1987

Design, fabrication and implementation of a hash table processor

Robert Paul Ketrick

A Thesis

Design, fabrication and implementation of a hash table processor.

Rochester Institute of Technology
Rochester N.Y.

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

July 04, 1987

Robert Paul Ketrick

Thesis Committee Approvals :

Dr. John Ellis (Advisor)

Dr. Peter G. Anderson

Dr. Roy S. Czernikowski

Abstract

Design of a hash table microprocessor
By Robert P. Ketrick

Hash tables have been used frequently to implement organized data table storage. However, there is a great deal of overhead in executing the hash algorithm every time the table is accessed. Alternatively, a content addressable memory (CAM) is a hardware implementation of an organized data table. A CAM requires very complex hardware design for each memory cell and therefore yeilds a very low cell density on each chip. This requires a large number of expensive chips to implement a large hash table.

In this thesis a simple hash function was designed into an integrated circuit using 4 micron NMOS technology. The scope of this thesis covers theoretical development, circuit design, simulation and fabrication. This chip performs a simple hashing algorithm using standard RAM to store the data and can interface to several 8-bit and 16-bit microprocessors.

The design of this device is aimed at improving the speed of compilers, assemblers, and whenever fast access to organized data tables are needed.

Contents

1) Introduction and Background 1
Problem Statement 4
Previous Work 6
Content Addressable Memories 8
Hashing 10
Uses of hashing and hash tables. 15
Theoretical and Conceptual Development 17
Principles of hash coding. 17
Theoretical design of a hash function 21

2) Project Description 24
Functional Specifications 25
Internal Architecture 33
Functional Description 38
Limitations and Restrictions 40
Problems encountered 41

3) Test Plan and Statistical data 44
Simulation Tests and Requirements 44
Programs Written 47
Statistical anlysis of the hash function. 47

4) Conclusion 55
System application for HTIC 56
Configuration Diagrams 59
Software Design Specifications 62
Terminal Emulator Program Specifications 63
Local Operation System Specifications 63

Glossary of terms and acronyms 64

Bibliography 66

Appendix A: Source code to generate the PLA. 67

Appendix B: Source code for hash function simulation. 74

Appendix C: Source code for generating test data. 85

Appendix D: State Diagram for PLA. 87

List of Illustrations

Figure 1. Mapping of Items to memory space. 15
Figure 2. Pinout Diagram 31
Figure 3. State Definition of S0 and S1 32
Figure 4. State Definition of L0 and L1 32
Figure 5. Internal Register Layout 36
Figure 6. Memory Cell Design 37
Figure 7. System Configuration 60
Figure 8. Microprcessor Configuration 61
Figure 9. Microprocessor Memory Map 62

INTRODUCTION AND BACKGROUND

Since the dawn of computers the goal has always been to make them operate faster and more efficiently. Engineers, software designers and VLSI technology have been the major contributors to the effort of improving computing efficency. Engineers have made great improvements in logic design. Software designers have greatly improved code writing techniques and together both have made tremendous progress in the area of design automation. In particular, the VLSI advances are responsible for increasing the complexity of the chips resulting in the improvement of speed and a chip size reduction.

However, increasing the processor speed is only one way the designer can improve the processing power of the computer. One of the biggest overheads in operating a computer system is the time spent storing and retrieving data from memory when the data is stored in a table format. In order to help in this area, the development of data management units known as CAMs (Content Addessable Memories) evolved, to store data based on its content rather than by assigning it a sequential address location. However, if a hardware CAM is not available,

then the table storage can be performed by a method implemented in software known as hashing.

The hashing algorithm is a process whereby the data associated with a variable is stored in a table based on some manipulation of the bit stream that makes up the variable name. This table is referred to as a hash table. A hash table is very useful in several types of systems. It can be especially advantageous in interpreted languages such as Basic or Lisp, as well as in assemblers, compilers, data base management or whenever data must be stored in a table format where fast storage and retrieval is necessary.

Problem Statement

The problem is to design and implement an Integrated Circuit that will perform a simple hashing function. The objective of this is to show how a software based hashing function can be implemented in hardware. This will support a novel view of CAMs and show how the

device can be implemented into a system to improve the efficency of a compiler or an assembler.

The integrated circuit is designed to function with several different μP's, both 16-bit and 8-bit machines. The proposed system is a 16-Bit machine which will have a speed advantage over the 8-bit μP's. This speed advantage results from the 8-bit μP's having to perform two writes to the PIA (Peripherial Interface Adaptor) before the data can be loaded to the 16-bit data bus of the HTIC.

The HTIC has two large blocks of isolated memory that are used as a storage table. One block is for the key values (32-Bits wide) and the second block is for the data values (16-Bits) associated with each key. The 16-bit data values are used as pointers. When a value is stored into the table, its corresponding variable name is stored along with it. This is where this memory scheme gets its name Associative Memory. The data is stored in association with its variable name.

Previous Work

Hashing may be thought of as a software implementation of a CAM. However, hashing algorithms generate a lot of overhead since the hashing routine must be run everytime there is an access into the hash table. Traditional CAMs have a drawback in that they require special hardware to be designed into each memory cell in order to function, thereby producing a very low yield of memory cells per unit area. Our HTIC (Hash Table Integrated Circuit) is a hardware implementation of a simple but fast hashing algorithm which incorporates most of the software overhead from the CPU. The HTIC can be interfaced to conventional RAM which requires that no special hardware be designed into each memory cell and therefore makes this implementation more cost effective. This arrangement is a cross between the two standard implementations of table addressing.

The development of CAMs started during the mid 1950's. There are two implementations of content addressing. One uses a data dependent memory mapping implemented by pure programming techniques; the second one uses special hardware in memory cells to store and retrieve data items. Both principles were developed simultaneously, about 1955.[1]

Content Addressable Memories

The basis for our HTIC lies in the concept of content addressable memories (CAMs). A content addressable memory is a memory system that searches for data on the basis of their content rather than on the basis of their location.[2] This basically means that CAMs are a way of storing data in an ordered manner, much like alphabetizing.

Since content addressable memories require a great deal of special hardware associated with each memory cell in order to perform parallel reading of all cells, only a few thousand cells may be constructed on a single chip; [3] therefore very little information can be stored in one CAM chip. However, CAMs hold an advantage in that a readout operation can be accomplished in one cycle due to the CAMs ability to compare all locations in parallel. Some CAM's allow the ability to do partial key matching or range checking. These devices require even more special logic for each cell.

---

[1] Content-Addressable Memories , T. Kohonen, pg. 6

[2] Content-Addressable Memories , T. Kohonen, pg. 1

Hashing

Hashing is an alternative method of orginized table storage. A key is used to place data into the table. The key is a tag that is associated with each data. In many cases the key is the actual variable name of the data being processed. Implementation is accomplished with pure programming techniques. Hence, everytime the hash table is accessed the hashing algorithm must be executed in order to store or retrieve the data.

Hashing can serve as a very fast method of storage and retrievial providing that the key and data are always in the same format. The domain of keys in a system is referred to as the data space and the domain of all memory locations is referred to as the address space. [4] Therefore, hashing can be viewed as mapping the data space to the address space.

Hashing is done by applying an algorithm to the key in order to obtain an address within the available memory to store the data. The same algorithm must be applied

---

[3] Associative Memory , T. Kohonen, pp. 13 - 15.

[4] Content-Addressable Memories , T. Kohonen, pg. 40

during recall in order to retrieve the data.

The hashing algorithm implemented in this thesis works as follows. We first form an address from the key by using a function f(key). This address is the initial probe into the hash table. Once the address is formed the contents of the location pointed to by the result of f(key) will be examined. If the location is empty then we are free to store the key and data there. If the location is used and equal to the key that we are looking for, then we may either read the data associated with the key or may overwrite the existing data value. If the location is not empty and not equal to the key we are looking for, then we must form a new address based on the current address with the addition of some offset. The offset can be chosen in several ways : (1) add 1, called linear rehash; (2) add k to the kth collision, called quadradic rehash; or (3) add g(key), called random rehash. The recalculation method choosen in this thesis is random rehashing. This rehashing continues until an empty location is found or the key is matched.

Since the hashing algorithm relies on the knowledge of which memory locations are filled when it selects a location to place data into the table, removing an entry once put into the table entails additional complexity

that we have chosen not to include in this project. If any entries were removed from the table then we would lose the ability to retrieve many of the keys that have already been stored. Locations that were previously written to and caused the hash algorithm to rehash over the used location would now appear empty. Therefore, once a key is put into the table it must remain until the application has finished and the table can be reinitialized for another run.

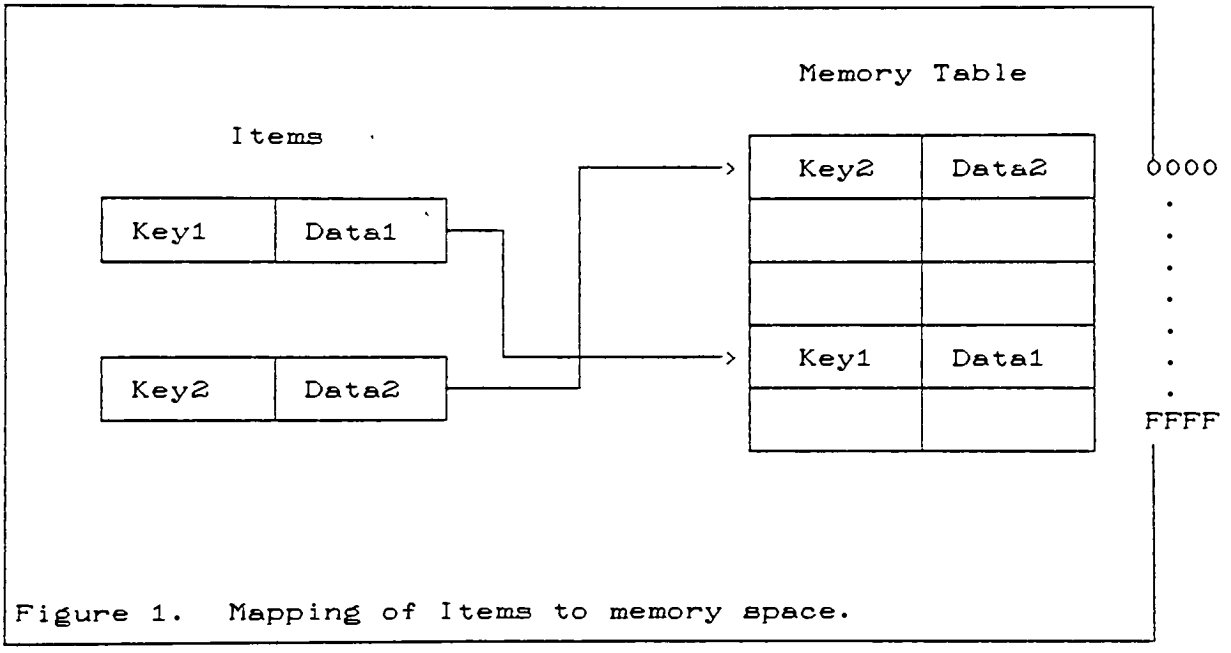


Figure 1. Mapping of Items to memory space.

Uses of hashing and hash tables.

One of the first applications of a hash table was the IBM assembler (1954). It used a hash table to set up and keep track of the symbols used during the assembler's execution. Each symbol was assigned a value in the table and that value represented the symbol's address for conversion to object code.

In recent years, practically all symbol tables for assemblers and compilers are implemented with the use of a hash table. Symbol tables are accessed frequently and in an unpredictable order during compilation and hash coding has proven most efficient in maintaining them.

Furthermore, the use of hash tables can be extended to the area of data base management, especially in multiple key word searching operations. One can see from this how useful a hash table processor can be to these types of applications.

Theoretical and Conceptual Development

Principles of hash coding.

The focus of hash coding is to map all possible data into the available address space with as uniform a distribution as possible in order to minimize search times. This is done in two steps. First, the keyword must be converted to a numeric form, then the numeric form must be mapped to the address space. This is accomplished by creating a hash function, *f(x)*. This function is used to create an address for each key: *Address = f(key1)*. Ideally, we would like our hashing function, *f(x)*, to generate a unique address that is uniformly distributed throughout the entire address space for every key value in the data space. That is, we would like *f(key1) ~= f(key2)* for all values of key1 and key2 not equal to each other. This would insure a one cycle operation for both reads and writes. However, since the data space is usually much larger than the address space, with every hash function there exists a finite, nonzero probability that *f(key1) = f(key2)*, where key1 ~= key2, for some values of key1 and key2 that exist in the

data space.[5]

The condition *f(key1) = f(key2)* is called a conflict or collision. We will refer to the nonzero probability, *P(c)*, as the probability of a collision occurring. If we choose a good hash function then *P(c)* will be low because our distribution will be close to uniform.

Since only one item is allowed to occupy any one location, our hashing algorithm must have some method of dealing with this problem. Kohonen [6] suggests adding one to the address until an empty location is found, this is called linear hashing. Once a collision occurs and we begin using the rehash function, we must mark each location after it has been used so that the rehash of a key does not overwrite an already existing entry in the table. One way to solve this is to add an extra bit to each location and set the bit when a location has been used.

Upon retrievial there is a problem in identifing which value of the hashed or rehashed address the data belongs to. Since *P(c)* does exist, we must have a positive way of

---

[5] Content-Addressable Memories , T. Kohonen, pg. 56

[6] Content-Addressable Memories , T. Kohonen, pg. 40

identifing which data goes with which key. As shown in Figure 1 on page 15, a solution to this is to set aside a portion of memory location to store the actual key or some unique function *f'(key)*. Now, when we probe into the table we can look at the contents of the key portion of the addressed location to determine if it is in fact the location that we are looking for. If it is, then our search is done. Otherwise, we must rehash and probe into another location until we find the location that we want.

Theoretical design of a hash function

The key used for this design is comprised of a bit stream 32 bits wide, the equivalent of four ASCII characters. Since we have no knowledge of the distribution of the data we assume that it has a uniform distribution. Therefore, we conclude that if we choose the odd bits or the even bits for our initial probe, we will get a uniform distribution. Hence, we will choose all the odd bits from the bit stream thereby creating a 16-bit address pointer into the hash table and a hash table size of 64K locations. However, as stated previously, every hash function needs a method of dealing with collisions. Therefore, we need some kind of increment or offset to add to the original address value.

Our increment for this function will be all the even bits of the bit vector and is therefore independent of the initial probe. The least significant bit of the increment is forced to a logical "1" and therefore forces the increment to be odd. When memory size is a power of 2, and the increment is odd, the set of probed addresses {(n*increment) mod (memory size)} is a permutation of all addresses of the memory; we therefore guarantee to probe the entire memory. Furthermore, experience has shown that it is almost impossible to do this wrong.[7]

---

[7] Peter G. Anderson, Committee member, private correspondence

PROJECT DESCRIPTION

The objective of this thesis is to design and analyze the performance of a specialized hash table processor. The circuit is designed as a hash table coprocessor unit that may be added to a system much like a math coprocessor is added for floating point operations. The chip is used as a memory mapped I/O device. The main CPU can use the device by issuing reads and writes to it. The CPU must also have an interrupt available in order to use this device. When the device completes processing a request it activates an interrupt request, similar to an I/O device.

Functional Specifications

The HTIC is a 64-pin DIP fabricated using 4 micron NMOS VLSI technology. The pinout for this chip is shown in Figure 2 on page 31. The definition of the I/O is as follows :

GND	Ground (0 Volts)
Vdd	+5 Volts
Ap0 - Ap15	Address Pointer lines 0 to 15. These lines are used to locate data in the hash table. When the HTIC probes into the table, the contents of the HREG are placed onto these

address lines. These lines are held until the IC generates another address pointer.

HtD0 - HtD15 Hash Table Data Lines. These data lines access the contents of the key table. They are isolated from the data lines of the CPU in the system.

CpuD0 - CpuD15 CPU Data Lines. These data lines connect directly to the hosts CPU's data bus and are the means by which the key value gets passed to the chip.

S0 and S1 Status 0 and Status 1 lines. There are three different types of status that the chip can return for any request: found, not found or empty. These status lines are read by the CPU after the HTIC has issued an interrupt request. See Figure 3 on page 32 for state definitions.

L0 and L1 Load 0 and Load 1 lines. These are sourced by the CPU and are used to instruct the chip which register to load. See Figure 4 on page 32 for the state definitions.

IPRR Interrupt Processor Result Ready. This line is active low and indicates to the CPU that the HTIC has finished processing the given data and is now ready to process the next key.

EHT Enable Hash Table. This line is sourced by the HTIC and acts as a chip select to the hash data table. It is activated when the HTIC is reading or writing to the table. This line is active high.

Write_Hash Write Hash line. This line is sourced by the Hash HTIC and is the read/write line to the hash data table. Write is active low and is delayed internally by 1/2 cycle (about 200 ns.) in order to create enough setup time for the RAM.

HW High Word enable line. This line is sourced by the HTIC and determines which byte of the key will be written into the data table. The most significant word is written when the line is high and the least significant word is written when the line is low.

CS1, CS2 Chip Selects 1 and 2. These lines are sourced by the host CPU and when selected will activate the HTIC. CS1 is active low and CS2 is active high.

R/W Cpu Read/Write line. This line is sourced by the CPU and is the read/write line from the CPU.

Reset This is a state reset line only, not a master reset. Since all internal registers must be initialized before the HTIC can function, only a state reset is needed. This pin is used to break the HTIC out of the rehashing loop. This pin must be activated via external logic. Activating this pin will not destroy the contents of any of the internal registers.

Clock Clock input. This is the clock input to the chip. A 2.5 Mhz crystal will be driving the input. All internal clocks will be generated from this input.

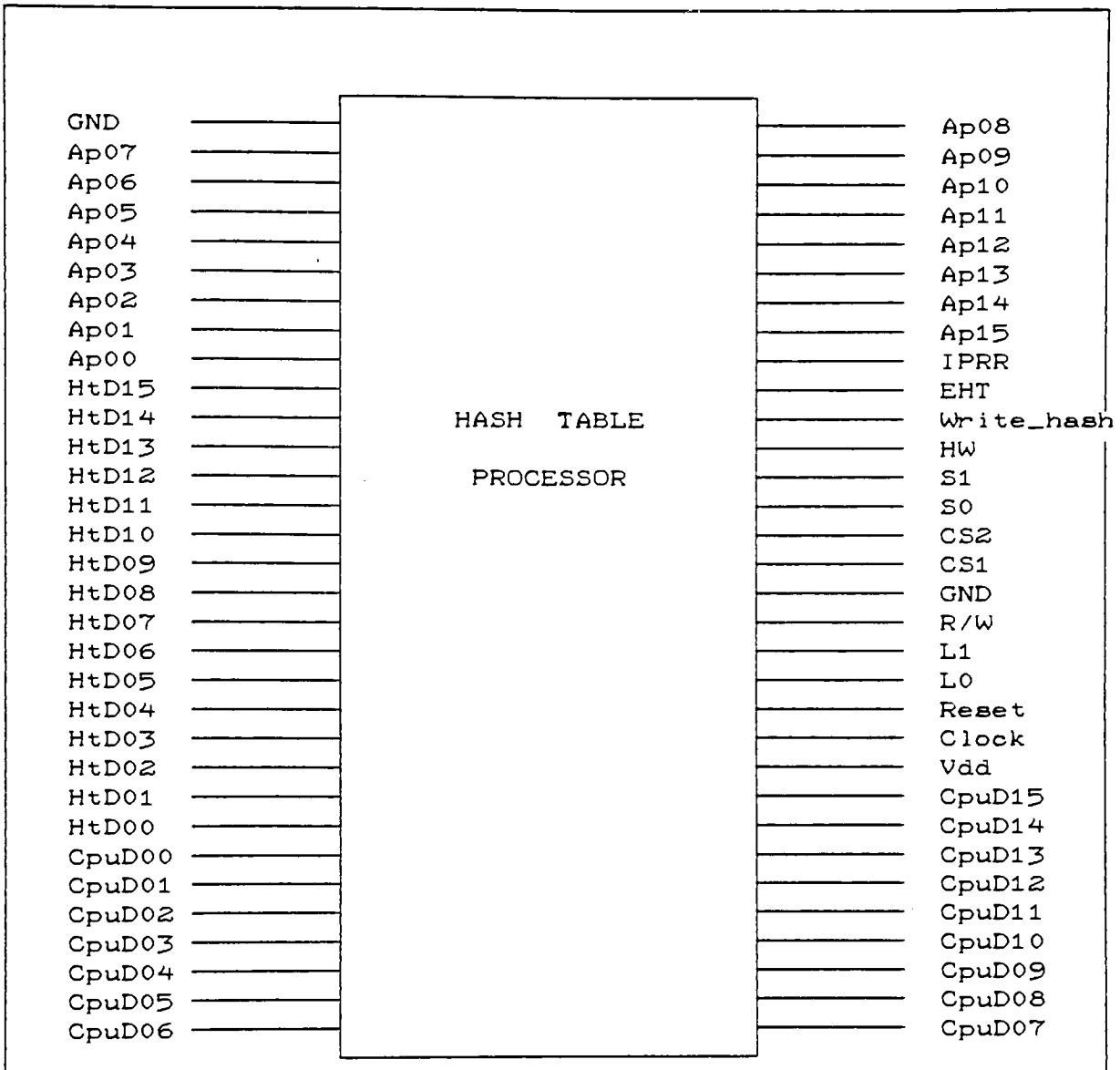


Figure 2. Pinout Diagram

S0	S1	Definition
0	0	Key not found in table
0	1	Key found in table
1	0	Location probed is empty
1	1	Undefined state

Figure 3. State Definition of S0 and S1

L0	L1	Definition
0	0	Load Key Memory Register 1
0	1	Load Key Memory Register 2
1	0	Load Empty Register
1	1	Write the new key into table

Figure 4. State Definition of L0 and L1

Note: The case of writing a new key is only valid if the HTIC has found an empty location.

The bit naming convension will be as follows :

MSB LSB

b31,b30,b29.....b3,b2,b1,b0

These 32 bits correlate to the 4 ASCII characters that

represent the key being processed.

Internal Architecture

The device consists of five internal 16-bit data registers, a carry chain ALU to perform the rehashing computation, a PLA (Programmable Logic Array) to generate the microcode for the hash table processor algorithm and a 16-bit comparator to compare hash table entries with the contents of the internal data registers. The diagram of the internal data path is shown in Figure 5 on page 36. The internal registers are defined as follows: key memory register #1 (KMR1), key memory register #2 (KMR2), empty register (EREG), delta register (DREG) and a hash register (HREG). The KMR1 and KMR2 hold the data passed to the chip from the host μP. This data is the ASCII representation of the characters that form the key being processed. KMR1 holds the most significant word and KMR2 holds the least significant word. The EREG holds a user defined value that is used to flag an empty location in the hash table. This value is written to all locations during the initialization phase on the device. After the KMR1 and KMR2 are loaded, the hashing network loads the DREG with the even bits of the key and the HREG with the odd bits per specification of

the hashing algorithm. Once this has been accomplished, the HREG and DREG look like:

HREG

b31	b29	b27	....	b5	b3	b1

DREG

b30	b28	b26	....	b4	b2	0

The data bus is split into three parts. The top consists of a 16-bit comparator. The middle section consists of three 16-bit registers, KMR1, KMR2 and EREG. The bottom section of the data bus consists of the bus driver, hash network, delta register, 16-bit carry chain adder and the hash register.

The comparator is split off only when the chip is comparing the hash table data to the internal registers. During loading of the internal registers, the comparator is not split from the middle section of the bus. The middle section of the bus is always precharged to +5 volts during $\phi 2$ of the clock cycle. This design uses a two phase non-overlapping clock scheme. During $\phi 1$ of the clock cycle registers may be written or read. During $\phi 2$ of the clock cycle, all registers are refreshed and the data bus is

precharged to +5 volts. The data registers cannot be accessed during ϕ2. Precharging is accomplished by connecting the data bus directly to the +5 volt bus during the ϕ2 clock cycle. Precharging is advantageous because of the circuit characteristics. The data bus is a large piece of diffusion which takes a long time to pull up to +5 volts due to its large capacitance. Furthermore, the time for the pullup transistor to charge the bus relative to the time that it takes the pulldown transistor to discharge the data bus is significantly greater. (See Figure 6 on page 37 for diagram of pullup and pulldown transistors.) Therefore, to increase the maximum speed of operation, precharging is used.

The bus driver is a set of 16 super buffers used to maximize the current drive through the long lines of diffusion in the hash network. The hash network manipulates the data from the KMR1 and KMR2 to form the HREG and DREG. The least significant bit of the DREG is grounded and the Carry_in_0 on the ALU is set to +5 volts to insure that the increment is always odd.

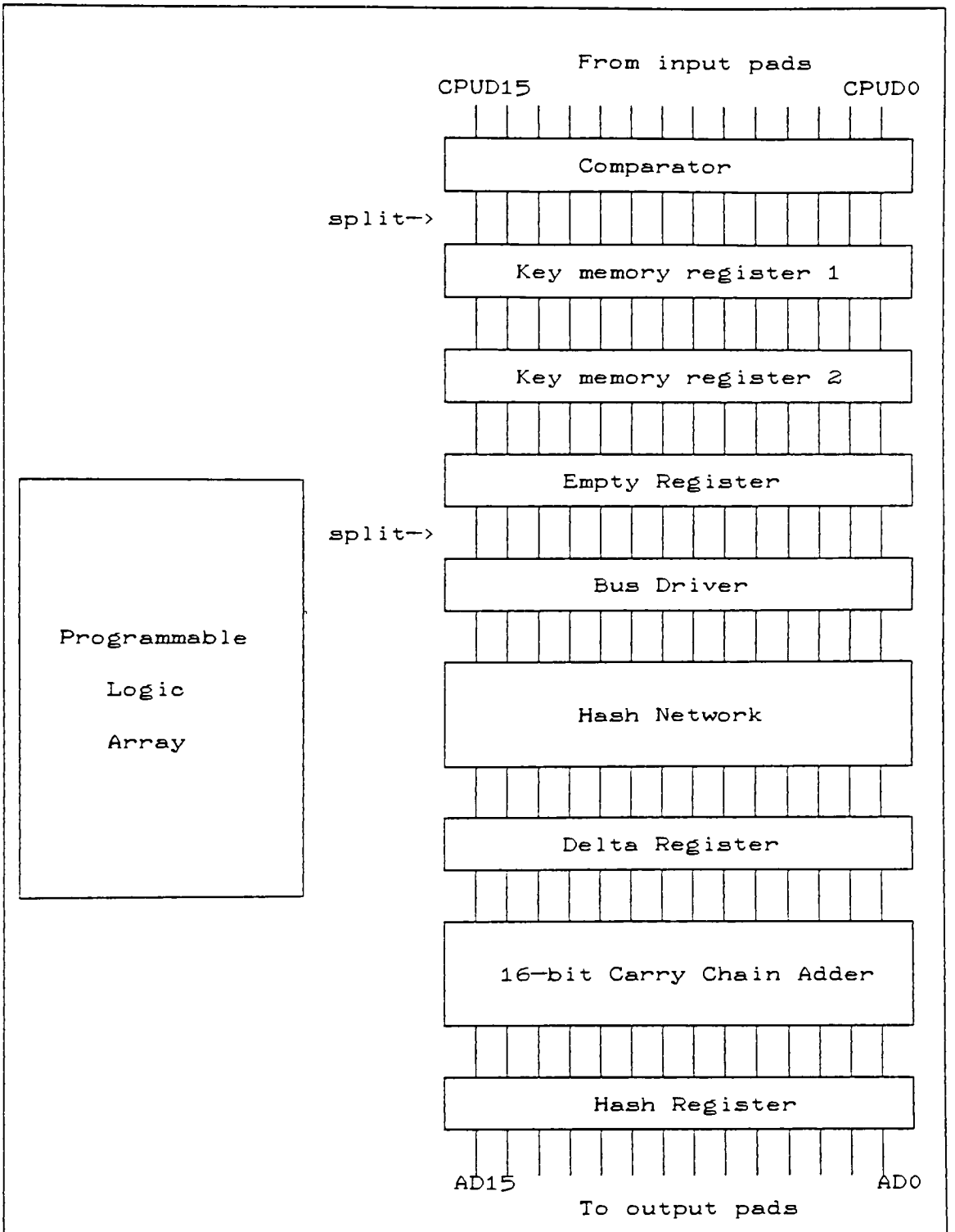


Figure 5. Internal Register Layout

The design of the internal data registers is shown in

Figure 6 on page 37. Each bit of the register consists of a pair of inverters. During phase two (#2) of the clock cycle the output of each inverter pair is fed back to the input thereby causing each cell to refresh itself. Each cell is also connected to the data bus by two pass transistors. One for writing and the other for reading the register. Both pass transistors have control signals attached to their gates.

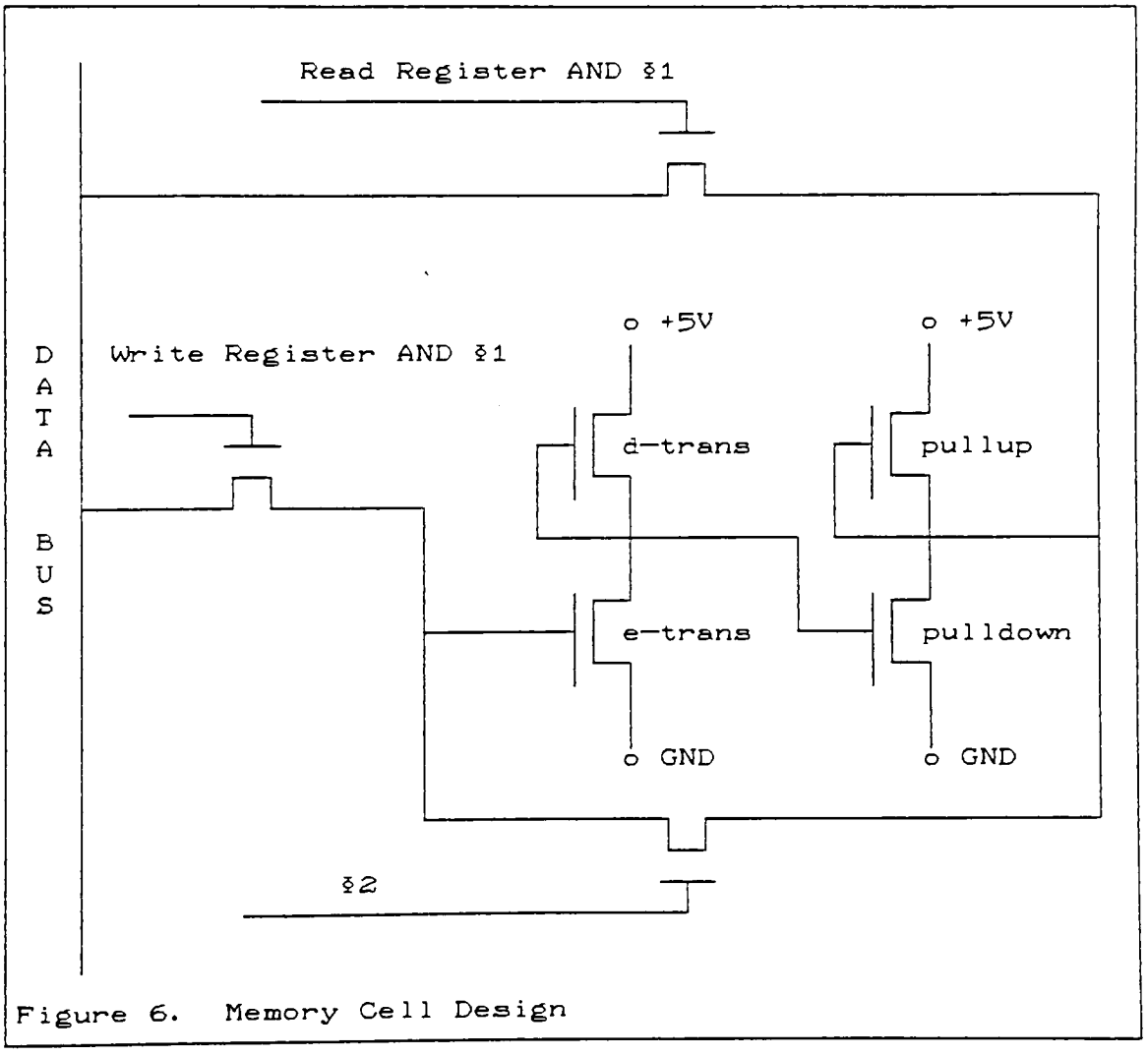


Figure 6. Memory Cell Design

Each control signal is qualified with ~ϕ1 or ~ϕ2 of the

non-overlapping clock. This is to insure that all register operations occur on the proper clock phase.

Functional Description

The chip is first reset to put its PLA into a known state. Then the EREG must be set to some value in an initialization sequence before the chip is ready to function. The initialization sequence is as follows. The user must first write 0000h to KMR1, this is to give the HTIC an initial location to start writing the empty value into the hash table (i.e., location 0000h to initialize the entire table). Then the host CPU must write the EREG with the value that has been selected to represent an empty location in the key table. Once the EREG has been written, this will trigger the state machine to start writing the value of the empty register into the hash table. The HTIC stays in a loop incrementing the address pointer by one and writing the value of the EREG into the table until Carry_out_15 equals one, thereby initializing the entire table.

The user must write the most significant word of the key value to KMR1 then write the least significant word to

KMR2. Once the KMR2 has been written, the chip will start processing the given key.

Next, the PLA will gate off the input pads and begin performing the hash function. The hash function takes 4 machine states to load the HREG and DREG. During the first state, KMR1 is gated onto the internal data bus and the hashing network writes the odd bits of this register into the lower 8 bits of the HREG. During the second state of this operation, the contents of KMR2 are gated onto the bus and loaded into the upper 8 bits of the HREG. This process is repeated to load the DREG with the even bits of the key value.

Once the chip is loaded, the first address that will be pointed to by the HTIC will consist of a concatenation of b31,b29,b27,b25.....b5,b3,b1 from the original key. At this point we obtain the first address pointer by placing the contents of the HREG onto the address pointer lines (AP0-AP15) and strobing the Enable Hash Table (EHT) line active.

The key portion of this location is examined by the 16-bit compatator to test for equality. Once the contents of the key location have been determined, there are three possible cases that may occur. The location is either equal, not equal or empty. If the location is equal, this

means the HTIC has found the location that it is looking for. If the location is not equal, the HTIC will perform a rehash function, look into the table again and repeat this process until the RESET line is strobed low. Thereby, breaking the PLA out of the rehash loop and returning the HTIC to state zero. If the location is empty, then the HTIC will interrupt the host CPU and tell the it that it has found an empty location. When this occurs, the host CPU will determine what action must be taken. This is detailed in the "System application for HTIC" on page 56.

The rehash function is accomplished by adding the DREG to the HREG and placing the result back into the HREG. The ALU (Arithmetic Logic Unit) that is used is a 16-bit carry chain adder and the overflow is thrown away.

Limitations and Restrictions

The system's limitation is mainly in the speed at which the chip can function. The faster the chip can run the better the system's efficiency will be.

The number of probes into the table must be limited by some MSI logic around the HTIC in order to break it out of

the loop that the state machine enters once rehashing begins. This MSI (Medium Scale Integration) logic can consist of a binary counter and an AND gate. When the counter overflows, it will activate an interrupt line to tell the CPU that it has finished. Since the Status lines will read "Not Found" the CPU will simply think the entry is not present in the hash table. This is not to be confused with the case of writing data to the hash table. If data is to be written, there must be an empty location or the entry must be found within the specified number of probes.

Problems encountered

There were several things that could have been done better. There were a great deal of chip design errors. Many of these are attributed to the lack of complete simulation and modeling done before the actual layout of the device began. There was a sufficient amount of simulation for the bit slices of the device and for the complete data path. However, the major downfall was in the lack of simulation of the complete device; that is the data path, the PLA, and the I/O pads together. Simulation proved to be very valuable in the places where it was used and at

the same time hurt the design effort in the places that it was not used.

There were also problems in that several things had to be designed into the chip as the design effort progressed. This was due to an incomplete design specification at the begining of the design effort. During this project, the specifications, modeling and layout of the device all took place concurrently. In actuality, the specification should have been completed first, followed by the design being modeled. In modeling the design, any updates to the specification found from modeling should be incorporated into the specifications. Once the model works and the specifications are up to date, then the actual layout should begin.

At the end of the layout effort a problem was encountered. This problem is known as simultaneous switching. It occurs when a device has a large percentage of its outputs changing from a logical 1 to a logical 0. If too many outputs change at the same time the chip can not sink all the current from the I/O pads. When this occurs the current across the ground wire induces a large voltage and raises the voltage level of ground as high as +3.5V to +4.0V thereby actually shutting down the chip and causing it to fail. The way to fix this problem is to use more than one ground pin to distribute the current surge. This

will reduce the voltage induced across the ground wires. Therefore, the CS0 pin was tied to the ground rail and its input pad was changed to a second ground pin in order to distribute the current flow inside the chip.

There are three known problems in the current chip. One is that the PLA only checks the high word when it is testing for equality to the empty value. The second is that the IC needs external logic to break the device out of its rehashing loop. Third, the LSB of the DREG has power connected to it. None are significant problems and could be easily removed in the next release of this device if one were to be designed.

The simulation speed of the device was evaluated to be 2.5 Mhz before the device began to fail. This speed is not fast enough to show any great benefit from using this device. However, if the speed were say 100 Mhz (10 times the speed of the CPU), then even at very high table densities the store or retrieve operations would be done very quickly. In order to improve the speed of this device, it would have to be redesigned in CMOS and use smaller demensions, maybe 1 or 2 microns, for the chip layout.

TEST PLAN AND STATISTICAL DATA

The following sections define the specifications for the test plan and actual procedures that were used in the testing of this device. In the following test descriptions the word "will" implies that this is a logical state or sequence leading to the test condition. The word "should" implies that if the circuit behaves as the statement describes, then the requirement is fulfilled. All simulation requirements were fulfilled before the HTIC was released for fabrication.

Simulation Tests and Requirements

Test Number : 1

Requirement : The HTIC's state machine must return to state 0, the idle state, following a reset.

Description : The simulation will be started and the RESET line will be set to a logical 0 (its active state). The state machine should go to state 0 and stay there.

Test Number : 2

Requirement : The HTIC's KMR1 can be loaded by using the L0 & L1 control lines.

Description : The RESET line will be enabled and the HTIC will be set to state 0. The chip will be selected and the R/W line will be set low. Data will be placed on the CPU Data lines. L0 & L1 will be set low to load KMR1. The internal register will be monitored and the data should appear one (1) clock cycle later.

Test Number : 3

Requirement : The loading of KMR2 will start the IC into its hashing algorithm and probing the hash table.

Description : The RESET line will be enabled and the HTIC will be set to state 0. The chip will be selected and data will be written to KMR2. The state machine should go to state 4 and perform the hash function on the data. Once the hash function is done, the chip should place a valid address on the AP0-AP15 lines and enable the hash table.

Test Number : 4

Requirement : The HTIC will perform the proper hash function.

Description : There will be several cases run for this test. In every case, the first address produced by the HTIC should be all the odd bits in the 32 bit key given to the HTIC. Every address thereafter should be the previous address plus the even bits with the LSB of the offset set to '1'. This test will also verify that the ALU is functioning properly.

Test Number : 5

Requirement : If a key is not found within a specific number of probes, the state machine shall abort the search via an external RESET sourced by the GLUE logic.

Description : The KMR1 and KMR2 will be loaded. The HTIC will be allowed to function for several rehash cycles. Then the RESET line will be pulled down and the state machine should go to the state 0, the idle state. Status lines 0 and 1 should read S0 = 0, S1 = 0.

Test Number : 6

Requirement : If the Key is found the rehash algorithm is stopped and the HTIC notifies the host CPU via the interrupt line that the HTIC is done

and the status lines have the proper status.

Description : The KMR1 and KMR2 will be loaded and the HTIC will be allowed to calculate its first address. Next, data will be placed on the Hash Table Data lines that is equal to the contents of KMR1 and on the next cycle the data equaling the contents KMR2 will be placed on the Hash Table Data lines. At this point, the HTIC should think that it has found the location that it is looking for. The HTIC should interrupt the host CPU and return a status of found (S0 = 0, S1 = 1).

Test Number : 7

Requirement : When an empty location is probed the HTIC will stop processing and inform the host CPU that it has encountered an empty location.

Description : The KMR1 and KMR2 will be loaded and the HTIC will be allowed to function. At the time that the HTIC is making its first probe into the hash table, data representing the programmed 'Empty Value' will be placed on the data bus. The HTIC should think that it has encountered and empty location and report this to the host CPU via interrupting it and posting 'Location Empty' status on the status line (S0 = 1, S1 = 0).

Test Number : 8

Requirement : Load the Empty register and initialize the entire hash table.

Description : The KMR1 will be loaded with 0000h as the starting address for the hash processor to begin initializing. Then the EREG will be written with 8000h. The chip will be allowed to function and it should write all 64K locations with the value stored in the EREG and interrupt the host CPU when it has finished.

Programs Written

There have been two programs written to simulate this device. One written in 'C' which simulates the hash table processor functioning in a system. The source code for this can be found in appendix B . The second program is a model of the HTIC and was simulated on the RITCV computer system at RIT before the chip layout was started. The layout of the chip was simulated to insure that it matches the original logic of the circuit.

Statistical anlysis of the hash function.

The program written in C that simulates the hash function shows that it is a good hash function. The program simulates the hash processor trying to store 64K keys into the hash table. Some limitations were put on the simulation so that it could finish running. If the entry was already in the table then the second appearance of the key was thrown out of the data sample and the first entry was left in the table. Once the table reaches 98% full the simulation is ended.

Samples were taken every 1K entries that were put into

the table. Listed below are the results from one of the simulation runs. The simulation was run with six (6) different sets of data and all results were consistant. The input data for these simulations was generated from a random character string generator (Appendix C for source code). All bit patterns are considered legal except for the string "****" which is reserved to represent the empty string or location. The collision density table shows the collision distribution for each 1K set of table entries. The table reads as follows: In the first 1K table entries, 997 keys were placed into the table with no collisions, 26 entries had 1 collision and 1 entry had 2 collisions.

```
Initializing table, please wait .... Done.

Total collisions = 28 , # of entries = 1024
 1.5625% full, average # of collisions = 0.0273.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     997  26   1   0   0   0   0   0   0   0   0
Maximum collisions = 3

Total collisions = 104 , # of entries = 2048
 3.1250% full, average # of collisions = 0.0508.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     958  57   8   1   0   0   0   0   0   0   0
Maximum collisions = 4

Total collisions = 257 , # of entries = 3072
 4.6876% full, average # of collisions = 0.0837.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     891 115  17   0   1   0   0   0   0   0   0

Total collisions = 449 , # of entries = 4096
 6.2501% full, average # of collisions = 0.1096.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     852 155  14   3   0   0   0   0   0   0   0

Total collisions = 677 , # of entries = 5120
 7.8126% full, average # of collisions = 0.1322.
```

```
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     825 174  22   2   1   0   0   0   0   0   0

Total collisions = 994 , # of entries = 6144
 9.3751% full, average # of collisions = 0.1618.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     750 237  32   4   1   0   0   0   0   0   0
Maximum collisions = 5

Total collisions = 1405 , # of entries = 7168
 10.9377% full, average # of collisions = 0.1960.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     683 282  50   8   0   1   0   0   0   0   0

Total collisions = 1845 , # of entries = 8192
 12.5002% full, average # of collisions = 0.2252.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     689 252  65  14   4   0   0   0   0   0   0
Maximum collisions = 6

Total collisions = 2399 , # of entries = 9216
 14.0627% full, average # of collisions = 0.2603.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     625 288  80  21   8   1   1   0   0   0   0
Maximum collisions = 7

Total collisions = 3105 , # of entries = 10240
 15.6252% full, average # of collisions = 0.3032.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     551 319 103  35  10   3   0   3   0   0   0
Maximum collisions = 10

Total collisions = 3971 , # of entries = 11264
 17.1878% full, average # of collisions = 0.3525.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     462 352 147  46  11   3   1   1   0   0   1
Maximum collisions = 13

Total collisions = 4903 , # of entries = 12288
 18.7503% full, average # of collisions = 0.3990.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     447 348 156  47  17   3   1   3   1   0   1

Total collisions = 5645 , # of entries = 13312
 20.3128% full, average # of collisions = 0.4241.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     561 287 114  37  15   6   2   2   0   0   0

Total collisions = 6585 , # of entries = 14336
 21.8753% full, average # of collisions = 0.4593.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     477 306 152  60  13   9   3   1   1   1   1
```

```
Total collisions = 7495 , # of entries = 15360
 23.4379% full, average # of collisions = 0.4880.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     480 313 146  56  19   4   4   1   0   0   1

Total collisions = 8710 , # of entries = 16384
 25.0004% full, average # of collisions = 0.5316.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     414 297 152  92  37  18   8   2   2   1   1
Maximum collisions = 16

Total collisions = 9820 , # of entries = 17408
 26.5629% full, average # of collisions = 0.5641.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     425 292 201  70  20   3   5   2   2   0   4

Total collisions = 11192 , # of entries = 18432
 28.1254% full, average # of collisions = 0.6072.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     351 295 212  95  35  15  10   6   2   1   2
Maximum collisions = 17
Maximum collisions = 18

Total collisions = 12990 , # of entries = 19456
 29.6880% full, average # of collisions = 0.6677.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     266 287 230 121  53  25  15   8   5   4  10

Total collisions = 14597 , # of entries = 20480
 31.2505% full, average # of collisions = 0.7127.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     332 269 206 107  50  27  12   8   4   3   6

Total collisions = 16081 , # of entries = 21504
 32.8130% full, average # of collisions = 0.7478.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     445 204 170  91  52  16  13  12   5   6  10
Maximum collisions = 27

Total collisions = 17837 , # of entries = 22528
 34.3755% full, average # of collisions = 0.7918.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     262 297 241 103  51  33  17  10   2   0   8

Total collisions = 19892 , # of entries = 23552
 35.9380% full, average # of collisions = 0.8446.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     257 247 210 142  79  35  23   5   6   6  14
Maximum collisions = 36

Total collisions = 22165 , # of entries = 24576
 37.5006% full, average # of collisions = 0.9019.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     279 208 232 123  62  38  20  17  12  10  23
```

```
Total collisions = 24330 , # of entries = 25600
 39.0631% full, average # of collisions = 0.9504.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     299 238 179 114  82  33  27  14   3  12  23
Maximum collisions = 66

Total collisions = 26957 , # of entries = 26624
 40.6256% full, average # of collisions = 1.0125.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     167 215 227 179  97  55  28  19   7   9  21

Total collisions = 29505 , # of entries = 27648
 42.1881% full, average # of collisions = 1.0672.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     177 223 260 162  84  40  24   8  15   7  24

Total collisions = 32132 , # of entries = 28672
 43.7507% full, average # of collisions = 1.1207.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     192 189 245 167 104  51  29   9   9   6  23

Total collisions = 33952 , # of entries = 29696
 45.3132% full, average # of collisions = 1.1433.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     390 216 149 107  70  36  14  10   8   6  18

Total collisions = 37422 , # of entries = 30720
 46.8757% full, average # of collisions = 1.2182.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     168 143 205 146 102  81  55  37  15  13  59

Total collisions = 41360 , # of entries = 31744
 48.4382% full, average # of collisions = 1.3029.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                      99 130 215 182 132  84  50  33  19  14  66

Total collisions = 45327 , # of entries = 32768
 50.0008% full, average # of collisions = 1.3833.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                     154 128 174 156 133  71  51  35  30  22  70

Total collisions = 49067 , # of entries = 33792
 51.5633% full, average # of collisions = 1.4520.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     175 144 156 162 126  72  59  37  15  12  66

Total collisions = 52572 , # of entries = 34816
 53.1258% full, average # of collisions = 1.5100.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     220 128 151 163 122  63  43  32  26  17  59
```

```
Maximum collisions = 82
Total collisions = 57597 , # of entries = 35840
 54.6883% full, average # of collisions = 1.6071.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      96  95 152 164 149  96  68  40  33  24 107

Total collisions = 62514 , # of entries = 36864
 56.2509% full, average # of collisions = 1.6958.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     104  84 140 164 135 122  70  51  22  17 115
Maximum collisions = 181

Total collisions = 67573 , # of entries = 37888
 57.8134% full, average # of collisions = 1.7835.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     184 106 127 134 120  99  53  47  27  24 103

Total collisions = 72699 , # of entries = 38912
 59.3759% full, average # of collisions = 1.8683.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     149 101 124 143 134  96  73  37  37  20 110

Total collisions = 78514 , # of entries = 39936
 60.9384% full, average # of collisions = 1.9660.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     121  80 126 128 125 112  73  52  43  31 133
Maximum collisions = 405

Total collisions = 84915 , # of entries = 40960
 62.5010% full, average # of collisions = 2.0731.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     102  91 121 135 132  95  67  52  40  37 152

Total collisions = 91756 , # of entries = 41984
 64.0635% full, average # of collisions = 2.1855.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      99  56 112 150 136  98  86  49  34  30 174

Total collisions = 99574 , # of entries = 43008
 65.6260% full, average # of collisions = 2.3152.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      94  63  85 107 108 109  86  72  48  33 219

Total collisions = 108659 , # of entries = 44032
 67.1885% full, average # of collisions = 2.4677.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      22  32  63 147 140 124  90  67  54  56 229

Total collisions = 116896 , # of entries = 45056
 68.7511% full, average # of collisions = 2.5945.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                     124  84  89  77 117  99  69  64  43  35 223
```

```
Total collisions = 125384 , # of entries = 46080
 70.3136% full, average # of collisions = 2.7210.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                      61  52  70 109 126  89  89  71  58  50 249

Total collisions = 134434 , # of entries = 47104
 71.8761% full, average # of collisions = 2.8540.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                      98  65  66  81 100  88  91  71  57  55 252
Maximum collisions = 414

Total collisions = 146360 , # of entries = 48128
 73.4386% full, average # of collisions = 3.0411.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      46  37  41  56  92 112  81  78  61  61 359
Maximum collisions = 474

Total collisions = 158022 , # of entries = 49152
 75.0011% full, average # of collisions = 3.2150.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      34  44  58  91 102 107  94  74  61  42 317
Maximum collisions = 1114

Total collisions = 169432 , # of entries = 50176
 76.5637% full, average # of collisions = 3.3768.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      66  65  71  92  97  87  89  70  68  56 263

Total collisions = 184332 , # of entries = 51200
 78.1262% full, average # of collisions = 3.6002.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                      39  28  29  36  78  68  79  84  61  57 465

Total collisions = 195029 , # of entries = 52224
 79.6887% full, average # of collisions = 3.7345.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                      40  56  45  78  87 100  90  97  70  47 314

Total collisions = 210482 , # of entries = 53248
 81.2512% full, average # of collisions = 3.9529.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      28  15  37  47  62  65  83  80  75  59 473

Total collisions = 225298 , # of entries = 54272
 82.8138% full, average # of collisions = 4.1513.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      55  48  54  51  59  63  58  80  77  63 416

Total collisions = 238403 , # of entries = 55296
 84.3763% full, average # of collisions = 4.3114.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      43  46  48  53  62  74  70  60  71  71 426
```

```
Total collisions = 251929 , # of entries = 56320
 85.9388% full, average # of collisions = 4.4732.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      46  53  47  59  65  39  79  73  61  55 447
Maximum collisions = 4122

Total collisions = 272431 , # of entries = 57344
 87.5013% full, average # of collisions = 4.7508.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      45  62  35  41  47  47  46  48  63  53 537

Total collisions = 293675 , # of entries = 58368
 89.0639% full, average # of collisions = 5.0314.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      21  18  20  34  39  44  47  58  40  67 636

Total collisions = 318086 , # of entries = 59392
 90.6264% full, average # of collisions = 5.3557.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      20  10  14  12  20  35  44  60  37  49 723

Total collisions = 344455 , # of entries = 60416
 92.1889% full, average # of collisions = 5.7014.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      29  13  22  19  24  22  23  37  45  44  746

Total collisions = 373176 , # of entries = 61440
 93.7514% full, average # of collisions = 6.0738.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      17  19  20  21  17  22  28  36  35  48 761
Maximum collisions = 10021

Total collisions = 406805 , # of entries = 62464
 95.3140% full, average # of collisions = 6.5126.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                      14  14  11  23  16  30  31  27  28  43 787

Total collisions = 464504 , # of entries = 63488
 96.8765% full, average # of collisions = 7.3164.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10-
                       2   8   7  10  12  18  26  17  26  34 864

Total collisions = 529266 , # of entries = 64512
 98.4390% full, average # of collisions = 8.2041.
Collision density -->  0   1   2   3   4   5   6   7   8   9   10+
                       0   2   2   3   6  12   7  10  14  13 955

Program completed normally.
```

CONCLUSION

Given more time I would actually implement the device into the proposed system for testing and evaluation. Simulation shows that at high table densities the average number of collisions is still small. Therefore, even as slow as the chip is now, we can expect fairly good results from it.

On the next pass the "known bugs" would be fixed and the layout would be done in CMOS to improve the speed of the device. As stated, the maximum speed at which the device simulates before failure is 2.5 Mhz. If that could be improved to a speed of say 50Mhz or 100Mhz, then the hash processor would have an apparent cycle time of one CPU cycle or less at lower table densities. At higher table densities the device will take longer to function.

The following paragraphs describe a host system configuration that the hash processor can be placed into for testing and evaluation.

System application for HTIC

The overall system shown in Figure 7 on page 60 describes a possible system configuration. The PC is used as a terminal to drive the 8086 μP. Data is transferred over a serial interface to an ACIA (Asynchronous Communication Interface Adaptor) on board the μP system. The μP system is shown in Figure 8 on page 61. The HTIC is memory mapped to location A020 - A023 hex. Hence, the host CPU uses the HTIC as a coprocessor and the instructions to use the device are memory mapped reads and writes. There is some additional code that must be used in order to interpret the return codes of the HTIC. A complete memory map of the system is shown in Figure 9 on page 62

The following is a sequence of these samples entered from the terminal. However, these "calls" can come from a compiler or an assembler in the same way when the program needs data relating to a key. The instruction set for the CPU is find_key(KEY) and write_key(KEY). The find_key() instruction calls a small assembler macro which will return the value of S0 and S1 to an internal register by doing a load accumulator from memory after the CPU has received an interrupt. The following cases are examples of how the system should operate.

1. VAR1 = 10: This case will write a new value into the hash table. Assume that the table is initially empty and this is the first request to place data in the table.

2. VAR1 = 7: This case will overwrite the value of 10 that was written in case 1.

3. VAR1 = ?: This case will retrieve an existing value from the table. The value that will be returned is 7 because case 2 replaced the first value of 10.

4. VAR2 = ?: This case will return with a status of not found in the table. This is because the key "VAR2" has not been written to the table yet.

In the first case let us assume that the system has just been initialized and the table is completely empty. We first call find_key(VAR1). This macro performs two writes to the HTIC. The first write is "VA" to KMR1 and the second write is "R1" to KMR2. Once the KMR2 has been written, the HTIC will execute the hash function on the key and begin its search into the table. Since the table is empty, the first location that the HTIC probes will interrupt the μP and return a status of "Location Empty". At this time the write_key(VAR1) macro is called and the data value associated with VAR1 can be written into the hash table's

memory. The write_key() macro will then select the HTIC with L0 and L1 at logical 1's, thereby telling the HTIC to write the key into the hash table. On the next clock cycle, the μP will place the data portion, in this case 10, on its data bus. This will be written into the data table at the same time the HTIC is writing the most significant byte, VA, into the hash table. The μP will then wait for an interrupt from the HTIC while it is writing the least significant byte into the hash table. After the HTIC has finished writing to the data table, it will send an interrupt to the processor and wait to process the next request.

The second case will overwrite an existing value. In this case, the value 10 in location f(VAR1) will be overwritten. The HTIC will be loaded in the same format as case 1 and execute the hash function. However, this time the HTIC will return a status of "Found" on the first probe and the host CPU will read the value from the data table latch. The LOS will determine that the value read from the table is different from the value passed to the μP from the user and the new value will be written into the data table. The host CPU will assume that the user wants to do a variable update. A message of "OK, VAR1 UPDATED." will be returned to the terminal.

In the third case, the system will retrieve an already

existing value from the data table. The LOS will interpret the question mark as a read only operation and attempt to find the specified key value in the hash table. Since we know that VAR1 already exists in the table, the HTIC will find the key in the table and interrupt the μP in the same way as in the previous case. However, since the LOS knows that this is a read only operation the system will read the data latch and return value found for the data location probed. A message of "VAR1 = 07" will be returned to the terminal.

The fourth case will try to read a value from the data table when there is no entry in the key table for the data. The HTIC will be loaded with the data and allowed to function. It will rehash until the binary counter overflows and interrupts the μP. At this point the status lines will contain "Not Found" and the μP will reset the PLA to state zero to break out of the rehash loop. Since the reset is a state only reset, the contents of all internal registers are preserved whenever the reset line is activated.

Configuration Diagrams

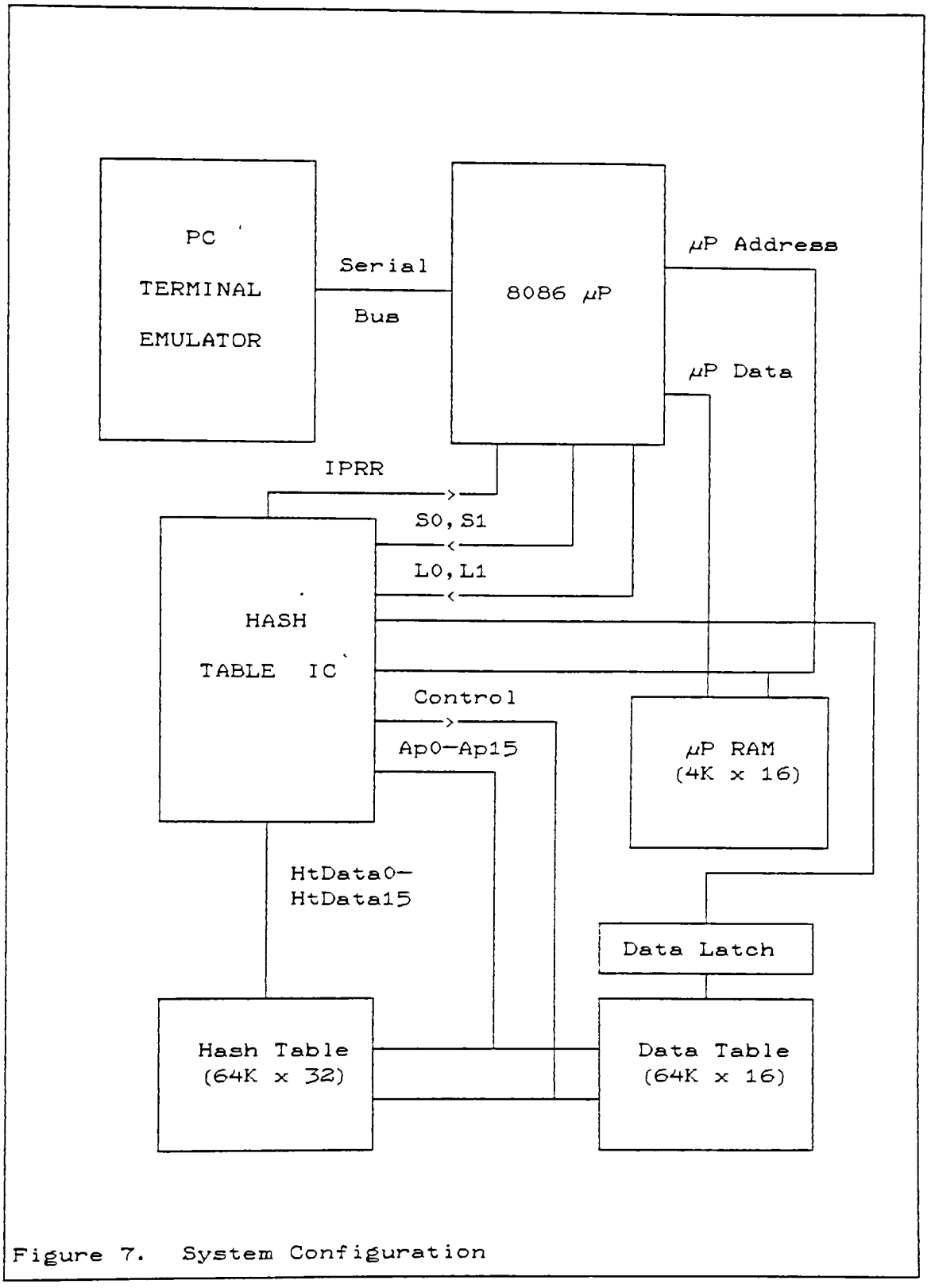


Figure 7. System Configuration

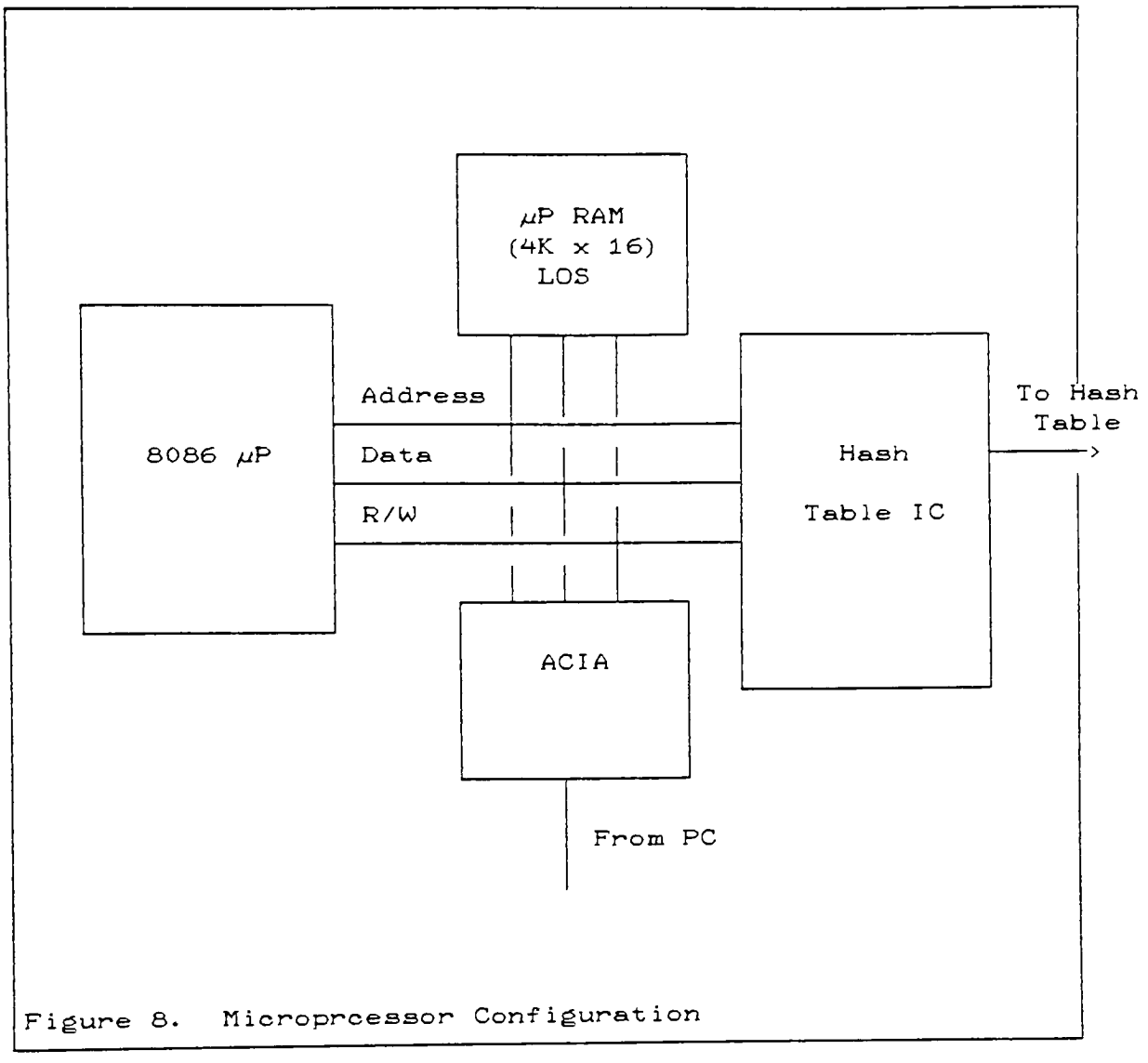


Figure 8. Microprcessor Configuration

Device	Address Range
LOS RAM	0000h 3FFFh
ACIA	A000h A003h
HASH IC	A020h A023h
DATA TABLE LATCH (Read) (Write)	A024h A025h

Figure 9. Microprocessor Memory Map

Software Design Specifications

The following sections define the specifications of the LOS that needs to reside in the microprocessor and the terminal emulation program that needs to reside in the PC.

Terminal Emulator Program Specifications

The terminal emulation program will handle the communications between the microprocessor and the PC. The terminal emulator will send the μP a data packet consisting of the four characters in the key and the two bytes of data. If all four characters are not specified, then the emulator program will append the appropriate number of blanks. The equal sign is implied and it is not necessary to send it to the μP system.

Local Operation System Specifications

The LOS has been broken into the following basic functions.

- To accept and receive data from the terminal.
- To interpret the data that has been sent and program the HTIC accordingly.
- To handle the interrupts that are sourced by the HTIC and process the data accordingly.
- Once the data has been processed, return status and data to the terminal emulation program.

GLOSSARY OF TERMS AND ACRONYMS

VLSI Very Large Scale Integration

IC Integrated Circuit

CPU Central Processing Unit. This is the main processing element in a computer system. This term will be used interchangeably with [P. (See Below for [P)

PLA Programable Logic Array. A electronic circuit that controls the functionality of the HTIC. Also known as a state machine.

KMR1 Key Memory Register #1. This register holds the first two characters of the varible name being processed.

KMR2 Key Memory Register #2

DREG Delta Register

EREG Empty Register

HREG Hash Register

ALU Arithmetic Logic Unit

μP Microprocessor

ASCII Americian Standard Code for Information Interchange.

PIA Peripheral Interface Adaptor

ACIA Asynchronus Communications Interface Adaptor. This device is used to send and receive data over a serial transmission line. The CPU will select this chip and pass it data 8 bits at a time. The ACIA will send this data out over its transmission line one bit at a time. The ACIA also adds extra information to each datum transmitted - start bits, stop bits and parity. All this extra information is user programmable from the CPU.

RAM Random Access Memory. This is where the CPU stores the instructions and data for operation. This memory can be written to or read from.

4K This term K is often used to denote 1024 bits or words. Therefore, 4K is 4 x 1024 or 4096 words.

LOS Local Operation System. This is a term used to

describe a set of instructions that the CPU executes while the system is in use.

DIP Dual Inline Package

NMOS N-type Metal Oxide Semiconductor

CMOS Complementary Metal Oxide Semiconductor

HTIC Hash Table Integrated Circuit

BIBLIOGRAPHY

- Content-Addressable Memories, Springer-Verlag Berlin Heidelberg 1980, By T. Kohonen
- Associative Memory, Springer-Verlag Berlin Heilelberg 1977, By T. Kohonen
- The comparative cost of associative memory, By R. M. Lea
- Searching and sorting, The Art of computer programming Vol. 3, Addison Wesley Publishing Co. 1973, By Donald E. Knuth
- Operational characteristics of a hardware-based pattern matcher , By Rodger L. Haskin and Lee. A. Hollar.

APPENDIX A: SOURCE CODE TO GENERATE THE PLA.

```
PLA Hash_IC_Controller;
VAR
    Reset,L0,L1,R_W,CS1,CS2,CS3,Equal,Cout15 : CLOCKED_INPUT;
    s: ARRAY[0..4] OF STATE;
    HtData,CPUData,Load,CKMR1,LKMR1,CKMR2,LKMR2,
    CEmpty,LEmpty,LHash,LDelta,Right,Left,Add,
    LResult,S0,S1,PadOut,HW,Write_Hash,EHT,IPRR : CLOCKED_OUTPUT;
    INORDER  Reset,L0,L1,R_W,CS1,CS2,CS3,Equal,Cout15,s;
    OUTORDER s,S0,S1,HW,Write_Hash,EHT,IPRR,LResult,Add,Right,Left,
             LDelta,LHash,LEmpty,CEmpty,LKMR2,CKMR2,LKMR1,
             CKMR1,Load,CPUData,HtData,PadOut;
BEGIN
IF ~Reset THEN
BEGIN
   CASE
     s = 0 :
          BEGIN
         IF ~( ~CS1 and CS2 and CS3 ) THEN
               BEGIN
                 Write_Hash := 1;
                 IPRR := 1;   { Keep interrupt line high }
                 s := 0;
               END;
         IF ~CS1 and CS2 and CS3  THEN
          BEGIN     { This state will determine the proper action  }
                    , { to take according to the user inputs L0 & L1 }
            IF ~L0 and ~L1 and ~R_W THEN
              BEGIN
                 Write_Hash := 1;
                 Load := 1;
                 LKMR1 := 1;
                 CPUData := 1;
                 IPRR := 1;
                 s := 0;
              END;
            IF ~L0 and L1 and ~R_W THEN
              BEGIN
                 Write_Hash := 1;
                 Load := 1;
                 LKMR2 := 1;
                 CPUData := 1;
                 IPRR := 1;
                 s := 4;
              END;
            IF L0 and ~L1 and ~R_W THEN
              BEGIN
                 Write_Hash := 1;
                 Load := 1;
                 LEmpty := 1;
```

```
                CPUData := 1;
                IPRR := 1;
                s := 16;
              END;
            IF L0 and L1 and ~R_W THEN
              BEGIN
                Write_Hash := 1;
                IPRR := 1;
                s := 2;
              END;
        END    { End the s0 if statment }
        ELSE s := 0;
    END;   { End the entire s0 }
s = 1 :
      BEGIN
        Write_Hash := 1;
        IPRR := 1;
      END;
s = 2 :
      BEGIN          { This state will write a new key into the table.}
       HW := 1;      { Enable the high word block of the hash table.  }
       Write_Hash := 0;    { Write to Hash Table.                      }
       EHT := 1;     { data value associated with the new key.       }
                     { Enable Hash Table (both Key and Data tables)   }
       CKMR1 := 1;
       Load := 1;
       HtData := 1;
       PadOut := 1;
       IPRR := 1;
       s := 3;
      END;
s = 3 :
      BEGIN
       HW := 0;
       Write_Hash := 0;    { Write to Hash Table. (This Is inverted  }
                           { before the output pad)                  }
       EHT := 1;
       CKMR2 := 1;
       Load := 1;
       HtData := 1;
       PadOut := 1;
       IPRR := 1;
       s := 0;
      END;
s = 4 :
      BEGIN       { This state will load the initial hash into the }
                  { lower 8 bits of hash register. (Ap0 - Ap7)      }
        Write_Hash := 1;
        CKMR1 := 1;
        LHash := 1;
        Right := 1;
        IPRR := 1;
        s := 5;
```

```
      END;
s = 5 :
      BEGIN      { This state will load the initial hash into the }
                 { upper 8 bits of hash register. (Ap8 - Ap15)    }
        Write_Hash := 1;
        CKMR2 := 1;
        LHash := 1;
        Left := 1;
        IPRR := 1;
        s := 6;
      END;
s = 6 :
      BEGIN      { This state will load the initial hash into the }
                 { lower 8 bits of delta register.                }
        Write_Hash := 1;
        CKMR1 := 1;
        LDelta := 1;
        Right := 1;
        IPRR := 1;
        s := 7;
      END;
s = 7 :
      BEGIN      { This state will load the initial hash into the }
                 { upper 8 bits of delta register.                }
        Write_Hash := 1;
        CKMR2 := 1;
        LDelta := 1;
        Left := 1;
        IPRR := 1;
        s := 8;
      END;
s = 8 :
      BEGIN
       CKMR1 := 1;        { Set up test for next cycle }
       HtData := 1;
       HW := 1;
       EHT := 1;
       IPRR := 1;
       Write_Hash := 1;
       s := 9;
      END;
s = 9 :
      BEGIN
       IF Equal THEN
                  BEGIN
                    CKMR2 := 1;      { Set up for next test, KMR2 := KEY }
                    HtData := 1;     { Access the Hash Table data }
                    HW := 0;         { Set low word of hash table active }
                    EHT := 1;        { Enable Hash Table }
                    IPRR := 1;
                    Write_Hash := 1; { Read from hash table }
                    s := 10;         { STATUS : KMR1 Equal, try KMR2 }
                  END;
```

```
        IF ~Equal THEN
                    BEGIN
                      CEmpty := 1;
                      HtData := 1;
                      HW := 1;
                      EHT := 1;
                      IPRR := 1;
                      Write_Hash := 1; { Read from hash table }
                      s := 11;   { STATUS : KMR1 Not Equal, try empty }
                    END;
      END;
s = 10 :
      BEGIN
       IF Equal  THEN
                    BEGIN
                       IPRR := 1;
                       Write_Hash := 1;
                       s := 14;    { STATUS : Found }
                    END;
       IF ~Equal THEN
                    BEGIN
                      CEmpty := 1;    { Set up for test on next cycle }
                      HtData := 1;
                      HW := 0;
                      Write_Hash := 1; { Read from hash table }
                      EHT := 1;
                      IPRR := 1;
                      s := 11; { STATUS : Not Equal, try Empty }
                    END;
      END;
s = 11 :
      BEGIN
       IF Equal THEN
                   BEGIN
                      Write_Hash := 1;
                      IPRR := 1;
                      s := 15    { STATUS : Location Empty }
                   END;
       IF ~Equal THEN
                    BEGIN
                       Write_Hash := 1;
                       IPRR := 1;
                       s := 12;   { STATUS : Not Empty and Not Equal }
                    END;                {    goto rehash.    }
      END;
s = 12 :
      BEGIN
       Write_Hash := 1;
       Add := 1;    { Compute the next hash }
       IPRR := 1;
       s := 13;
      END;
s = 13 :
```

```
        BEGIN
         Write_Hash := 1;
         LResult :=1;   { Load Result and re-probe table }
         IPRR := 1;
         s := 8;
        END;
    s = 14 :
        BEGIN
         Write_Hash := 1;
         IPRR := 0;     { Send Interrupt to Precessor. Result Ready }
         S0 := 0;       { RETURN STATUS : 0 1 (Found Key) }
         S1 := 1;
         s := 0;        { Go wait for more input }
        END;
    s = 15 :
        BEGIN
         Write_Hash := 1;
         IPRR := 0;     { Send Interrupt to Precessor, Result Ready }
         S0 := 1;       { RETURN STATUS : 1 0 (Location Empty) }
         S1 := 0;
         s := 0;        { Go wait for more input }
        END;
    s = 16 :
        BEGIN
          Write_Hash := 1;
          CKMR1 := 1;   { States 16 .. 19 will initialize the hash }
          LHash := 1;   { and delta to equal 0. This will set up   }
          Right := 1;   { the chip to write the EMPTY KEY VALUE to }
          IPRR := 1;    { all 64K memory locations.                }
          s := 17;
        END;
    s = 17 :
        BEGIN
           Write_Hash := 1;
           CKMR1 := 1;
           LHash := 1;
           Left := 1;
           IPRR := 1;
           s := 18;
        END;
    s = 18 :
        BEGIN
           Write_Hash := 1;
           CKMR1 := 1;
           LDelta := 1;
           Right := 1;
           IPRR := 1;
           s := 19;
        END;
    s = 19 :
        BEGIN
           Write_Hash := 1;
           CKMR1 := 1;
```

```
            LDelta := 1;
            Left := 1;
            IPRR := 1;
            s := 20;
         END;
      s = 20 :
         BEGIN
            IPRR := 1;
            HW := 1;                { Enable the high word. }
            Write_Hash := 0;        { Enable Hash Write.    }
            EHT := 1;               { Enable Hash Table.    }
            PadOut := 1;            { Change pads to output. }
            Load := 1;
            HtData := 1;            { Enable the data pads }
            CEmpty := 1;            { Write the contents of the empty }
            s := 21;                { register into memory.           }
          END;
      s = 21 :
         BEGIN
            IPRR := 1;
            HW := 0;                { Enable the low word. }
            Write_Hash := 0;        { Enable Hash Write.   }
            EHT := 1;               { Enable Hash Table.   }
            PadOut := 1;            { Change pads to output.  }
            Load := 1;
            HtData := 1;            { Enable the data pads }
            CEmpty := 1;            { Write contents into memory. }
            s := 22;
         END;
      s = 22 :
         BEGIN
            Write_Hash := 1;
            Add := 1;               { Increment address pionter by 1 }
            IPRR := 1;              { AP := Cin1 + Hash + Delta       }
            s := 23;                { Cin = 1; Delta = 0; -->         }
         END;                      { AP := Hash + 1 to get incrmenter }
      s = 23 :
         BEGIN
            Write_Hash := 1;
            LResult := 1;           { Capture Result of addition }
            IPRR := 1;
            s := 24;
         END;
      s = 24 :
         BEGIN
            IF Cout15 THEN
                        BEGIN
                          Write_Hash := 1;
                          IPRR := 0;      { Send interrupt to CPU }
                          s := 0;         { Go wait for input.    }
                        END;
             IF ~Cout15 THEN
                        BEGIN
```

```
                Write_Hash := 1;
                IPRR := 1;          { Not done, do not interrupt }
                s := 20;            { Go write next location.    }
              END;
        END;  (* CASE *)
        END
        END.
```

APPENDIX B: SOURCE CODE FOR HASH FUNCTION SIMULATION.

```
 /*****************************************************************/
 /*                                                               */
 /*                Include the following files :                  */
 /*                                                               */
 /*****************************************************************/
#include <c:\include\stdio.h>
#include <c:\include\math.h>
#include <c:\include\ctype.h>
#include <c:\include\malloc.h>
#include <c:\include\string.h>
 /*****************************************************************/
 /*                                                               */
 /*           Decloration of constants used in the program.       */
 /*                                                               */
 /*****************************************************************/
#define TABLE_SIZE         65535
#define KEY_SIZE               4
#define BIT_MASK               1
#define HASH_MASK     0x0000FFFF
#define MASK_0001         0x0001
#define MASK_0002         0x0002
#define MASK_0004         0x0004
#define MASK_0008         0x0008
#define MASK_0010         0x0010
#define MASK_0020         0x0020
#define MASK_0040         0x0040
#define MASK_0080         0x0080
#define MASK_0100         0x0100
#define MASK_0200         0x0200
#define MASK_0400         0x0400
#define MASK_0800         0x0800
#define MASK_1000         0x1000
#define MASK_2000         0x2000
#define MASK_4000         0x4000
#define MASK_8000         0x8000
```

Line# Source Line Microsoft C Compiler Version 5.00

```
 1    /*   Program Name : Hash
 2     *   Written By   : Robert P. Ketrick
 3     *   Program Description :
 4
 5  This program is submitted in patrial fullfillment of the requirements
 6 stated in the Hash  Memory IC Thesis proposal. The intent of this code
 7 is to  gather statistical information on the hashing function  used in
 8 the integrated circuit designed in this thesis.
 9  The  data for this  program is generated from  "gendata.c" which will
10 generate ramdom strings of four (4) characters  that are used as keys.
11 This program will take those keys as input and attempt to find a place
12 in the hash table for each key until the table table exceeds 98% full.
13
14  In the first  case the entry is thrown out of the  sample and another
15 key is read. In the second case the program will terminate.
16     */
17
18  #include "hash.h"
19
20  /* These are the global variables to be used in the hash program */
21
22  unsigned char key[KEY_SIZE];
23  char table[TABLE_SIZE][4] , empty_string[4] ;
24  long int  hash, delta ;
25
26  main(argc, argv)
27  int argc;
28  char *argv[];
29  {
30
31  /* These are the local variables to be used in the hash program */
32
33      FILE *in, *out, *fopen();
34      int collision = 0, finished = 0 , count = 0;
35      int max_collisions = 0 ;
36      long int i ;
37      register j ;
38      unsigned long int entries = 0 ;
39      unsigned long int total_collisions = 0 ;
40      float average_hit = 0.0, percent_full = 0.0  ;
41      char *newkey ;
42
43      int coll_00 = 0,      /* These ints track the collision density */
44          coll_01 = 0,      /* of the hash function.                  */
45          coll_02 = 0,
46          coll_03 = 0,
47          coll_04 = 0,
48          coll_05 = 0,
49          coll_06 = 0,
```

```
50          coll_07 = 0,
51          coll_08 = 0,
52          coll_09 = 0,
53          coll_10 = 0;
54
55
56        if (argc == 1)    {
57
58             /* no args; print error and exit */
59
60             fprintf(stderr, "hash: must have file as input.\n\n");
61             fprintf(stderr, "usage : hash <infile> \n");
62
63       }  else  {
64
65              /* get the data file for processing */
66
67             if ((in = fopen(*++argv, "r")) == NULL) {
68                 fprintf(stderr, "hash: can't open %s\n", *argv);
69                 exit(1);
70
71             }  /* endif fopen() */
72
73       }  /* endif argc == 1 */
74
75      fprintf(stdout,"Initializing table, please wait .... ");
76
77
78     /*****************************************************************/
79     /*                                                               */
80     /*   The table is initialized to all '****' to show that the     */
81     /* table is empty.                                               */
82     /*                                                               */
83     /*****************************************************************/
84
85      for (i = 0; i <= TABLE_SIZE ; i++)
86        for ( j  = 0; j < 4; j++) table[i][j] = '*' ;
87
88      for ( i = 0; i < 4; i++) empty_string[i] = '*' ;
89
90      fprintf(stdout,"Done. \n");
91
92      while ((fread(key,5,1,in) > 0)  &&
93             (entries < (TABLE_SIZE - 1020)))  {
94
95             finished = collision = 0 ;
96             key[4] = '\0';
97             hashkey(key);
98
99             while ( finished == 0 )  {
100
101
102    /*****************************************************************/
```

```
103     /*                                                                  */
104     /*  Since we are only looking for  statistacal information on       */
105     /* the performance of the hash function we will discard but         */
106     /* notify when we encounter duplicate entries in the table.         */
107     /*                                                                  */
108     /********************************************************************/
109
110                 if ((j = comp(key)) == 1) {
111
112                     finished = 1 ;
113                     fprintf(stdout,
114                         "Duplicate entries in table for %s. \n",key);
115
116                 }   /* endif */
117
118
119     /********************************************************************/
120     /*                                                                  */
121     /*    If the pointer in the table points to a "empty_string"        */
122     /*  then add one to the number of entries in the table and          */
123     /*  allocate the space and copy the new "key" string into the       */
124     /*  table.                                                          */
125     /*                                                                  */
126     /********************************************************************/
127
128                 if ((j = comp(empty_string)) == 1) {
129
130                     for (i = 0 ; i < 4; i++)  table[hash][i] =  key[i]
131                     entries ++ ;
132                     count ++ ;
133                     finished = 1 ;
134
135       /*****************************************************************/
136       /*                                                               */
137       /*  Keep track of the collision density per 1K table entries.    */
138       /*                                                               */
139       /*****************************************************************/
140
141                       switch (collision) {
142                         case 0 :
143                             coll_00 ++ ;
144                           break;
145                         case 1 :
146                             coll_01 ++ ;
147                           break;
148                         case 2 :
149                             coll_02 ++ ;
150                           break;
151                         case 3 :
152                             coll_03 ++ ;
153                           break;
154                         case 4 :
155                             coll_04 ++ ;
```

```
156                         break;
157                       case 5 :
158                           coll_05 ++ ;
159                         break;
160                       case 6 :
161                           coll_06 ++ ;
162                         break;
163                       case 7 :
164                           coll_07 ++ ;
165                         break;
166                       case 8 :
167                           coll_08 ++ ;
168                         break;
169                       case 9 :
170                           coll_09 ++ ;
171                         break;
172                       default:
173                           coll_10 ++ ;
174                         break;
175                     } /* endswitch */
176
177               }  /* endif for strcmp() */
178
179     /*****************************************************************/
180     /*                                                               */
181     /*   This section will perform the re-hash function and record   */
182     /*  that a collision has taken place. Also, if the number of     */
183     /*  collisions exceed 100 then the entry is thrown out of the    */
184     /*  sample.                                                      */
185     /*                                                               */
186     /*****************************************************************/
187
188               if ((j = comp(key)) == 0) {
189
190
191                     /* Re-hash function for the chip */
192
193                   hash = (hash + delta + 1) & HASH_MASK ;
194
195                collision++ ;
196                finished = 0 ;
197
198                if (collision > max_collisions)  {
199
200                   max_collisions = collision ;
201                   fprintf(stdout,"Maximum collisions = %d \n",
202                                   collision);
203                }
204
205               }  /* end comp */
206
207          } /* endwhile not finished */
208
```

```
209             total_collisions += collision ;
210
211     /*************************************************************/
212     /*                                                           */
213     /*  Generate the statistical information for the hashing     */
214     /* function.                                                 */
215     /*                                                           */
216     /*************************************************************/
217
218           if ( count == 1024 )  {
219
220              fprintf(stdout,
221                  "\nTotal collisions = %ld \%, # of entries = %ld \n",
222                   total_collisions,entries);
223              fprintf(stderr,
224                  "\nTotal collisions = %ld \%, # of entries = %ld \n",
225                   total_collisions,entries);
226
227
228              count = 0 ;
229              average_hit = (float)total_collisions / (float)entries ;
230              percent_full = (float)entries / (float)TABLE_SIZE * 100.00
231
232              fprintf(stdout,
233                 " %03.4f full, ave. coll = %03.4f.\n",
234                  percent_full,average_hit);
235              fprintf(stderr,
236                 " %03.4f full, ave. coll = %03.4f.\n",
237                  percent_full,average_hit);
238
239              fprintf(stdout,
240                 "Collision density -->  0    1    2    3    4    5    6
   7    8    9   10+ \n");
241              fprintf(stdout,
242                 "                        %4d %4d %4d %4d %4d %4d %4d %4d %4
d %4d %4d  \n",
243                 coll_00,coll_01,coll_02,coll_03,coll_04,coll_05,coll_06,
coll_07,coll_08,coll_09,coll_10);
244              fprintf(stderr,
245                 "Collision density -->  0    1    2    3    4    5    6
   7    8    9   10+ \n");
246              fprintf(stderr,
247                 "Frequency -->          %4d %4d %4d %4d %4d %4d %4d %4d %4...
d %4d %4d  \n",
248                 coll_00,coll_01,coll_02,coll_03,coll_04,coll_05,coll_06,
coll_07,coll_08,coll_09,coll_10);
249
250              coll_00 = coll_01 = coll_02 = coll_03 = coll_04 = coll_05 =
 0 ;
251              coll_06 = coll_07 = coll_08 = coll_09 = coll_10 = 0 ;
252
253           } /* endif for stats calculation  */
254
```

```
255         } /* endwhile fread  */
256
257         fprintf(stdout,"\nProgram completed normally.\n");
258
259   }    /* end of main program */
260
261   hashkey(newkey)
262   char *newkey ;
263   {
264   register int l,m ;
265   int index ;
266   unsigned char temp ;
267
268
269     /*****************************************************************/
270     /*                                                               */
271     /* Assign the ASCII characters to there corresponding integer    */
272     /* values by converting then to type int.                        */
273     /*                                                               */
274     /*****************************************************************/
275
276       hash = 0;
277       delta = 0;
278
279       for (l = 0; l < 4; l++){
280
281               temp = newkey[l] ;
282               index = 8 * l ;
283
284         for (m = 0 ; m < 8 ; m++) {
285
286                 if ((temp & BIT_MASK) == 1) setbit(m+index) ;
287                 temp >>= 1;
288
289          } /* endfor */
290
291       } /* endfor */
292
293   }
294   setbit(x)
295   int x;
296   {
297
298       switch (x) {
299
300             case 0 :
301               delta |= MASK_8000 ;
302               break;
303
304             case 1 :
305               hash |=   MASK_8000 ;
306               break;
307
```

```
308             case 2 :
309               delta != MASK_4000 ;
310               break;
311
312             case 3 :
313               hash !=  MASK_4000 ;
314               break;
315
316             case 4 :
317               delta !=  MASK_2000 ;
318               break;
319
320             case 5 :
321               hash !=  MASK_2000 ;
322               break;
323
324             case 6 :
325               delta !=  MASK_1000 ;
326               break;
327
328             case 7 :
329               hash !=  MASK_1000 ;
330               break;
331
332             case 8 :
333               delta != MASK_0800 ;
334               break;
335
336             case 9 :
337               hash !=  MASK_0800  ;
338               break;
339
340             case 10 :
341               delta != MASK_0400 ;
342               break;
343
344             case 11 :
345               hash !=   MASK_0400 ;
346               break;
347
348             case 12 :
349               delta != MASK_0200 ;
350               break;
351
352             case 13 :
353               hash !=  MASK_0200 ;
354               break;
355
356             case 14 :
357               delta !=  MASK_0100 ;
358               break;
359
360             case 15 :
```

```
361               hash |=  MASK_0100 ;
362               break;
363
364             case 16 :
365               delta |= MASK_0080 ;
366               break;
367
368             case 17 :
369               hash |=   MASK_0080 ;
370               break;
371
372             case 18 :
373               delta |= MASK_0040 ;
374               break;
375
376             case 19 :
377               hash |=  MASK_0040 ;
378               break;
379
380             case 20 :
381               delta |=  MASK_0020 ;
382               break;
383
384             case 21 :
385               hash |=  MASK_0020 ;
386               break;
387
388             case 22 :
389               delta |=  MASK_0010 ;
390               break;
391
392             case 23 :
393               hash |=  MASK_0010 ;
394               break;
395
396             case 24 :
397               delta |= MASK_0008 ;
398               break;
399
400             case 25 :
401               hash |=  MASK_0008  ;
402               break;
403
404             case 26 :
405               delta |= MASK_0004 ;
406               break;
407
408             case 27 :
409               hash |=   MASK_0004 ;
410               break;
411
412             case 28 :
413               delta |= MASK_0002 ;
```

```
414               break;
415
416             case 29 :
417               hash |=  MASK_0002 ;
418               break;
419
420             case 30 :
421               delta |=  MASK_0001 ;
422               break;
423
424             case 31 :
425               hash |=  MASK_0001 ;
426               break;
427
428             default:
429               err_badbit();
430               break;
431
432           } /* endswitch */
433
434   }
435   comp(buff)
436   char buff[];
437   {
438   int equal ;
439   register i = 0 ;
440
441           while ((table[hash][i] == buff[i]) && (i < 4)) { i++ ; }
442           if (i == 4 ) return(1); else return(0);
443   }
444   err_badbit()
445   {
446     fprintf(stderr,
447             "A bad bit was attempted to be set in routine setbit.\n");
448     fprintf(stderr,"Program aborted. \n");
449     exit(10);
450   }
```

PAGE 11
03-20-88
21:50:13
Microsoft C Compiler Version 5.00

No errors detected

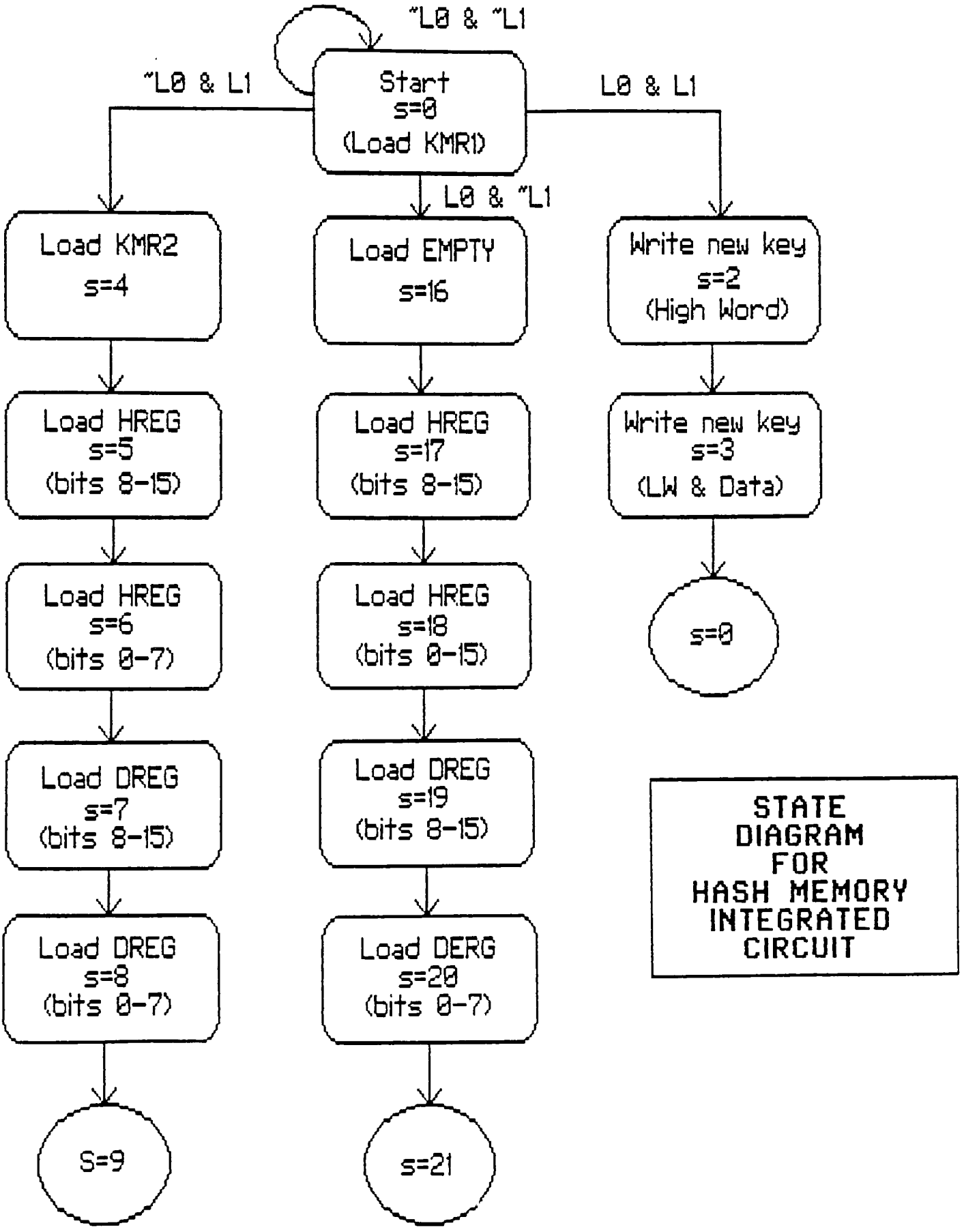
APPENDIX C: SOURCE CODE FOR GENERATING TEST DATA.

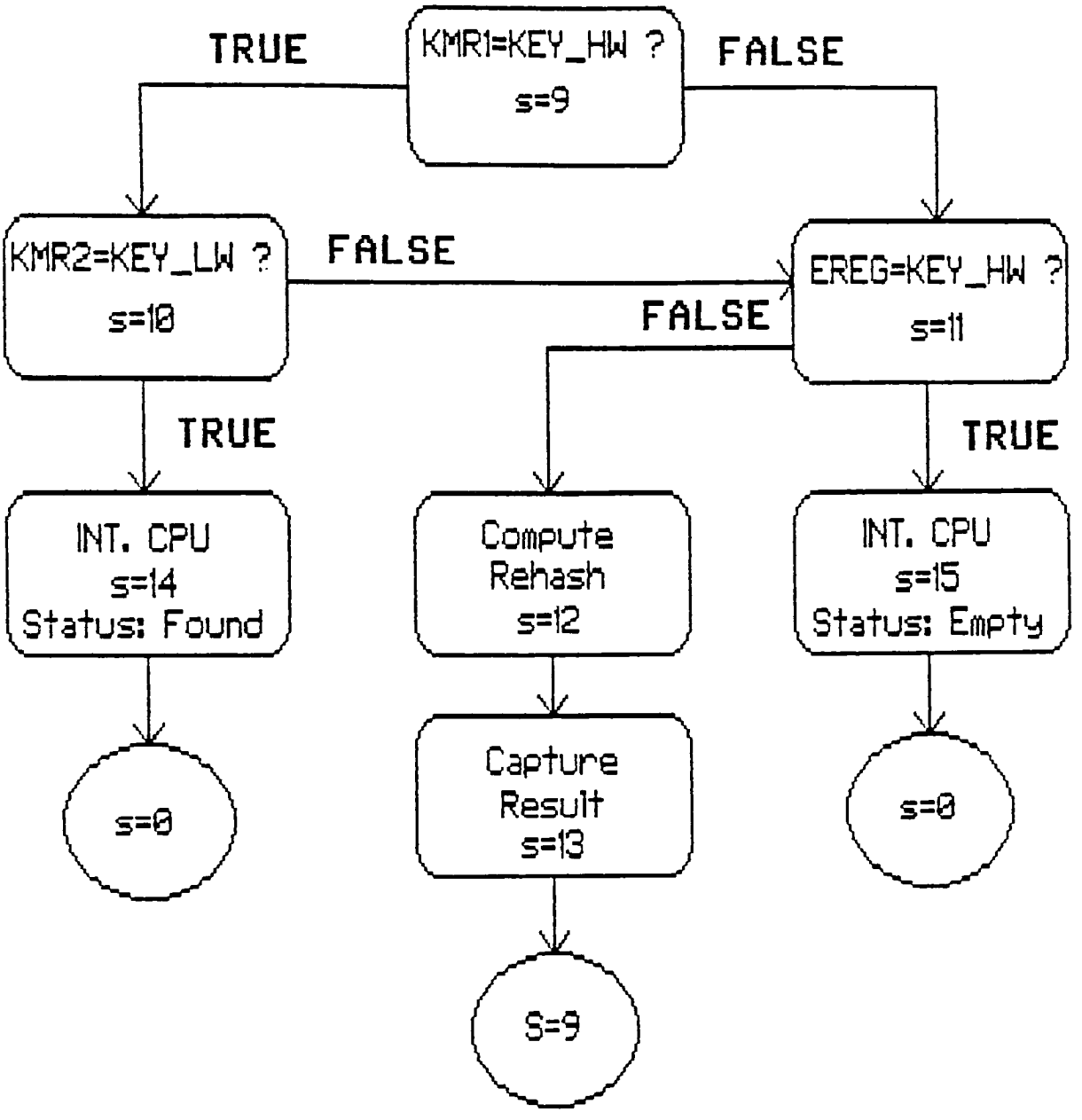
PAGE 1
03-20-88
15:09:40

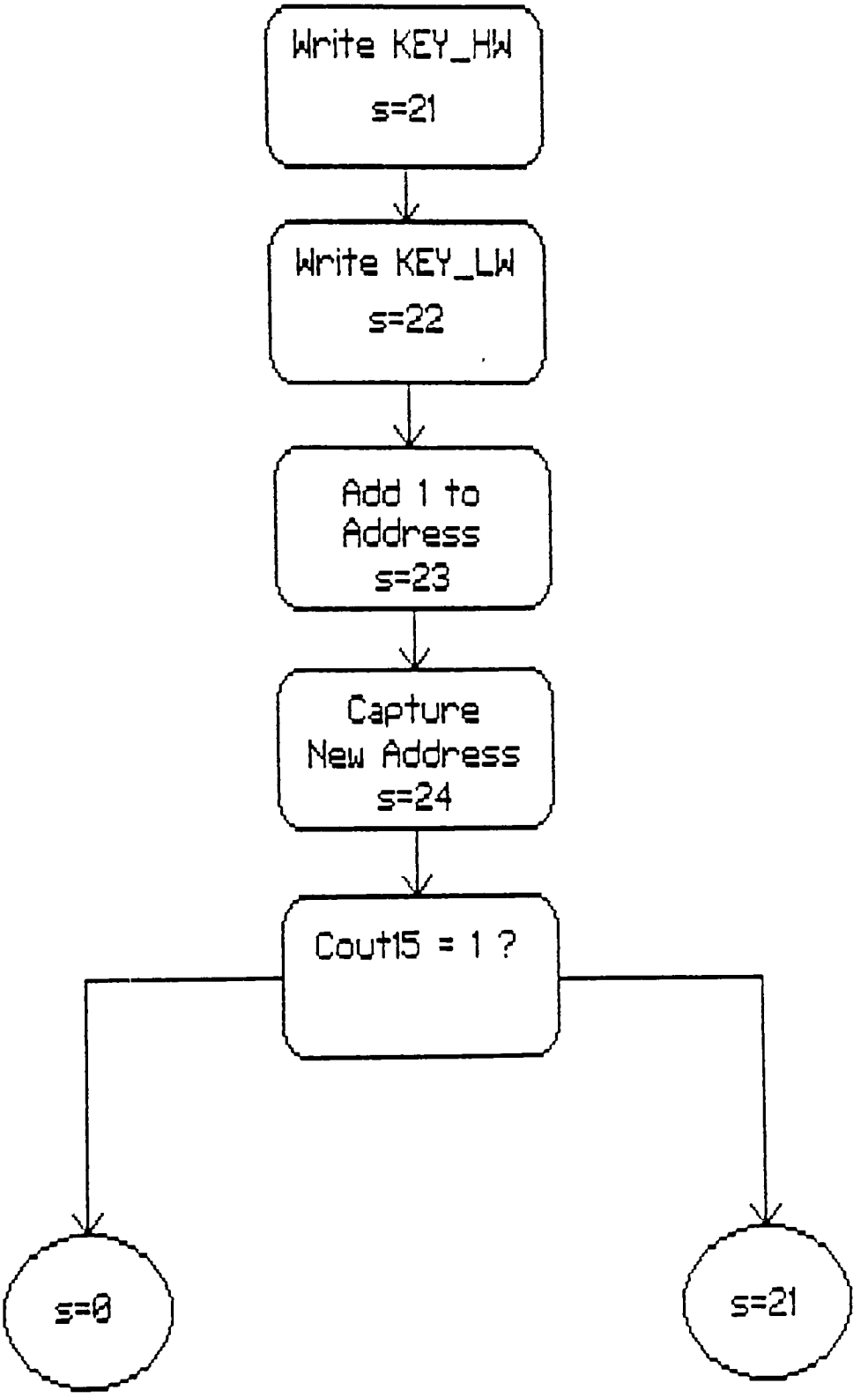
```
Line#  Source Line              IBM Personal Computer C Compiler Version 5.00
    1    /*    Program Name : GENDATA
    2     *    Written By   : Bob Ketrick
    3     *    Program Description :  This program generates strings of
    4     *      printable ASCII characters that are used as input to the
    5     *      hash table processor simulation program. The program will
    6     */     generate TABLE_SIZE number keys.
    7
    8  #include "c:\ibm\include\stdio.h"
    9  #include "c:\ibm\include\stdlib.h"
   10  #include "c:\ibm\include\string.h"
   11
   12  #define  TABLE_SIZE   70000
   13
   14
   15  main(argc, argv)
   16  int argc;
   17  char *argv[];
   18  {
   19
   20  /* These are the local variables to be used in the hash program */
   21
   22      FILE *out, *fopen();
   23      int rnd ;
   24      char *key , *empty = "****\0" ;
   25      unsigned long int count = 0 ;
   26      register i = 0 ;
   27
   28        if (argc != 3)    {
   29
   30             /* no args; print error and exit */
   31
   32             fprintf(stderr, "usage : gendata <outfile> <seed> \n");
   33             exit(0);
   34
   35       }  else  {
   36
   37              /* get the data file for processing */
   38
   39             if ((out = fopen(argv[1] , "w")) == NULL) {
   40                 fprintf(stderr, "hash: can't open %s\n", *argv);
   41                 exit(1);
   42
   43             }  /* endif fopen() */
   44
   45       }  /* endif argc != 3 */
```

```
46
47              srand(*argv[2] & '\x0F');
48              fprintf(stdout,"Seed value = %d \n\n",(*argv[2] & '\x0F'));
49
50              while ( count <= TABLE_SIZE ) {
51
52
53                      rnd = rand() % 127 ;
54
55                      if ((rnd >= '\x30') && (rnd <= '\x7D')) {
56
57                              key[i++] = rnd ;
58
59                              if (i == 4)  {
60
61                                i = 0 ;
62                                if (strcmp(key,empty) != 0) {
63
64                                    key[4] = '\0' ;
65                                    fprintf(out,"%s\n",key);
66                                    count++ ;
67                                    if ((count >= 10000) &&
68                                       ((count % 10000) == 0))
69                                      fprintf(stdout,
70                                          "%ld keys generated.\n",count);
71
72                                } /* endif */
73
74                              } /* endif */
75
76                      } /* endif */
77
78              } /* endwhile */
79
80  }   /* end of main program */
No errors detected
```

APPENDIX D: STATE DIAGRAM FOR PLA.


~L0 & ~L1
Start
s=0
(Load KMR1)
~L0 & L1
L0 & L1
L0 & ~L1
Load KMR2
s=4
Load EMPTY
s=16
Write new key
s=2
(High Word)
Load HREG
s=5
(bits 8-15)
Load HREG
s=17
(bits 8-15)
Write new key
s=3
(LW & Data)
Load HREG
s=6
(bits 0-7)
Load HREG
s=18
(bits 0-15)
s=0
Load DREG
s=7
(bits 8-15)
Load DREG
s=19
(bits 8-15)
Load DREG
s=8
(bits 0-7)
Load DERG
s=20
(bits 0-7)
S=9
s=21
STATE DIAGRAM FOR HASH MEMORY INTEGRATED CIRCUIT


KMR1=KEY_HW ?
s=9
TRUE
FALSE
KMR2=KEY_LW ?
s=10
FALSE
EREG=KEY_HW ?
s=11
FALSE
TRUE
TRUE
INT. CPU
s=14
Status: Found
Compute
Rehash
s=12
INT. CPU
s=15
Status: Empty
s=0
Capture
Result
s=13
s=0
S=9


Write KEY_HW
s=21
Write KEY_LW
s=22
Add 1 to
Address
s=23
Capture
New Address
s=24
Cout15 = 1 ?
s=0
s=21