Design and Use of an Electronic Sieve

by

Cameron Douglas Patterson

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
The Department of Computer Science

Winnipeg, Manitoba, 1983

DESIGN AND USE OF AN ELECTRONIC SIEVE

BY

CAMERON DOUGLAS PATTERSON

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

ACKNOWLEDGEMENTS

The sieve project was initiated after a discussion with D. H. Lehmer and Robert Coffin. Doug Kimelman, Gilles and Gilbert Detillieux participated in the design and implementation of the software. Electronics problems were solved with the ready assistance of Jim Reimer. Brad Day helped to assemble the hardware. Glen Ditchfield's wirelist program aided construction immensely. Forbes Burkowski is especially thanked, for he was an inexhaustible source of advice and support throughout the project.

Hugh Williams is the patient user of the sieve, and he arranged funding for its construction. The author acknowledges the financial assistance of a Postgraduate Scholarship from the Natural Sciences and Engineering Research Council.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter I

INTRODUCTION

This document describes the use, evolution, design, and construction of an electronic device called a sieve. The device is a tool used in number theory to solve sets of linear congruences. The sieve performs 133 million operations per second, where one operation is defined to be the determination of whether an integer satisfies all congruences. This speed is not attainable on any von Neumann style computer currently known to the author.

There were no constraints upon the medium of implementation for the sieve. As a result, hardware, firmware, and software were created in a configuration that fulfilled the design goals. Although it is necessary to describe the sieve from several perspectives, the reader should regard the sieve as a unified system.

Since the author does not have extensive training in number theory, the applications of the sieve will not be explored in depth. It is the intent of this document to provide information to the users and maintainers of the sieve system.

Chapter II

THE SIEVE PROBLEM

Let $m_1, m_2, \ldots, m_k$ be integers > 1, and relatively prime in pairs. We specify permissible residues for each modulus m_i, as follows:

$$X \equiv r_{ij} \pmod{m_i}$$
$$i = 1, 2, \ldots, k$$
$$j = 1, 2, \ldots, n_i < m_i$$

The problem is to find a value for X that satisfies all congruences simultaneously; further, X must be within a certain range, say $A \leq X < B$.

There are three classes of sieve problems. If $n_i = 1$ for all i (i.e. we specify only one residue for each modulus), then X can be determined using the Chinese Remainder Theorem (CRT). An example of this type of problem is:

$$X \equiv 4 \pmod{5}$$
$$X \equiv 8 \pmod{13}$$
$$X \equiv 21 \pmod{31}$$

There is only one solution in the range given by A = 0, B = 5*13*31 = 2015.

If the first k prime numbers are used as moduli, and we specify all permissible residues except for zero (i.e. $n_i = m_i - 1$ and $r_{ij} = j$), the resulting problem is the famous Sieve

of Eratosthenes. Each X found cannot divide any of the m_i. Thus, if m_k is the largest prime not greater than $A = \sqrt{B}$, then all solutions in the range A to B will be prime. An example of a prime-finding sieve is:

$X \equiv 1 \pmod 2$

$X \equiv 1,2 \pmod 3$

$X \equiv 1,2,3,4 \pmod 5$

$X \equiv 1,2,3,4,5,6 \pmod 7$

This sieve produces all the primes between 10 and 100.

If roughly half of the residues are specified for each modulus (i.e. n_i is approximately $m_i/2$), then the problem is called a quadratic-type sieve. The name is well-chosen, since this situation arises when investigating second degree Diophantine equations and quadratic residues. An arbitrary X will have a 1 in 2 chance of satisfying a particular congruence. Hence, X will have a 1 in 2^k chance of satisfying all k congruences. The number of solutions expected is therefore $(B-A)/2^k$. The following exemplifies a quadratic-type sieve:

$X \equiv 1 \pmod 3$

$X \equiv 1,4 \pmod 5$

$X \equiv 1,2,4 \pmod 7$

Even a large computer becomes overwhelmed when attempting to enumerate all combinations of congruences in a quadratic-type sieve. Such a sieve with k moduli represents approximately $m_1 * m_2 * \ldots * m_k/2^k$ sieve problems that can be

solved using the CRT. Typical quadratic-type sieves use the first 35 prime numbers as moduli; this is the equivalent of $4*10^{46}$ CRT problems! Since a range is imposed on the solutions, it is necessary to solve all CRT problems, and check each result for inclusion in the range.

An alternative approach is to search sequentially the numbers in the specified range for solutions. This technique is acceptable as long as the search can proceed quickly. Unfortunately, general-purpose computers must serially apply the congruences to each number, and each congruence may require several machine instructions. In the next chapter, it will be shown how special-purpose hardware allows all congruences to be tested simultaneously.

Chapter III

THE SIEVE DEVICE MODEL

D. H. Lehmer was the first to employ custom hardware for solving the sieve problem. The connection between the concept of a modulus and the physical world was straightforward. If a closed loop or ring is divided into m discrete sections, the ring can represent the modulus m. After choosing which section of the ring will represent residue 0, we proceed to distinguish those sections corresponding to permissible residues in a given sieve problem. A ring will be constructed for each modulus. Finally, we will maintain a window on each ring that shows only one section at a time.

The algorithm for running the sieve problem is as follows:

1. Setup each ring so that the residue 0 section is in the window.
2. Check if all sections in the windows are distinguished. If so, we have a solution.
3. Shift all rings by one section.
4. Repeat from step 2.

When a solution is found, its value is the number of times that step 3 has been performed.

For the sake of speed, it is desirable to have all windows inspected simultaneously, and all rings shifted simultaneously. Table 1 lists the sieve devices that have been constructed by Lehmer.

TABLE 1

Lehmer's Sieves

DATE	DESCRIPTION	TRIALS/SEC	REFERENCES
1926	bicycle chains	60	1,7
1932	gears with photo-electric solution detector	5000	2,3,7
1936	16-mm film		7
1946 -	sieves on general-purpose computers		
	ENIAC		7
	SWAC		4,7
	IBM 7094	100,000	7
	ILLIAC IV	3,000,000	
1960's	delay line	1,000,000	6,7
1975	electronic shift register	16,000,000	7
1982	memory-driven microcomputer-based (under development)	?	

It might appear that the speed of a sieve device is limited strictly by the rate at which the rings can be shifted. However, using an idea suggested in [5], it is possible to test more than one number per ring shift. In a ring of length m, one can get the effect of s shifts in one shift time by loading the ring with residues in the order 0, s, 2*s, 3*s, ... (mod m). In order for all residues to be contained in the sequence, m and s must be relatively prime. A window is placed over residues 0, 1, 2, ... , s-1 (mod m). Hence, we initially have the value of the first s residues for each modulus. After one left shift of the ring, residues s, s+1, s+2, ... , 2s-1 (mod m) appear in the windows. Thus, we are checking for solutions s numbers per shift time.

Chapter IV

DESCRIPTION AND JUSTIFICATION OF THE DESIGN

Originally, it was planned to simply reconstruct the 1975 Berkeley electronic sieve, using a PDP11/03 microcomputer system as the user interface. Later, it was considered desirable to have a communications path between the 11/03 and a PDP11/45 minicomputer system. This would provide access to a variety of I/O peripherals, more secondary storage, and remote login. The final configuration omits the 11/03 entirely; the sieve is a direct peripheral of the 11/45. The sieve has its own microprogrammed control unit, and may be thought of as an array processor.

The above evolution is a consequence of the requirements of the sieve. Some of the more important objectives - and the ways of achieving them - are as follows:

1. The sieve must be capable of running unattended for months at a time. In particular, power failures should not require human intervention. The PDP11/45 automatically reboots when power resumes. The UNIBUS initialization signal causes the sieve microcode to enter a power-up sequence.
2. A user-friendly environment is required. The user is not assumed to be a programmer, or to be familiar

with the host operating system. The user simply deals with a workspace of active problems. Completed problems are automatically archived in order to keep the workspace uncluttered. The PDP11/45 UNIX system facilitates providing the desired environment.

3. Additional tests may have to be performed by the host computer on solutions detected by the sieve. For example, a user might wish to use moduli that are not implemented in the sieve hardware. This was the original motivation for including the PDP11/03 in the sieve system architecture. It was feared that the PDP11/45 system could not provide the response required to keep the sieve from waiting. Increasing the priority of the sieve process would be unacceptable to the other users of the 11/45. As it turns out, most sieve problems do not require a great deal of software testing. In addition, the superior speed of the 11/45 over the 11/03 more than offsets the fact that the 11/45 is not dedicated to servicing sieve interrupts.

4. The sieve system must be reliable. The floppy disk-based PDP11/03 has a greater likelihood of I/O errors, and does not perform parity checking on memory reads. The PDP11/45 system has an order of magnitude greater Mean Time Between Failure than the 11/03 system, even though the 11/45 system is more

complex. As well, the UNIX operating system has an excellent record for robustness.

Versatility is deemed the most important attribute of the sieve. Even speed is subordinate to this goal, although a minimum speed of 100 million trials per second was enforced during the design. Examples of some of the tradeoffs are:

1. It was known in advance that some problems would require moduli other than those implemented in hardware. As a result, a convenient and minimum number of hardware moduli was used.
2. Solution counting cannot proceed at full sieve hardware speeds, since the PDP11/45 is interrupted for each solution. Although it would have been possible to include solution counters in the sieve hardware, the problems that required software moduli could not exploit the solution counters. Eight solution counters would be necessary - one for each solution detector. The added hardware complexity could not be justified.
3. The hardware clock counter has a 100 day period. Overflows are remembered in software.
4. The sieve can operate at a cycle time of 50 ns, but a ring bit will flip on the average of once per day. A 60 ns cycle time was chosen, with the result that a ring bit flips roughly once per month.

The checkpointing performed by the sieve background process not only records the sieve hardware state, but verifies it as well. Knowing the initial state of the rings, and how far the counter has advanced, it is easy to predict the new contents of the rings. This checking gives the user confidence that the sieve hardware is performing properly. In addition, all sieve unit writes are immediately verified by a read of the unit. Solutions generated by the sieve are not verified by the sieve software, because this should rightfully be done by independent software. Also, the extra overhead would slow down the solution counting mode even more.

Table 2 indicates the division of functions between the sieve device and the host computer. An overriding concern is to keep the communication between the two units to a minimum. Solutions require only 4 16-bit data transfers, and the first provides the initial interrupt. Reads and writes of m-bit rings are performed in $\lceil (m/16) \rceil$ data transfers.

TABLE 2

Host Computer / Sieve Peripheral Division of Labour

HOST RESPONSIBILITIES	SIEVE RESPONSIBILITIES
user interface	generate interrupts for solutions, counter overflow
integrity checks	execute commands from host
initiate power up sequence	operate as a complete slave to the host (impose no real-time constraints upon it)
implement less common moduli	implement most common moduli
non-volatile data storage	

Chapter V

HARDWARE

It was decided not to thoroughly describe the sieve hardware in this document, because of the size and nature of the description. The board layouts, wiring lists, and backplane layouts remain in the possession of the author, and are decodable only by him.

However, some high-level hardware documentation is provided. The integrated circuit and packaging technology chosen will be summarized. Each major subsystem of the sieve will be characterized. A sample of the wirelist program output is included in Appendix D. Finally, the design circuit diagrams and the interrupt system timing diagram are folded in a pocket inside the back cover.

5.1 DESIGN OVERVIEW

The rings defined in Chapter 3 are implemented as recirculating shift registers. There are 32 of these, corresponding in length to the first 32 prime numbers or their powers. The first 4 rings have length 64, 81, 25, and 49 for two reasons:

1. a ring must be at least 8 bits long
2. this accomodates common moduli, such as 8.

Each one bit in a ring denotes a permissible residue. A 48-bit counter keeps track of the number of times that the rings have been shifted. There are 8 solution detectors (i.e. 8 numbers are tested per shift). A solution detector is logically a 32-input AND gate. The sieve hardware has a cycle time of 60 ns, and is completely synchronous. Maximum speed is therefore one trial every 60/8 = 7.5 ns, or 133 million trials per second.

A design requirement for the machine was that there be no overrun of the rings when a solution is found. It is undesirable to have to "back up" the rings, in order to continue the search for solutions. Hence, the sieve has to "stop on a dime", rather than "coast to a stop". The rough equivalent of momentum in electronics is the time required for signals to propagate through gates and wires. Setup and hold times must be respected. Global signals such as the clock and interrupt lines have a large fanout, and require a multi-level distribution tree. Such trees inevitably introduce skewing of the signals in different parts of the machine. In order to prevent overrun, the solution detection logic has a 1-level pipeline. The rings are shifted every cycle, although it takes 2 cycles for a solution to halt the machine. The search for solutions continues by restarting the rings and the solution detection pipeline in unison.

5.2 PHYSICAL CONSTRUCTION

The sieve consists of over 400 integrated circuits, packaged on three high density wirewrap boards. Advanced Schottky and low-power Schottky TTL are used exclusively. The 8-bit serial-in parallel-out shift registers are military grade, for enhanced reliability. Unfortunately, longer length serial-out shift registers could not be used, because of the distribution of the 8 windows in a ring. Including a 35 amp power supply and cooling fans, the sieve device occupies about 2 cubic feet.

Although the sieve is tightly packaged, some taps from the rings to the solution detectors are longer than they should be, and signal relections occur. Those taps that experience severe reflections have parallel resistor termination at the receiving end. This does not eliminate the problem, because of impedance mismatches along the taps (impedance is hard to control on wirewrap boards and backplanes).

Ripple in the power and ground distribution systems is a problem, because of the large number of devices that are clocked simultaneously. In order to reduce this ripple to a safe level, boards were chosen that have nearly continuous power and ground planes. There is at least one ceramic disc decoupling capacitor per integrated circuit, and several dozen tantalum electrolytic decoupling capacitors per board. The power supply has approximately 0.2 farads of load capacitance.

The wirewrap wire colour convention is as follows:

black : clock distribution

red : interrupt lines

blue : taps from rings to solution detectors

white : all else

5.3 SIEVE HARDWARE SUBSYSTEMS

5.3.1 Control Unit

The functions of this circuitry are described in Chapter 6. Note that the control unit is clocked independently from the other subsystems. The control unit is on the top board of the rack.

5.3.2 Host Interface

The job of parallel-to-serial and serial-to-parallel conversions is accomplished by 74199's. A 74LS161A is used to control the number of shifts. Line drivers and series damping resistors are used for signals sent to the host. Parallel resistor termination and line receivers improve the fidelity of signals received from the host. These measures allow for a flat cable length of 25 feet from the host computer to the sieve (this is the maximum length allowed by the DEC DR11-C parallel interface). All host interface circuitry is on the top board of the rack.

5.3.3 Counter

The 48-bit counter is implemented by 12 4-bit synchronous binary counters. The least significant counter is a 74S161, while the remainder are 74LS161A's. Counter overflow is detected when the most significant counter asserts its Terminal Count output. The broadside load and read capability of these chips is exploited. For the purposes of reading and writing, the counter is split up into 3 individually addressable 16-bit units. The counter circuitry is on the top board of the rack.

5.3.4 Solution Detectors

A solution detector is implemented by three parallel 13-input NAND gates, followed by an AOI gate. The result implements the AND function with roughly a 12 ns propagation delay. The output of the solution detectors is pipelined by D-type flip-flops. All interrupt sources are ORed together to form a single interrupt line. This signal is then distributed via a 1-level distribution tree to the AND gates that drive the clock inputs of the rings and counter. The other inputs to these AND gates are the clock line, and unit select line. The solution detectors occupy the center portion of the middle board in the rack.

5.3.5 Rings

All rings are implemented using AMD 25LS164DM's. All shift registers in a ring share the same clock driver. A 2-input OR gate is spliced into a ring in order to write the ring. The first ring is special in that for the purposes of reading or writing, it appears as a single 64-bit recirculating shift register. However, it splits up into 8 8-bit recirculating shift registers while the sieve is searching for a solution. This transformation is accomplished by the use of a 2-input multiplexer between adjacent shift registers. The first 8 rings are located on the left side of the middle board. The next 12 rings are located on the right hand side of the middle board. The remaining 12 rings occupy the entire bottom board in the rack.

5.3.6 Clock System

The clock signal for the host interface, counter, solution detection pipeline, and rings is generated by an AMD 2925. While the sieve is searching for a solution, the crystal frequency is used. During I/O operations, the crystal frequency is divided by 10, and the single step facility of the 2925 is exploited. This 2925 is located on the middle board of the rack.

5.3.7 Remote Control

A handheld remote control box connects to the backplane of the rack. Basically, it controls the operation of the AMD 2925 oscillator that clocks the control unit. The control unit may be reset or single stepped from one microinstruction to the next. A multiplexer defines the signals affected by the remote control box when it is disconnected. The sieve will be in a "run" state when the remote control box is disconnected.

Chapter VI

FIRMWARE

6.1 MICROPROGRAMMING MODEL

Microprogramming is the use of a programmed "engine" to control a larger machine. A simple microengine configuration is shown in Figure 1. The most important unit is the m word by n bit memory, which is the source of the control signals. The memory is driven by a sequencer that simply determines the next address to be issued to the memory. The next-address selection defaults to the current address + 1 (modulo m). Alternatively, the next address is a function of addressing information in the current memory word, external data signals, and addresses "remembered" by the sequencer. All of the control variations permitted on von Neumann computers are usually available in microprogramming: sequential, unconditional and conditional jumps, subroutine calls and returns, bounded and unbounded loops, and even multi-way branches (case statements).

The "pipeline register" (Advanced Micro Devices' terminology) exists to allow the microengine to operate faster. Without it, the cycle time would include the access time of the microprogram memory. The pipeline register holds the current microinstruction, while the next

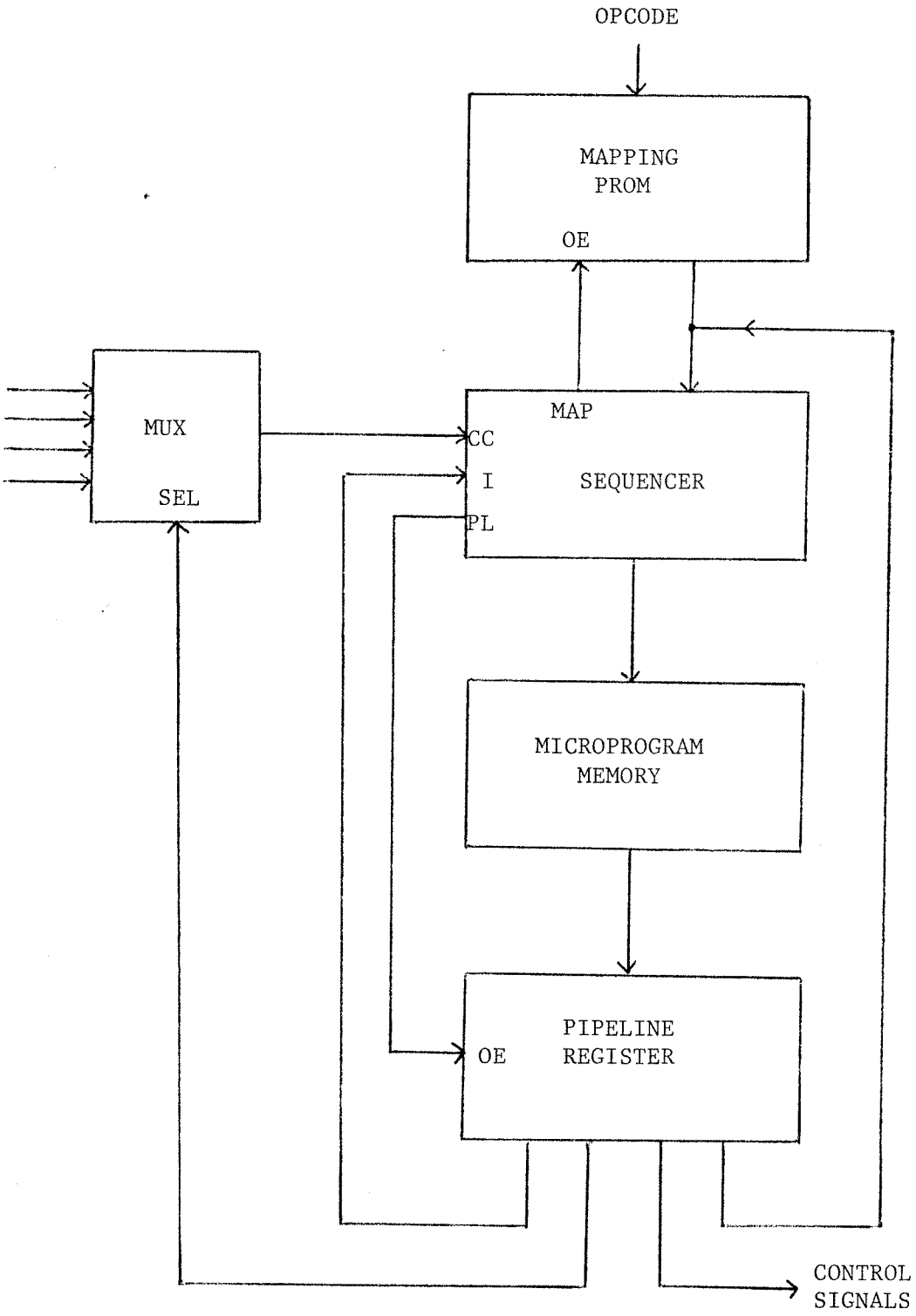


Figure 1: Basic Microengine Configuration

microinstruction is being addressed in the memory. Thus, the memory access time has been removed from the critical path.

The use of microprogramming to generate control signals in a digital machine is analagous to the use of structured programming to manage control flow in software. In both cases, a simple solution helps to solve a complex problem. A structured approach to control has the following advantages, in either a hardware or software environment:

1. The chances of coming up with a close-to-error-free design are improved when fundamental constraints are imposed upon control. Independently-designed modules are more easily integrated.
2. Debugging is easier, because errors can be isolated. Knowing that certain modules cannot interact helps to narrow down the possibilities.
3. Fixes or enhancements can be made with more confidence that we will not corrupt something already working.

Thus, we see that all phases of a hardware or software project benefit from controls on control.

6.2 SIEVE CONTROL UNIT

A schematic of the sieve control unit is given in the back cover pocket. It is very similar to the basic microengine model previously described. The Am2910 is a bipolar microprogram controller that can access up to 4K words of microprogram. It executes 16 different microprogram address modification instructions. The address provided on the Yi outputs may come from the Di input pins, the internal register/counter, microprogram counter, or stack. Five Am27S29 512x8 bipolar PROM's are arranged horizontally to give an overall microprogram memory size of 512 40-bit words. Am2920 octal registers are used to implement the pipeline register. Finally, an Am2922 octal multiplexer and an Am2920 allows the Am2910 to interrogate one of eight values to determine the outcome of conditional instructions. All clocked chips share the same 1 Mhz clock signal, which is produced by an Am2925 system clock generator.

It should be noted that the sieve control unit runs asynchronously with the rest of the machine. The control unit does not influence the speed of the rings. In fact, the control unit has a cycle time of 1000 ns, whereas the rings have a cycle time of 60 ns. While the sieve is searching for a solution, the control unit polls for assertion of the solution found signal, or for an instruction from the host computer. If a solution occurs, the rings and counter stop without assistance from the

control unit. Indeed, the control unit's only functions are to decode and execute instructions from the host computer, and to interrupt the host when a solution is detected.

6.3 MICROINSTRUCTION FORMAT

Each microinstruction field will be described in the following manner:

field mnemonic (bit positions) : field description

A field mnemonic having an "@" suffix indicates the signal is active low.

opcode (0-3) : Am2910 instructions. Only 4 of the 16 possible instructions are used. They are: conditional jump-to-subroutine, conditional jump-pipeline, return-from-subroutine, and continue.

address (4-12) : Am2910 direct input to the register/counter and multiplexer (i.e. pins D_0 - D_8). Pins D_9 - D_{11} of the 2910 are tied low. This field is used to directly access the 512-word microprogram memory.

me22@ (13) : Am2922 condition code multiplexer enable pin. This pin is held low throughout the microprogram.

re22@ (14) : Am2922 condition code multiplexer register enable pin. This allows the 2922 selection inputs (that are determined by the abc22 field) to be changed.

pol22 (15) : Am2922 condition code multiplexer polarity control pin. A low value on the Am2910 condition code input implies truth. A low value on pol22 inverts the signal that passes through the 2922.

abc22 (16-18) : Am2922 condition code multiplexer input selection pins. This field encodes the signal number that the Am2910 wishes to interrogate.

reqa (19) : DR11-C parallel interface "REQA" signal. It is used to acknowledge the receipt of data, or the execution of a command, by the sieve.

reqb (20) : DR11-C parallel interface "REQB" signal. It is used to interrupt the host computer when the sieve has data for it.

le373 (21) : 74S373 latch enable pin. This chip latches the ring or counter unit number that is contained in an instruction from the host computer to the sieve.

filler_1 (22) : Reserved for future use.

ccen@ (23) : Am2910 condition code enable pin. A high signal forces all conditional instructions to be unconditionally executed.

peoutbuf@ (24) : 74199 broadside load enable pins for the DR11-C parallel interface output buffer. On the rising edge of the clock, 16 data bits from the host computer to the rings or counter are loaded into a pair of 74199's.

peinbuf@ (25) : 74199 broadside load enable pins for the DR11-C parallel interface input buffer. On the rising edge of the clock, 16 data bits from the counter to the host computer are loaded into a pair of 74199's.

choose (26) : When asserted, only the unit (ring or counter) that is selected in the unit number demultiplexer will receive the clock signal. When deasserted, all units receive the clock signal.

feed (27) : When asserted, data can be serially introduced into a ring, or broadside-loaded into a counter, on the next rising edge of the clock. When deasserted, rings recirculate the data and counters count up, on the next rising edge of the clock.

dis_clk@ (28) : Prevents the sieve from halting due to counter overflow.

halt@ (29) : This signal is the input to a D-type flip-flop that is clocked synchronously with the rings and counter. The output of this synchronous halt flip-flop jams the sieve clock signal at the proper place in the clock period. Assertion and deassertion of halt@ is the means by which the control unit stops and starts the sieve looking for a solution.

dis_1@ (30) : Prevents the sieve from halting due to a solution detected on coincidence gate number 1.

dis_28@ (31) : Prevents the sieve from halting due to solutions detected on coincidence gates 2 through 8. In the current version of the microcode, this signal is manipulated identically to dis_1@. However, by asserting dis_28@ all the time, the sieve reverts to having just one coincidence detector (i.e. it checks for solutions one number at a time). This would allow the sieve to operate - at one-eighth speed - should any but the primary solution detector malfunction.

clk_halt@ (32) : Drives the run/halt pin of the Am2925 system clock generator. This master clock supplies the rings, counter, and host interface unit. When clk_halt@ is deasserted, the clock is free-running (although it may be jammed by solution detectors, counter overflow, or the halt@ signal). When clk_halt@ is asserted, the clock may be single-stepped.

clk_ss@ (33) : Causes the Am2925 master clock to produce a single square wave. The period of this wave is ten times the period of the free-running clock signal (i.e. the single-step period is 600 ns).

set_halt@ (34) : Connected to the direct clear input of the synchronous halt flip-flop. The output of this flip-flop must be low before the clk_halt@ signal is deasserted.

clr_halt@ (35) : Connected to the direct set input of the synchronous halt flip-flop. The output of this flip-flop must be high before the clk_ss@ signal is asserted. This signal also clears the presence of solutions from the solution detection pipeline.

ce199 (36) : Permits the sieve master clock signal to reach the 74199's that are in the host interface unit.

pe161@ (37) : Permits a broadside load into the 74LS161A that is used to limit the number of times a ring is cycled during a read or write ring operation. The load is accomplished on the next rising edge of the master clock.

ce161@ (38) : Permits the sieve master clock signal to reach the 74LS161A that is used to limit the number of times a ring is cycled during a read or write ring operation.

filler_2 (39) : Reserved for future use. Note that filler_1 is also available, and it is physically closer to the backplane.

Appendix C contains a documented listing of the sieve microprogram.

6.4 SIEVE INSTRUCTION SET

A DEC-supplied DR11-C parallel interface connects the sieve device to the UNIBUS of the host computer. Instructions and data are sent to the sieve, and data are received from the sieve, in 16-bit parcels. The transmission of data from the host computer to the sieve is done in lockstep, and is acknowledged with a handshake. The transmission of data from the sieve to the host computer is interrupt driven, and is also acknowledged with a handshake. Figures 2 and 3 describe the communications protocol in detail.

The format of a sieve instruction is:

opcode (o)	unit number (n)	shift count (c)	unused
(3 bits)	(6 bits)	(4 bits)	(3 bits)

least significant bits --> most significant bits

The bit ordering is from the perspective of the host computer. Data bits are used by the sieve beginning with the most significant bits. Data bits are generated by the sieve beginning with the least significant bits.

The opcode field specifies one of 8 possible instructions for the sieve device to execute. The instructions will be enumerated after the other fields are described.

The unit number field selects one of the rings (0-31), a 16-bit portion of the counter (32-34), or the output value of the coincidence gates (35).

HOST	SIEVE
(1) assembles command/data	{polling for CSR0 set}
(2) if REQA set, give "sync error"	
(3) write to DROUTBUF	
(4) assert CSR0	
{polling for REQA set}	(5) executes command/data when CSR0 set
	(6) asserts REQA
(7) if REQA not set after 100 microseconds, give "no response" error	{polling for CSR0 clear}
(8) deassert CSR0	
	(9) deassert REQA

Figure 2: Protocol for Write from Host to Sieve

The shift count determines the number of data bits to be used in a read or write ring command. An encoded value of n (where n is between 0 and 15) results in n+1 data bit shifts.

The instructions executed by the sieve are:

HOST	SIEVE
	(1) asserts REQB
(2) interrupt generated by REQB	{polling for CSR1 set}
(3) read DRINBUF	
(4) assert CSR1 to acknowledge data read	
{polling for REQB clear}	(5) deassert REQB
(6) if REQB set after 10 microseconds, give "REQB stuck" error, and disable interrupt capability of REQB	
(7) deassert CSR1	

Figure 3: Protocol for Write from Sieve to Host

read ring (o = 0) : The host computer needs the value of the next c bits of ring number n. The sieve responds with the data requested. Note that $\lceil (m/16)$ instructions are required to read an m-bit ring.

read counter (o = 1) : The host computer needs the value of counter unit n. The entire counter is 48 bits, but it is broken up into 16-bit units for the purposes of reading and writing.

write ring (o = 2) : The host computer needs to write c bits of data to ring number n. The data value is sent to the sieve after it has acknowledged the instruction. Note that $\lceil (m/16) \rceil$ instruction-data pairs are required to write an m-bit ring.

write counter (o = 3) : The host computer needs to write to counter unit n. A 16-bit data value is sent after the instruction is acknowledged.

go (o = 4) : The sieve is instructed to search for a solution using all units. The sieve will interrupt the host when a solution is found. This instruction should be issued with n=35 so that the coincidence gate values are presented during the interrupt.

only (o = 5) : The sieve is instructed to search for a solution using only unit number n. This instruction permits isolated testing of units, and is therfore used in diagnostics.

stop (o = 6) : This instruction causes a simple ackowledgement. As a side effect, it terminates an outstanding go or only command. Because of its simplicity, it is also used to test if the sieve control unit is alive.

pulse (o = 7) : All sieve rings and the counter are single-stepped. This is a diagnostic instruction.

Chapter VII

SOFTWARE

The essential sieve software consists of three programs: the "sieve" command, the sieve background process "sivmon", and the diagnostic program "sivdiag". The characteristics of these programs will be discussed. The files and data structures used by the above programs will also be described.

7.1 THE SIEVE COMMAND

This interactive program is the sole user interface to the sieve system - no other shell commands are required. Use of the sieve command is documented in Chapter 8.

Error detection is a significant part of this program. The return code of every I/O system call and request for memory is checked. Complete syntax and semantic error checking is performed on all user input. If an error is found, it is first reported to the user, and then as much "cleaning up" is done as possible (e.g. outstanding memory allocations are freed, files are closed, and the command is terminated). The program never aborts itself. Unfortunately, the multiprecision package aborts if a memory allocation request fails.

As a precaution, copies are made of the queue and done files before they are modified. These backups have the name ".../files/temp/backup.xxxxxx", where xxxxxx is obtained from the "mktemp" UNIX utility. The backup of the queue index and string files remains in existence throughout manage mode. A backup of the done index and string files is made during execution of the retrieve command.

Signals must not be allowed to terminate the sieve command. The hangup signal is ignored. The quit signal will abort the sieve command, and should be used only to debug the command. The interrupt signal sets a flag that is interrogated by some commands. These signals are set to their default value, though, before invocation of the editor or a shell escape.

7.2 THE SIEVE BACKGROUND MONITOR SIVMON

The environment of sivmon is very different from the sieve command. Sivmon is a non-interactive program that can send messages to the user only indirectly through the log file. The standard input, standard output, and standard error files are not used. Much effort was expended to make sivmon compact and efficient, since this process is ready or running almost all the time.

The error-handling philosophy of sivmon is also very different from that of the sieve command. There are two classes of errors: recoverable and unrecoverable. Only

queue file errors and unsuccessful sieve device commands are considered unrecoverable. An unsuccessful sieve device checkpoint (e.g. a ring is found to have flipped bits) causes a backup to the previous successful checkpoint, and the range of numbers where the error occurred is retried. The retry will be performed up to n times, where n is the value of the constant MAX_N_BACKUP. If this limit is exceeded, sivmon moves on to the next problem in the queue.

Both recoverable and unrecoverable errors generate a log file message. The maximum number of recoverable errors tolerated per invocation of the sivmon process is given in the constant MAX_N_ERROR. If this maximum is exceeded, an unrecoverable error is signalled. Unrecoverable errors cause sivmon to abort with a core dump. Hence, sivmon should execute in a directory that does not have nightly "core" file removals.

Although precautions are taken, sivmon is not immune to errors resulting from a system crash during an I/O operation. The results are unpredictable when sivmon is reinvoked.

The monitor process ignores the hangup, interrupt, quit, and broken pipe signals (a broken pipe is associated with the filter program, and causes sivmon to terminate the current problem with an error status). The terminate signal causes sivmon to perform an orderly shutdown. Any outstanding transaction with the filter program is allowed

to complete. In fact, the terminate signal may initiate a new filter program transaction, if a potential solution is encountered during the checkpoint.

7.3 SIVDIAG

Sivdiag is an unsophisticated interactive program to aid in sieve hardware troubleshooting. It cannot be run while sivmon is active. Sivdiag gives the user manual control of all sieve hardware functions, including some diagnostic operations that are not used by sivmon. The help command "?" provides a description of the functions available. Sivdiag does not use the normal read and write system calls to access the sieve device, since timeout errors will result if the sieve device does not respond in a certain amount of time. Communications protocol may be performed as slowly as desired, thereby allowing the sieve device to be single-stepped via its remote control box.

7.4 FILES

Note that the following file names are not fully qualified. The directories for these files are assigned by the UNIX administrators.

queue.index

This file contains problems that are accessible in manage mode of the sieve command. There is a header at the beginning to store global values that must be

remembered from session to session of the sieve command (e.g. the last problem number assigned). The fixed-length data defining each problem follows. Problems are stored in order of execution by sivmon (i.e. ascending problem number within descending priority). This file and the queue.string file are completely rewritten each time the sieve command's manage mode is quit.

queue.string

This file contains the variable-length character string data defining each problem in the queue.index file. There is a set of 5 character strings associated with each problem. The character string sets are in the same sequential order as the problems in the queue.index file. Each character string is terminated with a newline character, to aid manual viewing and editing, should the need arise.

done.index

This file has the same format as queue.index, except there is no header. It contains the fixed-length data on problems that have a "done" status (i.e. it is an archive). Whenever manage mode of the sieve command is quit, queue problems with a status of "done" are appended to this file. This file and the done.string file are completely rewritten each time the retrieve command is used.

done.string

This file is analagous in function to queue.string, except that it works in conjunction with done.index.

sivmon.lock

This file exists only while the sivmon background process is alive. It contains the process number of sivmon, so that the sieve command can terminate sivmon. It also performs a secondary function of allowing only one sivmon process to run at a time.

sieve.lock

This is an empty file, used to prevent multiple concurrent invocations of the sieve command. It is created at the beginning of the sieve command, and removed when the command is quit.

sivmon.log

This is a text file, containing messages from sivmon to the sieve command user. Sivmon appends messages to this file. At the end of a sieve command session, the user has the option of deleting the file.

problem/xxx

This file contains the binary equivalent of the user ASCII problem file whose unqualified name is xxx. Such a file is created by a successful test or ptest command in the prepare mode of the sieve command.

Both the ASCII and binary problem files are deleted by the destroy command. The binary problem file is read into memory when sivmon starts on a problem that specifies problem file xxx.

result/nnn

This file contains data generated from the execution of problem number nnn by sivmon. This file is created by sivmon the first time that it executes the problem. There is a small fixed-length header at the beginning of the file. Solutions follow in a variable-length binary format. At the end of the file are the solution count and clock count values, which are in the same format as solutions. Actual solution and clock count values are determined by adding the problem starting value (which is stored in the queue.string or done.string files) to what is stored in the result file.

temp/chkp.nnn

This file exists only while sivmon is executing problem nnn. The solution count and clock count are stored in this file during every checkpoint. If the system were to crash while problem nnn is running, then the values in this checkpoint file will be used to restart the problem. The solution count and clock count are appended to "result/nnn", and

"temp/chkp.nnn" disappears when sivmon terminates the problem, or when sivmon itself is forced to terminate. Note that "temp/chkp.nnn" is necessary because "result/nnn" is potentially growing all the time, as solutions are appended to it.

temp/spool.xxxxxx

This file contains ASCII text to be printed. It is created when a sieve command user enters a command that routes its output to the printer. "xxxxxx" is filled in by the UNIX utility "mktemp". The file is removed after the printing is complete.

temp/backup.xxxxxx

This is a temporary copy of a file such as queue.index, queue.string, done.index, or done.string. These backups are created and deleted automatically during the execution of the sieve command. The user can manually replace a damaged file with the backup, if the backup still exists.

temp/hardbits

This file is created and deleted automatically during execution of the test or ptest commands in prepare mode of the sieve command. It contains the residue bit strings of all hardware moduli encountered in the problem file being translated.

temp/virtbits

This file is created and deleted automatically during execution of the test or ptest commands in prepare mode of the sieve command. It contains the residue bit strings of all virtual (non-hardware) moduli encountered in the problem file being translated.

Chapter VIII

USER'S GUIDE TO THE SIEVE

A sieve user interacts with the machine strictly through software. The command modes and commands are described later in this chapter. The user need only have a rudimentary knowledge of the UNIX operating system in order to use the sieve. After logging in, the user should type "sieve" in response to the shell prompt ("$ " in Version 7 of UNIX). After a delay, a colon (":") will appear as the new prompt.

Messages may appear before the first colon prompt. These messages are of the form:

date time problem # message type: text of message

These messages were generated by the background program "sivmon" while it was running "problem #" at the date and time specified. "Message type" is one of:

error : an abnormal situation occurred that was handled by sivmon. Examples are: running out of memory, file errors (except for the queue file), and unsuccessful sieve checkpoints. Most errors cause termination of the problem (with an error status), and selection of the next problem in the queue.

fatal : an abnormal situation occurred that caused sivmon to abort. Queue file I/O errors and sieve hardware errors are usually fatal.

"Text of message" succinctly describes the error context.

If any messages did appear, the user has the option of saving them or deleting them when he exits from the sieve command. If they are saved, then they will reappear the next time that the sieve command is issued. Any new messages will be appended to the previous messages.

The sieve software does not support multiple simultaneous users. Errors could result if the sieve command was issued while another sieve session was already in progress. Therefore, the first action performed by the sieve command is to check if the lockout file "sieve.lock" exists in the sieve file directory. If so, the command terminates with the message

sieve command already in use by userid

where userid is the user identification of the person currently using the sieve command. If the lockout file does not exist then it is created, and remains in existence until the user exits from the sieve command. Note that if the sieve command was to end abnormally, no further sieve sessions would be permitted. In this case, the lockout file must be manually deleted.

It is suggested that the sieve command be issued only by the "sieve" userid. This promotes consolidation of all

sieve-related work. It also reduces the likelihood of a problem with the preparation of problem files. Qualification of a problem file name is not retained when the bit string equivalent is created in the sieve file directory. As an example, the preparation of a problem file ending in "/cubres" will overwrite the bit string file for any other problem file ending in "/cubres". Therefore, all problem file names must be unique.

8.1 UNIVERSAL COMMAND ATTRIBUTES

Before describing the commands available in the sieve program, the features common to all commands will be given:

1. Command names may be abbreviated, by omitting characters from the right end of the name. The abbreviation may continue until the command name becomes ambiguous, or until just one character remains. There are exceptions:
   a) commands that destroy information must be spelled out in full
   b) if an ambiguous command matches a common command name, then the common command name is assumed.
2. Command names may be in upper and/or lowercase.
3. The operands to a command may be specified on the same line as the command name, or they may be omitted. Since blanks separate operands, and operands are positional, only the rightmost operands

may be omitted. A prompt is made for required but omitted operands.

4. Commands are separated by carriage returns and/or semicolons. However, commands separated by semicolons may not span sieve command modes.

5. The standard input to the sieve command may be redirected to a script file. Reprompts, though, will accept their information from the terminal only.

6. Shell commands may be issued while in the sieve command. A shell command is distinguished by an exclamation mark as its first character. All characters following the exclamation mark, and up to but not including the carriage return are passed to an invocation of the shell. When the shell command terminates, another exclamation mark is displayed.

7. All data that is typed at the sieve command undergoes syntactic checking before command execution starts. A reprompt is made for syntactically incorrect strings.

8. Control values may be typed in place of normal values. Control values begin with the at sign.

   @? : help. It generates a brief description of what may be entered at that point in the command line. A prompt is then made for the command or argument. Note that "@?" in place of a command name gives the list of commands.

@> or @ : explicit null. It has a special significance to certain commands (e.g. modify). It is different from just a carriage return in that it indicates that a token has a null value.

@q or @Q : abort the command. Any reprompting will cease.

9. The hangup signal is ignored by the sieve command. The quit signal (cntl-backslash) aborts the sieve command with a core dump. Note that this can trash files, and should only be used to debug the sieve command program. Scrolling of displays may be done with the cntl-s and cntl-q keystrokes. Entire input lines may be deleted with the cntl-u keystroke. The point at which the interrupt signal takes effect in a command is given in the individual command descriptions.

The commands available to the sieve user form a hierarchy, as illustrated in Figure 4. Command name references in this chapter will normally be qualified, in order to indicate the appropriate mode (e.g. manage.list).

Appendix B illustrates the use of some of the common commands.

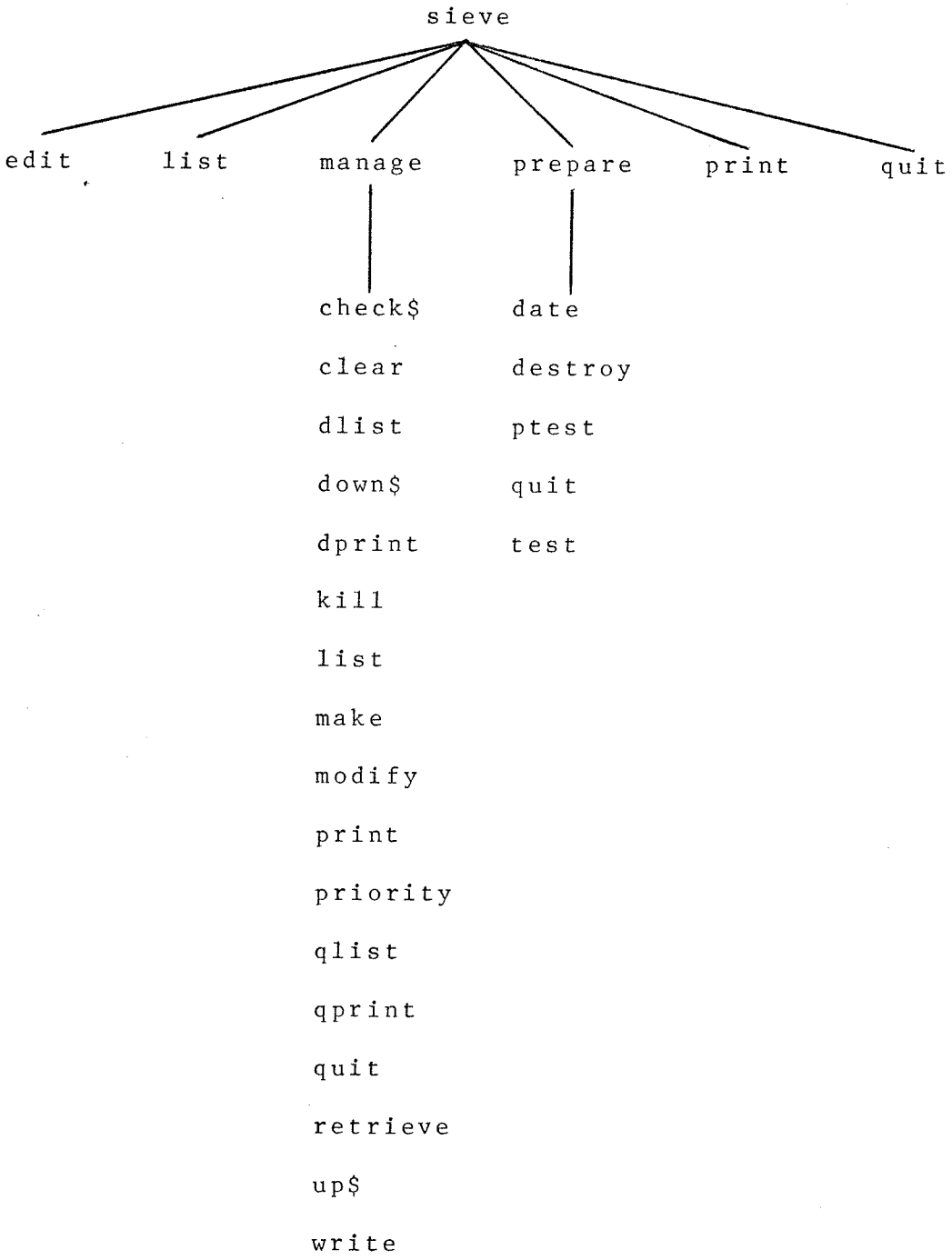


Figure 4: Sieve Command Hierarchy

8.2 COMMAND DESCRIPTION KEY

PROMPT : The prompt which must be displayed before the command name can be typed.

COMMAND : The full command name is given first. The shortest possible abbreviation of the command name follows in parentheses.

OPERANDS : The command operands are given in the order required. Square parentheses mean that the enclosed operand may occur 0 or 1 times. Curly parentheses mean that the enclosed operand may occur 1 or more times. A vertical bar indicates alternation. The type of each operand is enclosed in parentheses, and follows the name assigned to the operand. The operand types are:

number - an unsigned integer, containing an arbitrary number of digits, and an optional exponent of the form "en" or "En" (n is an unsigned integer < 32768).

file name - a string of up to 100 characters. A string containing special characters may be enclosed in single or double

quotes - the quotes will not be considered as part of the string. The following characters are considered special: control characters, blank, colon, semicolon, exclamation mark, at sign.

literals - strings of characters enclosed in quotation marks indicate specific values that must be used for an operand (the quotation marks are not actually entered). A list of literals is separated by the alternation symbol. Literals may be abbreviated in the same manner as command names.

FUNCTION : This section describes what the command is used for.

BREAK : Indicates at what point the interrupt signal (break or delete key) affects the command.

ERRORS : Specific error messages are given and explained, or an error class may be given. The error message classes are:

I/O errors - error status returned by an open, close, read, write, or seek operation.

process errors - an "exec" system call failed.

no memory errors - the required amount of main memory could not be allocated.

Note that the above error types should rarely occur. If such an error does occur, the command is immediately aborted. If a list of operands was supplied to the command, then those operands following the operand where the error occurred will not be processed. If commands were queued with semicolons, then the queue is flushed.

8.3 COMMAND DESCRIPTIONS

PROMPT : "$ "

COMMAND : sieve (sieve)

OPERANDS : none

FUNCTION : This command is typed to the shell. It causes the sieve program to be loaded into memory and executed. There is a typical delay of 15 seconds, and a maximum delay of 5 minutes, from the time "sieve" is typed to the time that anything is displayed on the terminal (this is the time required for the background sieve process to tidy up and exit). If the log file is not empty, then its contents are displayed on the terminal. Finally, the new prompt (":") is issued, which invites the user to enter a command. The standard input to the sieve command may be taken from a script file by using the redirection facility of the shell (e.g. sieve <script). Reprompts and verifications, though, accept their information from the terminal only.

BREAK : No effect.

ERRORS :

I/O errors

"had to use kill signal to terminate sivmon"

After 5 minutes from the time that the terminate signal was sent to the sivmon process, sivmon had not exited. This situation may arise if the filter program used by sivmon is processing a solution during the whole 5 minutes. If this message was issued, sivmon was not able to perform an orderly cleanup.

"sieve command already in use by userid"

The sieve command was not successful, because it is already executing. This message will be issued if the last sieve session ended abnormally. In this case, the "sieve.lock" file must be manually deleted.

PROMPT : ":"

COMMAND : edit (e)

OPERANDS : [problem file] (file name)

FUNCTION : The UNIX editor, ed, is invoked. The prompt issued during the edit session is "E:". Problem files are created and changed in this mode. For information on how to use ed, consult the UNIX user manual.

BREAK : Same as in a normal ed session.

ERRORS :

process errors

PROMPT : ":"

COMMAND : list (l)

OPERANDS : none

FUNCTION : Displays the contents of the log file at the terminal.

BREAK : Takes effect between lines of the log file.

ERRORS :

I/O errors

PROMPT : ":"

COMMAND : manage (m)

OPERANDS : none

FUNCTION : This command causes the sieve program to enter manage mode, which contains the majority of commands. The new prompt is "M:". In manage mode, the user creates instances of problems, schedules the problems, inspects results, and performs ancilliary housekeeping operations. The queue of problems is accessible only in this mode.

BREAK : No effect.

ERRORS :

I/O errors

PROMPT : ":"

COMMAND : prepare (pre)

OPERANDS : none

FUNCTION : Causes the sieve program to enter prepare mode. The prompt becomes "P:". In this mode, a problem file is inspected for errors, and translated from ASCII to binary. This translation must be performed before a problem file can be used by the background sieve process.

BREAK : No effect.

ERRORS : none

PROMPT : ":"

COMMAND : print (pri)

OPERANDS : none

FUNCTION : Produces a listing of the contents of the log file on the line printer.

BREAK : No effect.

ERRORS :

I/O errors

process errors

"log file is empty"

No messages are contained in the log file. A listing is not produced.

PROMPT : ":"

COMMAND : quit (q)

OPERANDS : none

FUNCTION : Causes the sieve program to terminate, and the user is returned to the shell. If the log file contains messages, then the user is asked if he wants those messages to be deleted. The valid responses are "yes" or "no".

BREAK : No effect.

ERRORS :

process errors

PROMPT : "M:"

COMMAND : check$ (check$)

OPERANDS : [time amount time units] (number "days" | "hours" | "minutes" | "seconds")

FUNCTION : This command is used to interrogate the checkpoint time interval (no operand supplied on command line) or to alter the checkpoint time interval (if operands are supplied on the command line). During a checkpoint, the background sieve process records on disk the state of the sieve device, and verifies the contents of the sieve hardware. The allowed checkpoint range is 1 to 65535 seconds. A suggested value is 30 minutes.

BREAK : No effect.

ERRORS :

"time must be > 0"

"amount of time is too big"

When resolved to seconds, time amount must be < 65536 seconds.

PROMPT : "M:"

COMMAND : clear (cl)

OPERANDS : {problem #} (number)

FUNCTION : This command can only be applied to problems in the queue that have an "error" status. Their status is changed to "ready", thereby allowing the problems to be executed. It is assumed that the error cause has been fixed.

BREAK : No effect.

ERRORS :

problem # is too big

can't find problem #

The specified problem # is not in the queue (note that the done file is not searched by this command).

problem # doesn't have an error status

PROMPT : "M:"

COMMAND : dlist (dl)

OPERANDS : none

FUNCTION : Dlist provides a display of the contents of the done file, which is the graveyard for problems that have completed. At the end of every manage mode session, all problems that have a "done" status are automatically archived in the done file. The done file is in order by time of archival, not problem number. The data given for a problem is the same as that provided by a manage.qlist command.

BREAK : Between problems.

ERRORS :

I/O errors

no memory errors

PROMPT : "M:"

COMMAND : down$ (down$)

OPERANDS : none

FUNCTION : This command prevents the background process "sivmon" from automatically starting up when the user exits the sieve command, or when UNIX reboots. Down$ is used only in exceptional circumstances (such as when maintenance is to be performed on the sieve hardware) or at the discretion of the UNIX administrators. The up$ command undoes the effect of down$.

BREAK : No effect.

ERRORS :

"already down"

A down$ command is already in effect.

PROMPT : "M:"

COMMAND : dprint (dp)

OPERANDS : none

FUNCTION : Same as dlist, except that the contents of the done file are printed on the line printer.

BREAK : No effect.

ERRORS :

I/O errors

process errors

no memory errors

PROMPT : "M:"

COMMAND : kill (kill)

OPERANDS : {problem #} (number)

FUNCTION : Kill removes problems with the specified problem #'s from the queue of problems. All solutions collected for a problem are deleted. The effect of kill cannot be undone; hence, the user is prompted for each problem # as follows:

kill problem # OK? Enter reply:

The user must answer "yes" or "no" (a quit control value is interpreted as a "no"). Caution should be exercised with this command.

BREAK : After processing of a problem #

ERRORS :

"problem # is too big"

"can't find problem #"

The problem is not in the queue (note that the done file is not searched).

PROMPT : "M:"

COMMAND : list (1)

OPERANDS : {problem #} (number)

FUNCTION : The list command displays information about problem #'s in the queue. The following data are given for a problem:

problem number

priority

status (new, ready, done, error)

recording mode (record, norecord)

problem file name

number of moduli used in the problem file

filter program name

starting value

stopping value

maximum number of solutions permitted

maximum time permitted

If the problem has a status other than "new", the following additional information is given:

solution count

clock count (how far the search for solutions has progressed)

solutions

BREAK : Between solutions or problems.

ERRORS :

I/O errors

no memory errors

"problem # is too big"

"can't find problem #"

The specified problem # is not in the queue (the done file is not searched).

PROMPT : "M:"

COMMAND : make (ma)

OPERANDS : recording mode ("record" | "norecord")
problem file (file name)
# of moduli (number)
filter program (file name)
start value (number)
stop value (number)
# of solutions (number)
time amount (number)
time units ("days" | "hours" | "minutes" | "seconds")

FUNCTION : The make command is used to create a new problem, and enter it into the queue. The operands supplied define the problem:

recording mode - "record" means record all solutions found. "Norecord" means just keep the largest solution found, and a count of all solutions.

problem file - the name of the problem file that specifies the moduli and residues to be used.

# of moduli - allows the use of a subset of the moduli in the problem file. Note that the order of use of moduli is hardware-implemented moduli first, then software-implemented moduli. This order may not be the same as the order of moduli in the problem file. However, the order within a modulus type (hardware or software) is the same as in the problem file. If you do not know how many moduli are in the problem file, but you want to use all of them, enter the value 0.

filter program - the name of the program that will perform additional tests on the potential solutions generated by the sieve. If this facility is not required, enter "-".

start value - the number from which the sieve is to begin the search for solutions. This number will be rounded down to a multiple of 8 (since the sieve hardware checks for solutions 8 numbers at a time).

stop value - the upper limit to be imposed on the solution search. This number will bw rounded up to a multiple of 8.

# of solutions - the maximum number of solutions allowed. Solutions must be accepted by the filter program to be included in the solution count.

time amount, time units - the aggregate real time that the problem will be permitted to run. The allowable range is 1 to 2147483647 seconds.

A problem is terminated when any of the stop value, maximum # of solutions, or time limit is exceeded, or if any unrecoverable error occurs. These termination criteria may be slightly exceeded, because of the pipelining of sieve hardware and software. Problems are given a default priority of 10. If all goes well, the message

problem # n entered successfully

is issued, where n is the number assigned to the new problem. Problem numbers are assigned sequentially from 1 to 2147483647, and are not reused.

BREAK : No effect.

ERRORS :

I/O errors

no memory errors

"problem file has not been prepared"

A binary version of problem file doesn't exist.

"# of moduli to use (0 for all) is too big"

More moduli were requested then are contained in the problem file.

"can't find filter program"

"filter program is the wrong type of file"

The file specified is not an executable program.

"no execute permission for filter program"

The program does not have the appropriate mode for invocation by the background sieve process.

"starting value is too big"

The number is too large to be stored (the exponent is > 32767).

"stop value is too big"

"stop value must be >= starting value"

The sieve cannot run backwards.

"max # of solutions is too big"

"max # of solutions must be > 0"

"amount of time is too big"

"time must be > 0"

PROMPT : "M:"

COMMAND : modify (mo)

OPERANDS : problem # (number)

recording mode ("record" | "norecord")

problem file (file name)

# of moduli (number)

filter program (file name)

start value (number)

stop value (number)

# of solutions (number)

time amount (number)

time units ("days" | "hours" | "minutes" | "seconds")

FUNCTION : This command allows the definition of a problem to be changed. The operands following problem # are the same as in the manage.make command. The previous values for these operands are first displayed. Next, a prompt is made for the operand values that were not on the command line. The null control value ("@" or "@>") should be specified for operands that are not to change from their previous value. A common use of this

command is to extend the problem termination criteria. Some operand changes are not permitted for problems that have already run on the sieve (e.g. changing the recording mode or start value). All of the error checks performed by the manage.make command are also performed by this command. In addition, the problem termination operands may not be decreased below the values of the variables that the termination operands monitor. Application of the modify command to a problem with an "error" or "done" status changes the problem status to "ready".

BREAK : No effect.

ERRORS : As for manage.make command, plus:

"can't change recording mode after problem has run"

"can't change starting value after problem has run"

"can't back up stop value"

"can't back up max # of solutions"

"can't back up amount of time"

PROMPT : "M:"

COMMAND : print (prin)

OPERANDS : {problem #} (number)

FUNCTION : Same as for the manage.list command, except that the problem data are sent to the line printer. Each problem is started on a new page. The order of problems appearing on the line printer may not be the same as the order of operands in the print command.

BREAK : No effect. Consider issuing the "cprt" command to the shell.

ERRORS : As for manage.list, plus:

process errors

PROMPT : "M:"

COMMAND : priority (prio)

OPERANDS : priority value (number)
{problem #} (number)

FUNCTION : The priority command affects the scheduling of the specified problem #'s within the queue. New and ready problems are run in descending order by priority value. The highest priority is 32767, and the lowest is 1. A priority of 0 prevents a problem from running. Problems with equal priority are run in ascending order by problem #.

BREAK : No effect.

ERRORS :

"priority # is too big"

"problem # is too big"

"can't find problem #"

The specified problem is not in the queue (note that the done file is not searched by this command).

PROMPT : "M:"

COMMAND : qlist (ql)

OPERANDS : none

FUNCTION : This command lists at the terminal the contents of the queue file. The problems are listed in scheduling order, except that some of the problems may have a done or error status. The data provided for each problem are:

- problem #
- priority
- status (new, ready, error, done)
- recording mode (record, norecord)
- problem file name
- # of moduli used in the problem file
- filter program name
- starting value
- stopping value
- maximum # of solutions permitted
- maximum time permitted

BREAK : Between problems.

ERRORS : none

PROMPT : "M:"

COMMAND : qprint (qp)

OPERANDS : none

FUNCTION : Qprint is equivalent to manage.qlist, except that the output is routed to the printer.

BREAK : No effect.

ERRORS :

I/O errors

process errors

PROMPT : "M:"

COMMAND : quit (q)

OPERANDS : none

FUNCTION : This command terminates manage mode. The new prompt will be ":". As a side effect, the queue of problems is "thinned" (i.e. problems with a status of "done" are moved to the done file).

BREAK : No effect.

ERRORS :

I/O errors

PROMPT : "M:"

COMMAND : retrieve (r)

OPERANDS : {problem #} (number)

FUNCTION : The retrieve command moves the specified problem #'s from the done file back into the problem queue. The status of these problems remains as "done"; hence, these problems will return to the done file when manage mode is quit, unless the problems are modified or killed. This command can take a long time to execute, because of the amount of I/O performed.

BREAK : No effect.

ERRORS :

I/O errors

no memory errors

"problem # is too big"

"problem # is already in queue"

"problem # not found in done file"

PROMPT : "M:"

COMMAND : up$ (up$)

OPERANDS : none

FUNCTION : "Up$" permits the background sieve process to begin execution when the user exits the sieve command. It negates the effect of a manage.down$ command.

BREAK : no effect.

ERRORS :

"already up"

An up$ command is already in effect.

PROMPT : "M:"

COMMAND : write (w)

OPERANDS : text file (file name)
{problem #} (number)

FUNCTION : This command is functionally equivalent to the manage.list command, except that the data on each problem # is appended to the UNIX text file. The file is created if it does not already exist. This command finds several uses: The text file created may be edited to remove unnecessary information (such as all lines containing a colon, and the "START PROBLEM" and "END PROBLEM" delimiters) and leave just the solution values. The text file may be subsequently printed. The manage.print command begins each problem on a new page; hence a write to a text file followed by a shell "upr text file" saves paper. Finally, the text file may be copied to tape or diskette, for transportation to other computer installations.

BREAK : Between problem #'s.

ERRORS :

I/O errors

no memory errors

"problem # is too big"

"can't find problem #"

The specified problem is not in the queue (the done file is not searched by this command).

PROMPT : "P:"

COMMAND : date (da)

OPERANDS : {problem file} (file name)

FUNCTION : Each operand is assumed to be the name of an ASCII problem file. Its last modification time (LMT) is compared with the LMT of its corresponding binary file (i.e. the one created by a prepare.test or a prepare.ptest command). If the LMT of the ASCII file is greater than the LMT of the binary file, or if no binary file exists, then the message

problem file is not current

is issued. This means that a prepare.test or prepare.ptest command should be performed on problem file in order for the binary version to reflect the changes made to the ASCII version. If this is not necessary, then the message issued is

problem file is current

BREAK : Between problem file's.

ERRORS :

"problem file doesn't exist"

A "stat" system call on the designated file was not successful.

PROMPT : "P:"

COMMAND : destroy (destroy)

OPERANDS : {problem file} (file name)

FUNCTION : This command removes both the ASCII and binary of each problem file specified. Each file name is specified as follows:

destroy file name OK? Enter reply:

The user must answer "yes" or "no" at this point (a quit control value is interpreted as a "no"). The ASCII problem file is removed by the "rm" UNIX command.

BREAK : After processing of a problem file.

ERRORS :

process errors

"problem file doesn't exist"

A "stat" system call did not succeed for the specified file.

PROMPT : "P:"

COMMAND : ptest (pt)

OPERANDS : {problem file} (file name)

FUNCTION : Same as for prepare.test, except that all diagnostics are sent to the printer instead of the terminal. This command would normally be used only when there are too many error messages to be easily viewed at the terminal. The name of each problem file is displayed at the terminal, just before it is processed.

BREAK : Between statements in a problem file, and between problem files.

ERRORS : As for prepare.test, plus:

process errors

PROMPT : "P:"

COMMAND : quit (q)

OPERANDS : none

FUNCTION : Causes an exit from prepare mode. The user is returned to the basic sieve command mode, with its prompt of ":".

BREAK : No effect.

ERRORS : none

PROMPT : "P:"

COMMAND : test (t)

OPERANDS : {problem file} (file name)

FUNCTION : The test command causes each problem file specified to be read and translated into a more compact binary form. The translation occurs only if no errors were detected in the problem file. The binary version of the problem file is created in a special directory that the user need not know about. The original problem file is not modified. For each problem file, a count of the number of errors detected is displayed. If no errors occurred, then a count of the number of moduli encountered is also displayed.

BREAK : Between statements in a problem file, and between problem files.

ERRORS :

I/O errors

no memory errors

syntax and semantic errors found in the problem file

All of these error messages begin with the line number in which the error was detected.

"no hardware moduli collected"

None of the valid moduli in the problem file are implemented in the sieve's hardware. At least one such modulus is required for the background sieve process to function properly.

8.4 PROBLEM FILE CREATION

The problem file contains the congruences to be used in one or more problems to be solved by the sieve. It can be created by the user through the editor. A library of utilities has been created to produce the quadratic residues, quadratic non-residues, and cubic residues, in the form of problem files (see programs "qres" and "cres"). Such automatically-generated problem files may be edited by the user, if necessary.

A problem file has a simple format. It consists of statements, where a statement is a list of unsigned integers, separated by "white space" (tab, newline, or blank characters), and terminated by a semicolon. The first number in a statement is assumed to be the modulus, and the remaining numbers are the residues desired for the modulus. Any set of consecutive residues i through j inclusive may be abbreviated as i:j or j:i. The maximum value of a modulus is 32767; hence, residues may have values from 0 to 32766. In Backus-Naur Form, the problem file syntax is:

```
problem_file   ::=  {statement}
statement      ::=  modulus  residue_list  ;
residue_list   ::=  {residue_item}
residue_item   ::=  residue
                    residue  :  residue
modulus        ::=  number
residue        ::=  number
number         ::=  {digit}
```

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

As an example, the congruence

$X \equiv 1,3,4,5,6,9 \pmod{11}$

is represented by the problem file statement

11 1 3:6 9;

Problem files can be annotated. All characters between a "%" character and the end of the line will be considered as a comment. It is recommended that every problem file begin with a few comments that summarize the congruences.

The following format is suggested for problem files: Moduli should be in ascending order. Each modulus begins on a new line, and is followed by a tab character. The residue list follows in ascending order. The residues or residue ranges are separated by a single blank character. At most ten residues should appear on a line. Each additional line required for the residues should be indented by a tab character. These rules make the moduli stand out, and permit an easy count of the residues for a given modulus. The problem file generation programs adhere to these rules.

All moduli in a problem file must be relatively prime, to guard against inconsistencies. At least one modulus must be implemented in the hardware of the sieve, since the hardware initiates solutions. For a given modulus m, the number of residues allowed is between 1 and m-1 inclusive. It is senseless (and an error) to specify all m residues, since this is the same as omitting the congruence altogether.

Finally, a particular residue may appear in the residue list only once.

Appendix B contains an example of a problem file.

8.5 FILTER PROGRAM CREATION

A filter program may be specified for a problem. Its purpose is to perform additional screening on numbers that have been generated by the sieve. This allows arbitrary tests to be applied to potential solutions before they are recorded, thus conserving secondary storage. In addition, the application of additional tests to a potential solution is carried on concurrently with the sieve peripheral's search for the next potential solution. Thus, the sieve peripheral is not necessarily idle while the filter program is active.

All filter programs accept as their standard input a multiple-precision binary-format number. Under Version 7 of UNIX, the "mp" package's MINT format is used. The standard output of the filter program is an integer with the following interpretation:

< 0 : error. Sivmon will terminate the current problem with an error status.

= 0 : the number is not a valid solution.

> 0 : the number is a valid solution.

Appendix B illustrates the format of a filter program.

Chapter IX

CONCLUSIONS

Two obvious ways of making an electronic shift register sieve faster are increasing the clock speed, and increasing the number of solution detectors. In this sieve, signal delays are a significant part of the cycle time. The specific problem is that some of the connections from the rings to the solution detectors are longer than they should be, and signal reflections occur due to impedance mismatches. These reflections must be allowed to damp out, in order for correct solution detection.

It may seem that the number of windows per ring may be arbitrarily increased, but there are limiting factors. First, there is the space required by a solution detector (6 integrated circuits in this sieve). Second, solution detectors have a large fan-in, and are therefore very sensitive to input signal skewing. As the number of solution detectors increases, the clock speed will probably be degraded. Finally, each solution detector must have a connection to every ring; signal routing becomes a problem.

The author envisions the construction of a single, 40 ring, 1 billion trial per second sieve, having 16 windows per ring, and a cycle time of 15 nanoseconds. ECL is the

only semiconductor logic capable of this speed. ECL requires a controlled impedance environment, but this is desirable for a sieve. Unlike TTL, ECL does not produce large current spikes in the power plane when it changes logic state; this is clearly advantageous when a large number of devices are switching simultaneously. Construction of the gigahertz sieve would be greatly simplified if each ring could reside on a single chip. The availability of low-volume, custom ECL fabrication is eagerly awaited.

A sieve has little use outside the field of number theory. However, building a sieve in a new technology is an excellent way to gain familiarity with that technology. The sieve algorithm is simple, and can be implemented in a variety of ways. Electronics has not yet been fully exploited. Optics promises even higher speeds.

REFERENCES

1. Lehmer, D. H. The Mechanical Combination of Linear Forms, American Mathematical Monthly, v. 35, 1928, p. 114-121

2. Lehmer, D. H. A Photo-Electric Number Sieve, American Mathematical Monthly, v. 40, 1933, p. 401-406

3. Lehmer, D. H. A Machine for Combining Sets of Linear Congruences, Mathematische Annalen, v. 109, 1934, p. 661-667

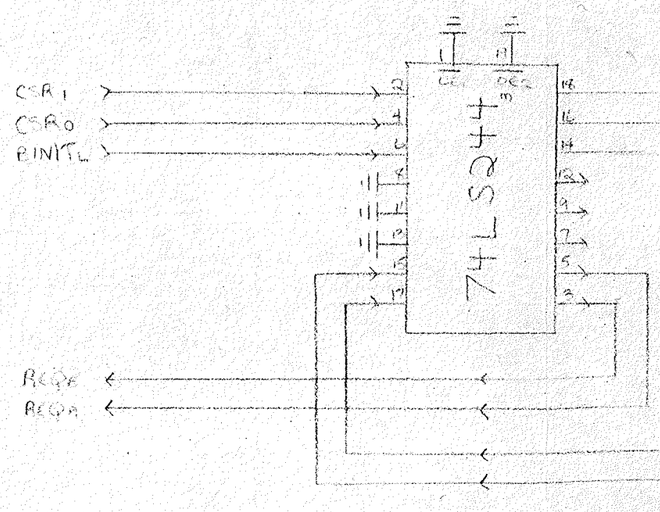
4. Lehmer, D. H. The Sieve Problem for All-Purpose Computers, Mathematical Tables and other Aids to Computation, v. 7, 1953, p. 6-14

5. Cantor, D. G., Estrin, G., Fraenkel, A. S., Turn, R. A Very High-Speed Digital Number Sieve, Mathematics of Computation, v. 16, 1962, p. 141-154

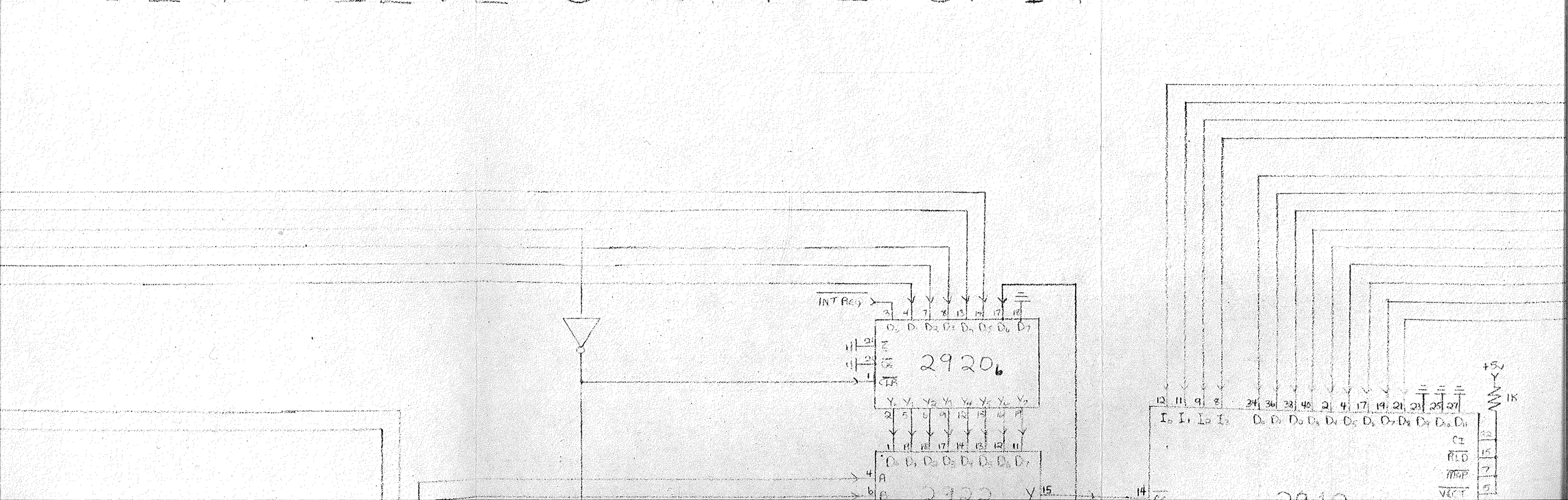
6. Lehmer, D. H. An Announcement Concerning the Delay Line Sieve DLS-127, Mathematics of Computation, v. 20, 1966, p. 645-646

7. Lehmer, D. H. A History of the Sieve Process, A History of Computing in the Twentieth Century, Academic Press, Inc., 1980, p. 445-456

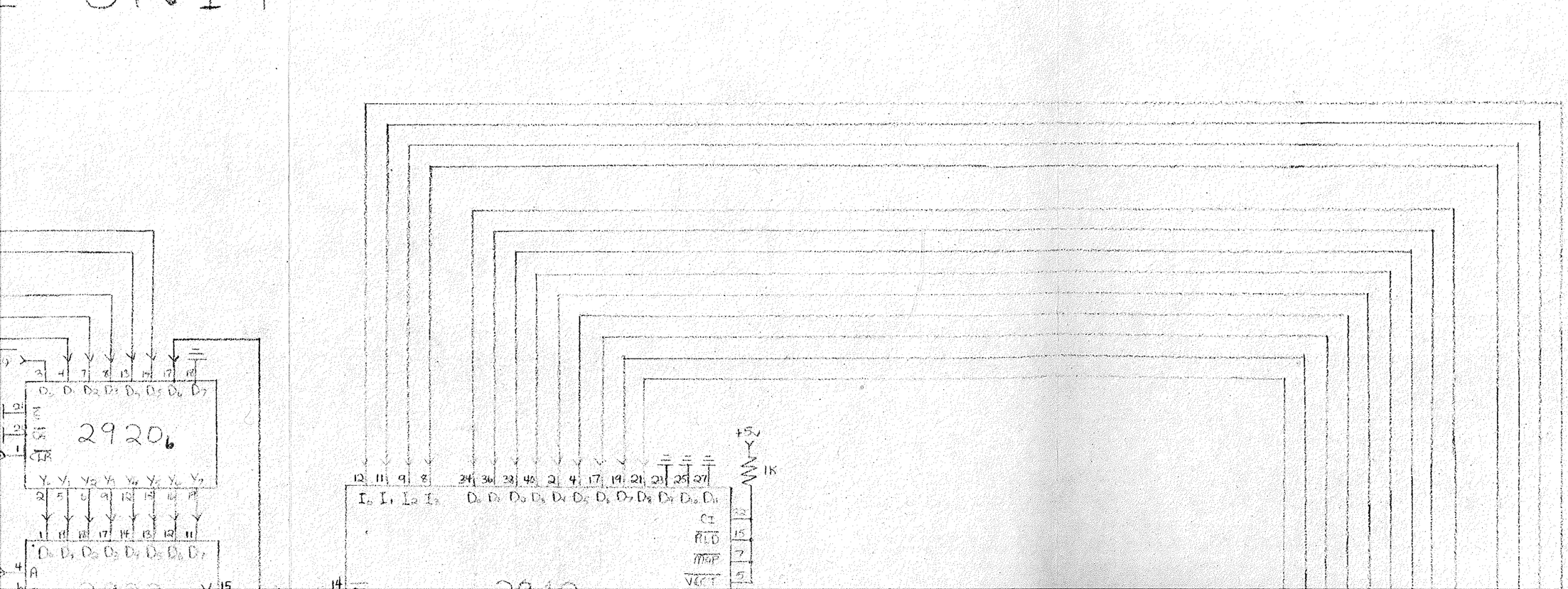
NUMBE

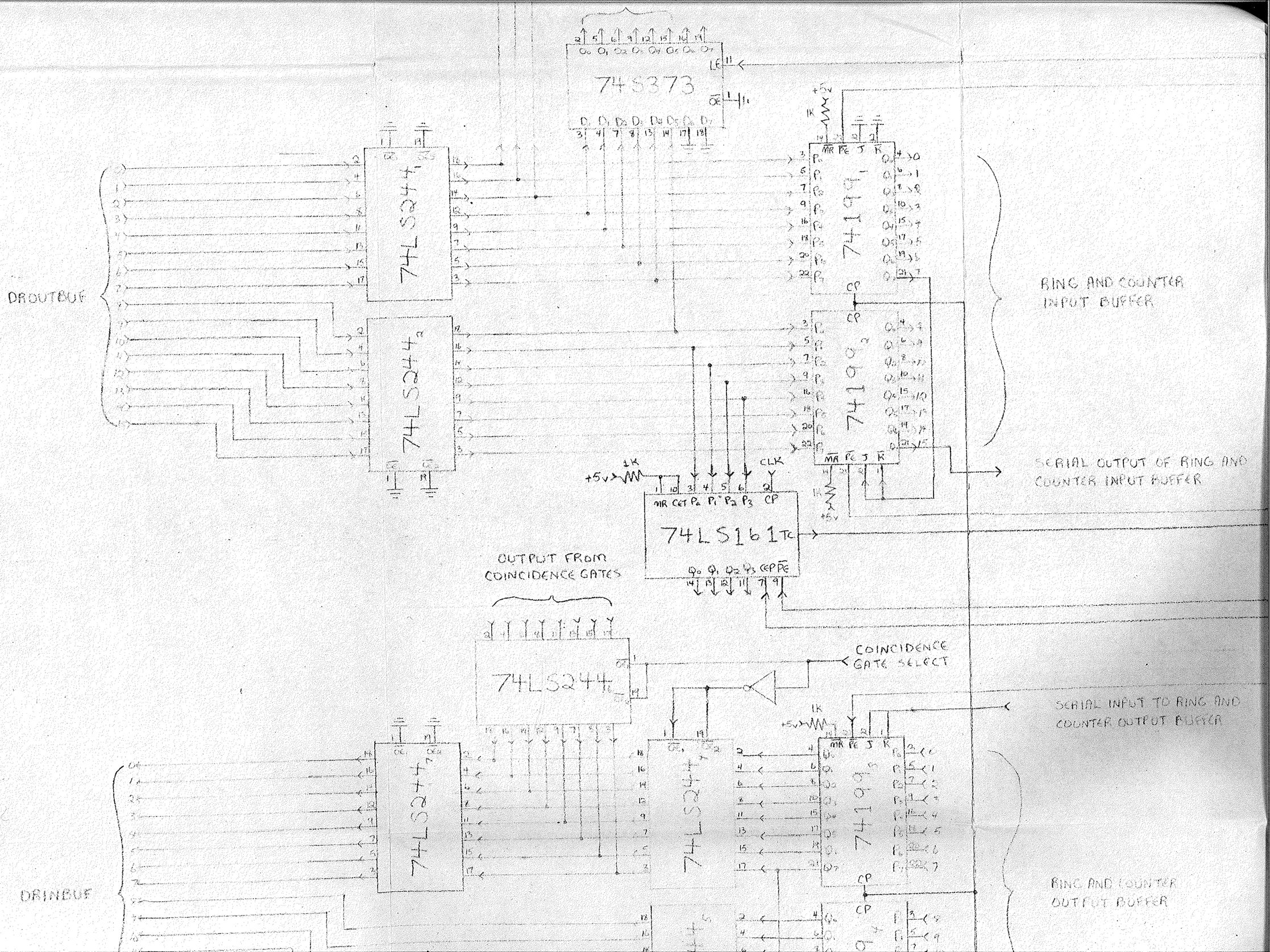


UMBER SIEVE CONTROL UNIT



L UNIT




74S373
74LS244
74LS244
74LS244
74LS244
74LS244
74190
74190
74LS161
74190
74190
DROUTBUF
DRINBUF
+5V
1K
CLK
MR CET P0 P1 P2 P3 CP
74LS161 TC
Q0 Q1 Q2 Q3 CEP PE
OUTPUT FROM COINCIDENCE GATES
COINCIDENCE GATE SELECT
RING AND COUNTER INPUT BUFFER
SERIAL OUTPUT OF RING AND COUNTER INPUT BUFFER
SERIAL INPUT TO RING AND COUNTER OUTPUT BUFFER
RING AND COUNTER OUTPUT BUFFER

RING AND COUNTER
INPUT BUFFER

SERIAL OUTPUT OF RING AND
COUNTER INPUT BUFFER

COINCIDENCE
GATE SELECT

SERIAL INPUT TO RING AND
COUNTER OUTPUT BUFFER

RING AND COUNTER
OUTPUT BUFFER

27529₁

27529₂

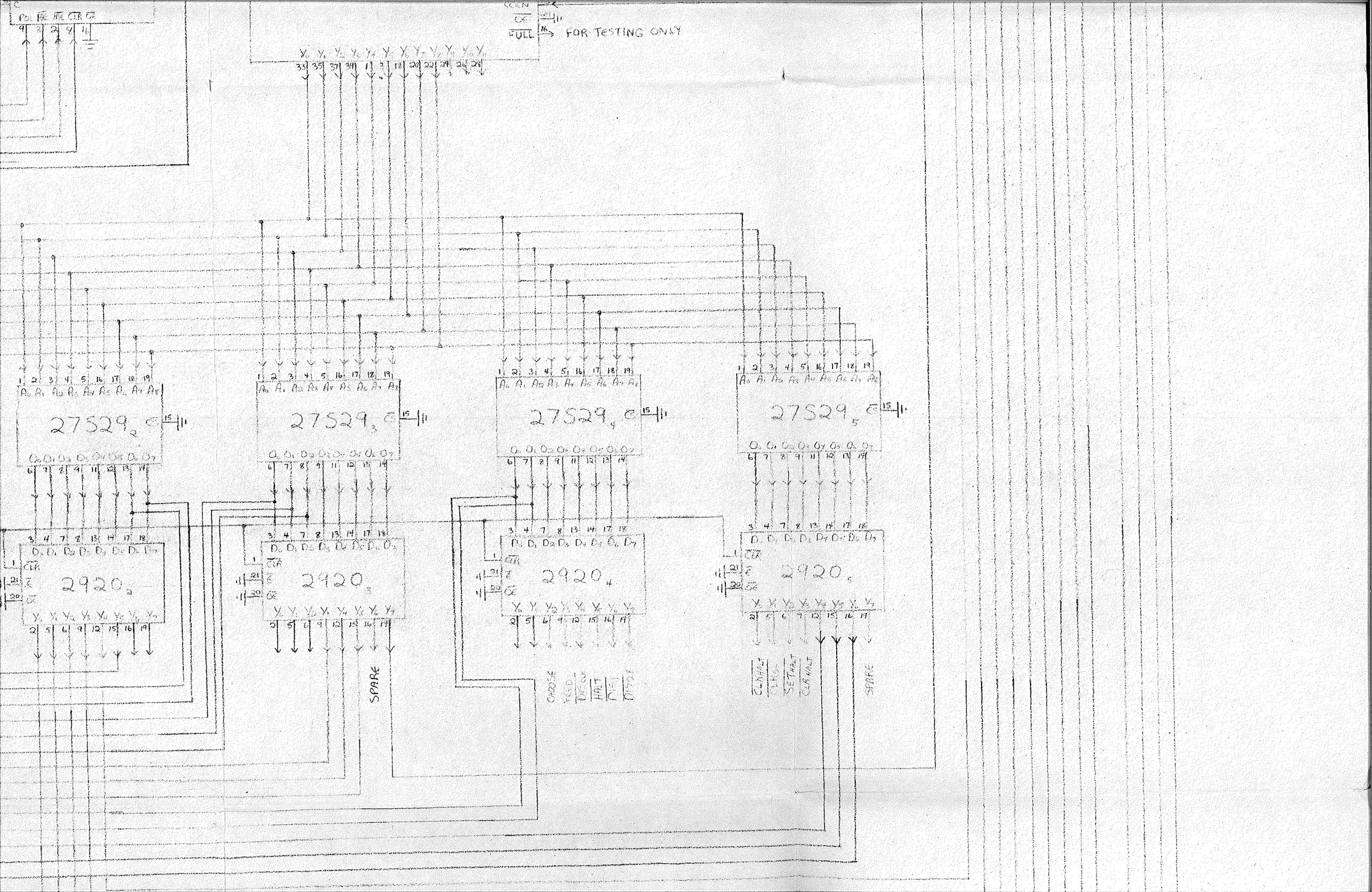
27529₃

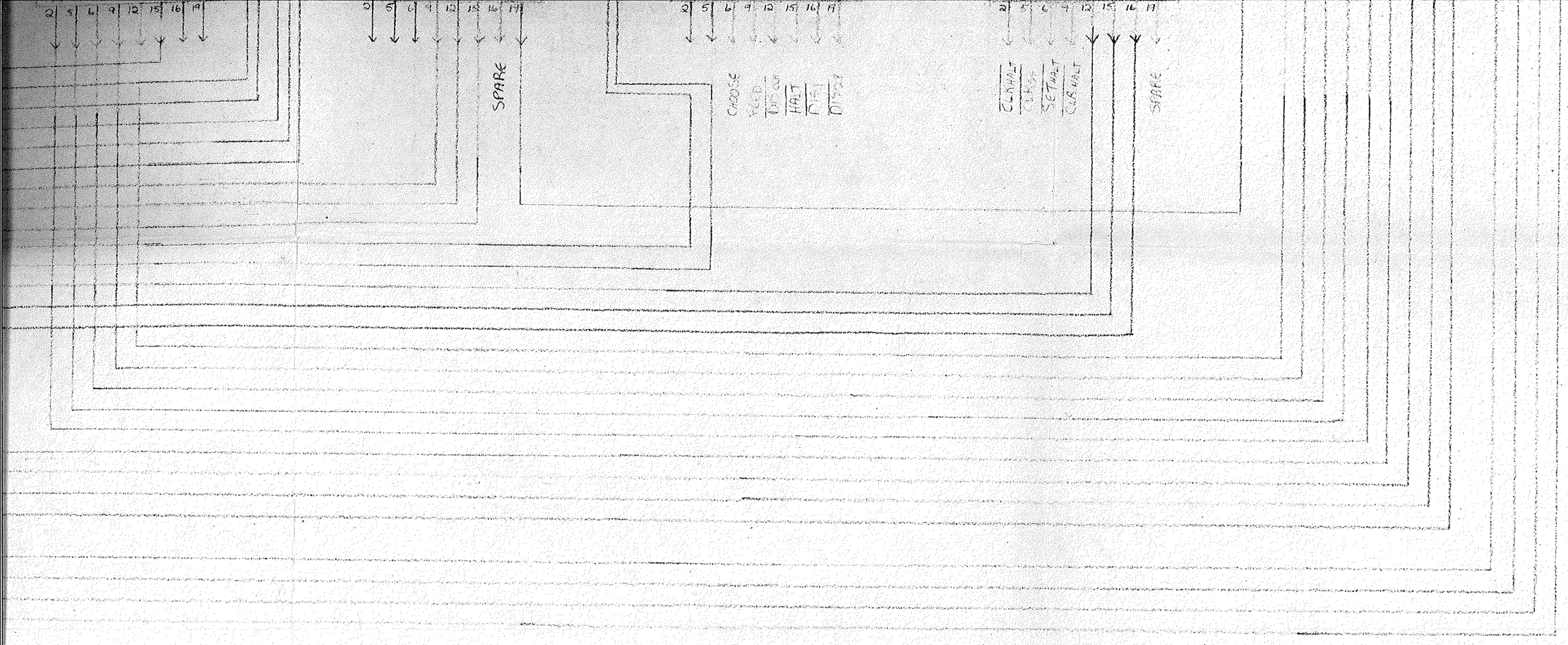
2920₁

2920₂

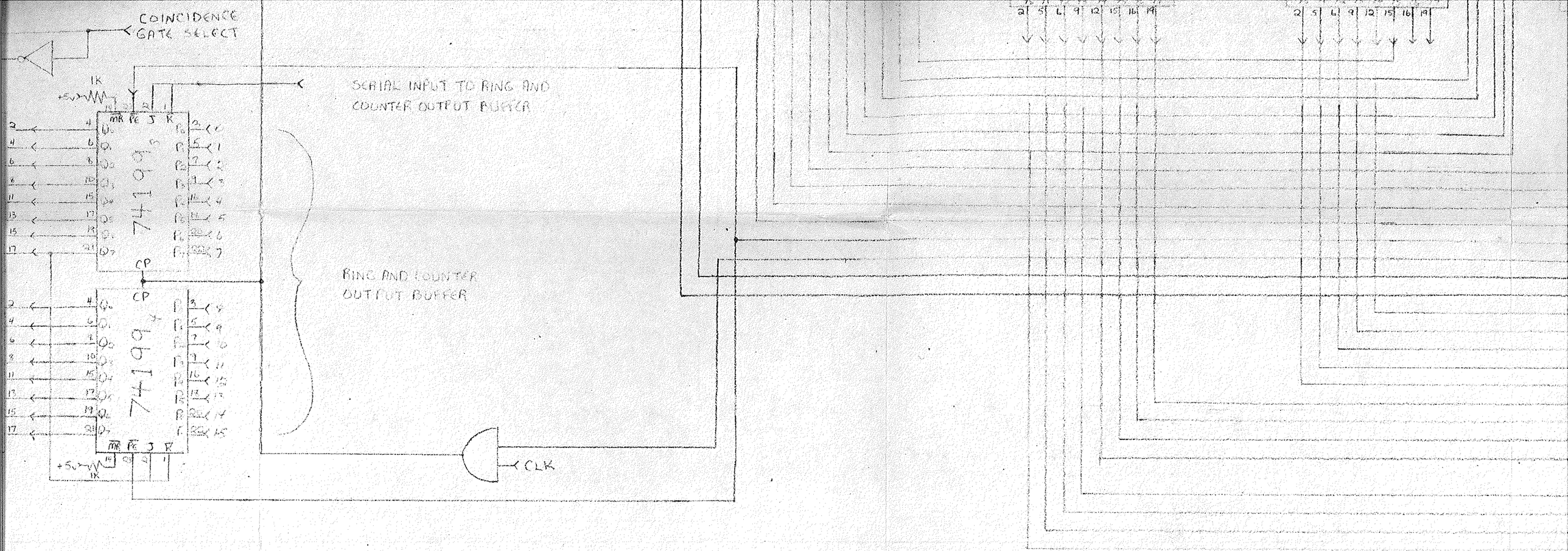
2920₃

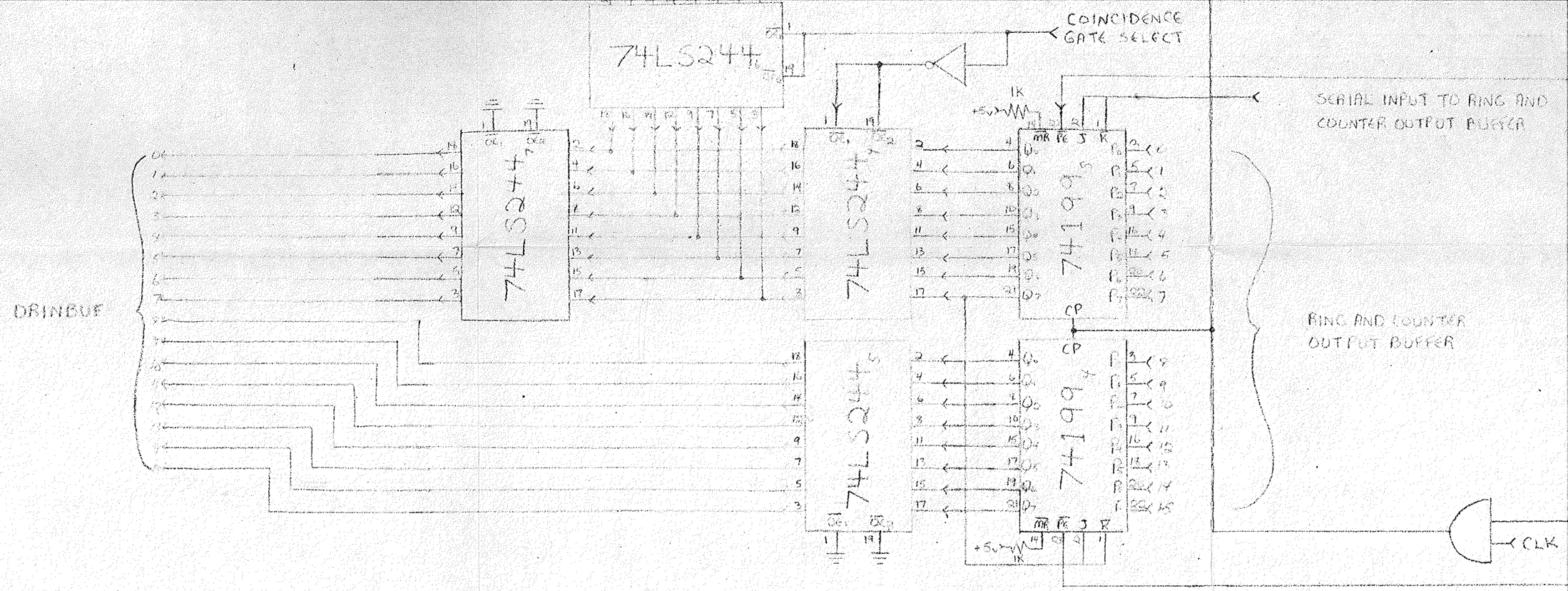
SPARE


$\overline{CEN}$
$\overline{OE}$
FULL
FOR TESTING ONLY
Y_0 Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7 Y_8 Y_9
33 35 37 39 1 3 18 20 22 24 26
A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8
1 2 3 4 5 16 17 18 19
O_0 O_1 O_2 O_3 O_4 O_5 O_6 O_7
6 7 8 9 11 12 13 14
D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7
$\overline{CLR}$
CP
$\overline{OE}$
21
20
15
27529_2
27529_3
27529_4
27529_5
2920_2
2920_3
2920_4
2920_5
SPARE
CHOOSE
HALT
SET
CLR
SPARE



C.D. PATTERSON
MAY 25/1981

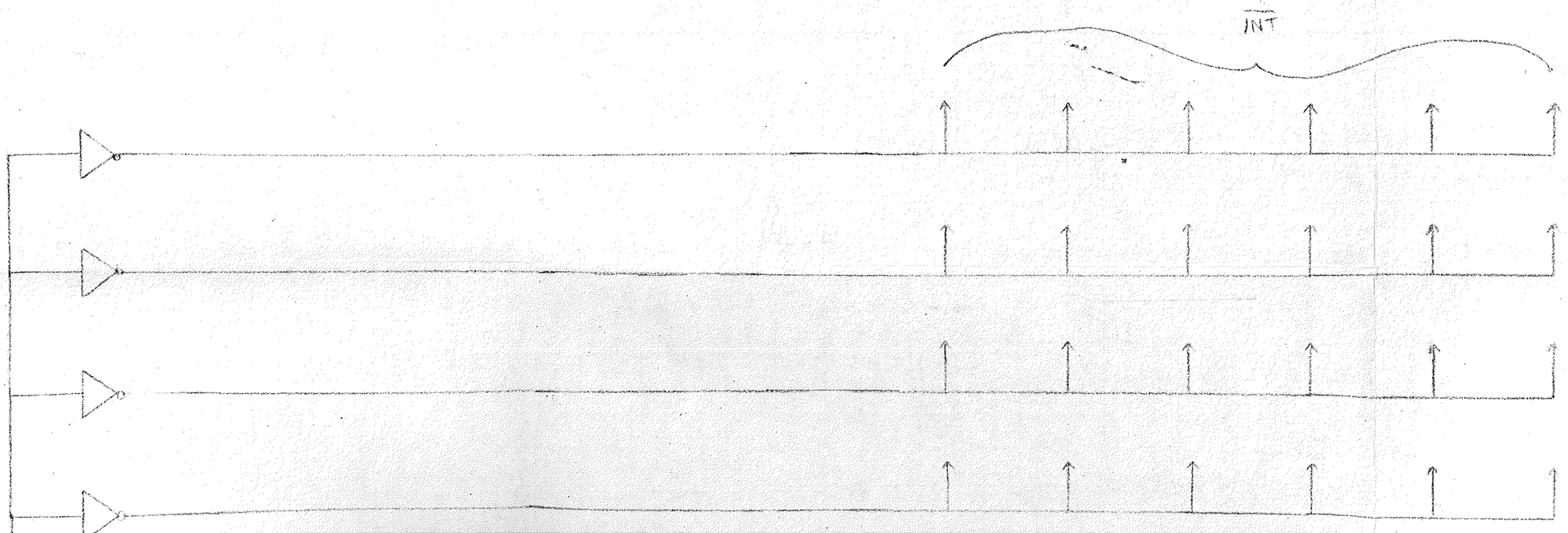

COINCIDENCE GATE SELECT
1K
+5V
SERIAL INPUT TO RING AND COUNTER OUTPUT BUFFER
74100
CP
RING AND COUNTER OUTPUT BUFFER
74199
CLK


74LS244
COINCIDENCE GATE SELECT
1K
+5V
SERIAL INPUT TO RING AND COUNTER OUTPUT BUFFER
DRINBUF
74LS244
74LS244
74LS244
74LS244
74LS194
74LS194
CP
CP
RING AND COUNTER OUTPUT BUFFER
CLK

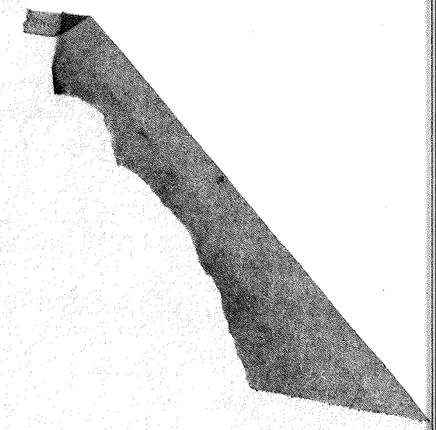
NUMBER SI

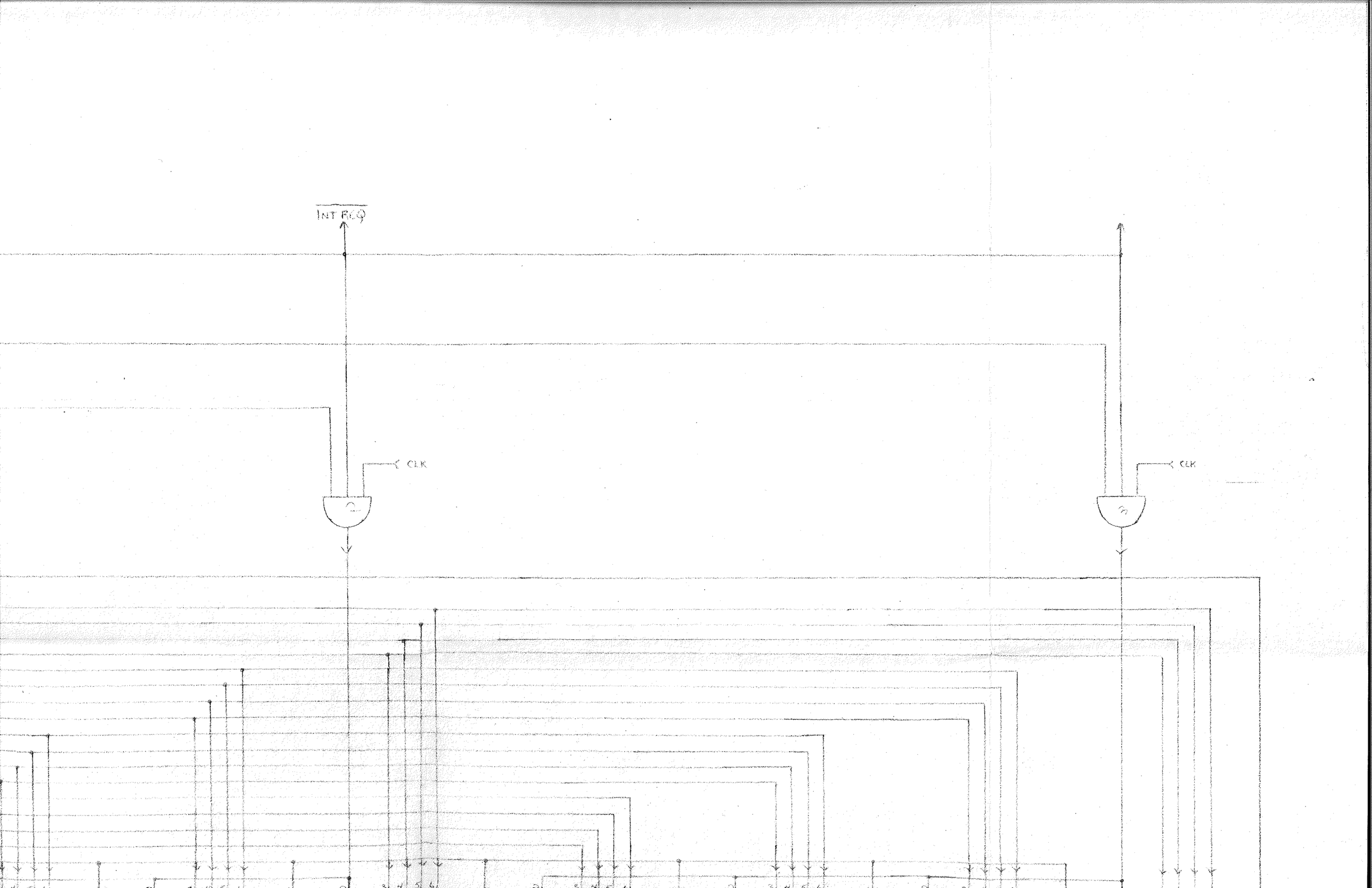
+5v
1K
U1(1)
U1(2)
U1(3)
DIS1
A
U1(4)
+
SD1
SD2
D1
CP
74
U1(5)
Q1
D2
U2(1)
U2(2)
U2(3)
U2(4)
U3(1)
U3(2)
U3(3)
U3(4)
U4(1)

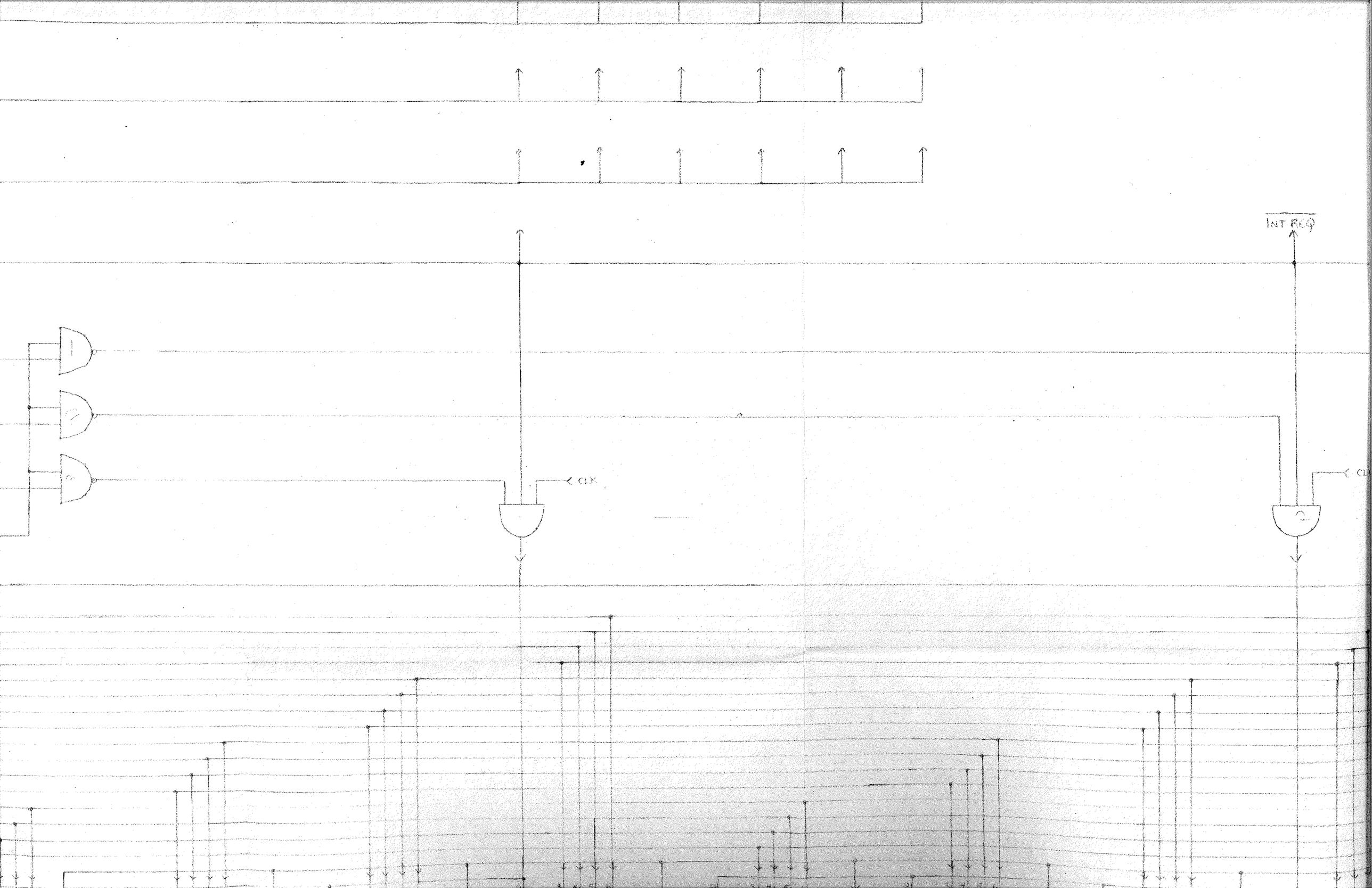
BER SIEVE COUNTER AND COINCIDENCE C

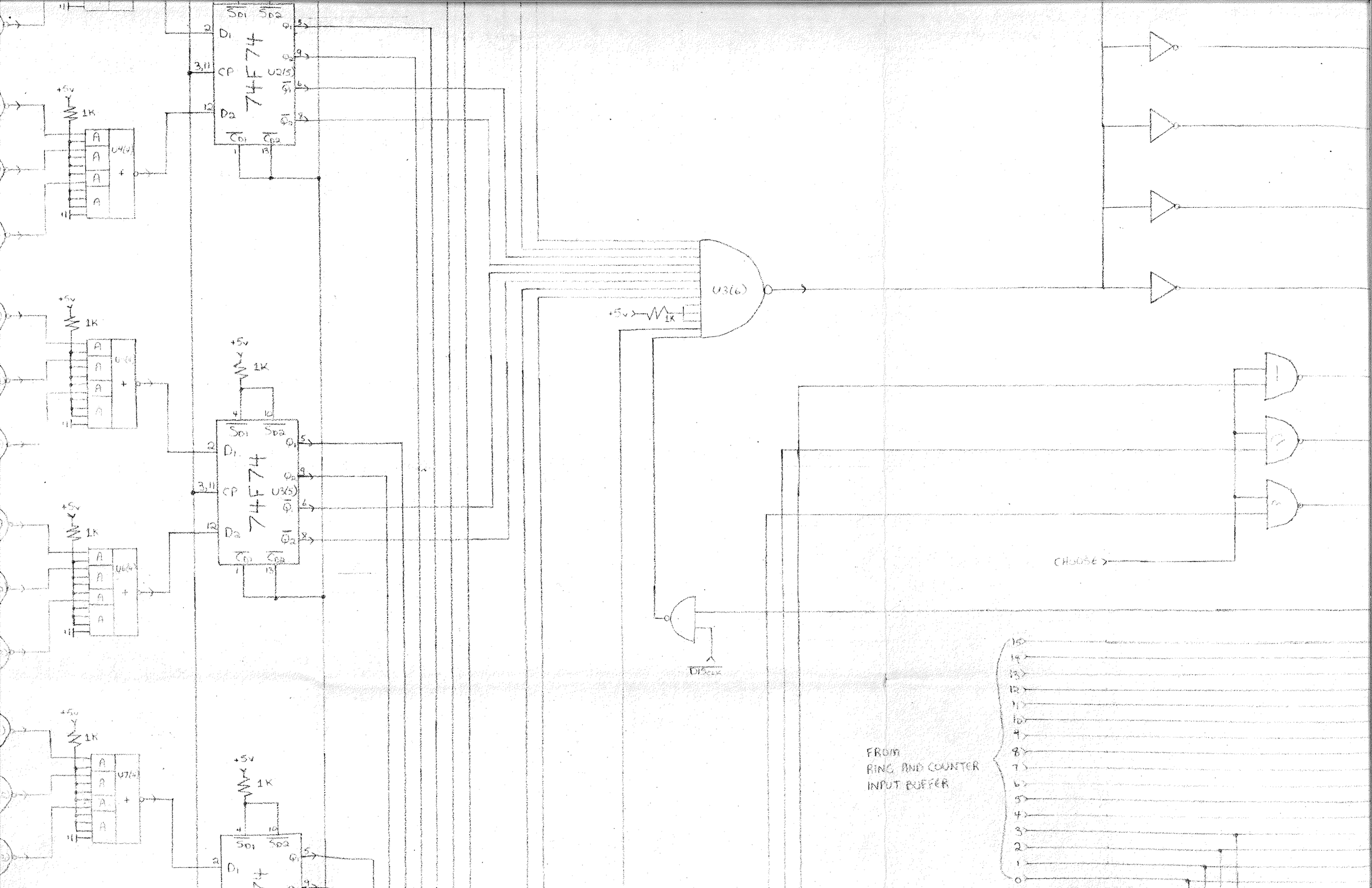


CE GATES



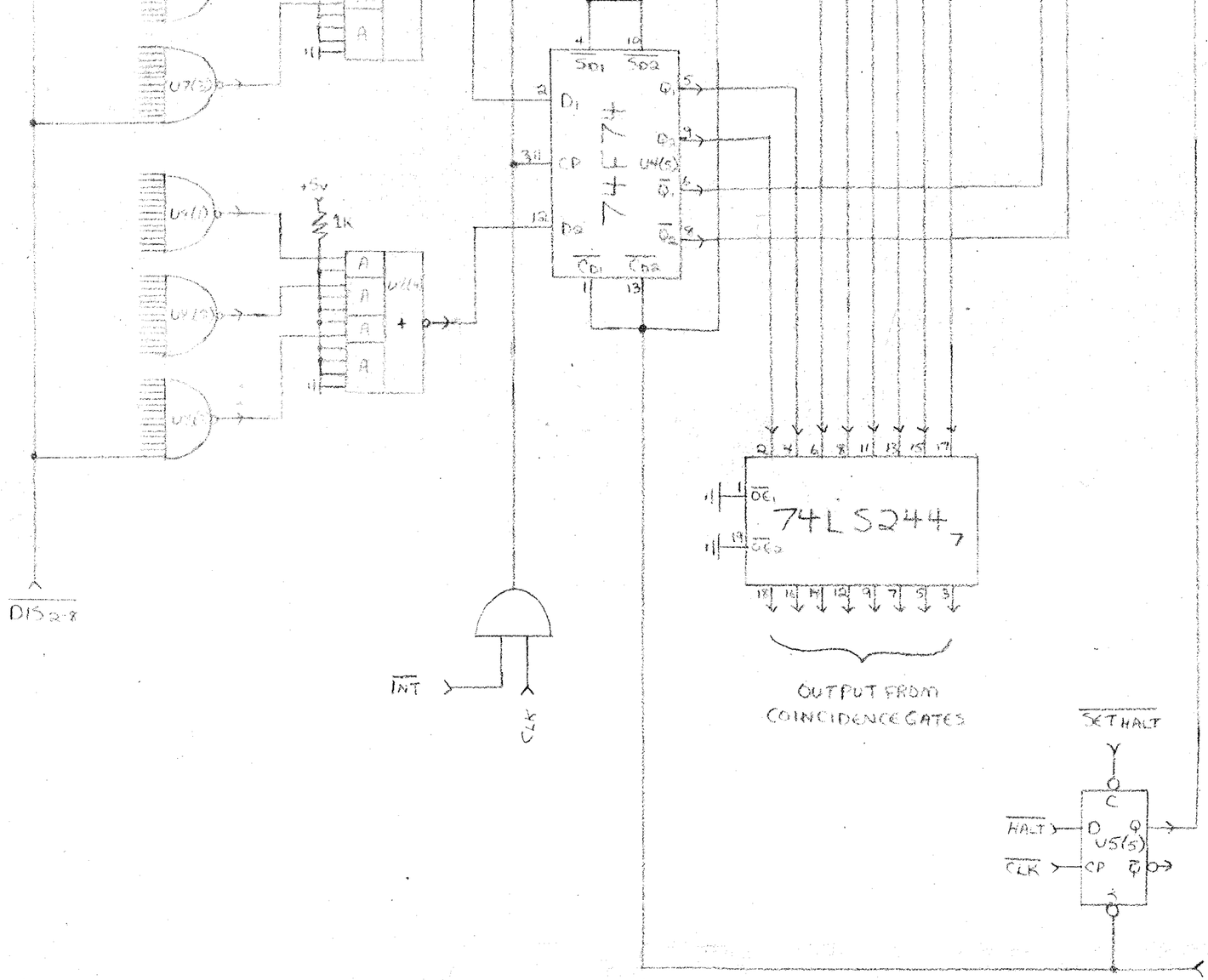

INT REQ
CLK
CLK

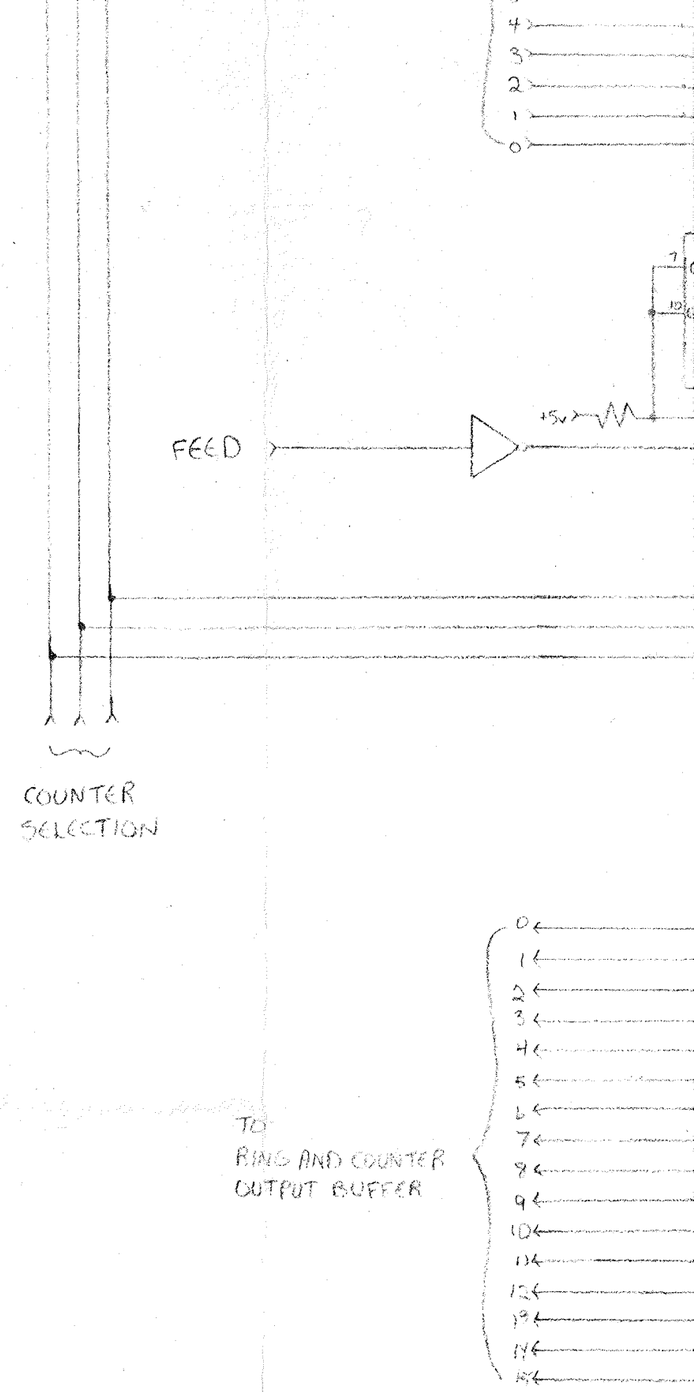

INT REQ
CLK

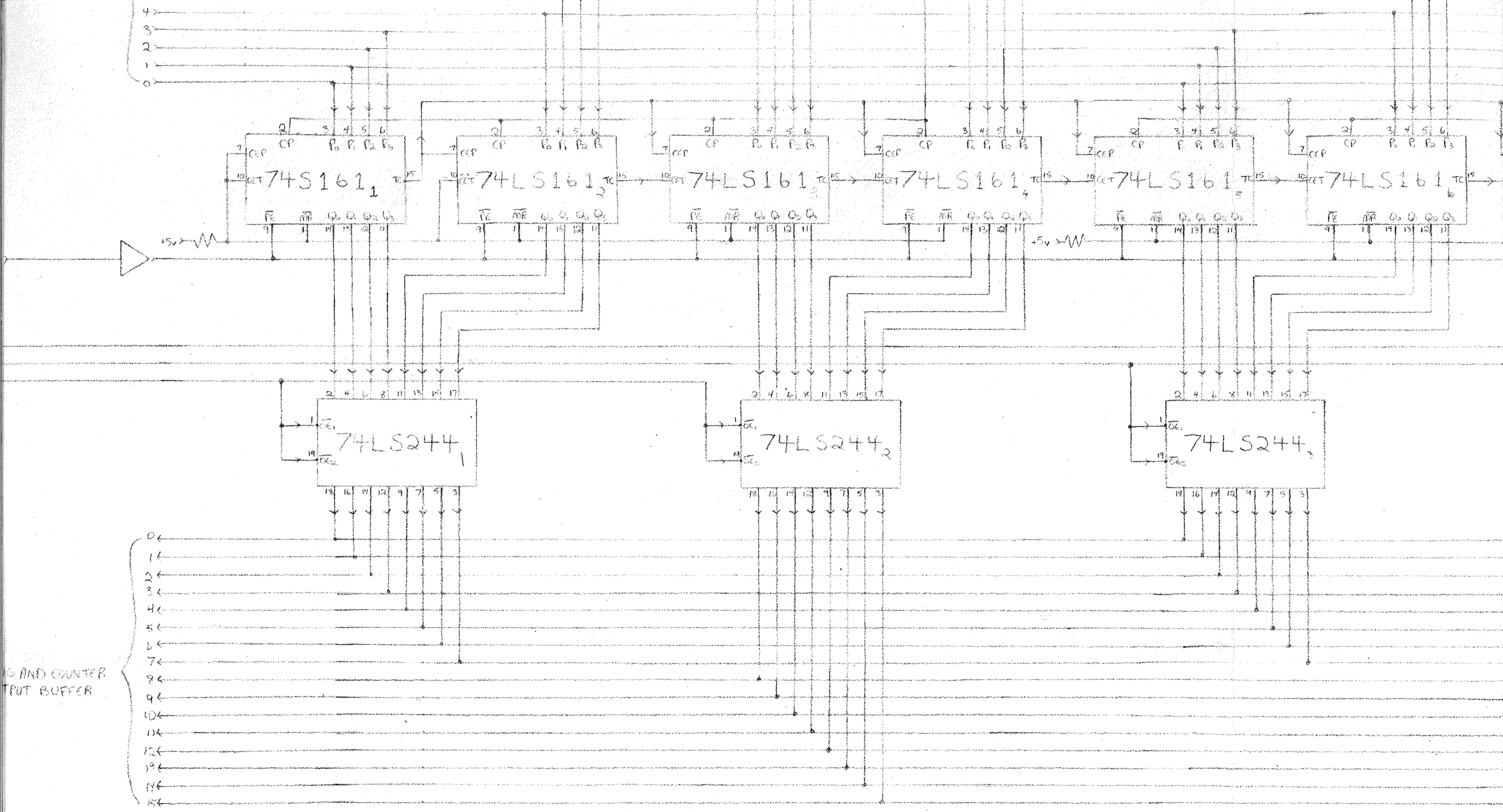

74LS74
U2(5)
U3(5)
U3(6)
1K
+5V
A
U4(4)
U5(4)
U6(4)
U7(4)
CHOOSE
DISCLK
FROM
RING AND COUNTER
INPUT BUFFER

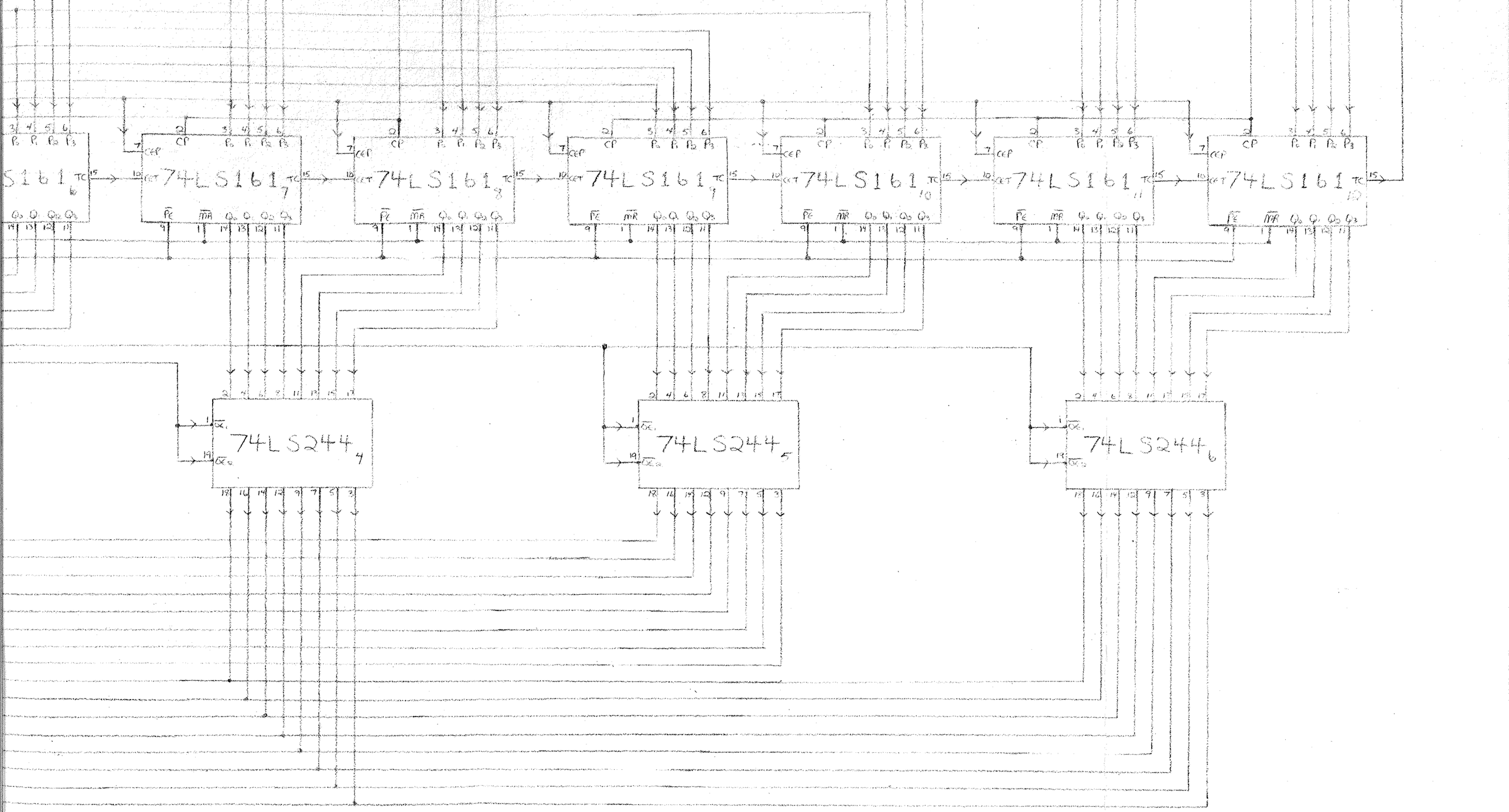
FROM
RING IND COUNTER
INPUT BUFFER

CH.056


+5v
1k
74LS244
OUTPUT FROM
COINCIDENCE GATES
INT
CLK
SET HALT
HALT
CLR HALT
C.D. PATTERSON
JUNE 16/1981


FEED
+5v
COUNTER
SELECTION
TO
RING AND COUNTER
OUTPUT BUFFER


74S161 1
74LS161 2
74LS161 3
74LS161 4
74LS161 5
74LS161 6
74LS244 1
74LS244 2
74LS244 3
+5v
+5v
AND COUNTER
TPUT BUFFER

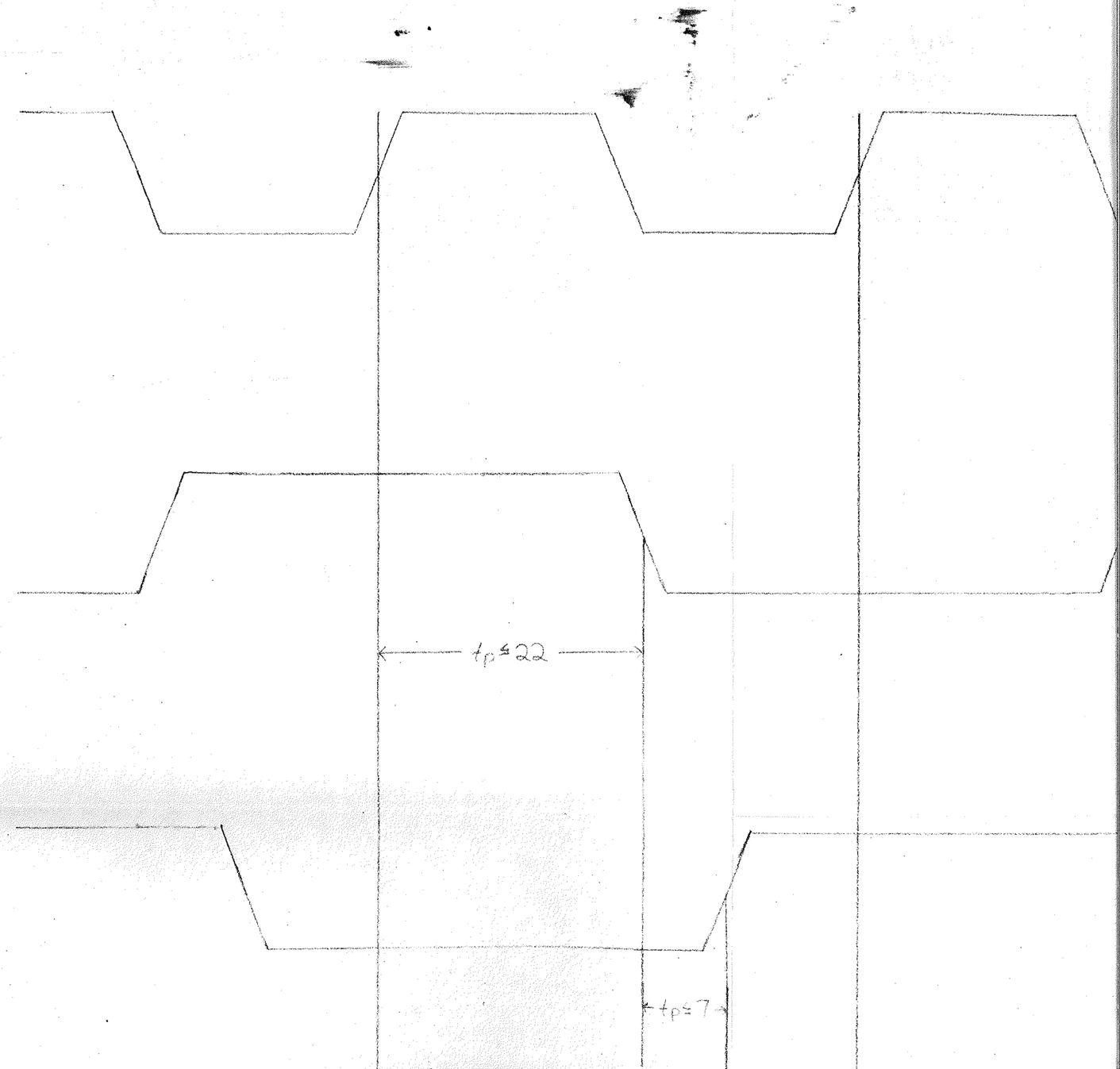

74LS161 6
74LS161 7
74LS161 8
74LS161 9
74LS161 10
74LS161 11
74LS161 12
74LS244 4
74LS244 5
74LS244 6

NUMBER SIEVE

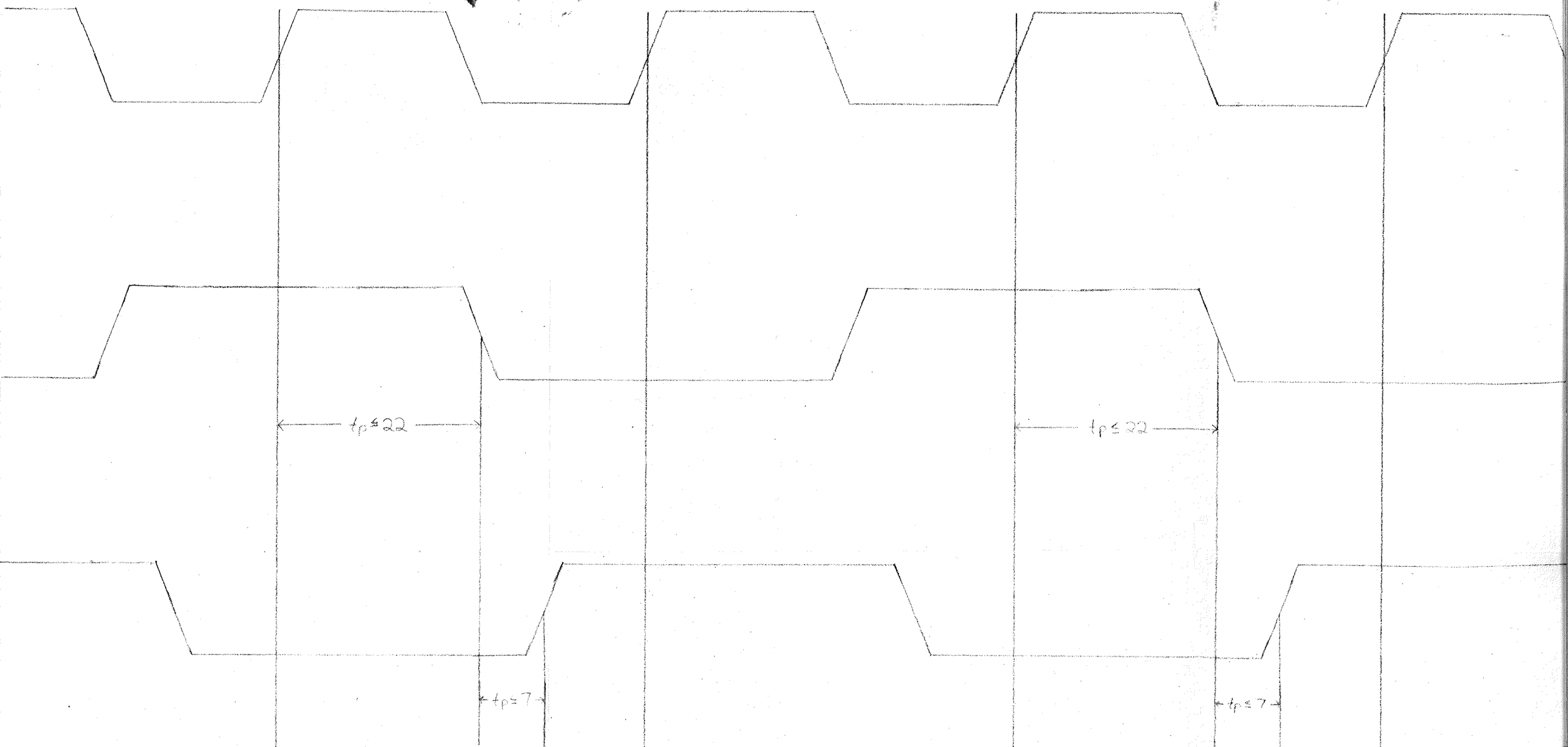
CLOCK (25 MHZ)

25LS164 OUTPUT

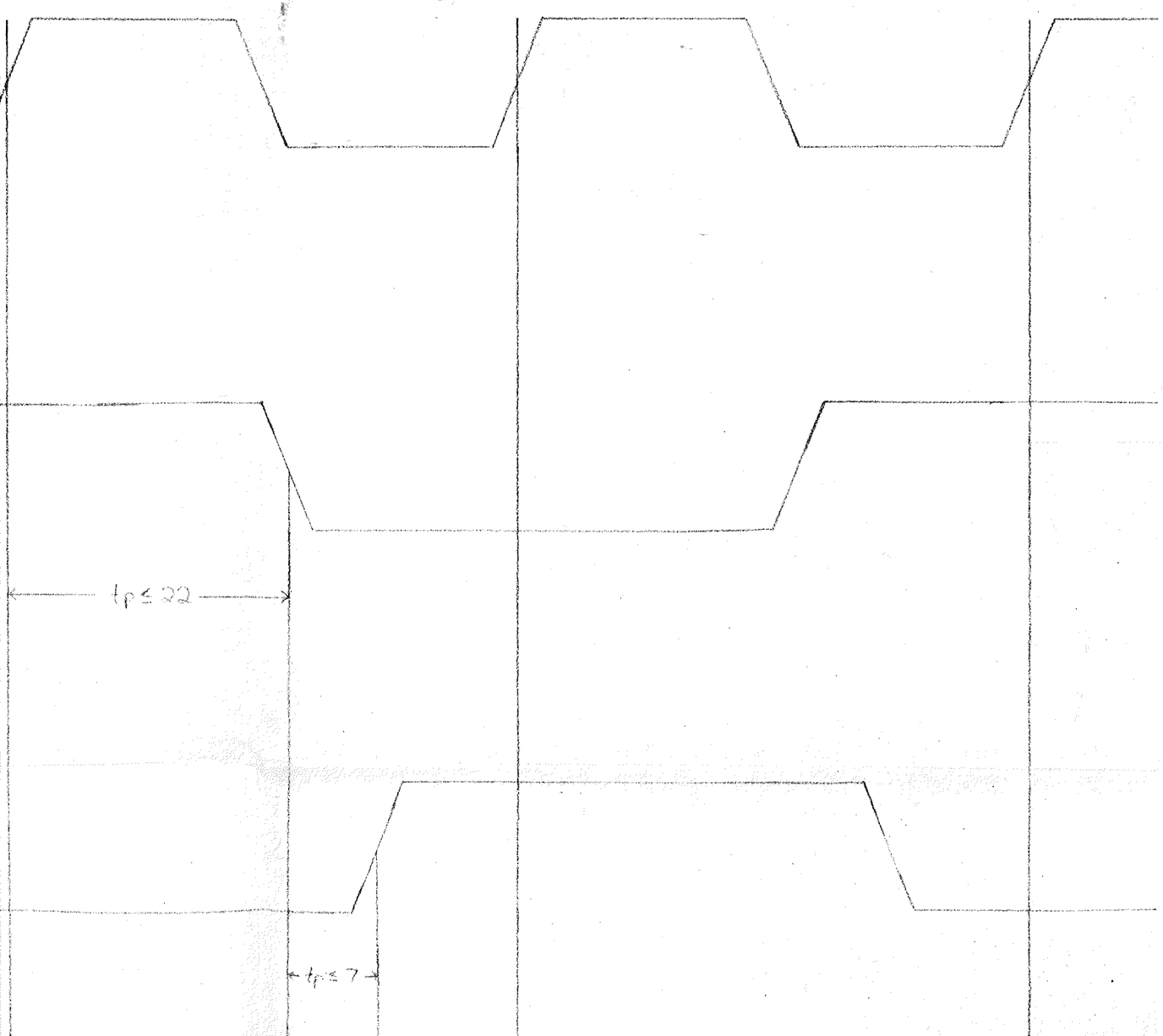
74S133 OUTPUT

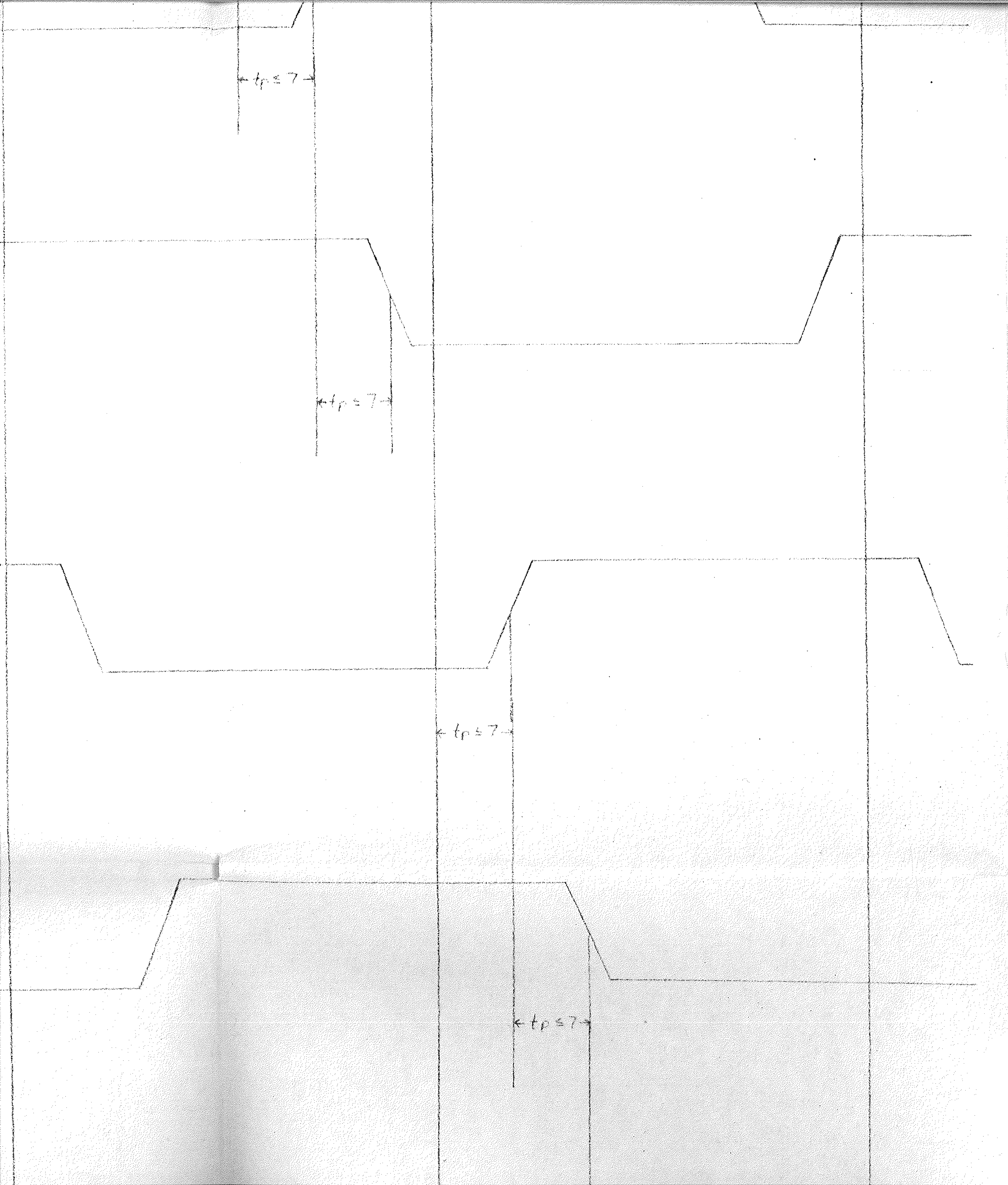


NUMBER SIEVE INTERRUPT TIMIN



ERRUPT TIMING





tp≤7
tp≤7
tp≤7
tp≤7

TPUT

$\leftarrow t_p = 7 \rightarrow$

$\leftarrow t_p = 7 \rightarrow$

TPUT

$\leftarrow t_p = 7 \rightarrow$

$\leftarrow t_p = 7 \rightarrow$

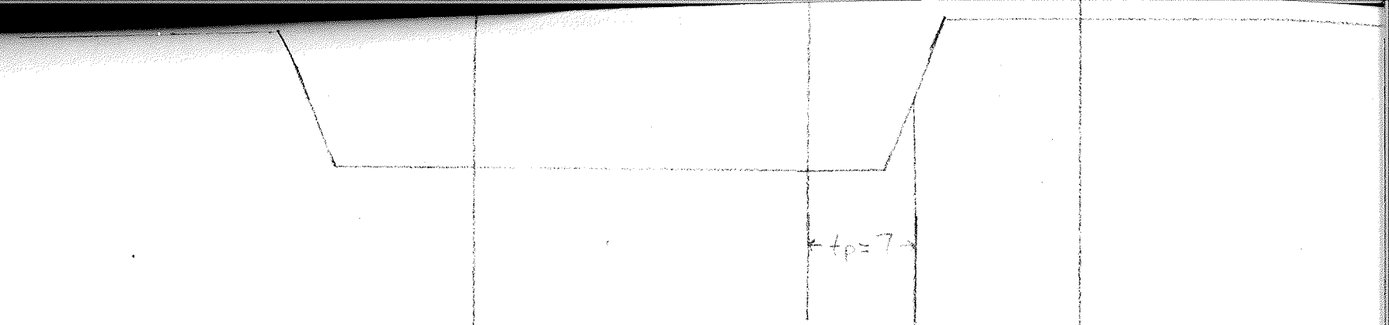
TPUT

$\leftarrow t_p = 7 \rightarrow$

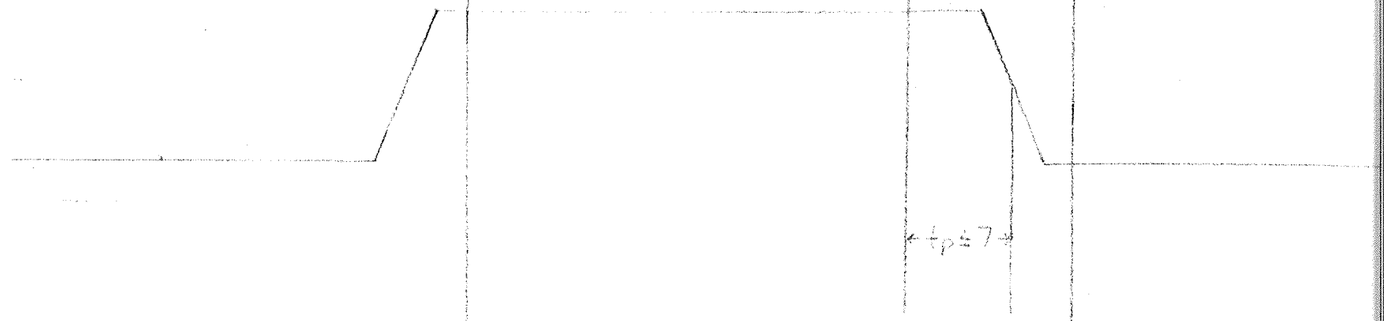
TPUT

$\leftarrow t_p = 7 \rightarrow$

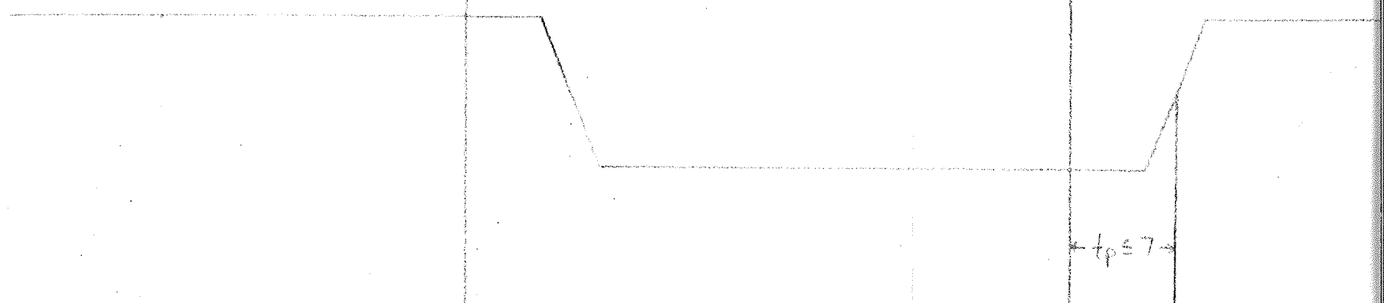
74S133 OUTPUT



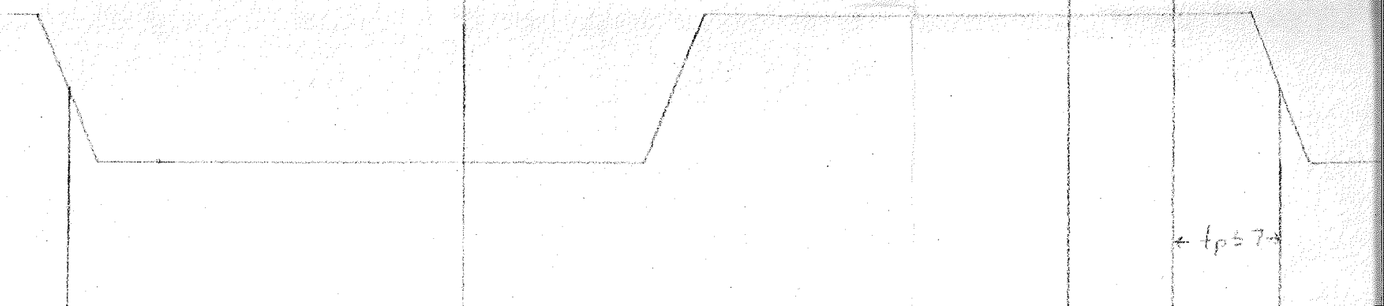
74HC4 OUTPUT



74F74 OUTPUT

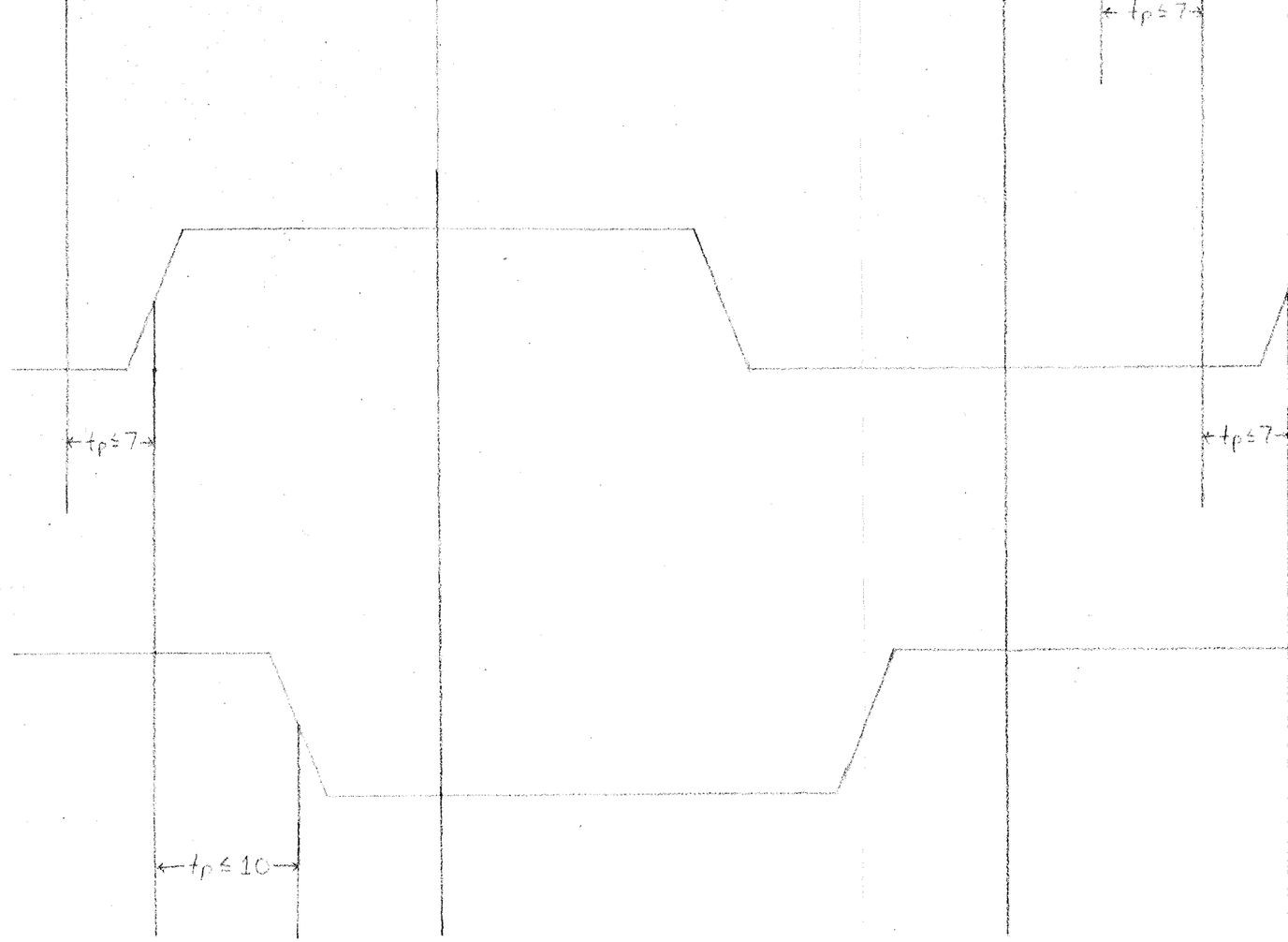


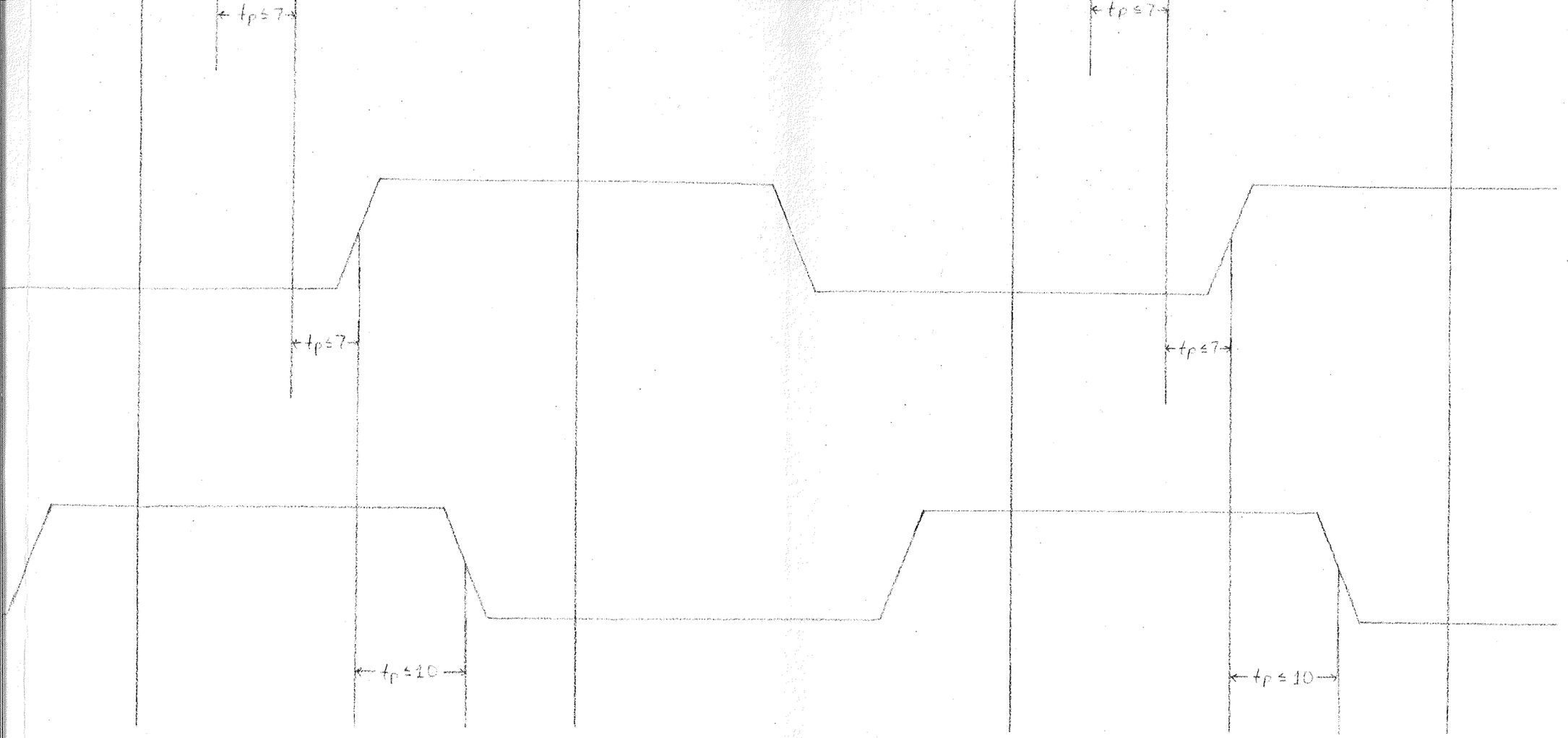
74S133 OUTPUT



74F04 OUTPUT

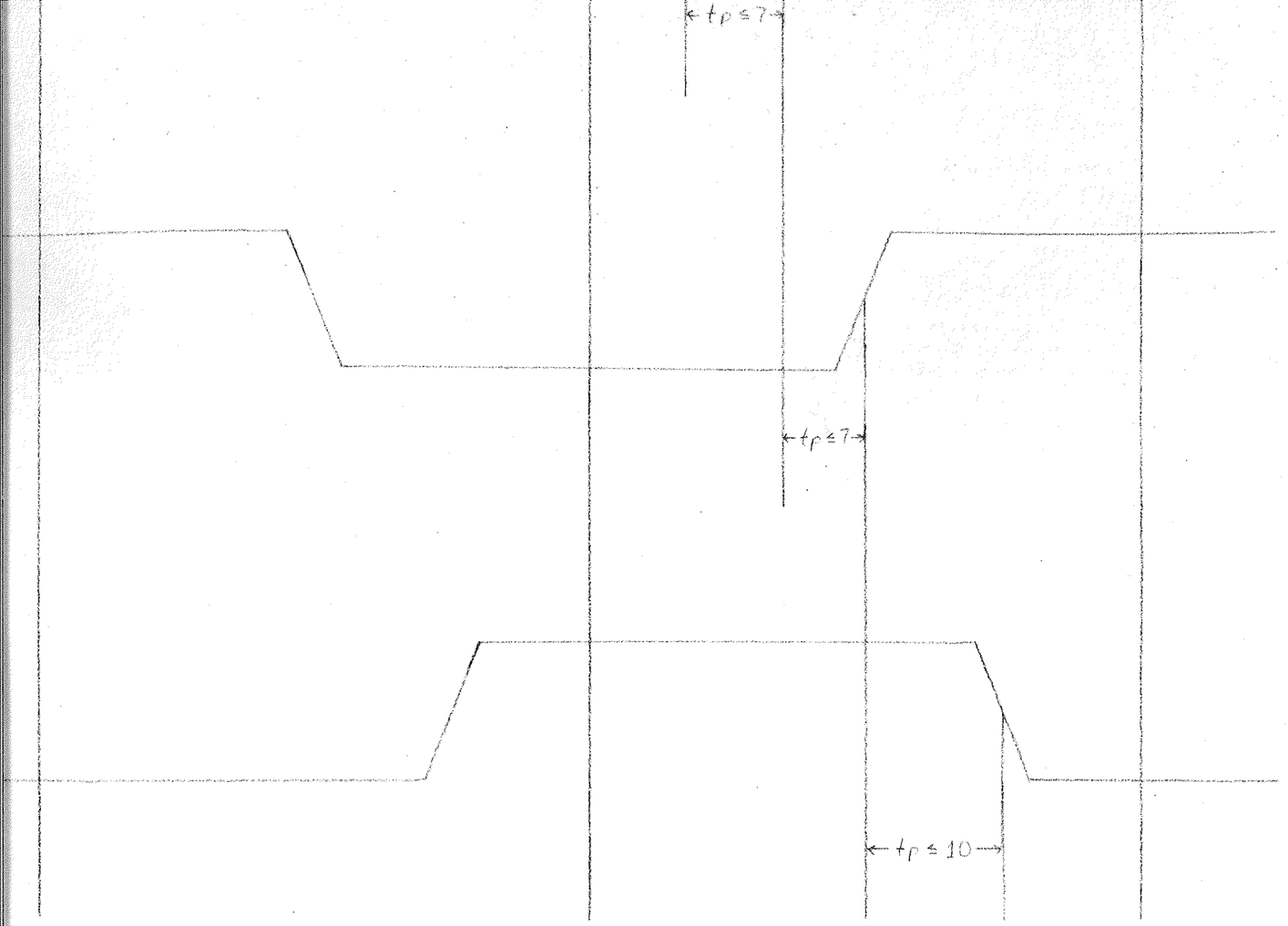
74S11 OUTPUT





SCAL

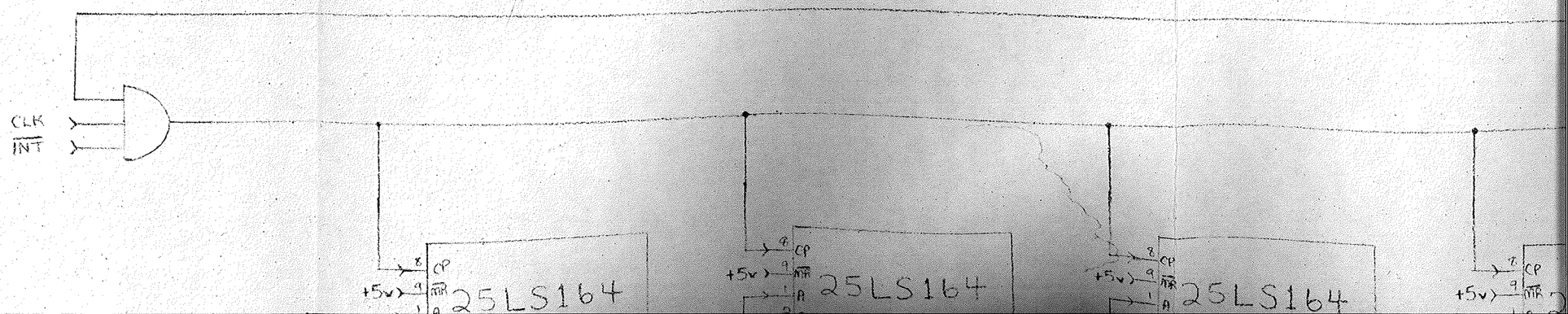
C.D. P
JUNE



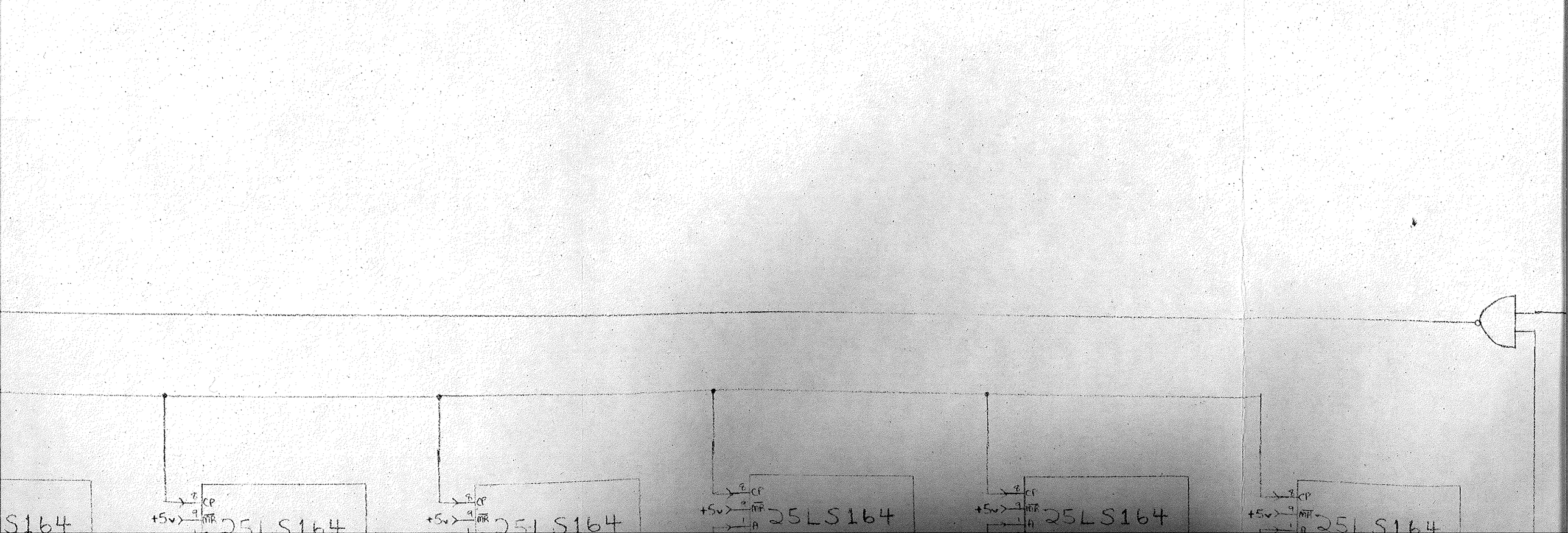
SCALE: 10 n.s. per inch

C.D. PATTERSON
JUNE 23/1981

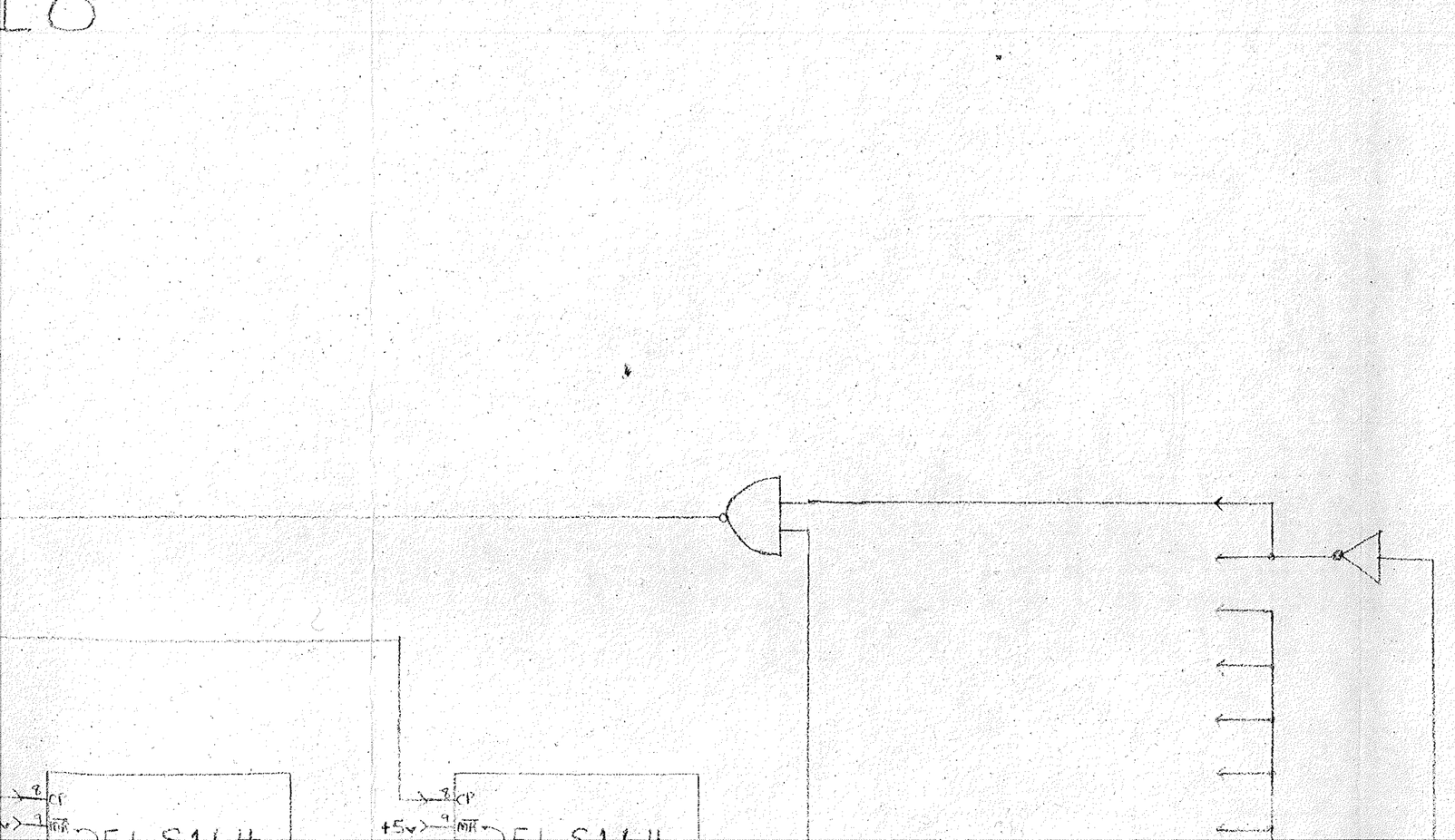
NUM

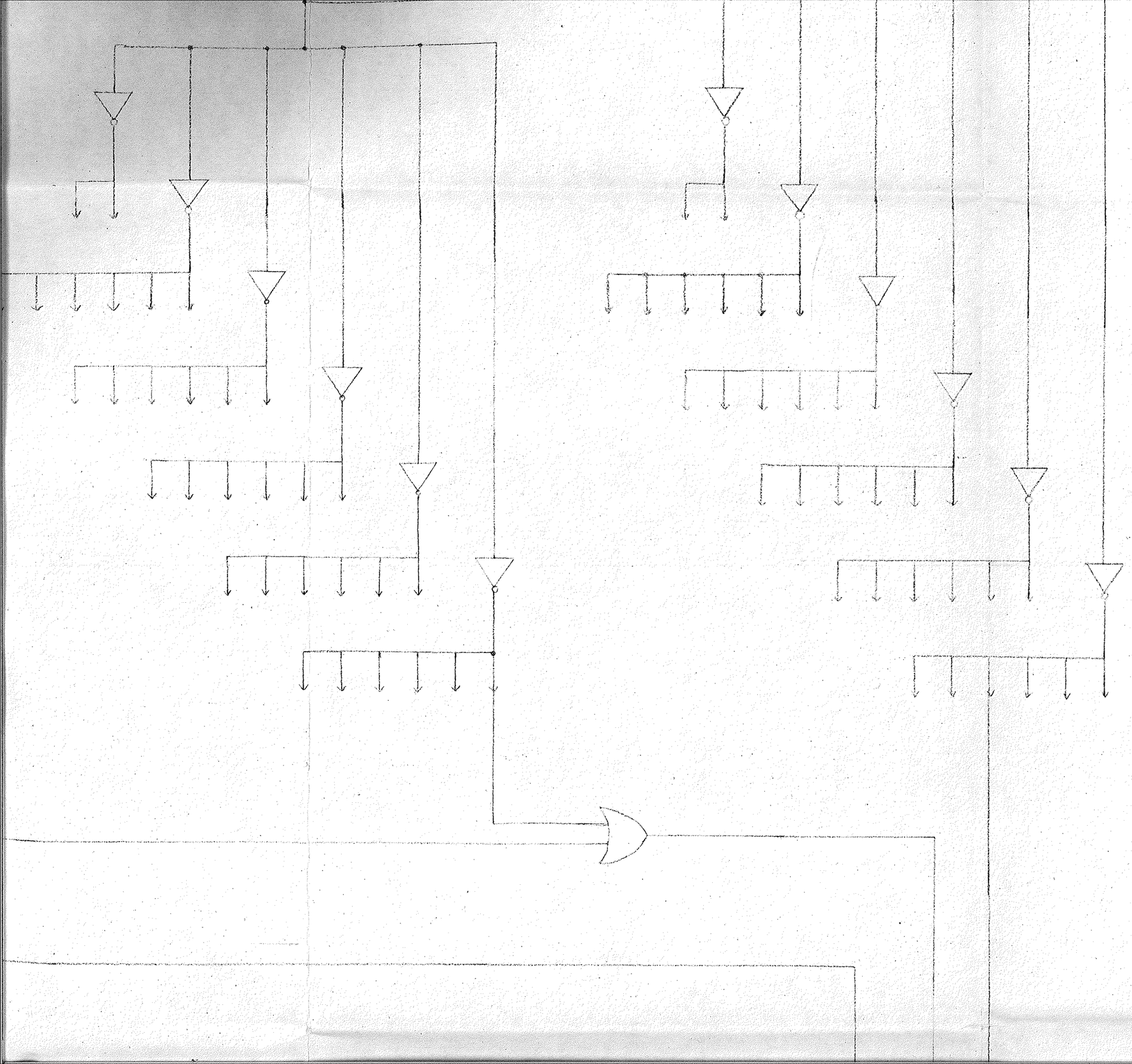


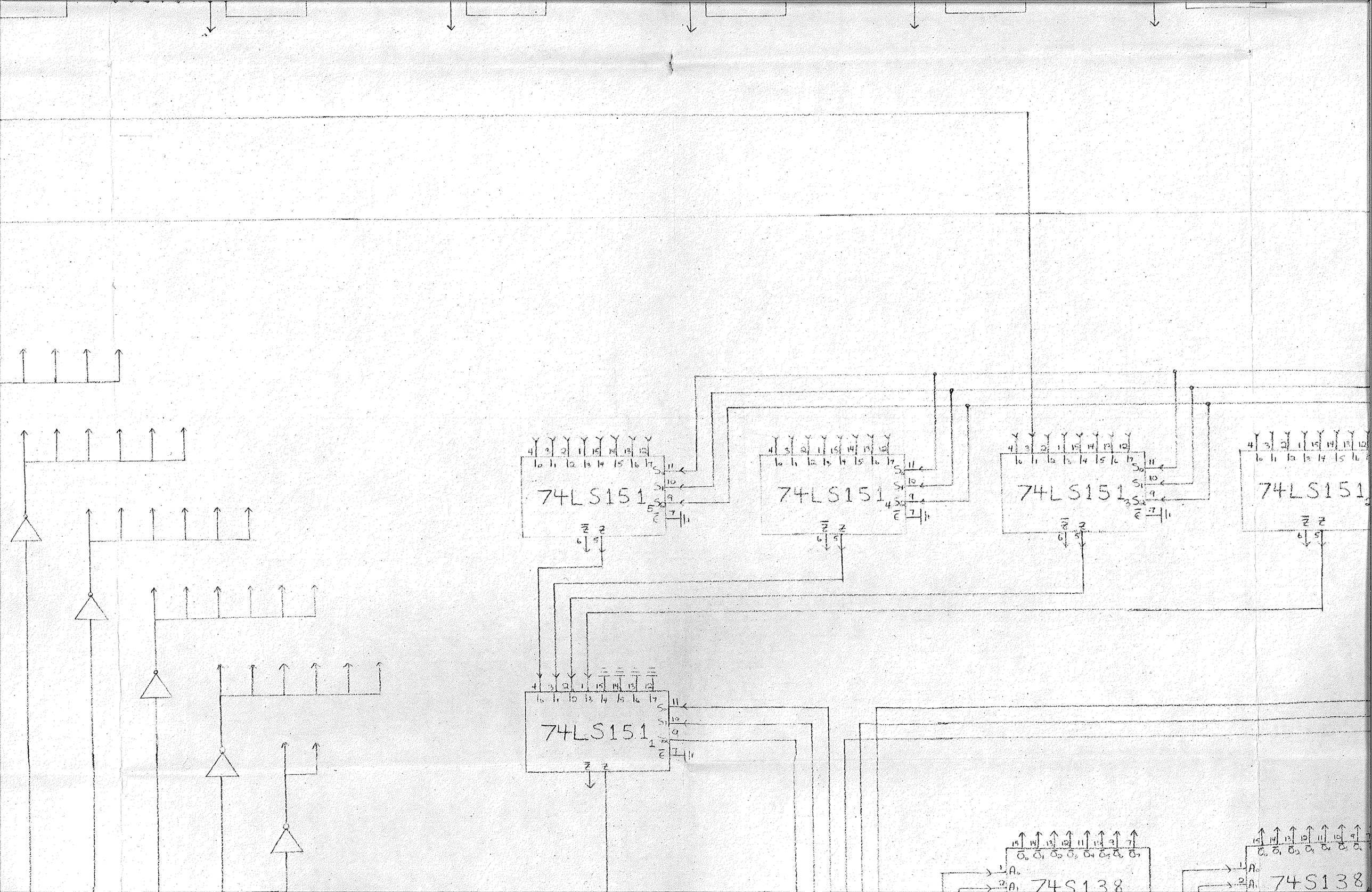
NUMBER SIEVE RING #18

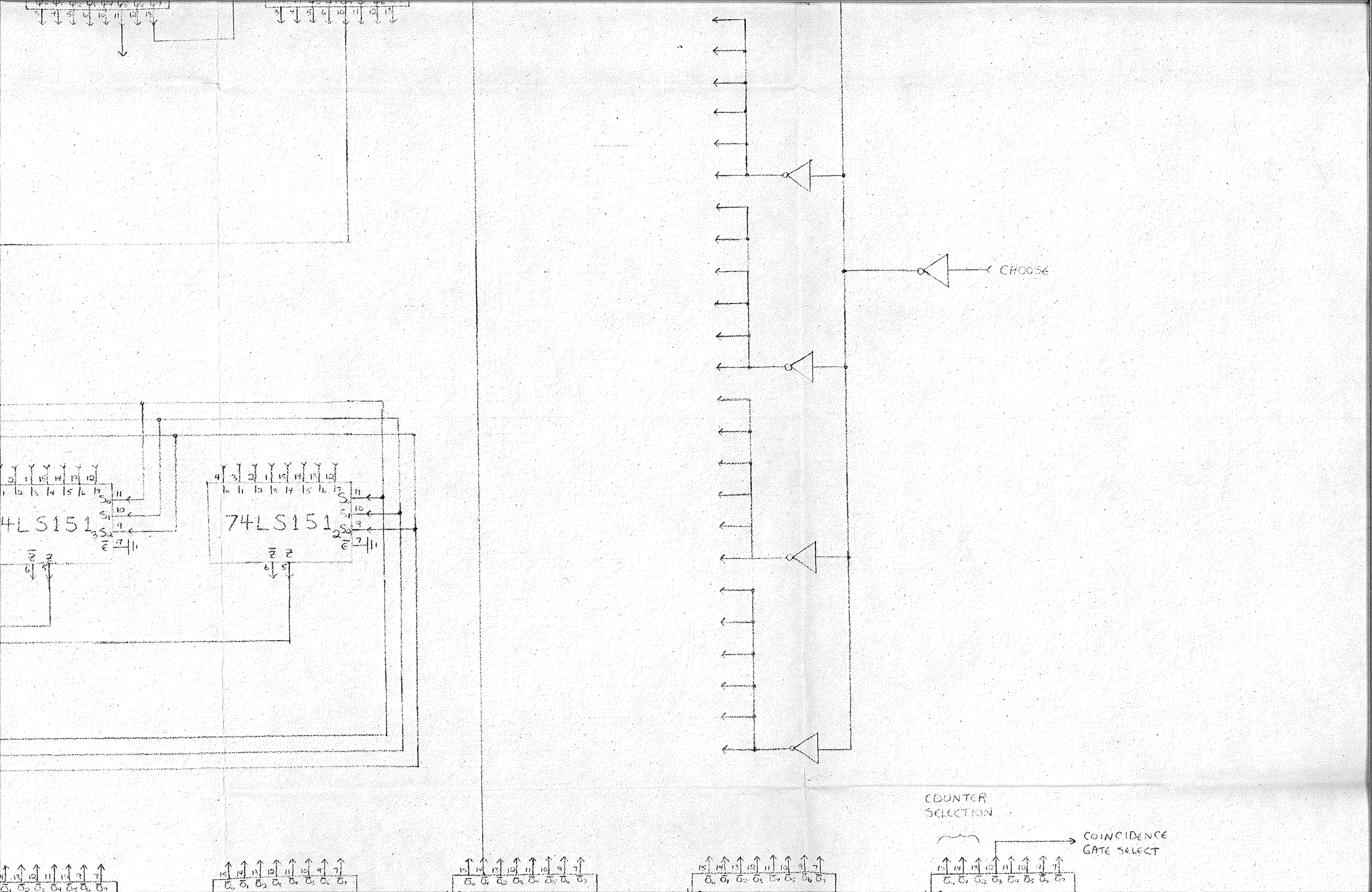


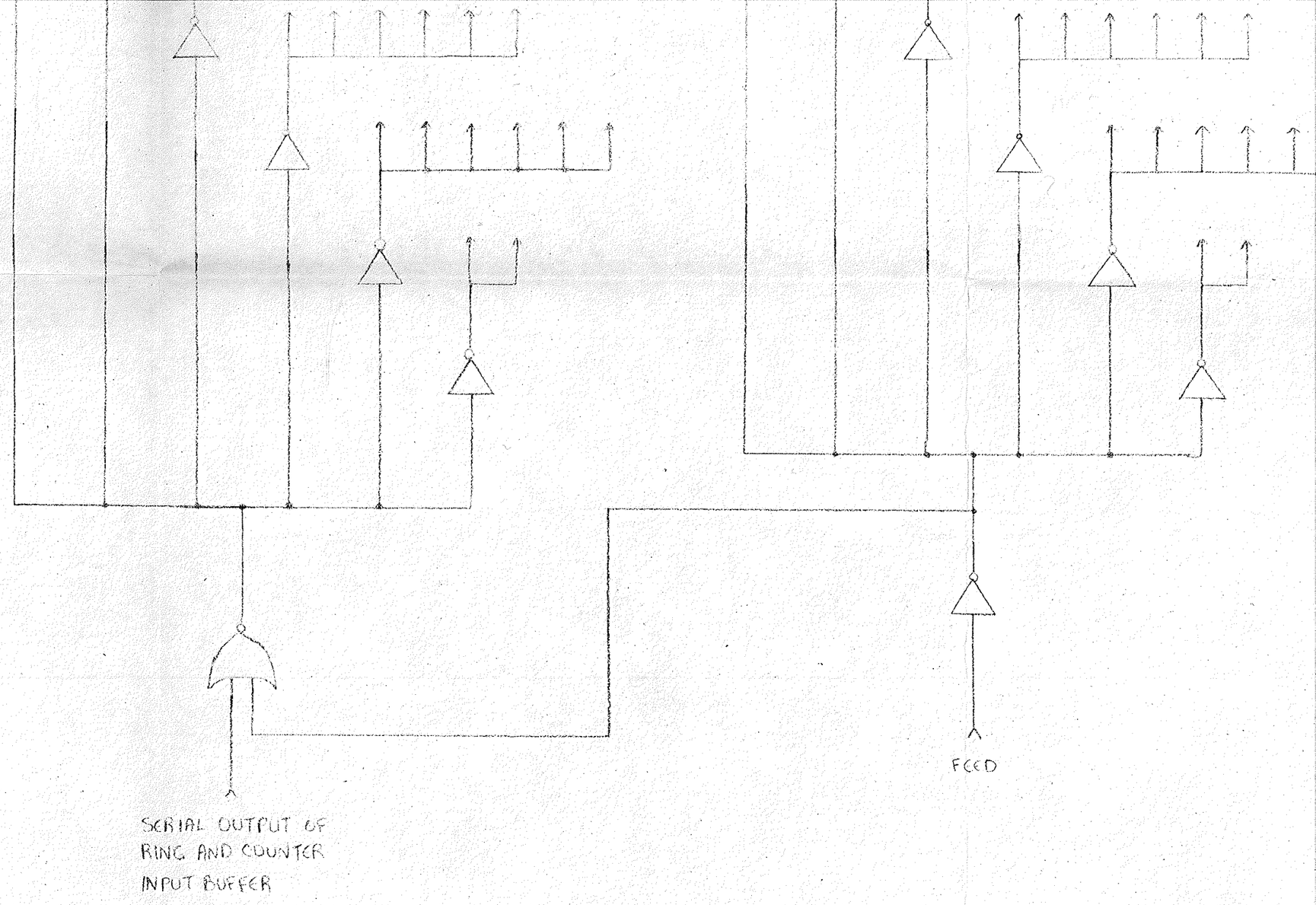
L8

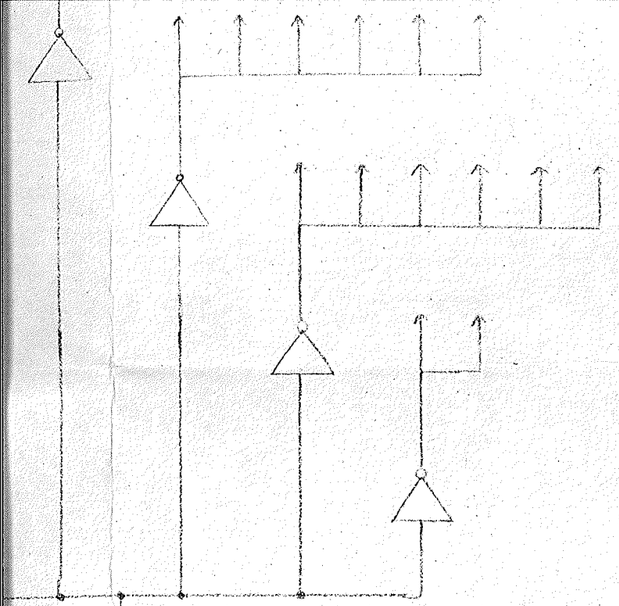


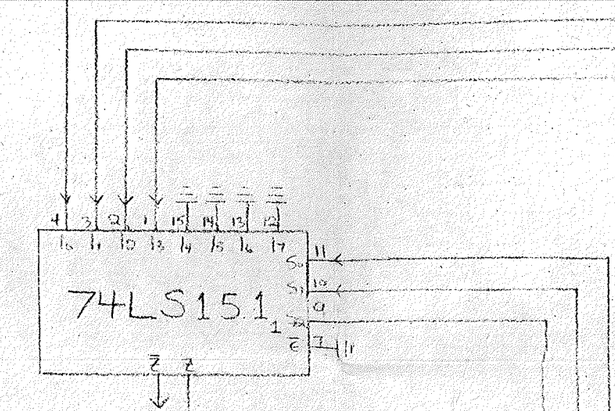



74LS151
74LS151
74LS151
74LS151
74LS151
74LS138
74LS138

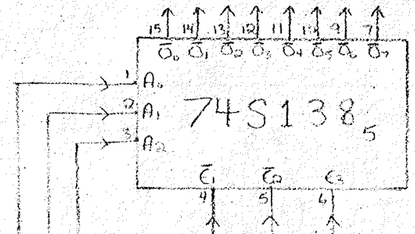

74LS151
74LS151
CHOOSE
COUNTER SELECTION
COINCIDENCE GATE SELECT

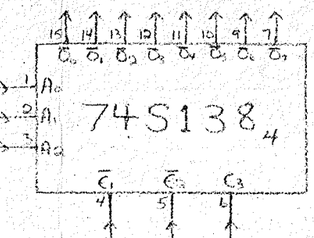

SERIAL OUTPUT OF
RING AND COUNTER
INPUT BUFFER
FEED



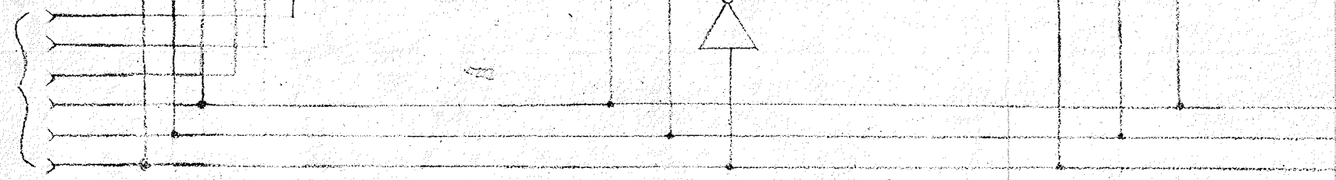


SERIAL INPUT TO
RING AND COUNTER
OUTPUT BUFFER

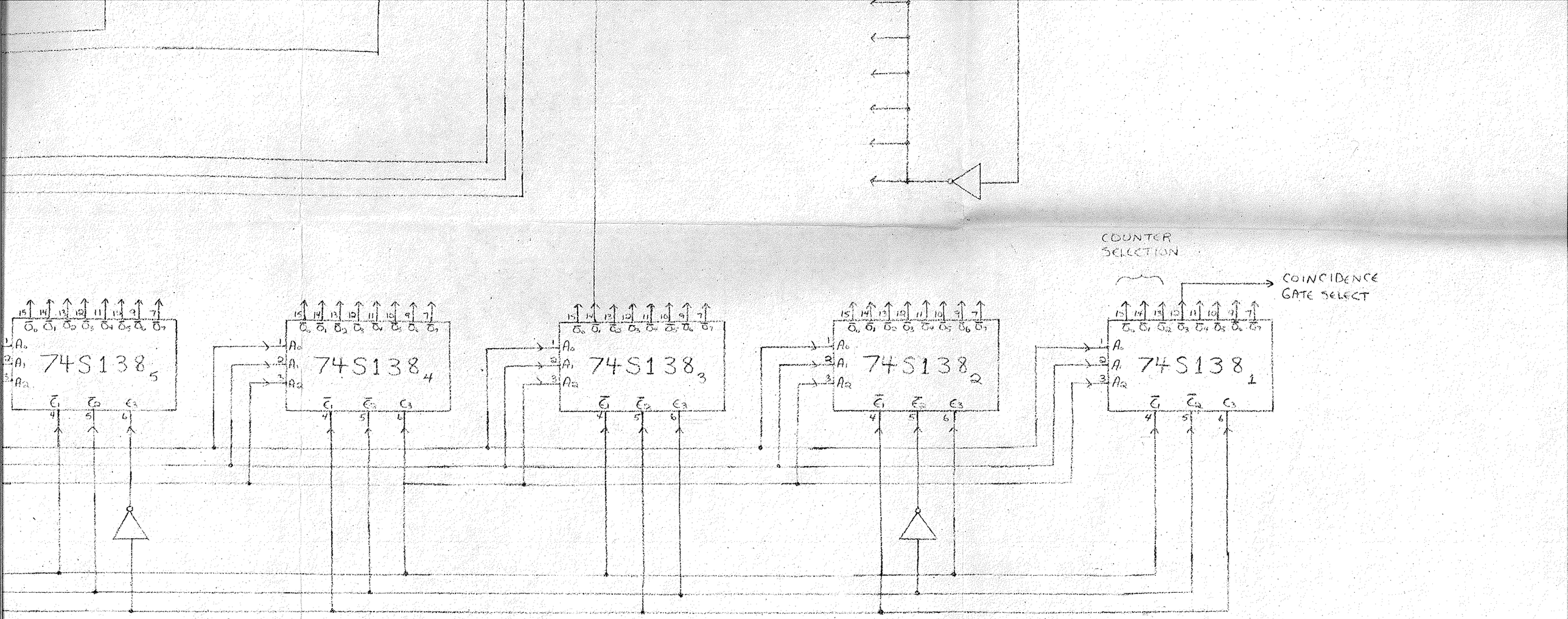




RING AND COUNTER
DEMULTIPLEXER INPUT



FEED



C. D. PATTERSON
JUNE 10/1981