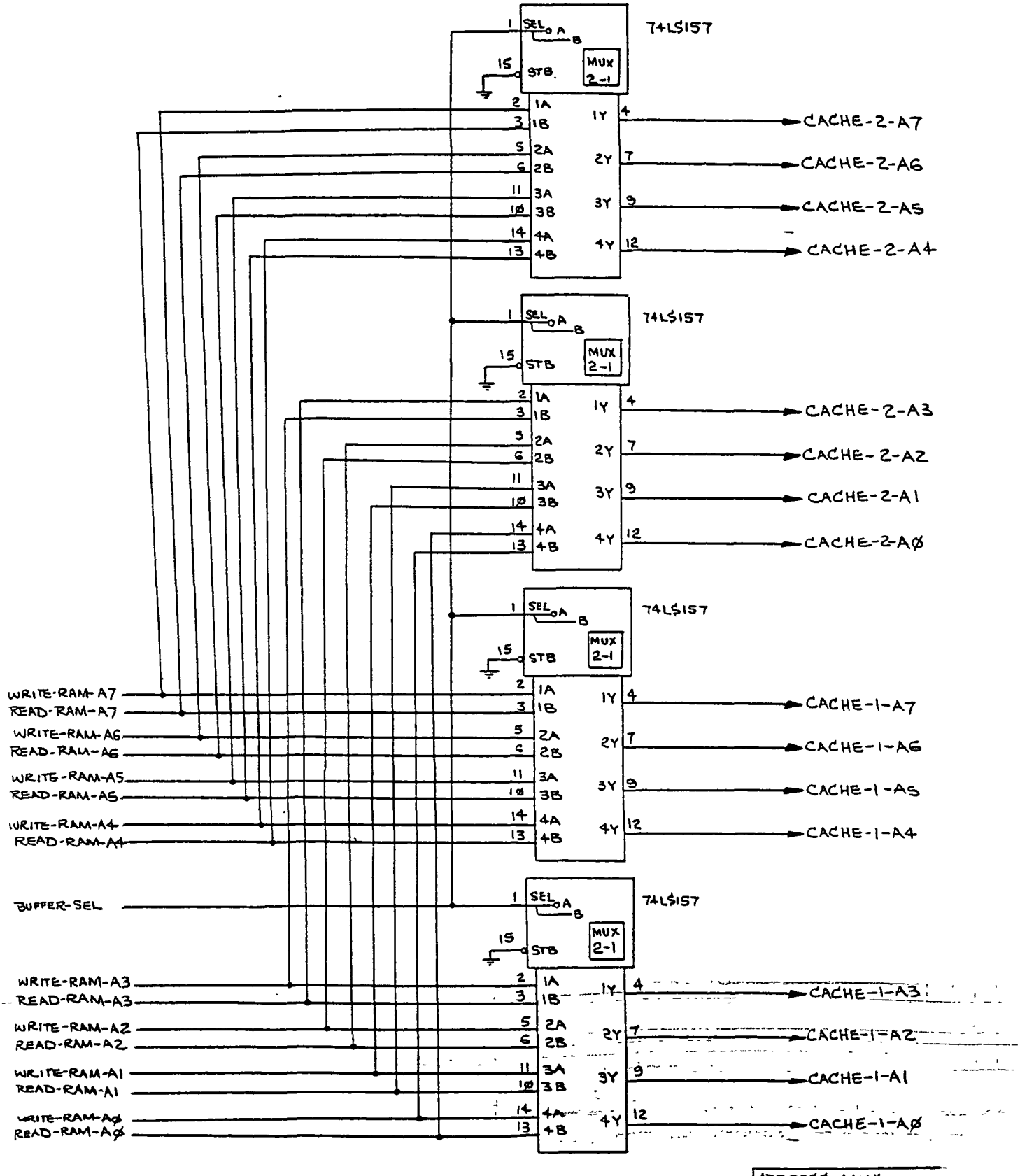


CACHE HI SPEED RAM
CRT CONTROLLER
#2

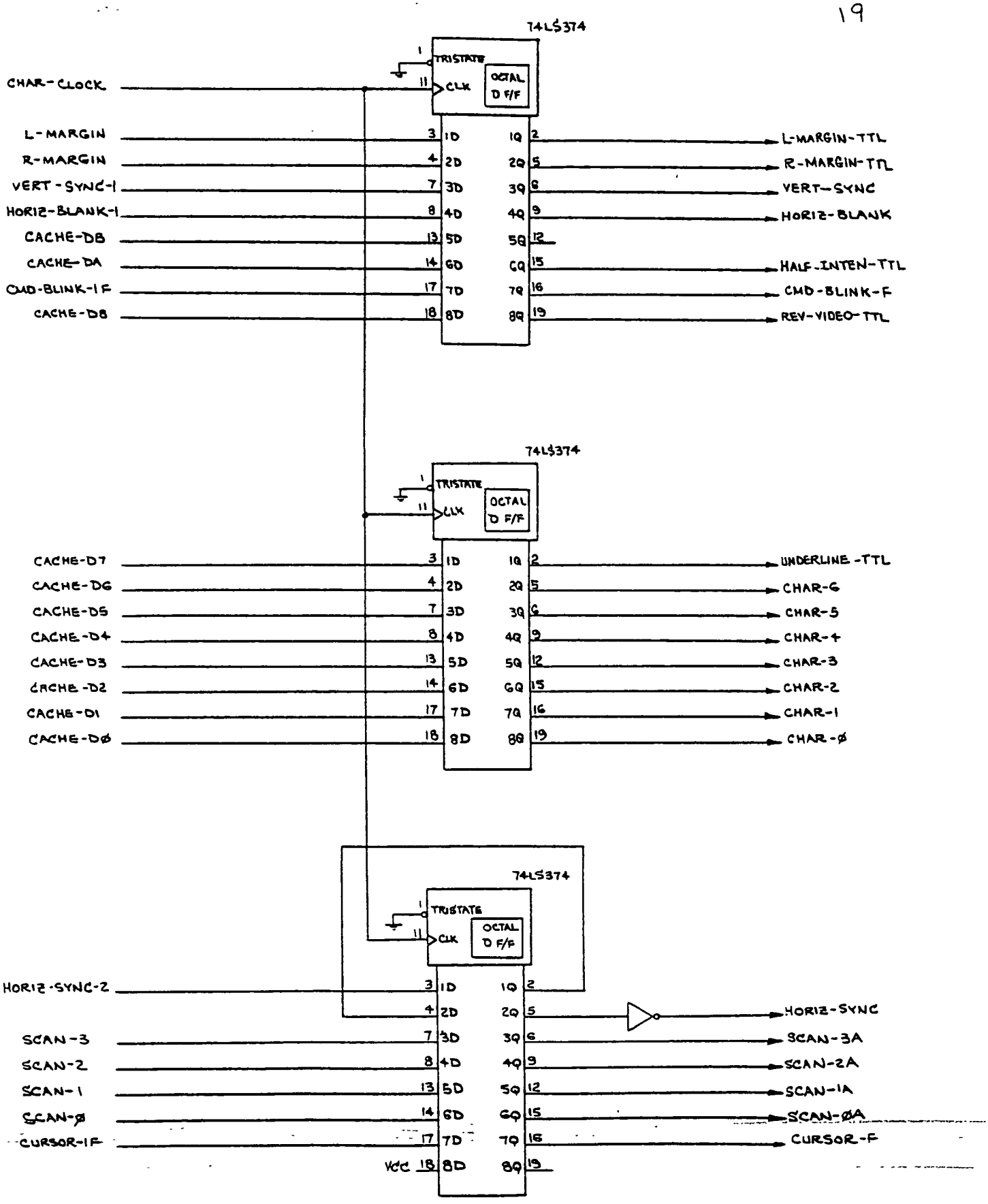


256-182 = 83 = 0123

CHAR. COUNTER
CRT. CONTROLLER

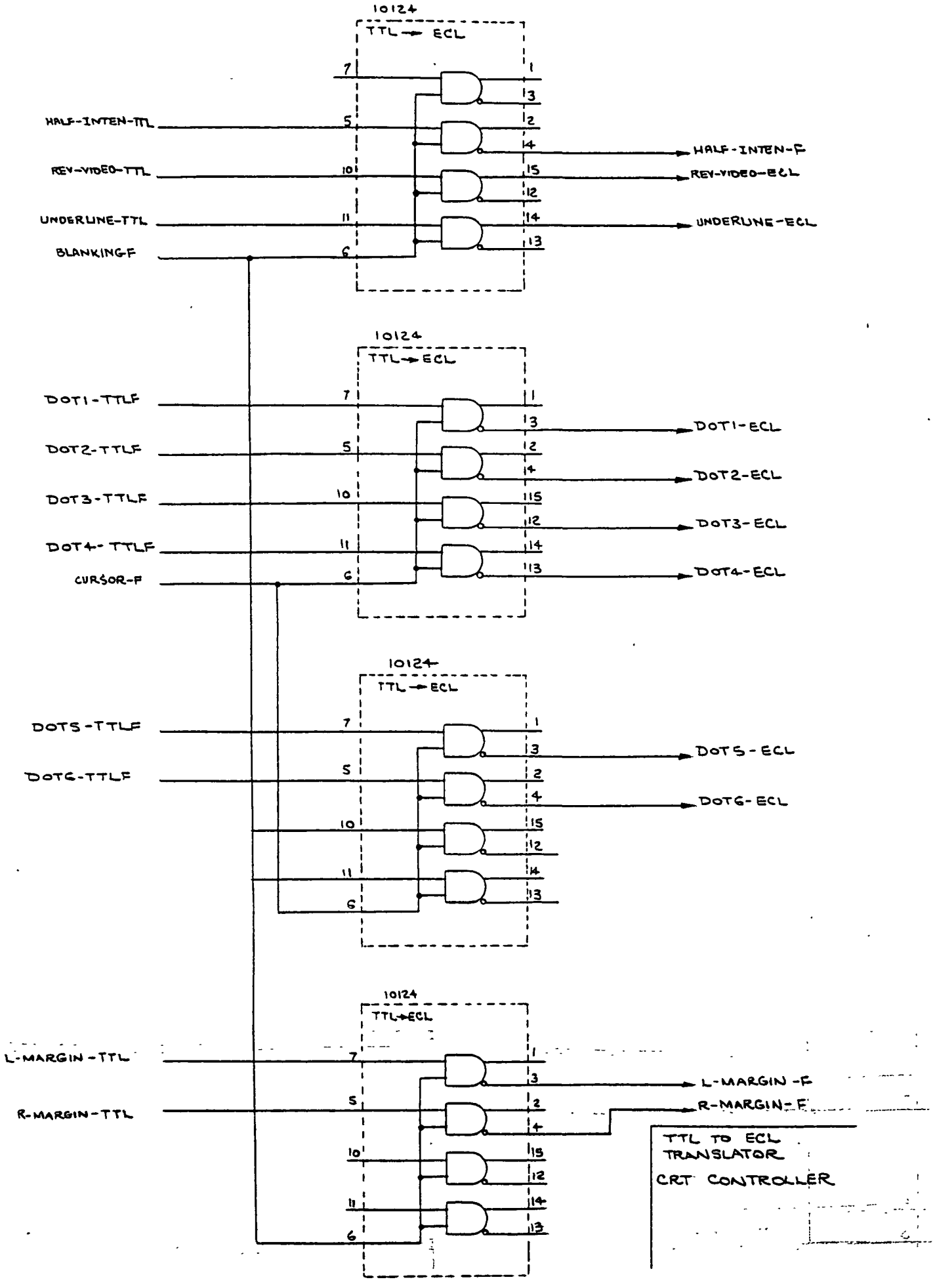


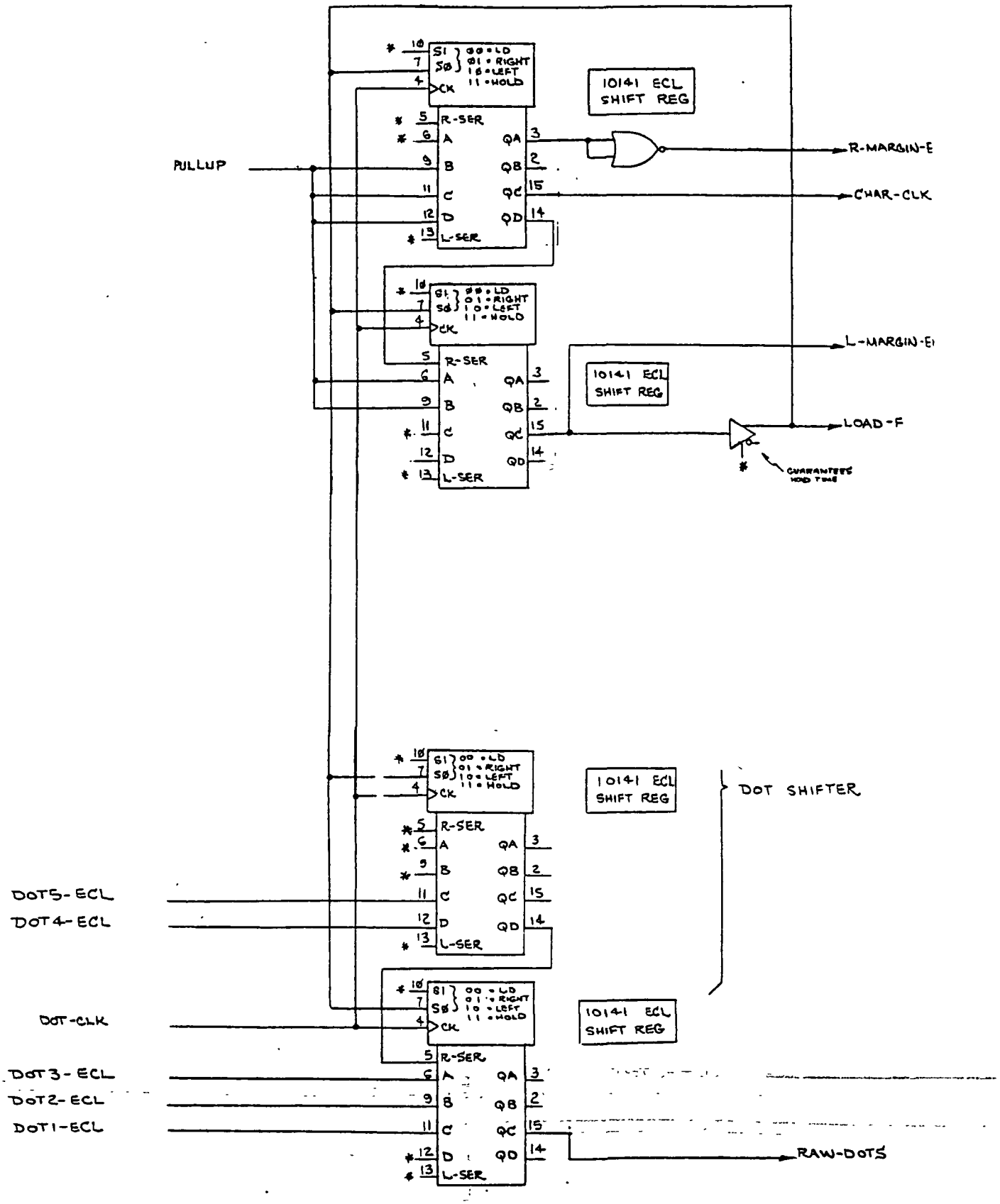
ADDRESS MUX
CACHE RAMS



PIPELINE REGISTER
CRT CONTROLLER

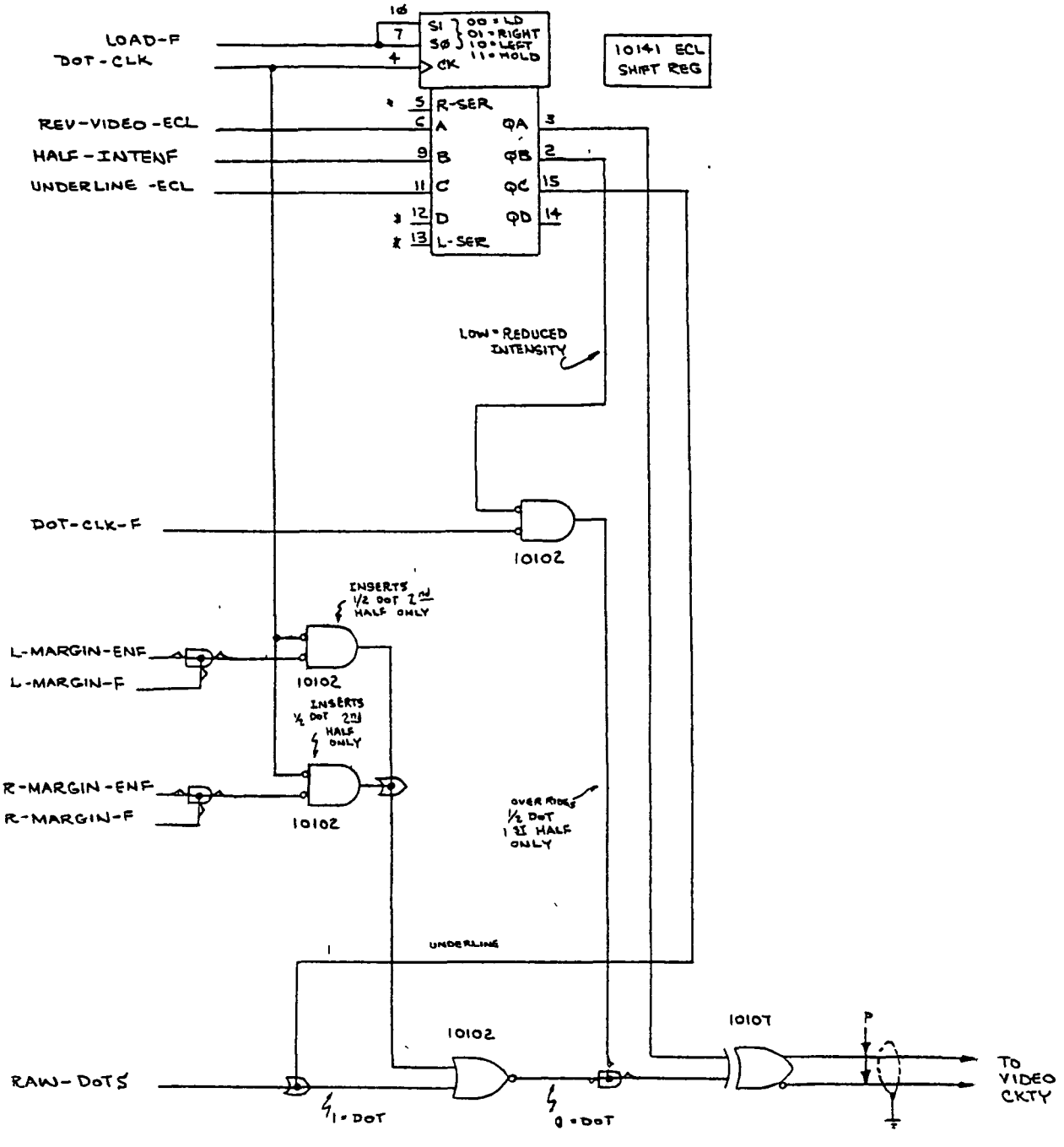
"Page missing from available version"


10124
TTL → ECL
HALF-INTEN-TTL
REV-VIDEO-TTL
UNDERLINE-TTL
BLANKING-F
HALF-INTEN-F
REV-VIDEO-ECL
UNDERLINE-ECL
10124
TTL → ECL
DOT1-TTLF
DOT2-TTLF
DOT3-TTLF
DOT4-TTLF
CURSOR-F
DOT1-ECL
DOT2-ECL
DOT3-ECL
DOT4-ECL
10124
TTL → ECL
DOT5-TTLF
DOT6-TTLF
DOT5-ECL
DOT6-ECL
10124
TTL → ECL
L-MARGIN-TTL
R-MARGIN-TTL
L-MARGIN-F
R-MARGIN-F
TTL TO ECL
TRANSLATOR
CRT CONTROLLER



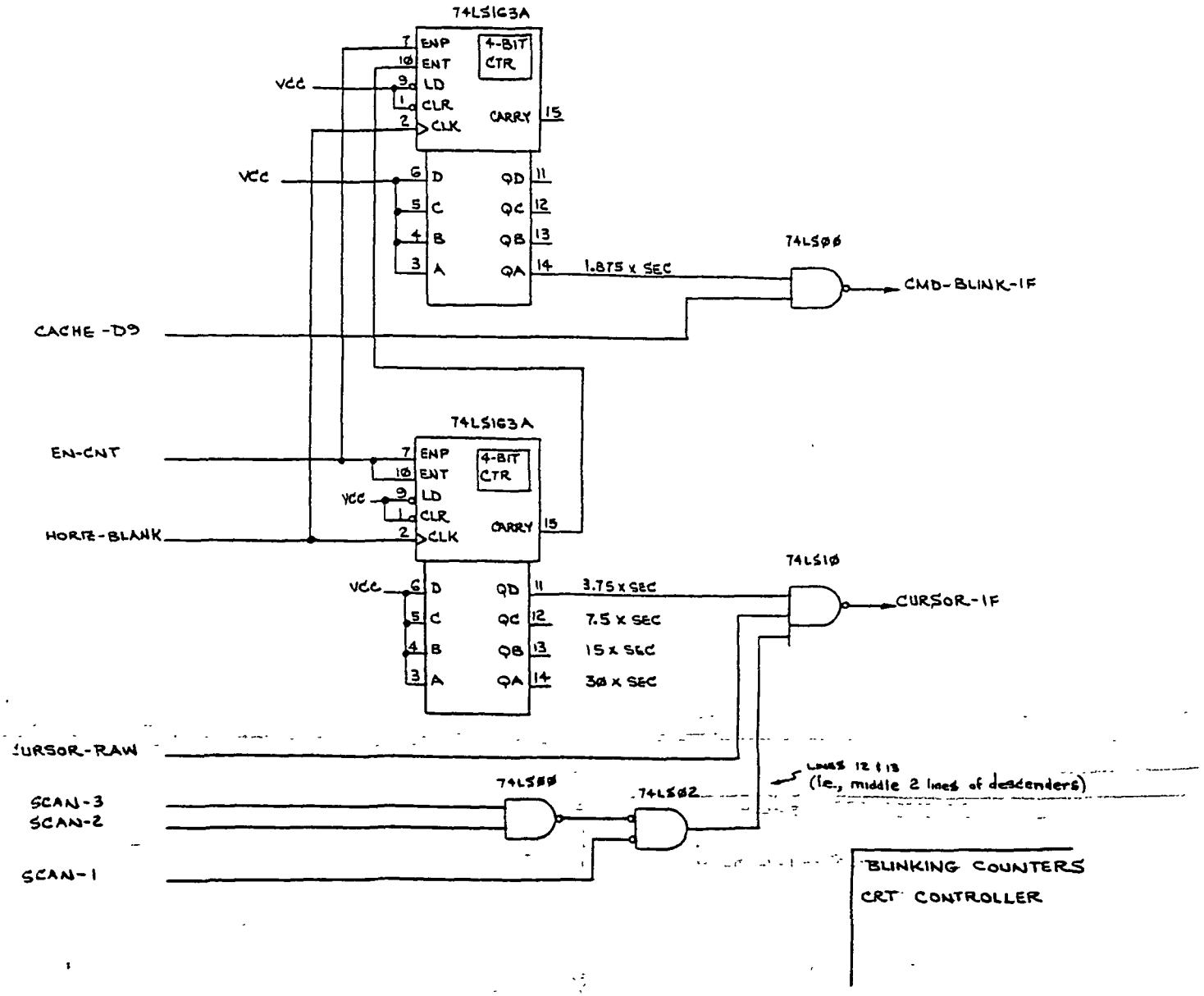
* NOTE: OPENS ON ECL 10,000 - LOGIC "0" = Acceptable design practice.

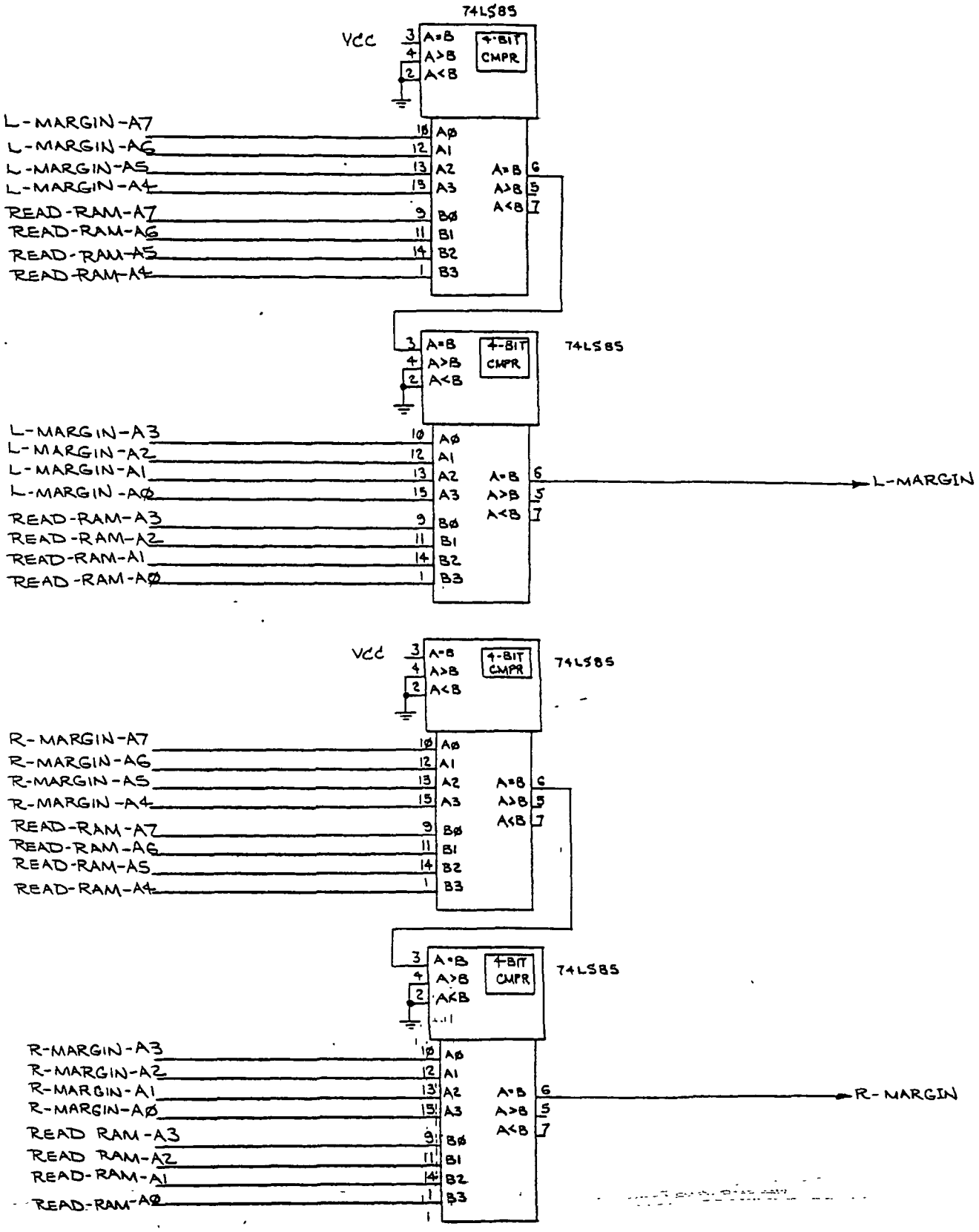
DOT SHIFTER & TIMING GENERATOR
CRT CONTROLLER



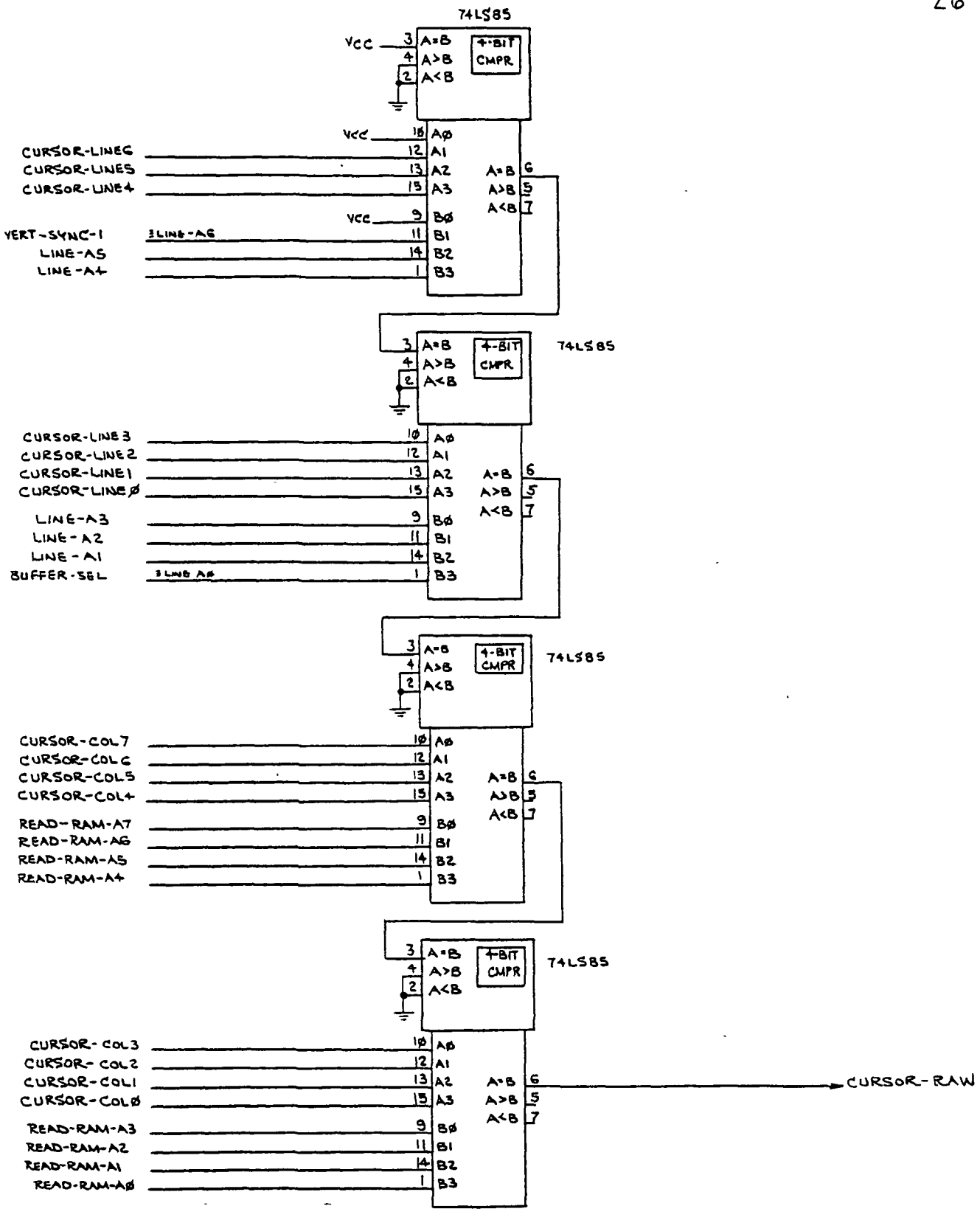
* NOTE: OPEN INPUTS ON ECL 10000 = LOGIC "Ø"
= ACCEPTABLE DESIGN PRACTICE

SPECIAL FEATURE INSERTION
CRT CONTROLLER

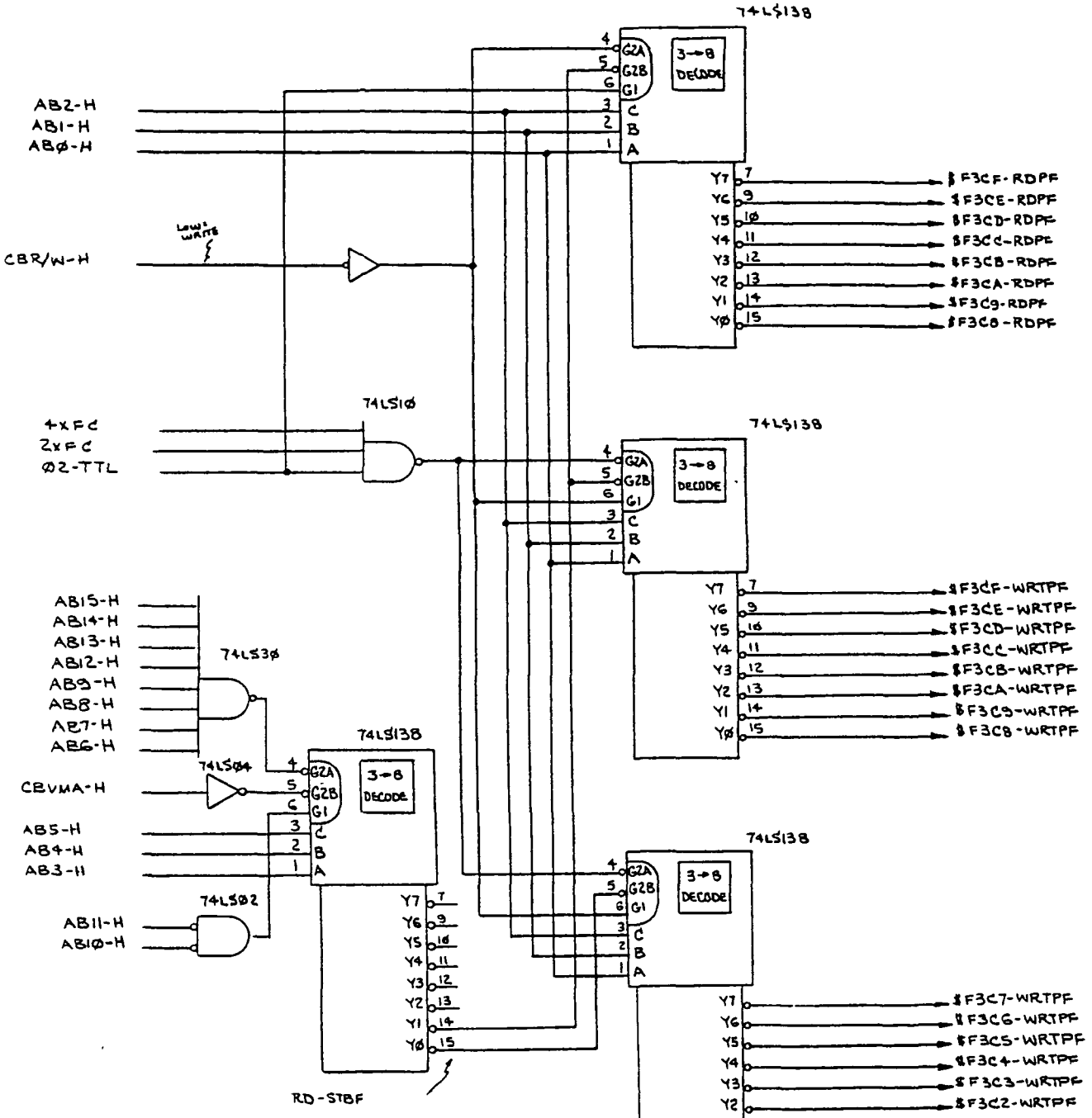

74LS163A
ENP
ENT
LD
CLR
CLK
4-BIT CTR
CARRY
VCC
D
C
B
A
QD
QC
QB
QA
1.875 x SEC
74LS00
CMD-BLINK-IF
CACHE -D9
EN-CNT
HORIZ-BLANK
3.75 x SEC
7.5 x SEC
15 x SEC
30 x SEC
74LS10
CURSOR-IF
CURSOR-RAW
74LS08
74LS02
SCAN-3
SCAN-2
SCAN-1
LINES 12 & 13
(ie., middle 2 lines of descenders)
BLINKING COUNTERS
CRT CONTROLLER



MARGIN DETECT LOGIC



CURSOR DETECTION COMPARATOR

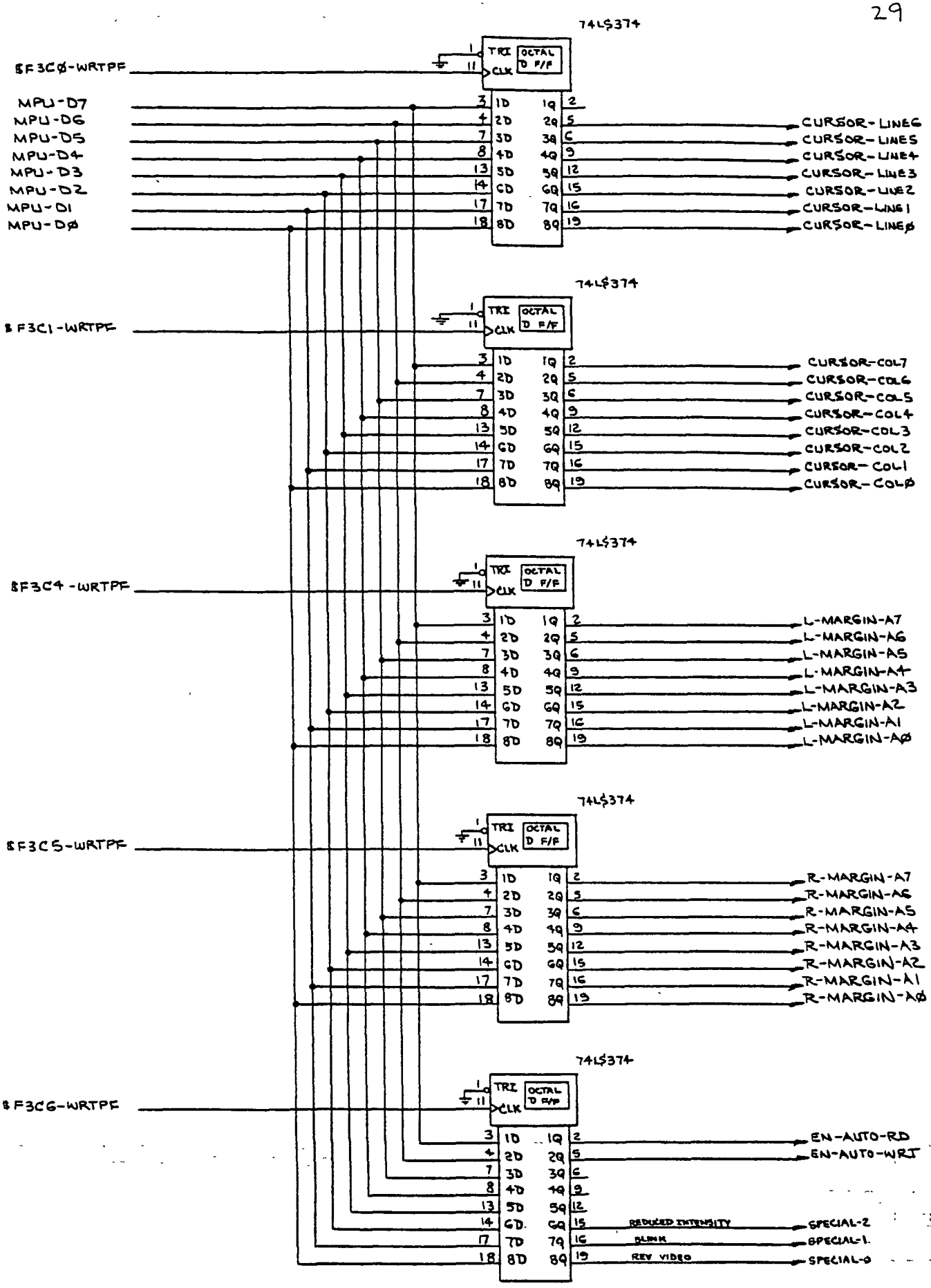


BASE ADDRESS = $F3C0

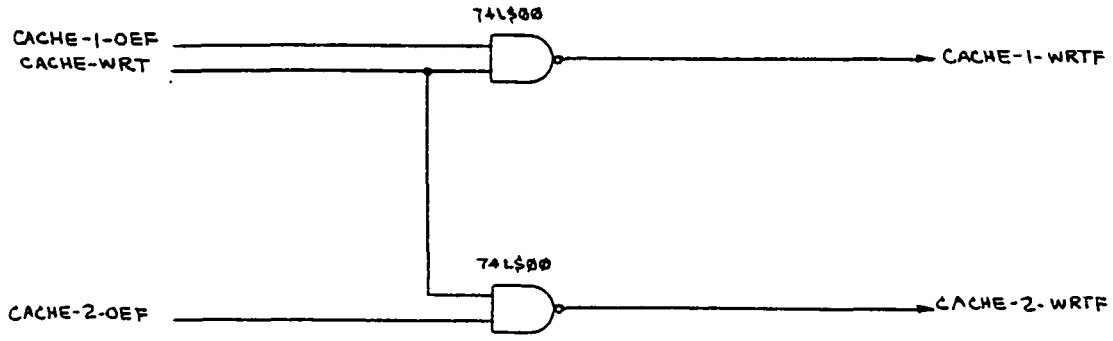
REGISTER ADDRESS DECODE AND WRITE STROBE LOGIC

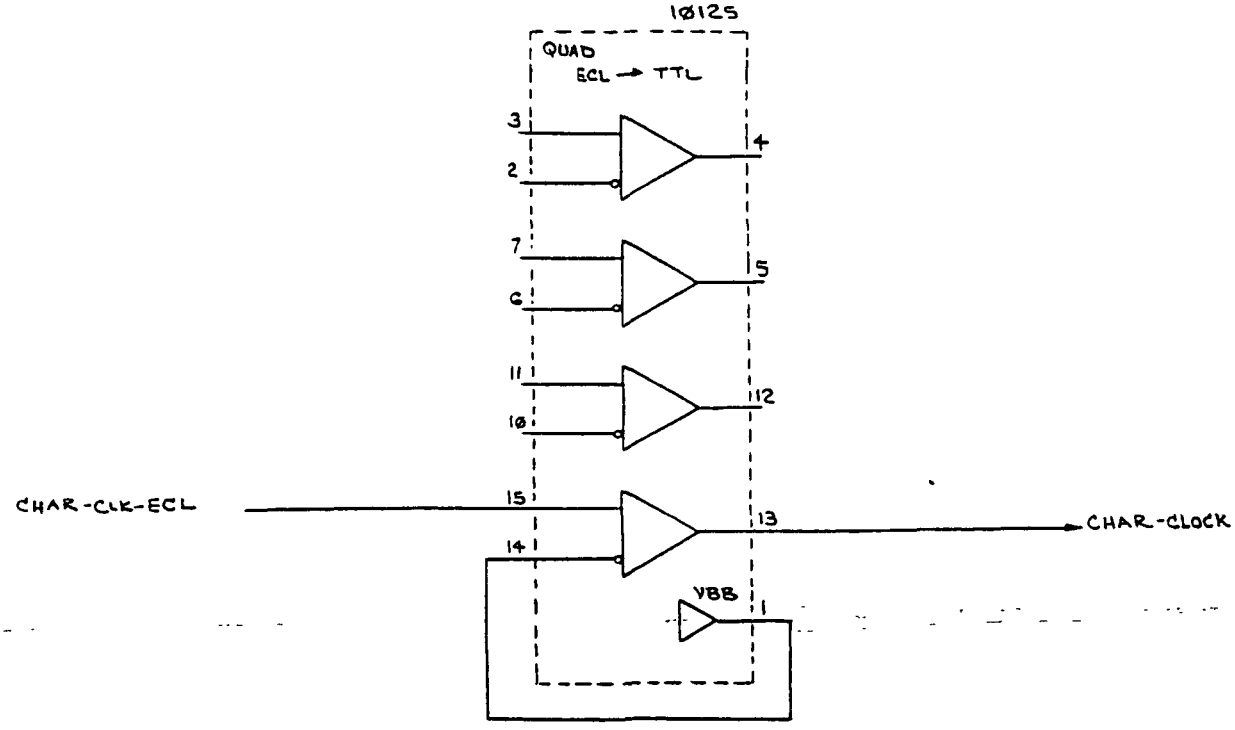
CRT CONTROLLER

"Page missing from available version"

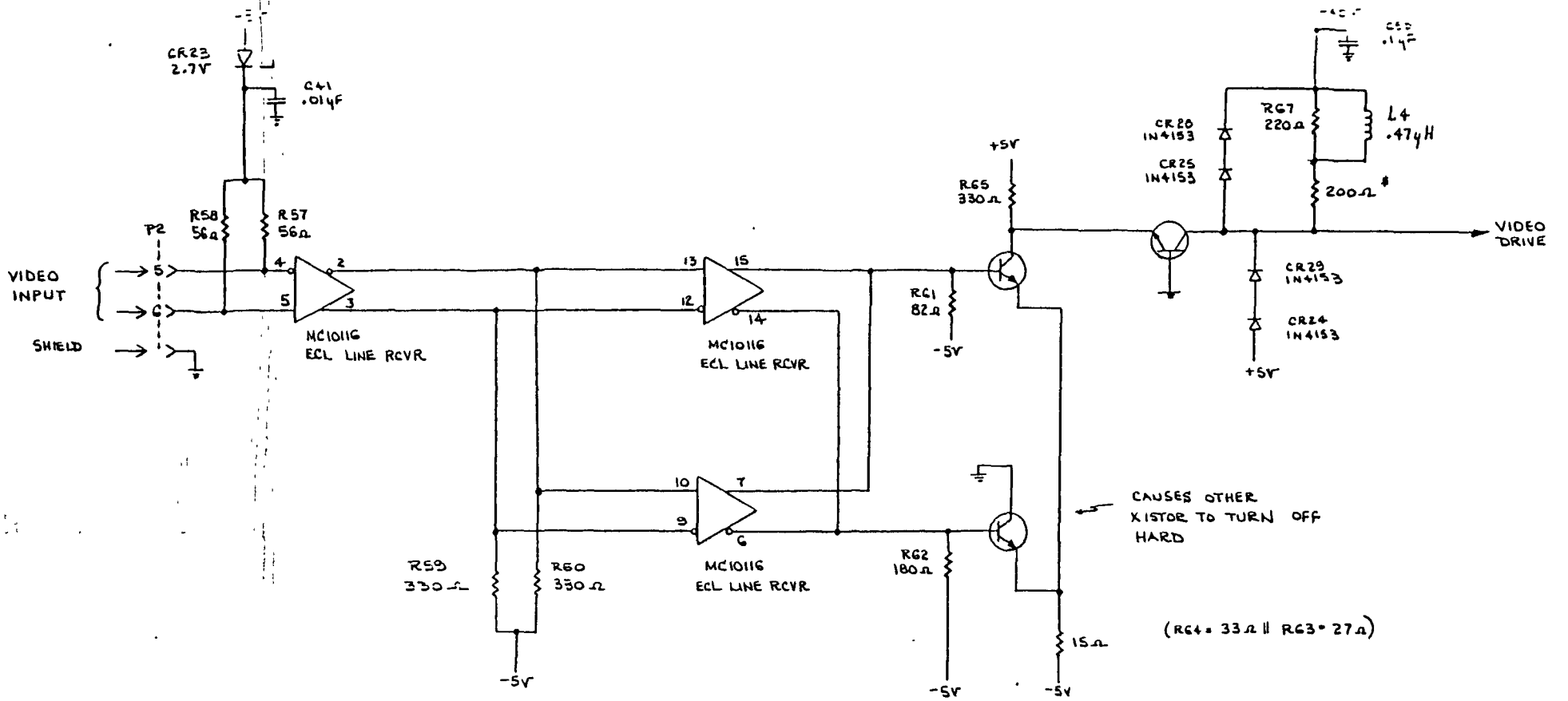


MARGIN, CURSOR, & SPECIAL FUNCTIONS REGISTERS



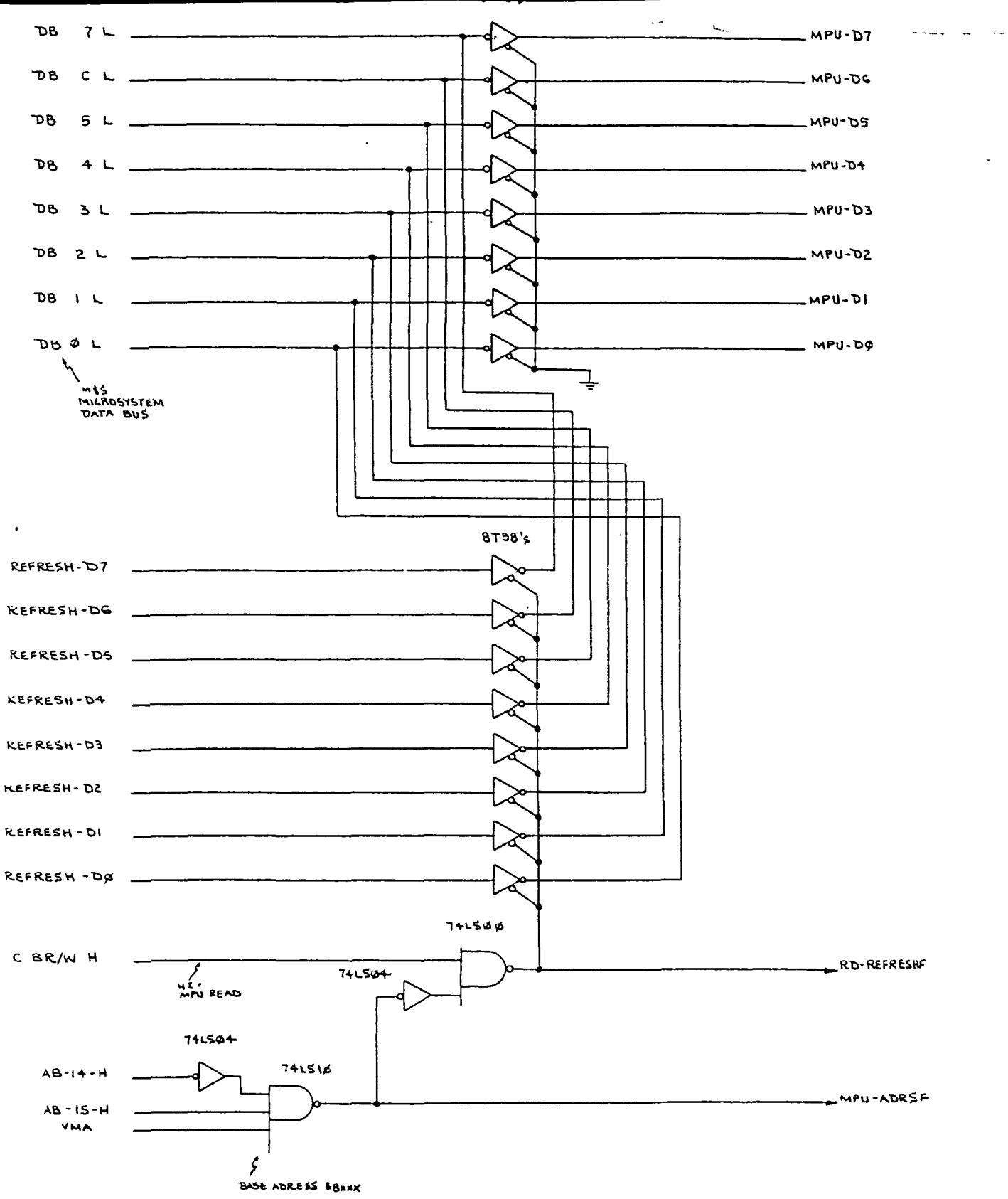


CHAR. CLOCK BUFFER &
CACHE WRITE STROBE
LOGIC - CRT CONTROLLER

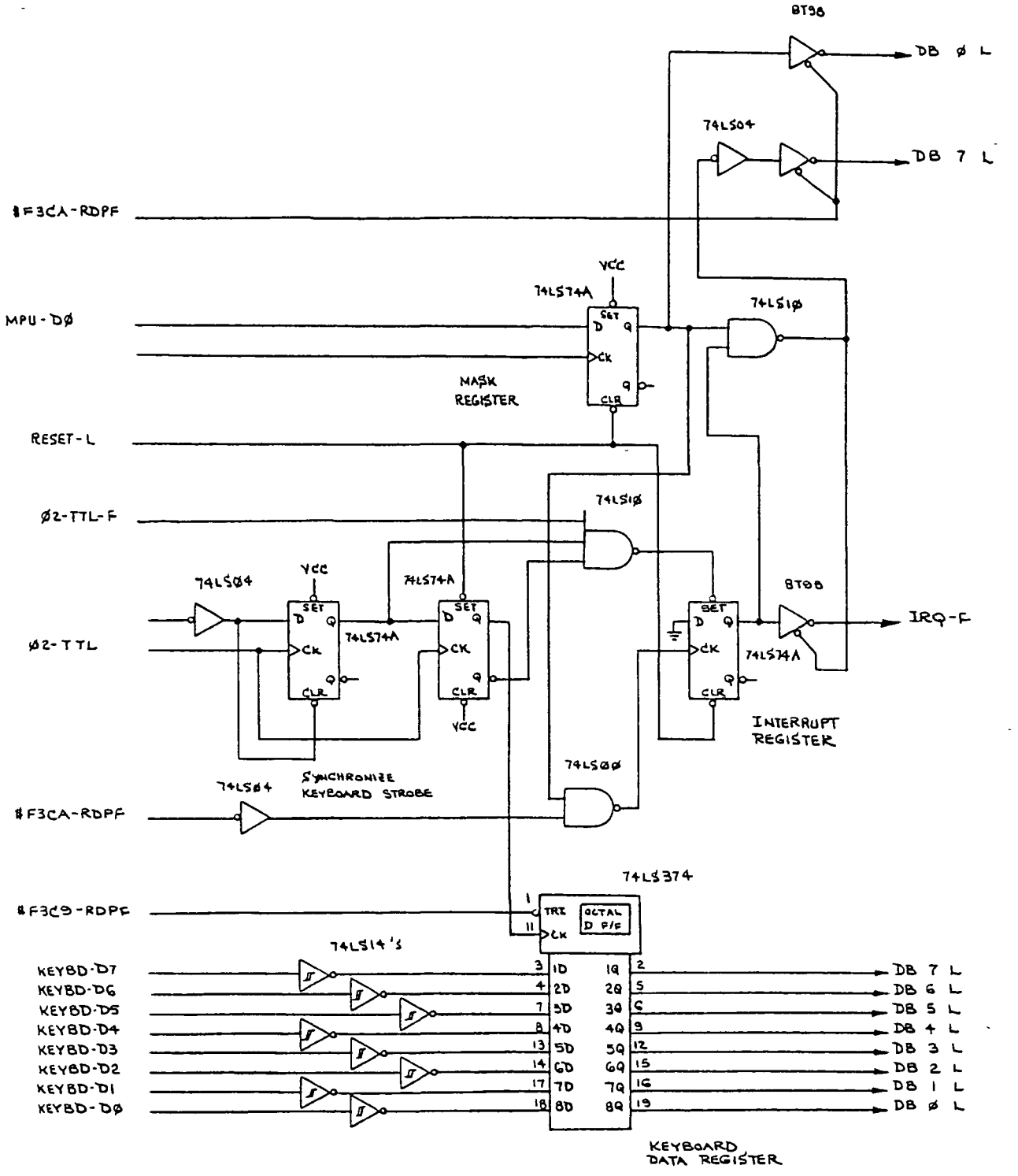


* 200Ω = RG8 ‖ RG9 ‖ R70 ‖ R71 ‖ R72
RG8 = RG9 = R70 = R71 = R72

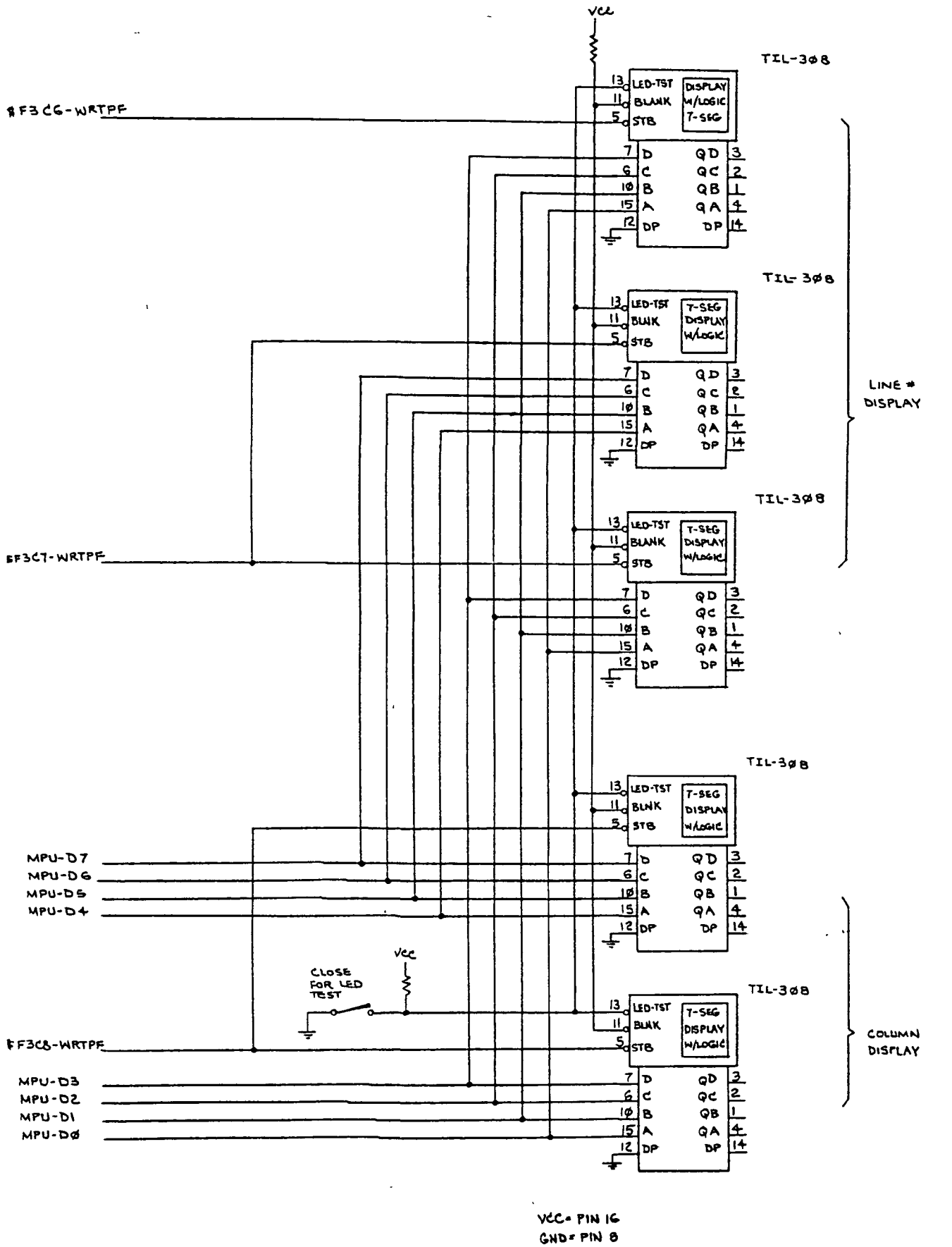
VIDEO DRIVE LOGIC
REDRAWN FROM
CPT CORPORATION # 910107



DATA BUS BUFFERS
& REFRESH MEMORY
ADDRESS DECODE

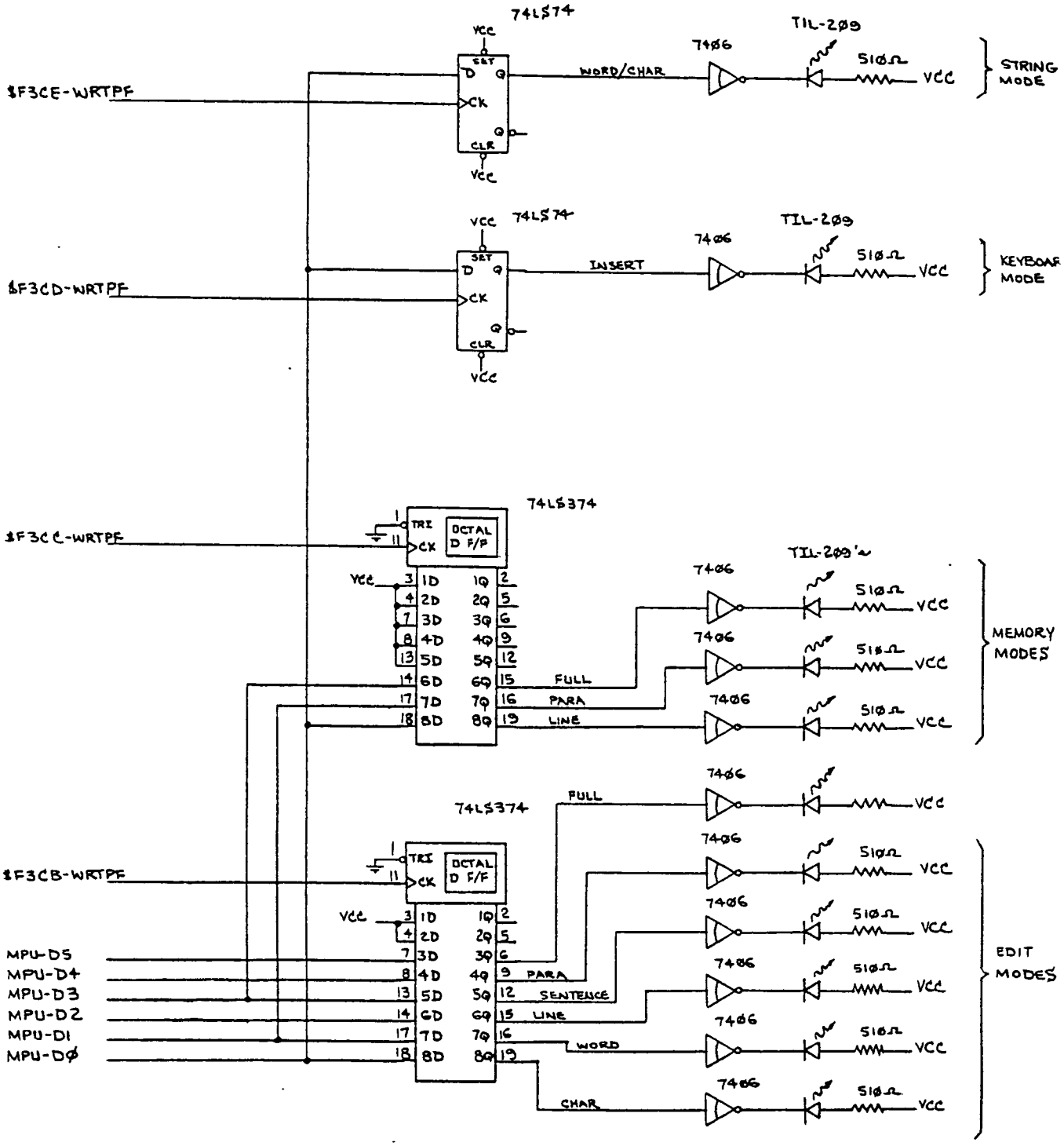


KEYBOARD INTERFACE
CRT CONTROLLER



VCC = PIN 16
GND = PIN 8

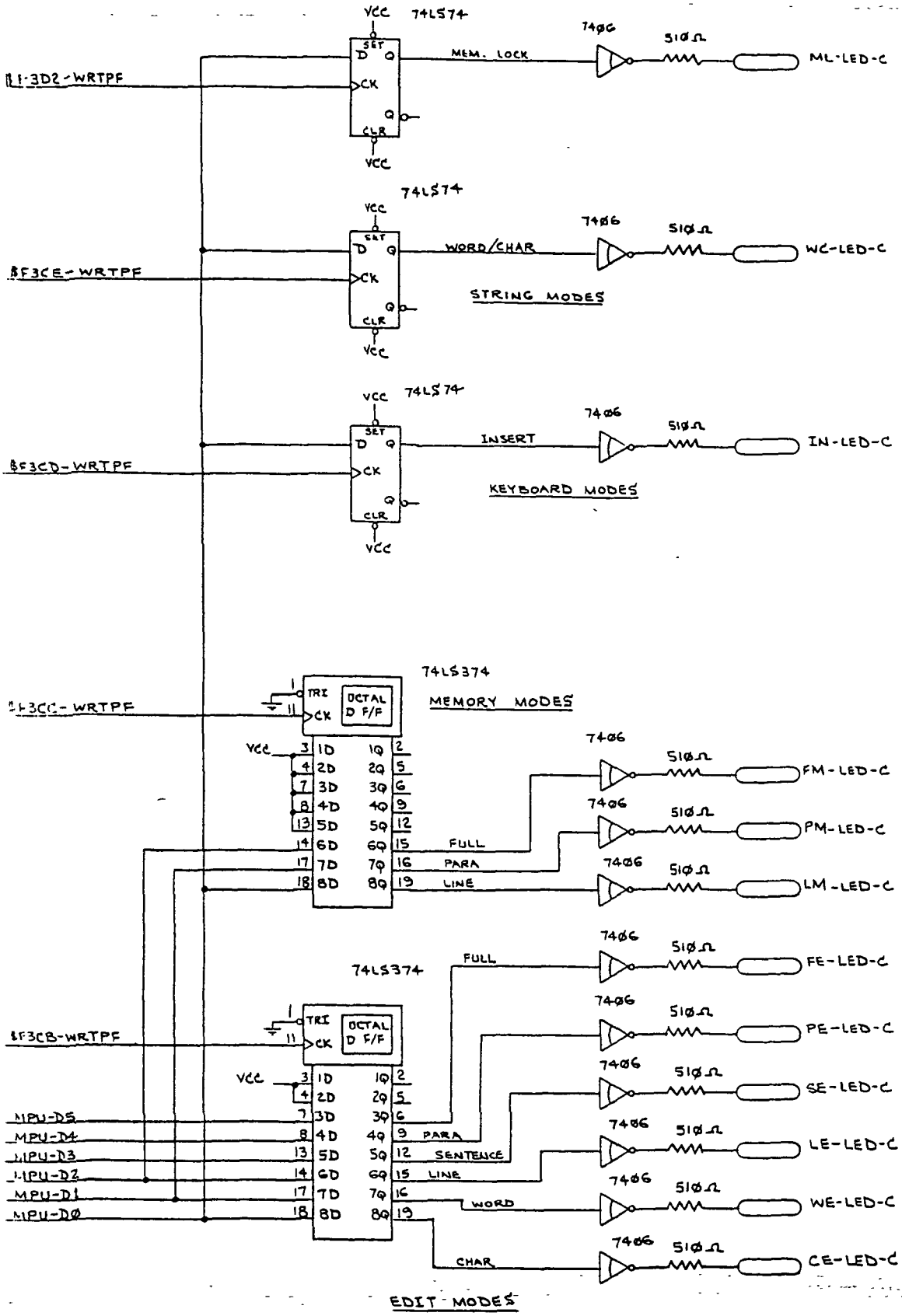
CURSOR POSITION DISPLAY
CRT CONTROLLER



TEXAS-INSTR- TIL-209 = MONSANTO MV5074B

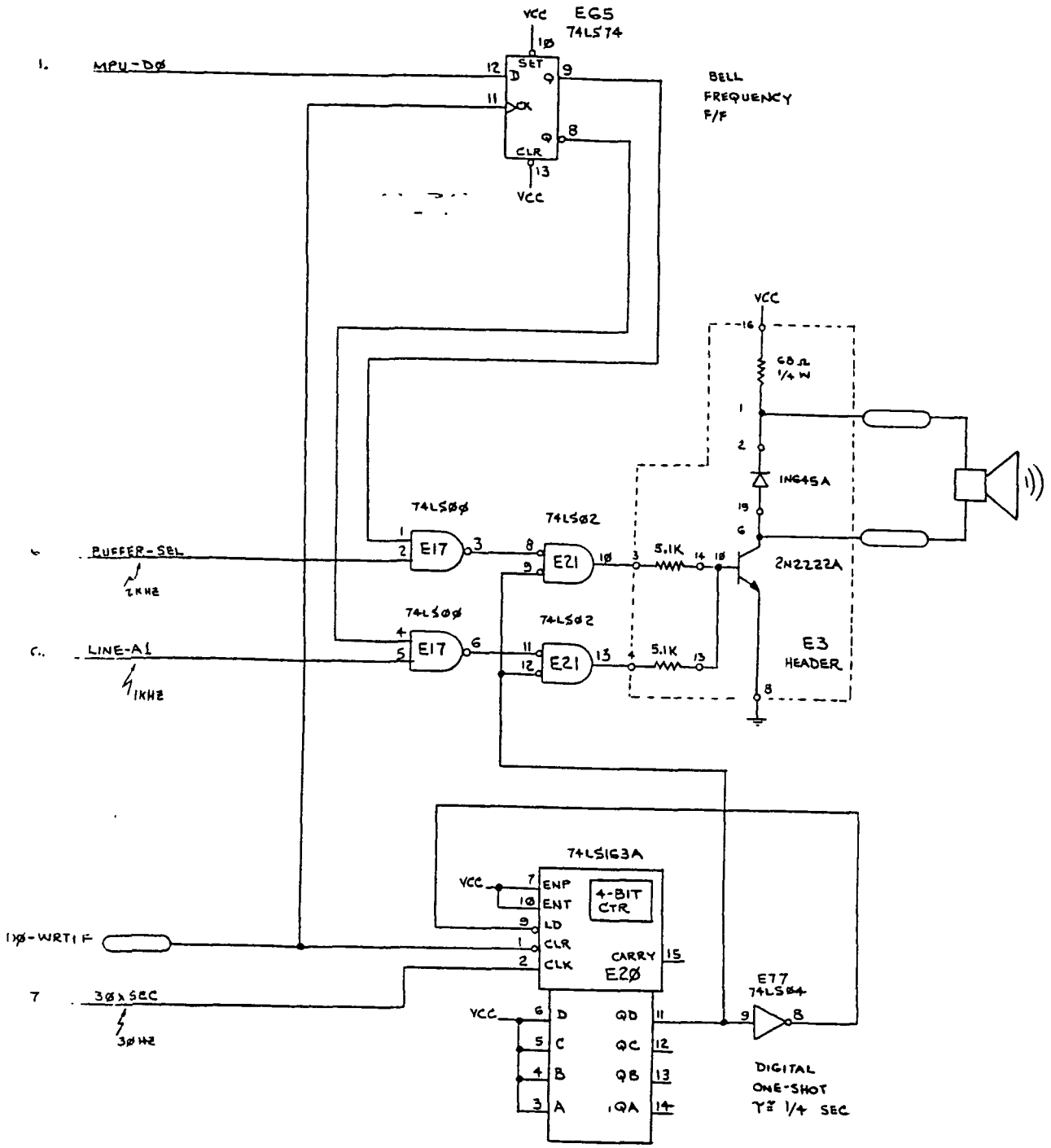
KEYBOARD DISCRETE DISPLAYS

CRT CONTROLLER



KEYBOARD DISCRETE DISPLAYS

CRT CONTROLLER



DUAL BELL CIRCUITRY
CRT CONTROLLER
VIDEO CARD

Final Report

Contract NAS8-34969

ENHANCEMENT OF INTELLIGENT EDITOR/PRINTER

This report was prepaired by Arizona State University under contract NAS8-34969 Enhancement of Intelligent Editor/Printer for Marshall Space Fligh Center of the National Aernautics and Space Administration.

A P P E N D I X

D

A One-Line Assembler/Dis-Assembler for the MC6800 MPU. This is a feature that is currently being added to MUDBUG which will greatly increase it ease of use, especially to the in-frequent or casual user. This capability will allow an engineer to debug in assembly language rather than machine language.

"Page missing from available version"

II THRU III

ABSTRACT

A user's manual describing the operation of a Motorola M6800-based One-line Assembler/Disassembler is presented. This One-line Assembler/Disassembler is a three-subroutine software package that is designed to increase the functionality of some MUFBUG commands, and it may have some other applications in the future. The One-line Assembler/Disassembler is capable of translating assembly-language instructions to machine codes, and it is also capable of translating machine-language instructions into assembly-language instructions. The assembler portion of the One-line Assembler/Disassembler is similar in function to a two-pass assembler. The primary difference between the assembler portion of the One-line Assembler/Disassembler and a normal assembler is that the One-line Assembler/Disassembler cannot accept labels in either the label field or the operand field. Numeric and self-relative values must be typed instead of using labels. The manual documents the One-line Assembler/Disassembler source syntax for instructions, expressions, and pseudo instructions, and it provides interface information for the internal One-line Assembler/Disassembler software package routines.

TABLE OF CONTENTS

Chapter 1

Introduction

MUDBUG is a Monitor-Utility-DeBUG software package that helps the programmer work with his or her machine-coded program on the Motorola M6800 microprocessor. The programmer can use the utilities offered by MUDBUG to enter, alter, and debug his or her own program. The MUDBUG system was written by Dr. David C. Pheanis in 1977. This One-line Assembler/Disassembler is designed for MUDBUG. The One-line Assembler/Disassembler was written by Mr. Yih-Liang Lien in 1983 with the help of some background information from Dr. Pheanis.

The One-line Assembler/Disassembler is a three-subroutine software package that occupies 4-K words of ROM, and it requires a 140-word block of RAM. It is designed to increase the functionality of certain MUDBUG commands. Especially in the memory-change (i.e., "C") command and memory-dump (i.e., "M") command, it increases the readability of the debugging transcript and helps the programmer debug his or her own program.

The One-line Assembler/Disassembler translates the machine-language instruction starting in the memory location that is addressed by the XR to an assembly-language

instruction, and the Disassembler outputs the assembly-language instruction to the terminal. Then the One-line Assembler allows the user to change the assembly-language instruction by typing a new assembly-language instruction. The user can alternatively keep the assembly-language instruction unchanged by typing only a terminator. The One-line Assembler/Disassembler accepts the five characters ",", CR, "*", ".", and "/" as legal instruction terminators. A user who attempts to use any other character except the above five characters as an instruction terminator will receive an error message.

An assembly-language instruction that is input to the One-line Assembler/Disassembler typically contains three fields: an opcode field, an operand field, and a terminator field. The user can type blank character(s) or tab character(s) or any combination of blanks and tabs between two fields. At least one column of white space is necessary between the opcode field and the operand field. The necessity for white space between the operand field and the terminator field depends on the terminator.

A comma terminator tells the One-line Assembler/Disassembler that it should next process the instruction at the same location again. A carriage return as a terminator tells the One-line Assembler/Disassembler to process the

next sequential instruction in memory after it finishes processing the current instruction. An asterisk terminator tells the One-line Assembler/Disassembler to process the previous sequential instruction in memory after it finishes processing the current instruction. A period terminator tells the One-line Assembler/Disassembler to return control to the top of MUDBUG after processing the current instruction. Finally, a solidus ("/") terminator tells the One-line Assembler/Disassembler to abort processing by returning control immediately to the top of MUDBUG without even finishing the processing for the current instruction.

There are three entry points for calling the One-line Assembler/Disassembler software package. One entry point, labeled DISASM, is used as the entry point of subroutine DISASM. Subroutine DISASM translates a machine-language instruction to an assembly-language instruction and prints the assembly-language instruction on the user's terminal. The other two entry points, labeled ASMTERM and ASSEM, are used as the entry points of subroutine ASMTERM and subroutine ASSEM, respectively. These two subroutines assemble assembly-language instructions to machine codes and store those codes into memory. Subroutine ASMTERM takes care of the terminator of the input assembly-language instruction and adjusts the XR to point to the location of the next instruction to be processed where the next

instruction is determined according to the input terminator. Subroutine ASSEM, on the other hand, returns the ASCII code of the terminator in the BR to the calling program instead of taking care of it.

The first word of the One-line Assembler/Disassembler ROM is referred to symbolically as location ASMROM, and the first word of the One-line Assembler/Disassembler RAM is similarly designated as location ASMRAM. The entire One-line Assembler/Disassembler program is written in a completely symbolic fashion, of course, so all of its memory addresses can easily be changed to suit any special application.

Chapter 2

Internal Routines and Subroutines

The One-line Assembler/Disassembler software package incorporates three internal subroutines, and these three subroutines are potentially useful for a wide variety of applications. Programmers who wish to invoke any of the three internal subroutines can simply call the desired routine(s) from their own programs.

The list on the next page summarizes the internal One-line Assembler/Disassembler routines that are available to the system's users, and interface characteristics such as calling-sequence requirements and return conditions are given for each subroutine in the pages that follow. Notice that the entry points for the available internal routines all occur in consecutive locations of a vector table that starts at the first location of the One-line Assembler/Disassembler ROM. This vector table is provided as a convenience for the user so that the entry-point addresses for the One-line Assembler/Disassembler's internal subroutines will not change from one sub-version of the One-line Assembler/Disassembler to another.

2.1. Summary of Internal Routines.

The list below summarizes the internal One-line Assembler/Disassembler routines that are available for direct access by the system's users. The symbolic label "ASMROM" is used here and throughout this document to represent the memory address of the first word of the One-line Assembler/Disassembler's ROM.

Name	Entry Point	Function
DISASM	ASMROM+$00	Translate the machine-language instruction starting in the memory location that is addressed by the XR to an assembly-language instruction, and output the assembly-language instruction to the terminal.
ASMTERM	ASMROM+$03	Translate an assembly-language instruction to machine language, and store the machine-language instruction into memory. The XR points to the proper memory location according to the input terminator upon return.
ASSEM	ASMROM+$06	Translate an assembly-language instruction to machine language, and store the machine-language instruction into memory. The BR contains the ASCII code of the input terminator upon return.

2.2. Subroutine Descriptions

This section contains interface information for the internal One-line Assembler/Disassembler software package routines that are available for direct access by users. Each routine is briefly described, and then its calling sequence and return conditions are documented as may be appropriate.

Subroutine DISASM: Entry Point = ASMROM+$00

Subroutine DISASM translates the machine-language instruction whose first byte is at the memory location addressed by the XR to assembly language and outputs it to the user's terminal.

LENBUF is a 21-byte buffer in ASMRAM. Each byte of LENBUF contains a number that represents the length of a machine-language instruction. The lengths of twenty consecutive previously-processed instructions can be kept in this buffer, so subroutine ASMTERM can use LENBUF to determine the address of a previously-processed instruction. LENPTR is a pointer that points to the location where the latest value was stored into the LENBUF area. LENPTR must point to the first word of the LENBUF area before subroutine DISASM is called for the first time, and subroutines DISASM

and ASMTERM automatically update LENPTR as they process instructions. The XR must always contain the address of the instruction to be disassembled before control enters subroutine DISASM.

Subroutine DISASM contains internal routines known as ONEBYTE, TWOBYTE, TRIBYTE, and FCDB/TE. These internal routines update LENPTR to point to a new entry in the LENBUF area, and then they store the value of an instruction's length into the LENBUF entry that is designated by the updated LENPTR value. These routines actually share a lot of common instructions, and they are essentially different entry points into one routine.

LENPTR must point to the first word of the LENBUF area before subroutine DISASM is called for the first time, but the user doesn't need to update the value of LENPTR after that because subroutines DISASM and ASMTERM automatically update LENPTR. The user can reinitialize LENPTR whenever she or he wishes to start processing a new group of instructions. Subroutines ASMTERM and ASSEM store a special sentinel value into location LENBUF to mark the beginning of the LENBUF table, and the first word in the LENBUF table is therefore not used to contain the length of an instruction.

Subroutine DISASM outputs "FCB $hh " on the user's terminal if the operating code that is addressed by the XR is an invalid code. The $hh value represents the invalid operating code.

Calling Sequence:

```
        LDX   =LENBUF          or       LDX   =LENBUF
        STX   LENPTR                    STX   LENPTR
        LDX   =(opcode address)         LDX   =(opcode address)
        JSR   DISASM                    BSR   DISASM
```

Return Condition: The XR contains the address of the next operating code, and the CC register is destroyed, but all other registers are preserved. The value of LENPTR has been incremented, and the value of the instruction's length has been stored into the LENBUF entry that is now designated by LENPTR. Also, the variable UNLOC has been set to point to the instruction that was processed.

Subroutine ASMTERM: Entry Point = ASMRUM+$03

Subroutine ASMTERM translates an assembly-language instruction to machine code and stores the machine code into

memory starting at the location to which INLOC points. Subroutine ASMTERM adjusts the XR to point to the location of the next instruction to be processed where the next instruction is determined according to the terminator of the input assembly-language instruction. INLOC is a two-byte memory location in ASMRAM. It contains the value of the address where the input assembly-language instruction's machine codes should go. The value of INLOC has been set by subroutine DISASM if control comes to subroutine ASMTERM following a call to subroutine DISASM.

Subroutine ASMTERM processes the input assembly-language instruction and updates the LENBUF entry that is addressed by LENPTR. LENPTR points to the location that contains the length value of the assembly-language instruction that was just displayed by subroutine DISASM. Subroutine ASMTERM calls subroutine PTRADJ to decrement the value of LENPTR by one, and then subroutine ASMTERM calls subroutine ONEBYTE or TWOBYTE or TRIBYTE or FOURBYTE depending on the memory size of the machine code for the input assembly-language instruction to update the LENBUF entry. The reason for calling subroutine PTRADJ first before calling subroutine ONEBYTE or TWOBYTE or TRIBYTE or FOURBYTE is that subroutines ONEBYTE, TWOBYTE, TRIBYTE, and FOURBYTE increment the value of LENPTR by one in order to point to next available location in the LENBUF area and then

store the value of the machine-language instruction's length into that location. The length value of the assembly-language instruction that was processed by subroutine DISASM is overwritten by the length value of the new assembly-language instruction that was just assembled by subroutine ASMTERM.

Subroutine ASMTERM updates the value of LENPTR according to the input assembly-language instruction's terminator. Since subroutine ASMTERM is designed to be used with subroutine DISASM, the value of LENPTR that is set in subroutine ASMTERM must match the function of subroutine DISASM.

When the terminator of the input assembly-language instruction is a carriage-return terminator, the value of LENPTR is unchanged but the value of the LENBUF entry that is designated by LENPTR has been updated. Later on, subroutine DISASM disassembles the next machine-language instruction to an assembly-language instruction and stores the value of the machine-language instruction's length into the next available entry in the LENBUF area.

When the terminator of the input assembly-language instruction is a comma terminator, the value of LENPTR is decremented by one after the value of the LENBUF entry that

was originally designated by LENPTR has been updated. Later on, subroutine DISASM disassembles the machine-language instruction that is in the same memory location addressed by the XR to an assembly-language instruction and stores the value of the machine-language instruction's length into the same entry in the LENBUF area as the entry that was addressed by LENPTR before the value of LENPTR was decremented by one.

When the terminator of the input assembly-language instruction is an asterisk terminator, the value of LENPTR is decremented by two after the value of the LENBUF entry that was originally designated by LENPTR has been updated. We can get the value of the previously-processed instruction's length when LENPTR is decremented by one. Then we decrement the value of LENPTR by one again. Later on, subroutine DISASM disassembles the machine-language instruction that is in the previous memory location addressed by the XR to an assembly-language instruction and stores the value of the machine-language instruction's length into the previous entry in the LENBUF area.

When the terminator of the input assembly-language instruction is a period terminator, the value of LENPTR is unchanged but the value of the LENBUF entry that is designated by LENPTR has been updated.

When the terminator of the input assembly-language instruction is a solidus terminator, the value of LENPTR is unchanged, and the value of the LENBUF entry that is designated by LENPTR is not changed, either.

The terminator is not so meaningful when subroutine ASMTERM is called alone. We still set the correct returned value of the XR when the terminator is a carriage return or a comma, but the returned value of the XR will be unpredictable when the terminator is an asterisk. This problem occurs because we have no idea about the value of the previous instruction's length.

If LENPTR reaches the end of the LENBUF area, the length value stored in the second location of the 21-byte buffer is swept out and the rest of the values stored in the LENBUF table are moved up one location, so a new value can be stored at the end of the buffer area.

An error message, "INVALID: the whole input instruction string", will be printed on the user's terminal when the assembly-language instruction is incorrect.

Calling Sequence:

1. General Calling:

LDX	=<address>	or	LDX	=<address>	
STX	INLOC		STX	INLOC	
JSR	ASMTERM		BSR	ASMTERM	

2. Call with subroutine DISASM:

LDX	=LENBUF	or	LDY	=LENBUF	
STX	LENPTR		STX	LENPTR	
LDX	=<opcode address>		LDX	=<opcode address>	
JSR	DISASM		BSR	DISASM	
JSR	ASMTERM		BSR	ASMTERM	

Return Condition:

1. Normal return. The XR contains the address of the next instruction to be processed where the next instruction is determined according to the terminator of the input instruction. The CC register is destroyed, but all other registers are preserved. The value of the instruction's length has been stored into the LENBUF entry that was designated by the original LENPTR value, and LENPTR has been updated according to the terminator.

2. Solidus terminator. If the input assembly-language instruction is terminated by a solidus, the value of the memory location addressed by the XR is unchanged and control goes directly to the top of MUDBUG instead of returning to the calling routine. The value of the LENBUF entry that is designated by LENPTR is unchanged, and LENPTR is unchanged, too.

3. Period terminator. If the input assembly-language instruction is terminated by a period, the value of the memory location addressed by the XR is updated and control goes directly to the top of MUDBUG instead of returning to the calling routine. The value of the instruction's length has been stored into the LENBUF entry that is designated by LENPTR, and LENPTR is unchanged.

4. Error return. The XR contains the value of INLOC and the CC register is destroyed, but all other registers are preserved. The value of the memory location addressed by the XR is unchanged. The value of the LENBUF

entry that is designated by LENPTR is unchanged, too. The value of LENPTR is decremented by one just as if a comma terminator had been used.

Subroutine ASSEM; Entry Point = ASMROM+$06

Subroutine ASSEM is similar to subroutine ASMTERM. It translates an assembly-language instruction to machine code and stores the equivalent machine-language instruction into memory starting at the location that is addressed by the XR. Subroutine ASSEM returns the ASCII code of the input instruction's terminator in the BR to the calling program. When subroutine ASSEM is called, the XR must contain the value of the address at which the machine-language instruction is to be stored.

An error message, "INVALID: the whole input instruction string", will be output to the user's terminal when the input assembly-language instruction is incorrect.

Calling Sequence:

```
LDX   =<address>      or      LDX   =<address>
JSR   ASSEM                   BSR   ASSEM
```

1. Normal return. The XR contains the memory address of the location that immediately follows the new instruction in memory when the terminator is a carriage return, a comma, an asterisk, or a period. The value of the XR is unchanged when the terminator is a solidus. The BR contains the ASCII code of the input instruction's terminator, and the CC register is destroyed, but all other registers are preserved. The value of the memory location addressed by the previous XR is updated when the terminator is a carriage return, a comma, an asterisk, or a period. The value of the memory location addressed by the original XR value is unchanged when the terminator is a solidus.

2. Error return. The CC register is destroyed, but all other register values are preserved. The value of the memory location addressed by the XR is unchanged.

Chapter 3

The One-line Assembler Source Syntax, Expressions, and Pseudo Instructions

3.1. One-line Assembler Source Syntax

A source line in the One-line Assembler's assembly language may contain up to three fields: an opcode field, an operand field, and a terminator field. Since the One-line assembler makes no connection from one source line to another, no symbol table is established, and labels are not accepted. A typical source line that is typed at a terminal generally looks like this:

```
LDAA     0, X     <CR>
----     ----     ----
opcode   operand  terminator
field    field    field
```

The One-line Assembler allows the user to type leading blank(s) and/or tab(s) with a source line input because many users like to insert white space at the beginning of an input source line for improved readability. No leading white space is required, of course, but white space is allowed until the first non-white character of the source line has been typed. The first field of a typical source line is the opcode field, and it must contain one of the valid M6800 opcode or pseudo-opcode mnemonics. (See the

M6800 Programming Reference Manual for the opcodes, and see section 3 of this chapter for the pseudo optodes' descriptions.) Some of the opcodes require an "A" or "B" extension (e.g., BIT or CMP) while other opcodes permit the "A" or "B" extension (e.g., ROL or ASR), and still other opcodes do not permit any extension (e.g., INX or BLE).

The opcode field may be terminated by one or more blanks and/or tabs. Most programmers use a single tab to advance to the operand field of the instruction. There are, however, some instructions that do not need an operand field, and for these instructions the opcode field may be terminated by one of the four terminators (";" or "," or "*" or "/") with at least one leading blank or tab or may be terminated by a carriage return with or without leading white space. A carriage return terminates the entire source line.

The second field of an instruction line is the operand field, which naturally contains the operand for the instruction. Most instructions require at least a simple operand field (e.g., STX SAM), while other instructions require or allow two subfields in the operand field (e.g., LDAA 3,X or STAB 0,X). Some instructions permit multiple operand-field entries (e.g., FCB and FDB).

Finally, there are some instructions (such as INX and DEX) that do not require any operand field at all.

If an operand field contains two or more subfields, one or more blanks and/or tabs may occur after the comma that terminates one subfield and before the beginning of the next subfield. This rule makes the One-line Assembler source syntax compatible with a standard typing rule that requires a blank space after a comma.

The operand field can be terminated by one or more blank spaces and/or tab characters (except after a comma). Most users type a single blank space to skip to the terminator field when the terminator is one of the four nonblank terminators (";" or "*" or "." or "/"). The user can type a carriage return without any leading white space to terminate both the operand field and the entire source line with a carriage-return terminator.

3.2. One-line Assembler Expressions

The One-line Assembler evaluates expressions that are found in the operand field. A simple operand field contains only one expression, but an operand field with subfields contains an expression in each subfield. An expression consists of a term or multiple terms connected by operators,

and an expression can include any of the following types of terms:

1. Decimal number. Any series of digits (0-9) starting with a nonzero digit is recognized as a decimal number. The value must be unsigned and in the range 1 through 65,535 (i.e., an unsigned 16-bit value).

2. Octal number. Any series of octets (0-7) starting with a zero is interpreted as an octal number. Octal numbers are permitted to occupy as many as 16 bits (including the sign bit) because users sometimes find it convenient to code 16-bit constants as octal values.

3. Binary number. A "%" prefix character followed by a series of bits (0s and 1s) is recognized as a binary number. Binary numbers can be as large as 16 bits, so users can specify full 16-bit values in binary if they are so inclined.

4. Hexadecimal number. A series of hextets (0-9, A-F, or a-f) that is prefixed with a "$" is interpreted as a hexadecimal (hex) number. Hex numbers, like numbers in other bases, can contain as many as 16 bits, so a hex value can contain as many as four hextets.

5. Present-location symbol. When an asterisk ("*") is used as a term, it represents the value of the assembler's location counter, which is always updated as the last order of business when an instruction is assembled. The value of the present-location symbol therefore equals the address of the current instruction, and for a two-word or three-word instruction the value of "*" is the address of the first word of the instruction. Most users of ordinary two-pass assemblers tend to avoid using the present-location symbol except in special circumstances since the use of "*+" or "*-" is an extremely poor programming practice. For example, anyone who codes an instruction such as JMP *+5 is making his or her program hard to read and hard to maintain. If any intervening code is inserted (or deleted) after the JMP *+5 and before the target address of the jump, the program will no longer operate correctly. However, since the One-line Assembler cannot accept labels, "*+" and "*-" addressing is useful during debugging sessions.

6. Single ASCII character. A single ASCII character enclosed by or preceded by apostrophes or quotation marks (e.g., 'A', "A", 'A, or "A) may be used as a term. The value that is generated for the quoted character is the eight-bit ASCII code for the character with the parity bit reset.

The terms of an expression are combined (using 16-bit operations) according to the expression's binary operators in left-to-right order with all operators having equal precedence, and the following binary operators are recognized:

+	Addition
-	Subtraction
*	Multiplication
/	Integer Division
.AND.	Logical AND
.OR.	Logical Inclusive OR
.XOR.	Logical Exclusive OR
.MAX.	Larger of Two Signed Terms
.MIN.	Smaller of Two Signed Terms
.HI.	Larger of Two Unsigned Terms
.LOW.	Smaller of Two Unsigned Terms
.LSR.	Logical Shift Right (Zero Fill)
.ASR.	Arithmetic Shift Right (Sign Fill)
.ROR.	Rotate Right
.ASL.	Arithmetic Shift Left
.ROL.	Rotate Left

Additionally, a single unary operator ("+" or "-") may appear at the beginning of an expression, and the unary operator will be applied to the first term of the expression.

Note that the assembler can tell by the syntax of an expression if an asterisk ("*") is being used to denote multiplication or the present-location symbol.

The One-line Assembler computes the value of an expression from left to right with equal precedence for all

operators. For example, the expression "4+5/2-4*3" evaluates to "9/2-4*3", then to "4-4*3", then to "0*3", and finally to zero. Notice that integer division is performed, and the remainder, if any, is discarded. Since an expression is evaluated from left to right, parentheses are not meaningful and are not allowed in an expression.

3.3. One-line Assembler Pseudo Instructions

Like most assemblers, the One-line Assembler recognizes some pseudo instructions. Pseudo instructions, which are sometimes called assembler directives, don't usually correspond to machine-language instructions on a one-to-one basis as normal assembler-language instructions do. Instead, a pseudo instruction may tell the assembler to take some action, so the user directs the assembler's operation through the use of pseudo instructions. The remainder of this section contains descriptions of the various pseudo instructions.

FCB. The FCB (Form Constant Byte) instruction tells the assembler to evaluate the operand expression into an 8-bit value and to generate an object word that contains the 8-bit value of the expression. The FCB instruction is thus used for creating constant values that can be accessed by the program at execution time. If the value of an FCB

expression field does not fit properly into 8 bits, the assembler reports an error message.

The FCB instruction permits multiple expression fields separated by commas, and each expression field generates one word of object code. The location counter is advanced after each expression field of an FCB statement is evaluated. Therefore, the location-counter symbol ("*") in an FCB instruction always refers to the memory address of the word that is being generated by the expression that contains the location-counter symbol.

FCC. The FCC (Form Constant Characters) instruction tells the assembler that the operand field contains a character string. The string must be enclosed between two identical delimiters, each being a single nonblank printable character.

The following examples illustrate ways to code strings with the FCC instruction:

```
"Try everything you can."
?Does John go with us?
WIf music be the food of love, play on.W
D!#$%&"()_*+ Try try try go go go@ hope you win !!!D
```

The FCC statement generates one object word for each character of the string (not including the string delimiters, of course). The FCC statement is used to generate character strings that are normally used by programmers for outputting messages or headings at execution time.

Although any nonblank character may be used as a string delimiter, most programmers use quotation marks (") as string delimiters. A programmer who wishes to include a quotation mark as a character of the string ordinarily uses apostrophes (') as string delimiters. A user who wishes to include both quotation marks and apostrophes in a string usually chooses some other special character (such as !, $, %, &, =, :, +, <, -, /, or ?) as a string delimiter.

The One-line Assembler carefully examines the string that appears with an FCC statement, and it reports an error if anything appears to be amiss with the string. For example, the One-line Assembler reports an error if a nonblank character immediately follows the closing string delimiter because the user probably tried to use the string delimiter as a character in the string in this case. The One-line Assembler also reports an error if it never finds the closing string delimiter, and it similarly reports an error if the string is empty.

FDB. The FDB (Form Double Constant Byte) instruction is similar to the FCB instruction, but the FDB instruction generates a 16-bit constant that forms two consecutive object words. The most-significant half of the 16-bit expression value is put into the first object word, and the least-significant half of the 16-bit expression value is put into the second object word.

The FDB instruction tells the One-line Assembler to evaluate the operand expression into a 16-bit value and to generate two object words that contain the 16-bit value of the expression. The FDB instruction is thus used for generating double-precision constant values that can be accessed by the program at execution time.

The FDB instruction permits multiple expression fields separated by commas, and each expression field generates two words of object code. The location counter is advanced after each expression field of an FDB statement is evaluated. Therefore, the location-counter symbol ("*") in an FDB instruction always refers to the memory address of the first word that is being generated by the expression that contains the location-counter symbol.

SK1. The SK1 (Skip 1 word) instruction tells the One-line Assembler to generate a one-word instruction that will

skip the next one-word instruction during execution of the assembled (user's) program. By employing this instruction (as opposed to a branch instruction) the programmer will save one word of memory. For example, the following instruction sequences execute the same way, but the code on the right uses one less word of memory:

```
          .                          .
          .                          .
          BRA     1F                 SK1
DECX      DEX               DECX     DEX
1H        STAA    0,X                STAA    0,X
          .                          .
          .                          .
```

The generated opcode value ($85) for the SK1 instruction is the same value that is generated for the first word of a two-word BITA instruction with immediate addressing. When the SK1 instruction is executed, therefore, the microprocessor takes the following word as the second word of a BITA instruction with immediate addressing. The net effect is to skip one word. As a side effect, the SK1 instruction may modify the N bit, the Z bit, and the V bit of the condition codes.

SK2. The SK2 (Skip 2 words) instruction tells the assembler to generate a one-word instruction that causes the next two memory words to be skipped at execution time. Similar to the SK1 instruction, the SK2 instruction saves

the programmer one word of memory. For example, the following instruction sequences execute the same way, but the code on the right uses one less word of memory:

```
        .                               .
        .                               .
        BRA     1F              SK2
LOOP    ADDB    #4      LOOP    ADDB    #4
1H      STAB    2, X            STAB    2, X
        .                               .
        .                               .
```

The opcode value ($8C) that is generated for the SK2 instruction is the same value that is generated for the first word of a three-word CPX instruction with immediate addressing. When the SK2 instruction is executed, therefore, the microprocessor takes the following two words as the second and third words of a CPX instruction with immediate addressing. The net effect is to skip two words. The SK2 instruction may therefore be used to skip two single-word instructions or one double-word instruction during program execution. As a side effect, the SK2 instruction may modify the N bit, the Z bit, the V bit, and the C bit of the condition codes.

Chapter 4

One-line Disassembler Syntax

The One-line Assembler translates each source line into the proper M6800 machine-language code and stores it into memory on a line-by-line basis at the time of entry. In order to display an instruction, the machine code must be disassembled, and the instruction mnemonic and operands are displayed. Those jobs, disassembling machine codes and displaying instruction mnemonics and operands, are done by the One-line Disassembler.

The One-line Disassembler increases program readability when the user checks his or her program in memory. The disassembled source line may not, however, look identical to the source line that was originally entered for the instruction. The disassembler always outputs numerical values in hexadecimal preceded by a "$" sign. For example,

```
        LDX     3+5*4/2, X
```

disassembles to

```
        LDX     $10, X
```

The disassembler cannot output labels. It outputs the value of a label instead of the character string of the label. For example, the following source line is assembled by the M6800 assembler on the VAX:

```
        STX     SAVEX   The address of SAVEX is $108.
```

This instruction disassembles to

```
        STX     $108
```

When an instruction uses the self-relative addressing mode, the disassembler displays the actual hexadecimal address of the destination instead of displaying the displacement between this instruction and the target address. For example, the following source line is assembled by the M6800 assembler on the VAX:

```
        BRA     LOOP    The address of LOOP is $1200
```

This instruction disassembles to

```
        BRA     $1200
```

Also, for some instructions, there are two valid mnemonics for the same operating code, and the disassembler may choose a form different from the one originally entered. For example,

a. BITA with immediate addressing is returned for SKIP1.

b. CPX with immediate addressing is returned for SKIP2.

c. BCC is returned for BHS.

d. BCS is returned for BLO.

Any invalid M6800 operating code will be displayed in the following way: "FCB $<machine code> ": for example,

```
        FCC     "ABCDE"
```

disassembles to

```
        FCB     $41
        FCB     $42
        COMA
        LSRA
        FCB     $45
```

Since the One-line Disassembler is often used with the One-line Assembler, we should especially note one particular situation. Suppose that the user types the following instruction as an input to the One-line Assembler:

```
        FCB 1,2,3,4
```

The instruction length for this pseudo instruction is four bytes. The value four is therefore stored into the LENBUF area after this pseudo instruction has been processed by the One-line Assembler. Suppose that the user continues to input new assembly-language instructions or to check his or her program at subsequent locations in a forward direction. Later on, the user uses asterisk ("*") terminators to back up to this pseudo instruction, and a NOP assembly-language instruction is displayed at the user's terminal. This NOP instruction appears because the value

$01, which is the value of the first byte of the four-byte FCB instruction, is the opcode for a NOP instruction. The One-line Disassembler sees the $01 and assumes that it is the opcode of a NOP assembly-language instruction. The user can continue to check the contents of the FCB pseudo instruction's memory locations by typing carriage-return terminators.

The FCB 1,2,3,4

disassembles to

```
NOP
FCB   $02
FCB   $03
FCB   $04
```

The One-line Disassembler is a very useful debugging tool, but the user still needs to have a clear idea about his or her own program. For example, a user who tries to disassemble a string of ASCII characters will only create confusion.

Chapter 5

The Application of the One-line Assembler/Disassembler in MUDBUG

As mentioned in Chapter 1, this One-line Assembler/Disassembler software package is designed for MUDBUG to increase the functionality of certain MUDBUG commands. When a source program is assembled and downloaded to memory, what the user sees in memory is the machine-language version of the program. It is sometimes difficult for the user to understand and debug his or her program in machine language. By using this One-line Assembler/Disassembler software package, the user can inspect his or her program in memory and change the program by simply typing assembly-language instructions directly instead of typing machine codes. The following two sections describe the application of the One-line Assembler/Disassembler in the memory-change (i.e., "C") command and the memory-dump (i.e., "M") command of MUDBUG.

5.1. The Application of the One-line Assembler/Disassembler in the "C" Command

The original "C" command, which has one parameter, displays the current hex value of the contents of memory location START, and then it accepts a new hex value to be input into that location. The user can read his or her

program in machine language and modify the program by typing hex values. After the One-line Assembler/Disassembler software package is applied, the "C" command, which still has one parameter, displays the current assembly-language instruction that starts at memory location START. Then the "C" command accepts a new assembly-language instruction, assembles it to machine language, and puts the machine code into memory starting at location START.

If the new assembly-language instruction is terminated by a carriage return with no other termination character, the One-line Assembler/Disassembler automatically continues the "C" command by setting START <--- START + (instruction length) and performing the "C" command's function for the next instruction in memory. The notation "(instruction length)" as it is used here means the length of the new instruction, not the length of the old instruction. If the user doesn't type a new instruction, however, the old instruction is used as the new instruction.

If a comma terminator is used to terminate the new assembly-language instruction for location START, the One-line Assembler/Disassembler will keep the same value in the START parameter without changing it. Then the One-line Assembler/Disassembler will perform the "C" command's function again for the same START location in memory.

If an asterisk terminator is used to terminate the new assembly-language instruction for location START, the One-line Assembler/Disassembler sets START <--- START - (previous instruction length) and automatically performs the "C" command's function with the preceding instruction in memory. The Assembler/Disassembler remembers the lengths of as many as the last 20 instructions that have been input or examined with the current invocation of the "C" command. The user can therefore use the "*" terminator to back up over as many as 20 instructions or to the original START address, whichever occurs first. An attempt to back up beyond either limit causes the "C" command to stay on the instruction at the limit.

If the new assembly-language instruction is terminated by a period, the "C" command returns control to the top of MUDBUG after it installs the machine code of the new assembly-language instruction.

Finally, if a solidus ("/") terminator is used, the "C" command is aborted and control returns at once to the top of MUDBUG with no change to the contents of locations starting at START.

If a termination character (CR * , . or /) is input alone without a new assembly-language instruction, the

contents of the memory locations remain unchanged, but the function of the termination character regarding the continuation or termination of the "C" command is still effective. MUDBUG uses the instruction length of the existing instruction in memory when the user doesn't type a new instruction. Users can therefore examine several consecutive instructions rather conveniently by using carriage-return or asterisk terminators, and they need to type new assembly-language instructions only when new assembly-language instructions are actually desired.

The comma, period, asterisk, and solidus terminators are different from the carriage-return terminator. The user who types a comma, period, asterisk, or solidus terminator must still type a carriage return after the comma, period, asterisk, or solidus. When we refer to a carriage-return terminator, therefore, we are actually referring to the lack of any special termination character immediately preceding the carriage return that always terminates the input line to the One-line Assembler.

The user can correct extensive typing errors by typing a backslash ("\") to rub out the entire input line before the carriage return is typed to terminate the input line to the One-line Assembler. Then the user can type another assembly-language instruction right after the backslash

character. However, the recommended procedure is to type a backslash followed by a comma followed by a carriage return before restarting the input line.

The One-line Assembler/Disassembler allows the user to correct typing errors by using the backslash as many times as desired. The user must type an entire new assembly-language instruction instead of typing part of the assembly-language instruction after the backslash is typed. Once the terminating carriage return is typed, no further corrections can be made.

The user can correct minor typing errors by typing backspace character(s). When the user types a backspace, the One-line Assembler deletes one input character and erases it from the screen of the user's terminal. The cursor is left in the position of the character that was erased. The user can use backspace characters to erase input characters repeatedly until the entire input line has been erased. After all input characters have been erased, the One-line Assembler refuses to accept additional backspace characters and rejects them by ringing the bell at the user's terminal.

The user who wishes to erase a tab character from an input line is advised to use a backslash to delete the

entire input line instead of trying to use a backspace to delete the tab character. The One-line Assembler doesn't adjust the cursor position to account for the possible multi-column aspect of a tab character that is deleted, and the user who deletes a tab character with a backspace may become confused by the misleading information that appears on the screen.

The One-line Assembler accepts a maximum of 80 input characters on a single input line. After 80 input characters have been accepted into the input buffer, the One-line Assembler refuses to accept any more input characters and rejects any additional input character by ringing the bell at the user's terminal. The only exceptions to this rule are as follows. The One-line Assembler accepts a backspace or a backslash even when the input buffer is full because these input characters remove characters from the input buffer. The One-line Assembler also accepts a carriage return when the input buffer is full because a carriage return terminates the input line.

5.2. The Application of the One-line Assembler/Disassembler in the "M" Command

The original "M" command requires two parameters, and it dumps locations START through STOP in hexadecimal format

to the terminal. The memory dump actually starts with the location whose address is FLOOR(START/16)*16, so the hexadecimal memory address of the first word of the dump always ends in zero. The memory dump is printed with 16 values per line (except possibly for the last line, which may be shorter), and it is formatted for ease of readability. Also, the memory address of the first value on the line is printed at the beginning of each line. What users see on the terminal are the machine codes of the contents of locations START through STOP.

After the One-line Assembler/Disassembler software package is applied in the "M" command, the "M" command still requires two parameters, and it dumps locations START through STOP in assembly-language instruction format to the terminal. The memory dump is printed with one assembly-language instruction per line, and the memory address of each assembly-language instruction is printed at the beginning of each line. If STOP is in the middle of an instruction, the "M" command prints the entire instruction instead of truncating the instruction. Truncating the instruction would confuse the user, so the real stopping location of the "M" command is STOP or STOP+1 or STOP+2 depending on the instruction length of the last instruction processed. Control returns directly to the top of MUDBUG

after the assembly-language instructions from START through STOP are printed on the terminal.

The output of the "M" command is illustrated by the following sample memory dump.

Sample Dump

```
2000 7F 0075   CLR  $0075
2003 86 01     LDAA =$01
2005 B7 8008   STAA $8008
2008 B7 F800   STAA $F800
200B CE 0050   LDX  =$0050
200E 6D 13     TST  $13, X
2010 26 05     BNE  $2017
2012 A6 12     LDAA $12, X
```

Chapter 6

Conclusion

This report has discussed the internal routines of the One-line Assembler/Disassembler software package, the source syntax, expressions, and pseudo instructions of the One-line Assembler, the One-line Disassembler syntax, and applications of the One-line Assembler/Disassembler to the existing functions of MUDBUG.

The One-line Assembler/Disassembler software package has following restrictions:

a. Labels and line numbers are not used. Labels are commonly used in assembly language to refer to other lines and locations in a program. The One-line Assembler has no knowledge of other program lines and therefore cannot make the required association between a label and the label definition located on a separate line.

b. Source lines are not saved. In order to read back a program after it has been entered, the machine code is disassembled and then displayed as mnemonics and operands.

c. Limited error indication. The One-line Assembler shows only one error message, "INVALID: the whole input string", to the user. In contrast, the cross assembler generates specific error messages for different types of errors.

d. Only a few directives (pseudo instructions) are accepted.

e. No conditional assembly is used.

The One-line Assembler is a true subset of the M6800 cross assembler. The format and syntax that are used with the One-line Assembler are acceptable to the cross assembler. The One-line Assembler/Disassembler is a powerful tool for creating, modifying, and debugging M6800 code.

BIBLIOGRAPHY

MC68000 Educational Computer Board User's Manual.
Motorola Inc., 1982.

David C. Pheanis, MUDBUG-2/D2 User's Manual.
Department of Computer Science,
Arizona State University, 1982.

David C. Pheanis, M6800 Assembler User's Guide.
Department of Computer Science,
Arizona State University, 1981.

David C. Pheanis, MUDBUG-2 Source Program.
Department of Computer Science,
Arizona State University, 1983.