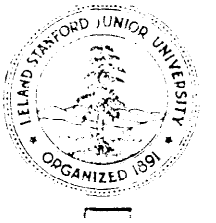
COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-4055



Sparse Distributed Memory Prototype: Principles of Operation

M. J. Flynn, P. Kanerva, B. Ahanin
N. Bhadkamkar, P. Flaherty, P. Hickey

Technical Report CSL-TR-87-338

February 1988

This research is supported by NASA under contract NAGW 419, in cooperation with RIACS (Ames Research Center).

Sparse Distributed Memory Prototype: Principles of Operation

by
M. J. Flynn, P. Kanerva, B. Ahanin
N. Bhadkamkar, P. Flaherty, P. Hickey

Technical Report CSL-TR-87-338
February 1988

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Sparse distributed memory is a generalized random-access memory (RAM) for long (e.g., 1,000 bit) binary words. Such words can be written into and read from the memory, and they can also be used to address the memory. The main attribute of the memory is sensitivity to similarity, meaning that a word can be read back not only by giving the original write address but also by giving one close to it as measured by the Hamming distance between addresses.

Large memories of this kind are expected to have wide use in speech and scene analysis, in signal detection and verification, and in adaptive control of automated equipment-in general, in dealing with real-world information in real time.

The memory can be realized as a simple, massively parallel computer. Digital technology has reached a point where building large memories is becoming practical. This research project is aimed at resolving major design issues that have to be faced in building the memories. This report describes the design of a prototype memory with 256-bit addresses and from 8K to 128K locations for 256-bit words. A key aspect of the design is extensive use of dynamic RAM and other standard components.

Key Words and Phrases: Neural networks, pattern recognition, adaptive control.

Contents

CONTENTS

List of Figures

LIST OF FIGURES

List of Tables

Chapter 1

Introduction to Sparse Distributed Memory

1.1 Introduction

Sparse distributed memory (SDM) is a generalized random-access memory (RAM) for long (e.g., 1,000 bit) binary words. These words serve as both addresses to and data for the memory. The main attribute of the memory is sensitivity to similarity, meaning that a word can be read back not only by giving the original write address but also by giving one close to it, as measured by the number of mismatched bits (i.e., the Hamming distance between addresses).

The theory of the memory is mathematically complete and has been verified by computer simulation. It arose from the observation that the distances between points of a high-dimensional space resemble the proximity relations between concepts in human memory. The theory is also practical in that memories based on it can be implemented with conventional RAM-memory elements. The memory array in the prototype memory makes extensive use of 1 M-bit DRAM technology, with array modules in concurrent execution. Consequently, the prototype is inexpensive compared to implementations of the memory on systolic-array, "connection machine," or general-purpose equipment.

In applications of the memory, the words are patterns of features. Some features are produced by a sensory system, others control a motor system, and the rest have no immediate external significance. There is a ***current*** (1,000 bit) ***pattern,*** which is the current contents of the system's focus. The sensors feed into the focus, the motors are driven from the focus, and the memory is accessed through the focus. What goes on in the world-the system's "subjective" experience-is represented internally by a

sequence of patterns in the focus. The memory stores this sequence and can recreate it later in the focus if addressed with a pattern similar to one encountered in the past. Thus, the memory learns to predict what is about to happen.

Wide applications of the memory would 'be in systems that deal with real-world information in real time. Such information is rich, varied, incomplete, unpredictable, messy, and dynamic. The memory will home on the regularities in the information and will base its decision on them. The applications include vision-detecting and identifying objects in a scene and anticipating subsequent scenes-robotics, signal detection and verification, and adaptive learning and control. On the theoretical side, the working of the memory may help us understand memory and learning in humans and animals.

For an example, the memory should work well in transcribing speech, with the training consisting of "listening" to a large corpus of spoken language. Two hard problems with natural speech are how to detect word boundaries and how to adjust to different speakers. The memory should be able to handle both. First, it stores sequences of patterns as pointer chains. In training--in listening to speech-it will build a probabilistic structure with the-highest incidence of branching at word boundaries. In transcribing speech, these branching points are detected and tend to break the stream into segments that correspond to words. Second, the memory's sensitivity to similarity is its mechanism for adjusting to different speakers—and to the variations in the voice of the same speaker.

1.2 Rationale for special hardware

Although the sparse distributed memory is a generalized random-access memory, its most important properties are not demonstrated by ordinary random accesses. For those properties to appear, the memory addresses must be at least 100 bits and preferably several hundred, and any read or write operation must manipulate many memory locations. When these conditions are met, the memory can use approximate addresses (in the Hamming-distance sense) to retrieve exact information as well as statistically abstracted information that represents natural groupings of the input data. Intelligence in natural systems is founded on such properties.

Simulation of the memory on a conventional computer is extremely slow. A properly designed, highly parallel hardware is absolutely necessary for dealing with practical problems in real time. Table 1.1 shows the estimated performance of different sized memories on a range of hardware implementations.

The Stanford prototype is designed to be a flexible, low-cost model of projected large-scale implementations. Experiments performed with the prototype are intended

to develop better applications support and especially faster, more efficient implementations.

Table 1.1: Realizing sparse distributed-'memory in different kinds of hardware.

Hardware	Dimension, n	Number of locations, m	Cycles per second	Task
Dedicated DEC 2060	128	10,000	.2–1	Demonstrate convergence properties of the memory
32-node Intel iPSC	128	50,000	1-5	Simple learning by trial and error
16K-processor Connection Machine	200	60,000	50-200	Word parsing in compacted text
Stanford Prototype	256	80,000	50	Word parsing in compacted text and possibly in speech
Present VLSI potential	1,000	100,000,000	1,000	Language understanding (?)

1.3 Basic concepts and terminology

This chapter presents a nonmathematical description of the operating principles behind SDM. Readers desiring a mathematical description of these concepts should consult the paper by Kanerva [3]. The papers by Keeler [4] and Chou [1] contrast the properties of SDM with a neural-network model developed by Hopfield [2] that resembles SDM in certain aspects of its operation.

There are six concepts that are central to describing the behavior of SDM. These are:

- writing to the memory
- reading from the memory
- address pattern (or reference address, or retrieval cue, or cue)
- data pattern (or contents, or data word)
- memory location (or hard location) and hard address

- distance from a memory location

The first two are operations on the memory, the middle two are external to the memory and have to do with the external world, while the last two are concepts relating to the internal aspects of the memory. Each of these is explained in more detail below.

Writing is the operation of storing a data pattern into the memory using a particular address pat tern.

Reading is the operation of retrieving a data pattern from the memory using a particular address pat tern.

Address Pattern. An N-bit vector used in writing to and reading from the memory. The address pattern is a coded description of an environmental state. (In the prototype, $N = 256$.)

Data Pattern. An M-bit vector that is the object of the writing and reading operations. Like the address pattern, it is a coded description of an environmental state. (In the prototype, ***M = 256.)***

Memory location. SDM is designed to cope with address patterns that span an enormous address space. For example, with $N = 256$ the input address space is 2^{256}. SDM assumes that the address patterns actually describing physical situations of interest are sparsely scattered throughout the input space. It is impossible to reserve a separate physical location corresponding to each possible input; SDM implements only a limited number of physical or "hard"' locations. The physical location is called a ***memory (or hard) location.***

Every hard location has associated with it two items:

- a fixed ***hard address,*** which is the N-bit address of the location.
- a ***contents*** portion that is M-bits wide and that can accumulate multiple M-bit data patterns written into the location. The contents' portion is not fixed; it is modified by data patterns written into the memory.

1.3.1 Distance from a memory location (to the reference address)

The distance from a memory location to a reference address used in either a read or write operation is the Hamming distance between the memory location's hard address and the reference address. The Hamming distance between two N-bit vectors is the number of bit positions in which the two ***differ***, and can range from 0 to N. SDM uses

the Hamming measure for distance because it is convenient for vectors consisting of 0s and 1s. However, other measures could equally well be used.

The operation of the memory is explained in the remainder of this chapter. However, the following is a brief preview:

- During a write, the input to the memory consists of an address pattern and a data pattern. The address pattern is used to select hard locations whose hard addresses are within a certain cutoff distance from the address pattern. The data pattern is stored into each of the selected locations.
- During a read, an address pattern is used to select a certain number of hard locations (just like during a write). The contents of the selected locations are bitwise summed and thresholded to derive an M-bit data pattern. This serves as the output read from the memory.

How this works is explained below.

1.4 Basic concepts

1.4.1 A simple example

These concepts and the basic mechanisms of SDM will be illustrated by a stylized example. For the sake of simplicity, assume the following:

1. Input vectors consist of

   (a) an integer address that can range from 1 to 1000, and

   (b) a data pattern (or content portion) that is an 8-bit vector.

   An example of an input vector is:

Address	*Data pattern*
867	0 1 1 0 1 0 1 0

   It should be emphasized that the data pattern is not a binary number. Rather, the 1s and 0s could be thought of as the presence or absence of specific features. In the actual implementation, described later, both the address and contents are 256-bit patterns.

2. The memory in this example implements only 99 hard locations. These have associated with them the addresses

   5.5, 15.5, 25.5, . . . , 995.5

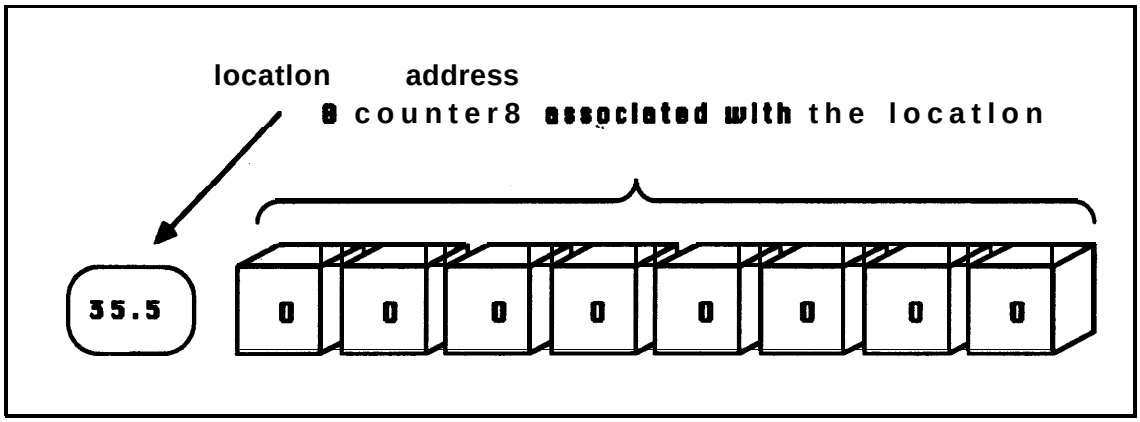


Figure 1.1: Example of a location.

The reason for the half addresses is merely to position each location symmetrically between 1 and 10. The need for this will be clear shortly.

3. Each hard location has associated with it 8 buckets-one bucket for each bit of an 8-bit data-pattern vector. Each bucket accumulates bits that are stored into it by acting as an up/down counter. Each bucket starts out holding the value 0. A binary 1 stored into the bucket causes its count to go up by 1, whereas a binary 0 stored into a bucket causes its count to go down by 1.

   As will be explained shortly, this facility is required because each location may have many inputs stored into it.

   An example of a location is shown in Figure 1.1. If an input vector with contents 1 0 0 1 0 0 1 1 is stored into this location, the location will look as shown in the upper half of Figure 1.2. If now another input vector with contents 1 0 0 0 0 0 1 1 is stored into the same location, the result is shown in the lower half of that figure.

   The contents of an individual location could be interpreted as follows. If a bucket has a count that is positive, it has had more 1*s* written into it than 0*s* and can be interpreted as a 1. Similarly, a bucket with a negative count can be interpreted as a 0. A bucket with a 0 count (in a location that has been written into) has had an equal number of 1*s* and 0*s* written into it and can be interpreted as a 1 or a 0, each with probability 0.5.

To understand the working of SDM, we will deal with the problem of retrieving the closest match. We want to store into memory the input vectors that the system

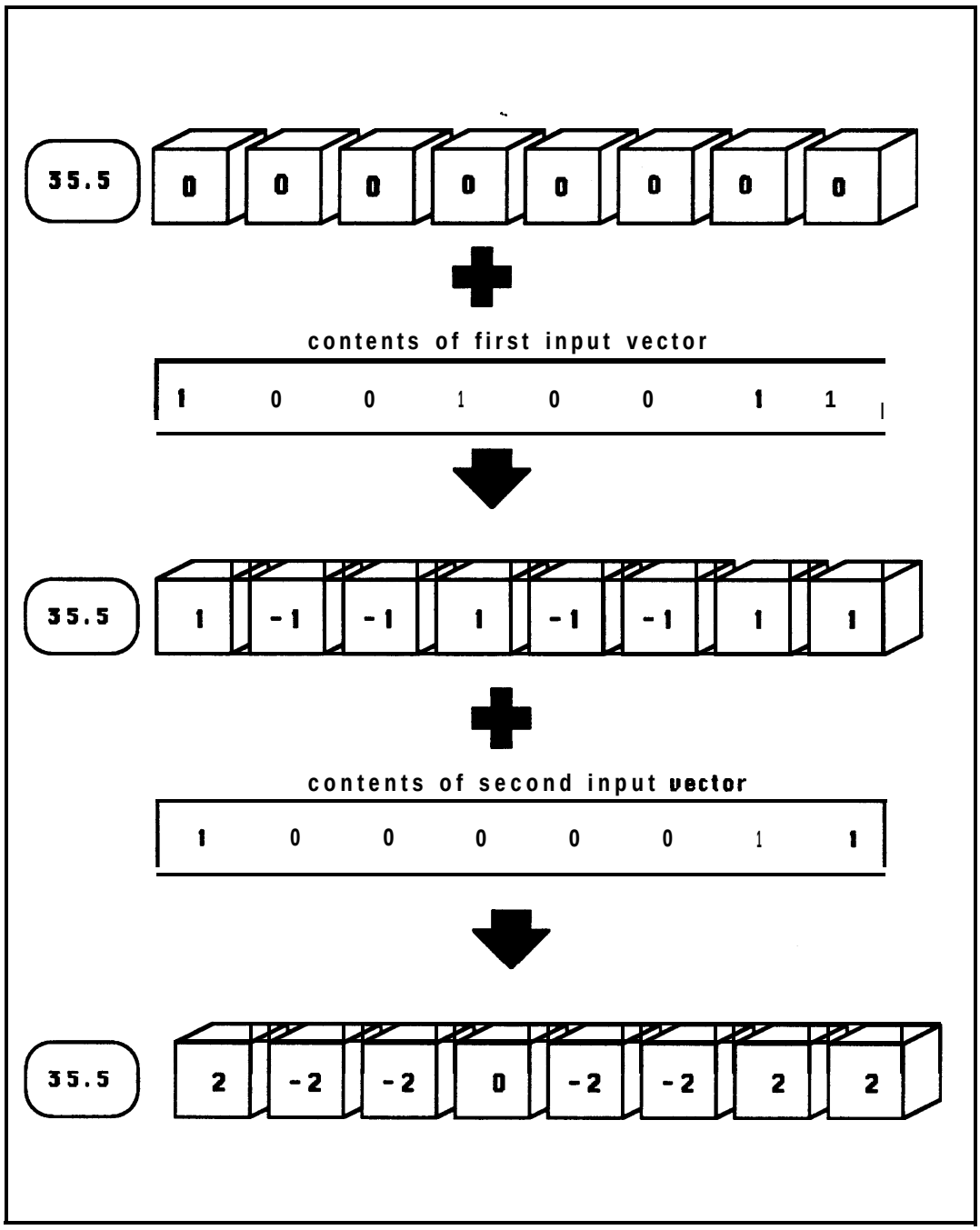


Figure 1.2: Example of a location.

encounters. At some later point, we want to present to the memory an address cue and have the memory retrieve the contents of the stored input vector with the address that is closest to the input cue.

1.4.2 A simple solution that does not work

An apparently simple way in which this best-match problem could be tackled is the following: Store each input into the closest hard location. This can be accomplished by making each hard location "sensitive" or addressable by any input with an address that is within f4.5 of the address of the location. Thus, any input with an address in the range 31 to 40 (both inclusive) would be stored into the location with the address of 35.5. Then, when presented with a retrieval cue, read out the contents of the closest hard location. Unfortunately, though this works sometimes, it often does not. To see this, consider the example below:

Input #1: 139 10101010
Input #2: 169 11001011
Retrieval cue: 150

Input #1 will be stored into the location with address 135.5. Input #2 will be stored into the location with address 165.5. The retrieval cue will activate the location 155.5. This location has nothing in it. One way to deal with this problem is to gradually increase the activation distance during retrieval. In the above case, if the activation distance were increased to •t 14.5, the system would retrieve the contents of the location 135.5, which contains the closest match. However, if the example is modified slightly so that the first input address is 131 and the second is 161, the method fails even after the activation range has been increased to 150 ± 14.5. SDM solves the problem using a statistical approach that is much more robust and has fairly simple mechanics.

1.4.3 The SDM solution

SDM overcomes the above problem by

1. distributing each stored input over many locations, and
2. retrieving from a distributed set of locations.

This is the reason for the word "distributed" in the name of the system. Now, instead of storing an input into the closest location, an input is stored into all locations within a certain distance of the write address. Similarly, when presented with a retrieval cue, all

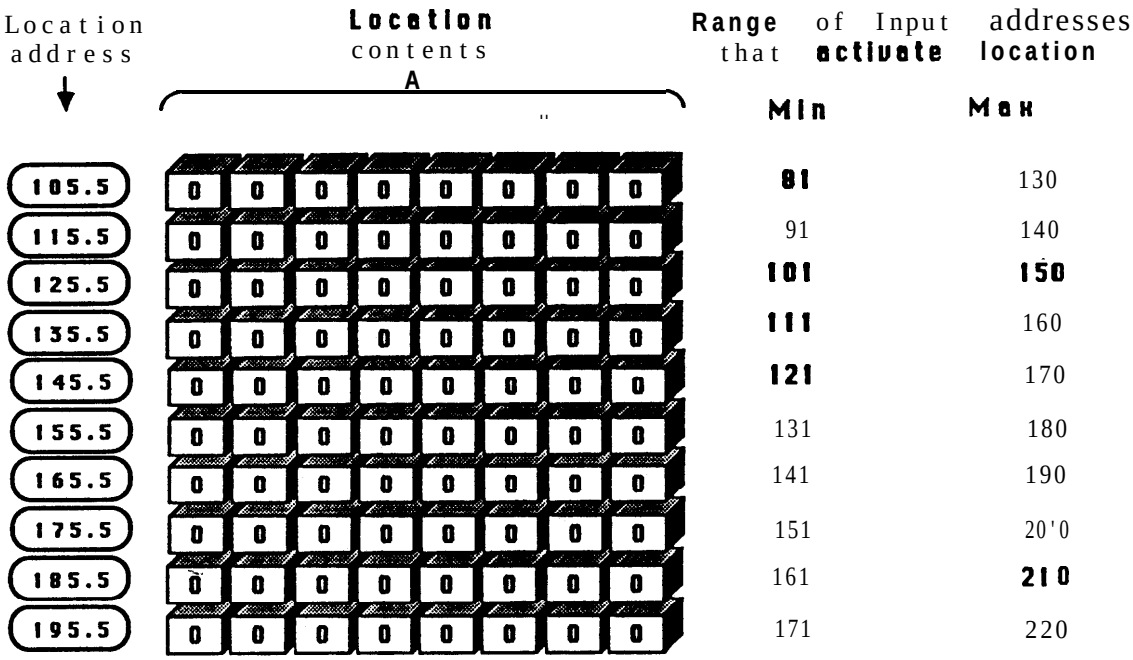


Figure 1.3: Example using SDM.

locations within a certain distance of the retrieval cue are read out and used to derive the output in a manner to be described. These two distances, namely, the activation distances during the storage and retrieval of patterns, need not be the same. The operation of SDM can be illustrated by continuing with the previous example. Instead of having each physical location be addressable by any address within f4.5 of it, assume that the activation distance is now f25. We will use the same activation distance during both the storage and retrieval phases, for simplicity. Figure 1.3 illustrates the initial state of a portion of the system, encompassing physical locations with addresses ranging from 105.5 to 195.5. Also shown is the range of addresses to which each location is sensitive.

When the memory is presented with the first input pattern, 139: 1 0 1 0 1 0 1 0, the memory locations with addresses 115.5, 125.5, 135.5, 145.5, and 155.5 are all activated. The contents of the input vector are written into each of the locations according to the rule described earlier. Figure 1.4 shows the state of the system after this occurs.

Now the system is presented with Input #2, namely 169: 1 1 0 0 1 0 1 1. This input activates the locations with addresses 145.5, 155.5, 165.5, 175.5, and 185.5. The vector being stored, 1 1 0 0 1 0 1 1, is accumulated, bit by bit, into the buckets of each of these

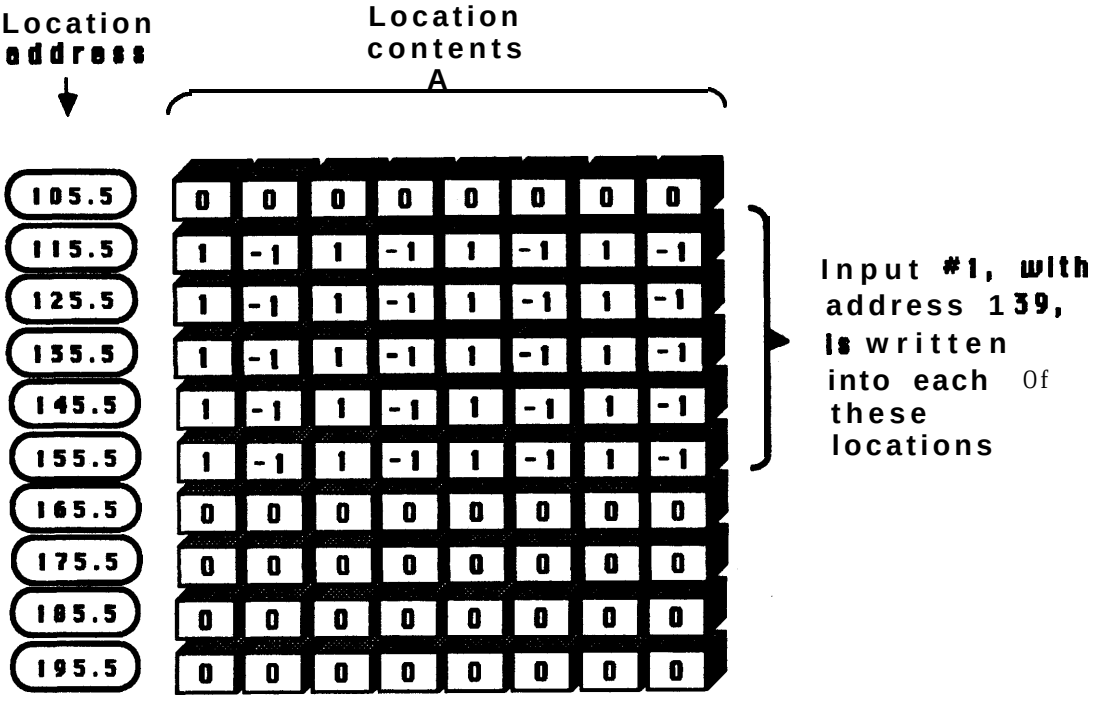


Figure 1.4: Example using SDM.

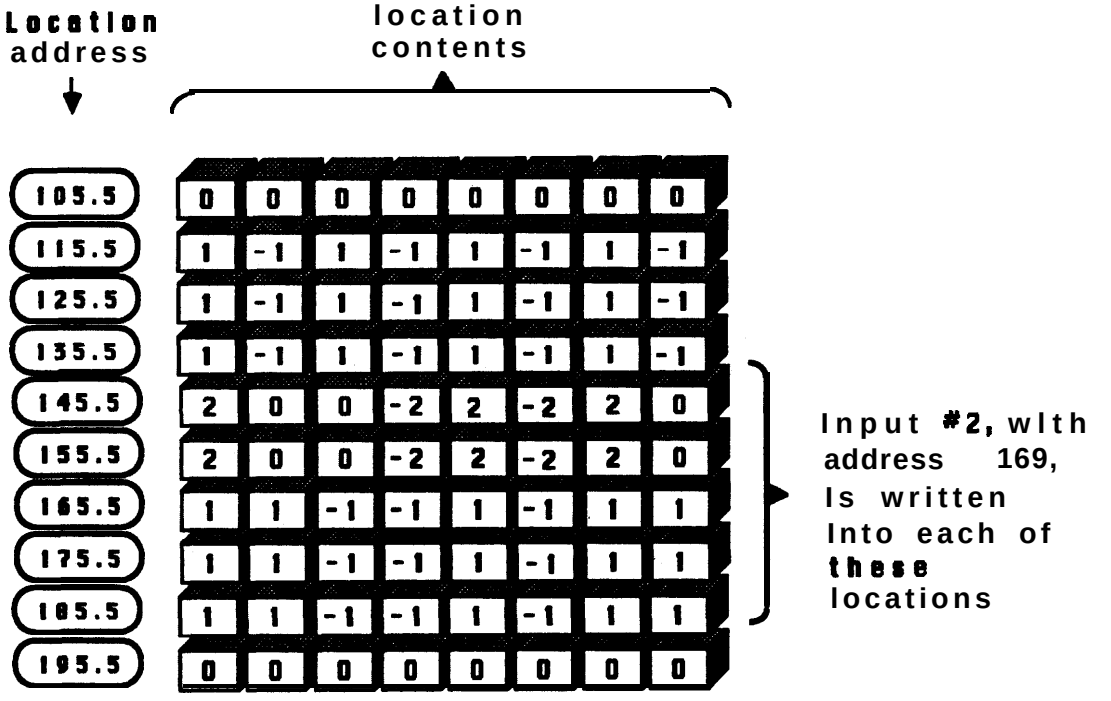


Figure 1.5: Example using SDM.

locations. The resulting state of the system is presented in Figure 1.5. Notice that the two locations at addresses 145.5 and 155.5 have each had both input vectors written into them. Both input vectors fell within the ±25 activation distance of each of these locations.

Now consider what happens when the system is presented with the retrieval cue 150. This address activates all locations with addresses in the range 150 ± 25, namely, the locations with addresses 125.5, 135.5, 145.5, 155.5, and 165.5. The retrieval mechanism's goal is to determine, for each bit position, whether more 1s or more 0s were written into all the selected locations and to output 1 or 0 accordingly. In the case of a tie, 1 or 0 is output with probability 0.5.

The way this works is illustrated in Figure 1.6. For ***each bit position,*** the following operations are performed:

1. The contents of the buckets of all the selected locations are summed arithmetically. (A positive sum indicates that more 1s were written into these locations, while a negative sum indicates more 0s.)
2. The sum is thresholded.

A positive sum yields 1, a negative sum yields 0, and a sum of 0 yields 1 or 0 based on the toss of a coin. In this particular case, this process yields the output 1 0 1 0 1 0 1 0. This is the system's response to the query "What is the content portion of the stored input with the address closest to the retrieval cue of 150?". Notice that the vector output by the system is in fact the content portion of Input #1, which was the stored input vector with the closest address to 150, the retrieval cue.

1.4.4 Differences between the simple example and the real model

The simplified model presented above was sufficient to explain some of the basic mechanisms of SDM. However, to understand the workings of the model in more depth we switch now to a discussion based on the real model. In the simple model above, input vectors consisted of an address portion that was a three digit integer, while the contents were an 8-bit vector. Distances between input vectors were measured by the magnitude of the arithmetic difference between addresses. In the actual model, input vectors have an address consisting of an N-bit vector and a contents portion consisting of an M-bit vector. N and M need not be the same in the most general case, but they are both 256 in the prototype we are implementing. This equality is required for certain modes of operation, described later in this report. Distances between vectors are measured by the Hamming distance between the N-bit vectors. Since the Hamming distance is the

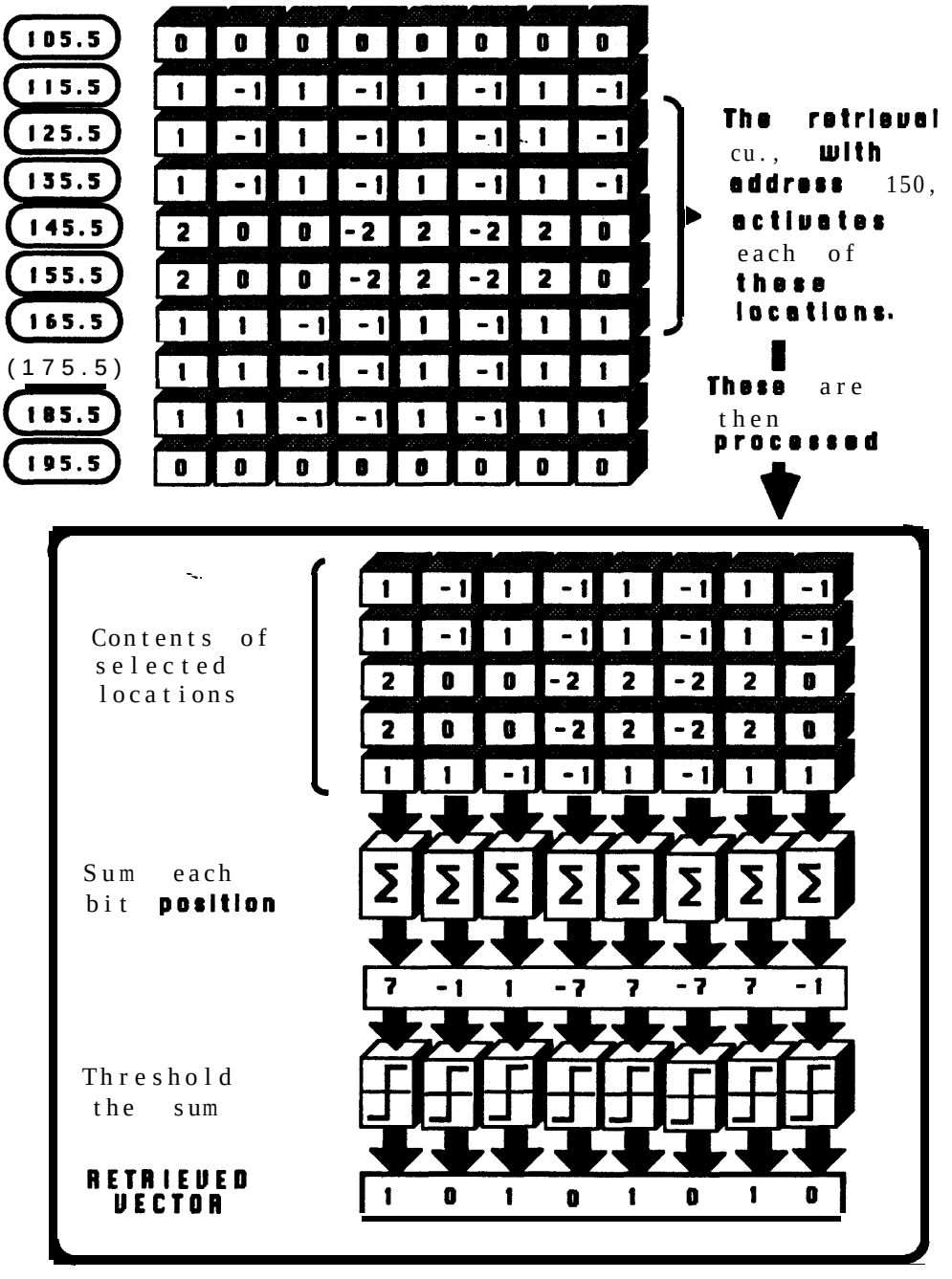


Figure 1.6: Example of data selection in SDM.

number of bit positions in which the two vectors differ, for N-bit vectors this distance will range from 0, for two identical addresses, to N, for two addresses that are bitwise complements. The potential address space is 2^{256}, compared to 1,000 in the simple example. Whereas the simple model had 100 hard locations, the basic module of the prototype system has 8,192 locations with addresses scattered randomly with uniform probability through the address space. Each hard location has associated with it a set of 256 buckets to accumulate the vectors that may be stored into that location. Each bucket is conceptually an 8-bit up/down counter that can hold a count in the range of -127 to +127, inclusive. (If more than 127 1s are stored into a bucket in excess of Os, the bucket will have an overflow. Similarly, an underflow will occur if the number of 0s stored into a bucket exceeds the number of 1s by more than 127.) For the discussion that follows, it is useful to visualize the 2^N input space as a two-dimensional rectangle. Figure 1.7 shows the address space in this fashion. The physical locations are indicated by the small black dots within the rectangle.

The process of writing or storing a vector into the memory consists of the following two steps: (a) Draw an N-dimensional sphere of Hamming radius d around the address of the input vector. In the plane this can be visualized as drawing a circle centered at the address of the input vector. (b) For each physical location that falls within this sphere, accumulate the contents portion of the input vector into each of the 256 associated buckets. This is depicted in Figure 1.8.

Given a retrieval address, the process of reading from the memory proceeds in a similar two-step fashion: (a) Draw an N-dimensional sphere of Hamming radius d' (which need not equal the radius d used for storing patterns) centered at the retrieval cue. (b) Derive the i^{th} bit of the output vector (i going from 1 to M) in a manner identical to that used in the simple example. Specifically, sum the contents of the i^{th} buckets of all the locations falling within the sphere drawn above, and then threshold the sum to either 1 or 0 based on whether the sum is positive or negative. This is depicted in Figure 1.9.

1.5 Autoassociative dynamics

We are now in a position to understand the reconstructive properties of SDM when used in the autoassociative mode. In this mode, the dimensionality of the address and contents portions of an input vector are the same. In fact, the contents are used also as the address. This means that the input vector can be viewed as consisting of a pattern vector P that forms its address vector, and the same pattern vector P that forms it contents vector. During storage, the pattern vector P serves as the address that is used to select a number of physical locations. The same vector P is also stored into each of

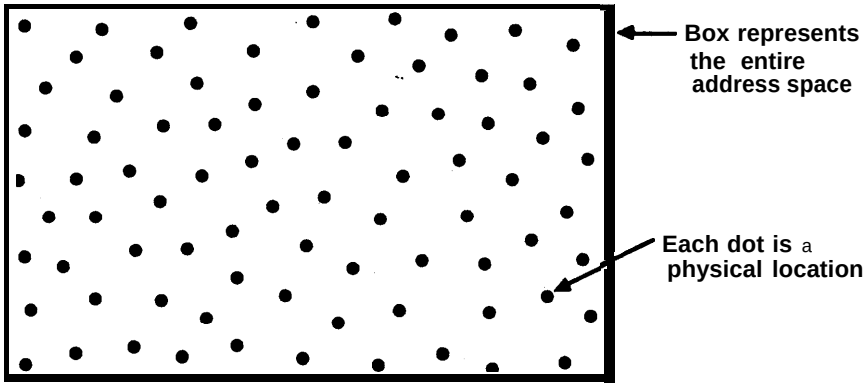


Figure 1.7: Space of locations.

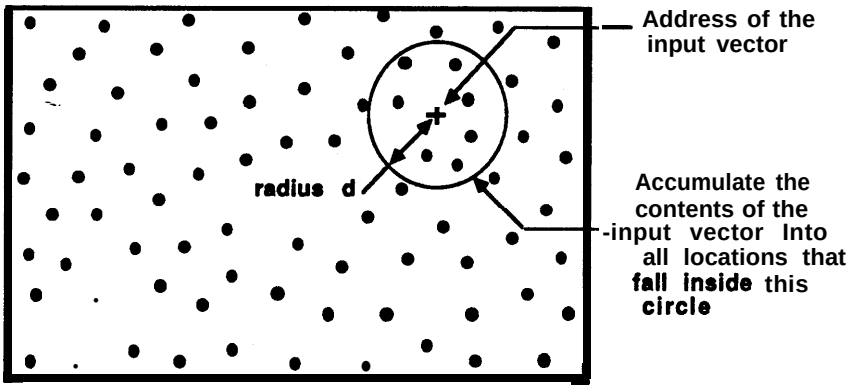


Figure 1.8: SDM storage activation radius.

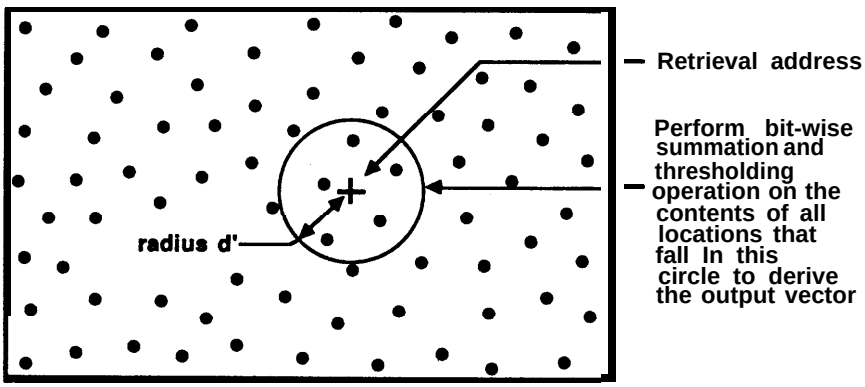
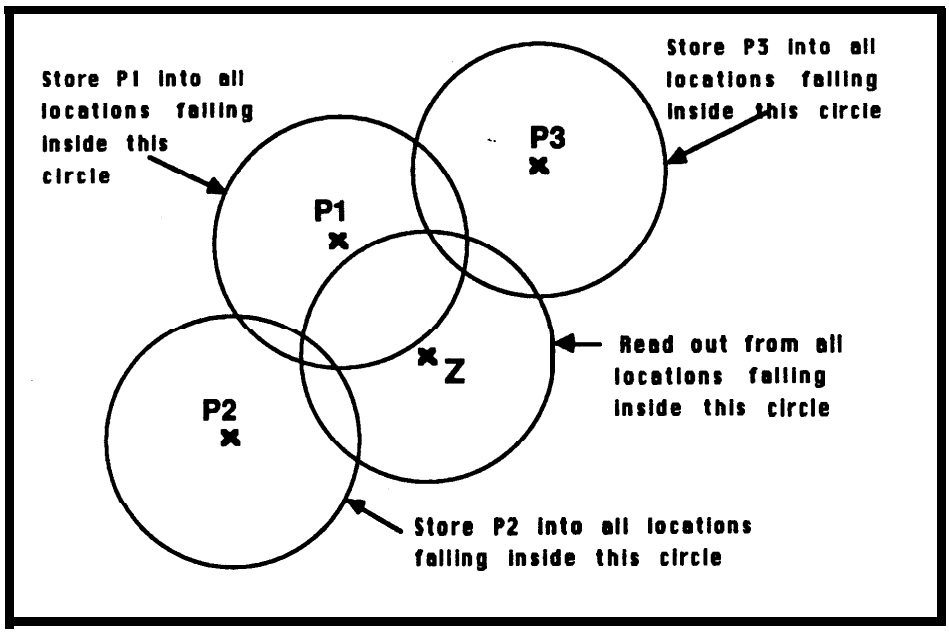


Figure 1.9: SDM retrieval activation radius.



For clarity, individual physical locations are not shown.

Figure 1.10: Overlapping radii.

the locations it activates. Figure 1.10 shows three pattern vectors P(l), P(2), and P(3) stored into locations in memory. Z is a contaminated or partial version of P(l), and the goal is to have the memory reconstruct P(1) when cued with Z. As shown in the diagram, the locations activated by Z are of six types:

1. Those that contain only $P(1)$.
2. Those that contain P(1) and P(2).
3. Those that contain P(1) and P(3).
4. Those that contain only P(2).
5. Those that contain only P(3).
6. Those that contain nothing.

This can be generalized to say that the locations activated by Z contain a mixture of P(l), which can be regarded as the signal, and non-P(l) patterns, which can be regarded as noise, using nomenclature from Keeler [4].

The signal-to-noise ratio is higher (a) the closer Z is to P(1) and (b) the less densely populated the memory is, i.e., the fewer patterns that have been written into it. It can be shown mathematically [3] that if Z is closer than a certain critical distance from P(l), then summing and thresholding the contents of the locations activated by Z results in a vector Z(1) that is closer to P(1) than Z was. The critical distance is a function of how densely populated the memory is. The greater the population of the memory, the smaller the critical distance. The fact that Z(1) is closer to P(1) than Z was is of great benefit because now Z(1) can be used as a new retrieval cue. The output Z(2) will be even closer to P(1) than $Z($ 1) was, and it too can now be used as a new retrieval cue. This iterative process, which is a form of feedback, therefore produces a sequence of outputs Z(l), $Z(2)$, . . . , Z(n) that converges rapidly to either P(1) or a minimally noisy version of $P($ 1). This is depicted in Figure l.lla. If the cue vector Z is beyond the critical distance from P(l), i.e., if it is too contaminated or incomplete, the sequence of vectors Z(l), $Z(2)$, . . . will not converge to P(1). Instead, it will be a diverging sequence that will wander through address space. It may eventually wander, by chance, into the attracting zone of some other stored pattern, say, $P(k)$, and thereby converge onto $P(k)$. This process is depicted in Figure l.llb.

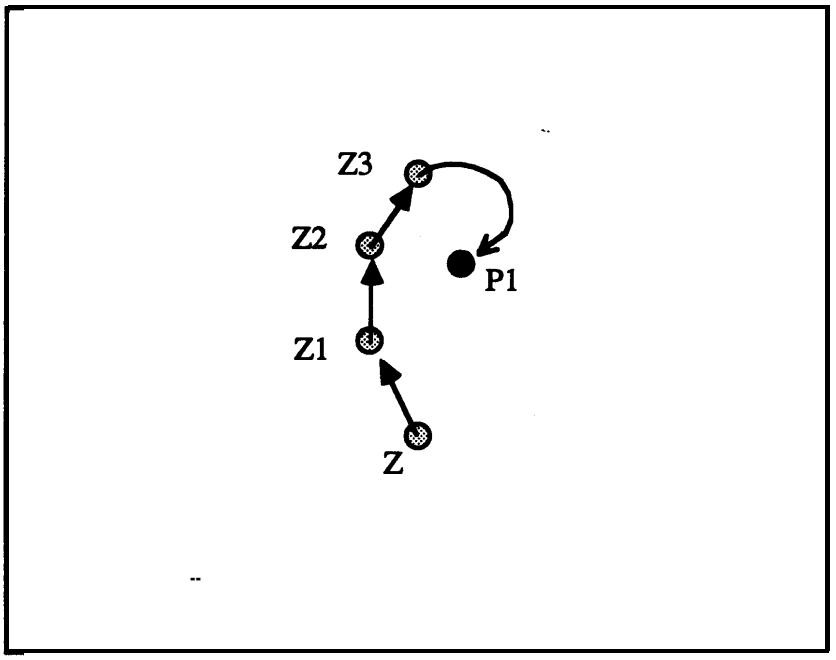
Experimentally, convergence to the correct value P(1) occurs rapidly (40 iterations), whereas the diverging sequence, if it eventually converges to some other pattern, takes a large number of iterations. The two situations are easy to distinguish in practice.

There is an upper bound to the value of the critical distance referred to above. In a sparsely populated memory in which there is little or no overlap between the locations in which patterns have been stored, the critical distance is necessarily less than the sum of the Hamming radii used during storage and retrieval. If the cue Z is beyond this distance from P(l), none of the locations activated by it during the retrieval process will contain any copies of $P($ 1).

The behavior of SDM illustrated in Figures l.lla and b is similar to the dynamic behavior of the Hopfield net used in its autoassociative mode. In that model, stored patterns act like attractors for input cues within a certain critical distance. The behavior of the Hopfield model, however, is driven by an energy-minimization mechanism in which the stored patterns behave like local minima of an energy function associated with the address space.

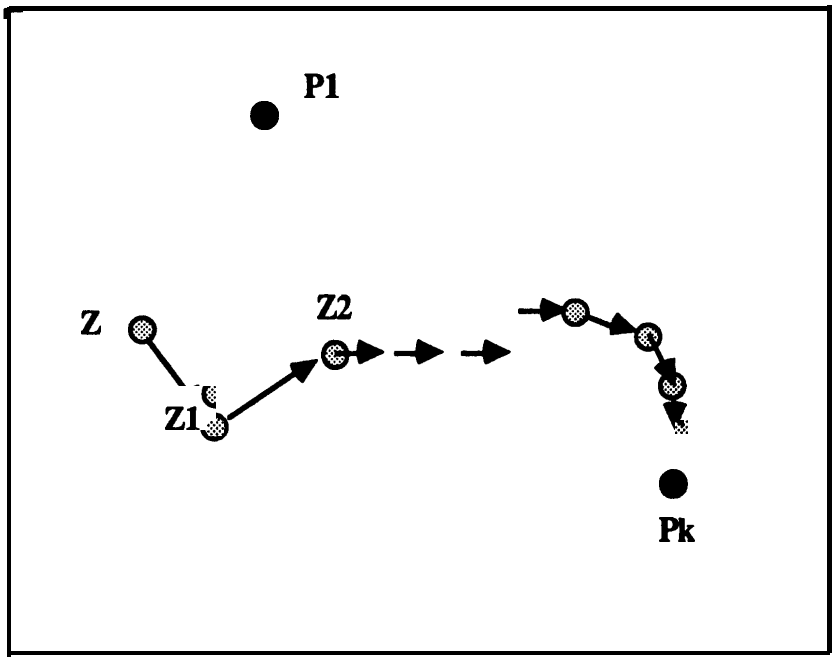
1.6 Heteroassociative dynamics (sequences)

SDM can also be used in a heteroassociative mode. In this mode the contents portion of an input vector is not generally equal to its address portion. In general, the two do



(a)

Cueing the memory with **Z** sufficiently close to **P1** **results** in a rapidly converging **seqence** of values. The stored pattern **P1** acts like an attractor.



(b)

If **Z** is too distant from **P1,** the iteratively retrieved sequence does not converge. It may eventually converge on some other pattern Pk.

Figure 1.11: SDM autoassociative mode.

not have to be of the same dimension either. However, when they do have the same dimensionality, SDM lends itself to the storing and recalling of sequences. Consider a sequence of N-dimensional pattern vectors

$$P(1);\ P(2),\ P(3),\ldots$$

Examples of such sequences might be (a) a sequence of motor-control parameters for controlling the trajectory of a robot arm, or (b) a sequence of musical notes that comprises a tune.

Imagine forming the input vectors shown below, where the address portion of an input vector is the previous data pattern element in the sequence.

Address pattern	*Data pattern*
$P(1)$	$P(2)$
$P(2)$	$P(3)$
$P(3)$	P(4)
$\vdots$	$\vdots$
$P(i)$	$P(i+1)$

These vectors can be used to write into the memory. The effect of this is that at the locations selected by the address P(l), the pattern ***P(2)*** is stored; at the locations selected by the address ***P(2),*** the pattern ***P(3)*** is stored, and so on. In general, at the locations selected by the address ***P(i),*** the pattern $P(i+1)$ is stored. This is illustrated in Figure 1.12.

Now imagine cueing the memory with a pattern close to one of the elements of the stored sequence. For example, suppose we cued the system with the address ***P(2)**** that is close to ***P(2).*** Just as in the autoassociative case, if ***P(2)**** is sufficiently close to ***P(2),*** the retrieved pattern will be even closer to the pattern that was stored in the locations activated by the address ***P(2).*** In this case the retrieved value $P(3)*$ will be closer to the stored value ***P(3)*** than ***P(2)**** was to ***P(2).*** If ***P(3)**** is now used to cue the memory, the retrieved pattern P(4)$*$ will be closer to P(4) than ***P(3)**** was to ***P(3).*** Continuing in this manner, we observe that cueing the memory with the pattern ***P(2)**** allowed us to iteratively recover the sequence: ***P(3)*, P(4)*, P(5)*,*** . . . that converges onto the stored sequence ***P(3), P(4), P(5), ...*** This is illustrated in Figure 1.13.

Just as in the autoassociative case, if the initial cue ***P(2)**** is too distant from ***P(2),*** the retrieved sequence would not converge to the stored sequence. However, unlike the autoassociative case where convergence can be easily distinguished from divergence, in the case of sequences the difference is unfortunately hard to tell by looking at the retrieved patterns.

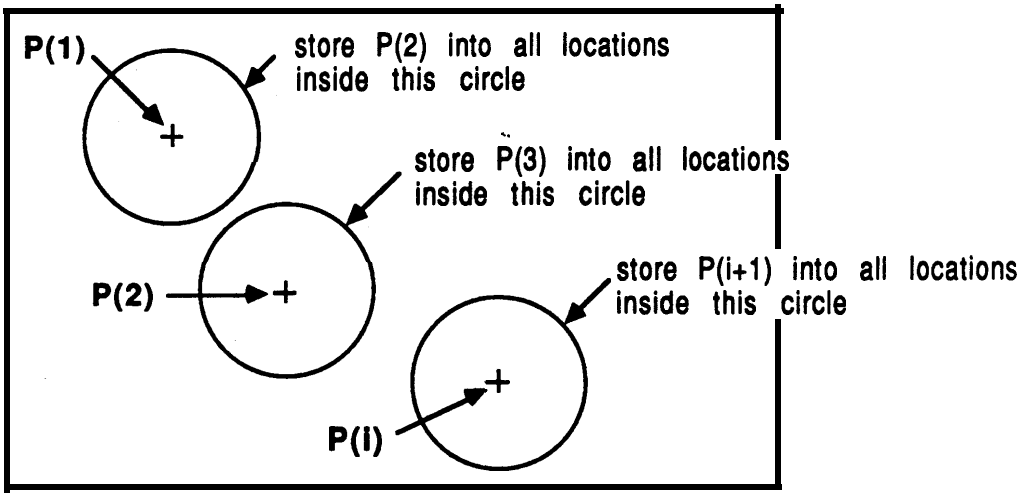


Figure 1.12: SDM heteroassociative mode: storage.

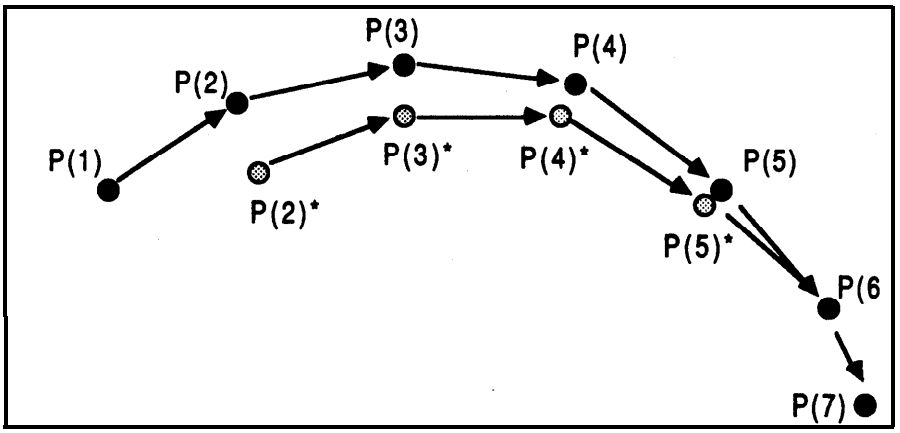


Figure 1.13: SDM heteroassociative mode: retrieval.

1.6.1 Folds

The ability to store sequences endows SDM with the capability to behave as a predictor. The values recovered from stored sequences provide a prediction of the most probable future event. This is illustrated in the following example. Suppose the sequence $A \rightarrow B \rightarrow C \rightarrow D$ occurs more often than the sequence $A \rightarrow B \rightarrow E \rightarrow D$. Suppose, further, that each sequence that the system encounters is written into it in the manner previously described. If the system now encounters $B*$ (close to B), what is likely to happen next? Cueing the system with $B*$ will recover $C*$ (close to C) rather than $E*$ (close to E), because in the locations activated by B there were more copies of C stored than of E, simply because it occurred more often. Thus, the retrieval mechanism predicts the most likely next step in the sequence.

The examples used so far have associated the next element in a sequence with the one before it. This is often insufficient as a basis for prediction. For example, consider the two equiprobable sequences:

$$A \rightarrow B \rightarrow C \rightarrow D$$

$$E \rightarrow B \rightarrow C \rightarrow F$$

Given an event $C*$, we have insufficient information to predict the next event. In fact, to do so we need to look not only one but two steps back in the sequence to know which sequence we are in. SDM handles such situations by utilizing "folds" of different "orders" and combining the results from different folds to arrive at the result. In general, a k^{th}-order fold is a complete set of SDM locations in which sequences are stored with pattern $P(i)$ serving as the address and pattern $P(i + k)$ serving as the contents. More specifically, a first-order fold is one in which the pattern stored is the one that immediately follows the pattern that forms the address. A second-order fold is one in which the stored pattern is the one that follows the address pattern by two steps, and in a third-order fold the stored pattern follows the address pattern by three steps. The two sequences listed above would result in the following storage in each of three folds:

Sequence	$A \rightarrow B \rightarrow C \rightarrow D$	$E \rightarrow B \rightarrow C \rightarrow F$
lst-order fold	at A store B	at E store B
	at B store C	at B store C
	at C store D	at C store F
2nd-order fold	at A store C	at E store C
	at B store D	at B store F
3rd-order fold	at A store D	at E store F

Figure 1.14 shows how to use multiple folds to arrive at a prediction. Imagine

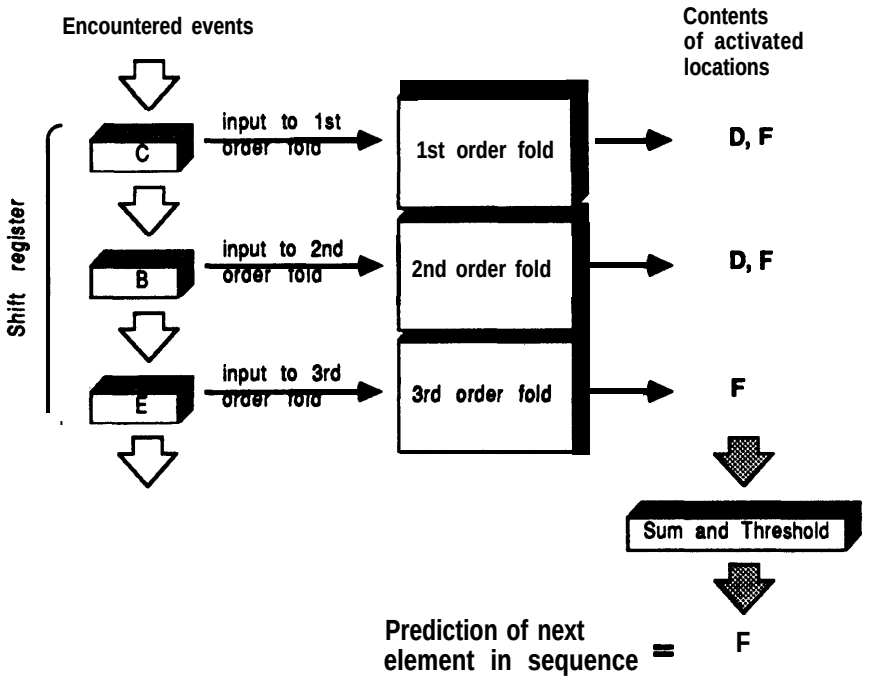


Figure 1.14: Folds.

that the system has previously encountered the sequences $A \rightarrow B \rightarrow C \rightarrow D$ and $E \rightarrow B \rightarrow C \rightarrow F$ in an equiprobable way and that it has stored patterns into its three folds in the manner shown above. Now, assume that the system encounters the patterns E, B, and C in that order. The input being encountered is fed into a mechanism like a shift register in which each register holds a pattern. The shift-register contents are used as input cues to successively higher order folds. In this case, the most recent input pattern C is used as an input cue to the first-order fold, ***B*** is used as a cue to the second-order fold, and ***E*** is used as a cue to the third-order fold. To derive a result, the standard summing and thresholding operation is performed on the contents of *all* the locations activated, not fold by fold. The locations activated by the cue C in the first-order fold have had an equal number of ***D*** and ***F*** patterns written into them, as have the locations activated by the cue ***B*** in the second-order fold. The cue ***E*** activates locations in the third-order fold that have only had ***F*** patterns written into them. The result of summing all of these together and then thresholding is that the pattern ***F*** is recovered with high probability. This pattern is the system's prediction of the next event that is likely to occur.

Chapter 2

Description of the Prototype

This chapter provides an overview of the physical and functional design of the SDM prototype. It explains how the concepts introduced in Chapter 1 are implemented and acts as a introduction to the more detailed descriptions of the hardware and software presented in Chapters 3 and 4, respectively.

Some of the parameters of the prototype design were described in Section 1.4.4 on page 11. Specifically, input vectors (referred to as "words" from here on) are 256 bits long. The system implements 8,192 hard addresses (locations). Words written into the system are accumulated, bit wise, into buckets that hold an S-bit binary count. These and some performance characteristics of the system are summarized below in Table 2.1.

The prototype system was designed around four modules. This modular approach provides the flexibility to modify memory parameters for the present project and makes it easy to upgrade specific portions of the system in future designs. Figure 2.1 shows a physical and block diagram of the system, while Table 2.2 shows the hardware and

Word size:	256 bits
Counter size:	8 bits
Capacity per fold:	8192 words
Number of folds:	1 to 16
Hamming Radius:	0 to 255
Number of reads or writes per second:	50

Table 2.1: Parameters of SDM prototype.

Module	Underlying Hardware	Underlying Soft ware
Executive Module	Sun 3/60 workstation, or any microcomputer with SCSI port	Custom C-code
Control Module	M-68020 based single-board computer	Assembly language code
Address Module	Custom design using LSI corn-ponents on wire-wrap board	None
Stack Module	M-68000 based single-board computer (1 per fold)	Assembly language code

Table 2.2: Hardware and software components for prototype system.

software used to implement each module.

Each of these modules is described briefly below, and the description is expanded upon later in this chapter.

2.1 Physical description

The Executive Module provides the interface between a user or application and the rest of the system. The system's "focus," described in the Introduction, resides in the Executive Module. It is a software module that resides on a Sun 3/60 workstation. It communicates with the rest of the system via the Small Computer Systems Interface (SCSI) port on the Sun. The Executive Module software is written in C and is designed to be easily portable to any microcomputer with an SCSI interface. The rest of the system, namely the Control, Address and Stack Modules, reside in a custom card cage. The Control Module acts as the link between the Executive Module and the rest of the system and controls the operation of the Address and Stack Modules. It is implemented on a single-board microcomputer based on a Motorola 68020 microprocessor. The board has 4 MB of random-access memory. The operating program is written in assembly language. The Control Module communicates with the Executive Module via an SCSI bus, and with the Address and Stack Modules over a VME bus that links all modules on the card cage.

The Address Module performs the task of determining which Hard Addresses are within the specified Hamming distance of the Reference Address. Because of the compu-

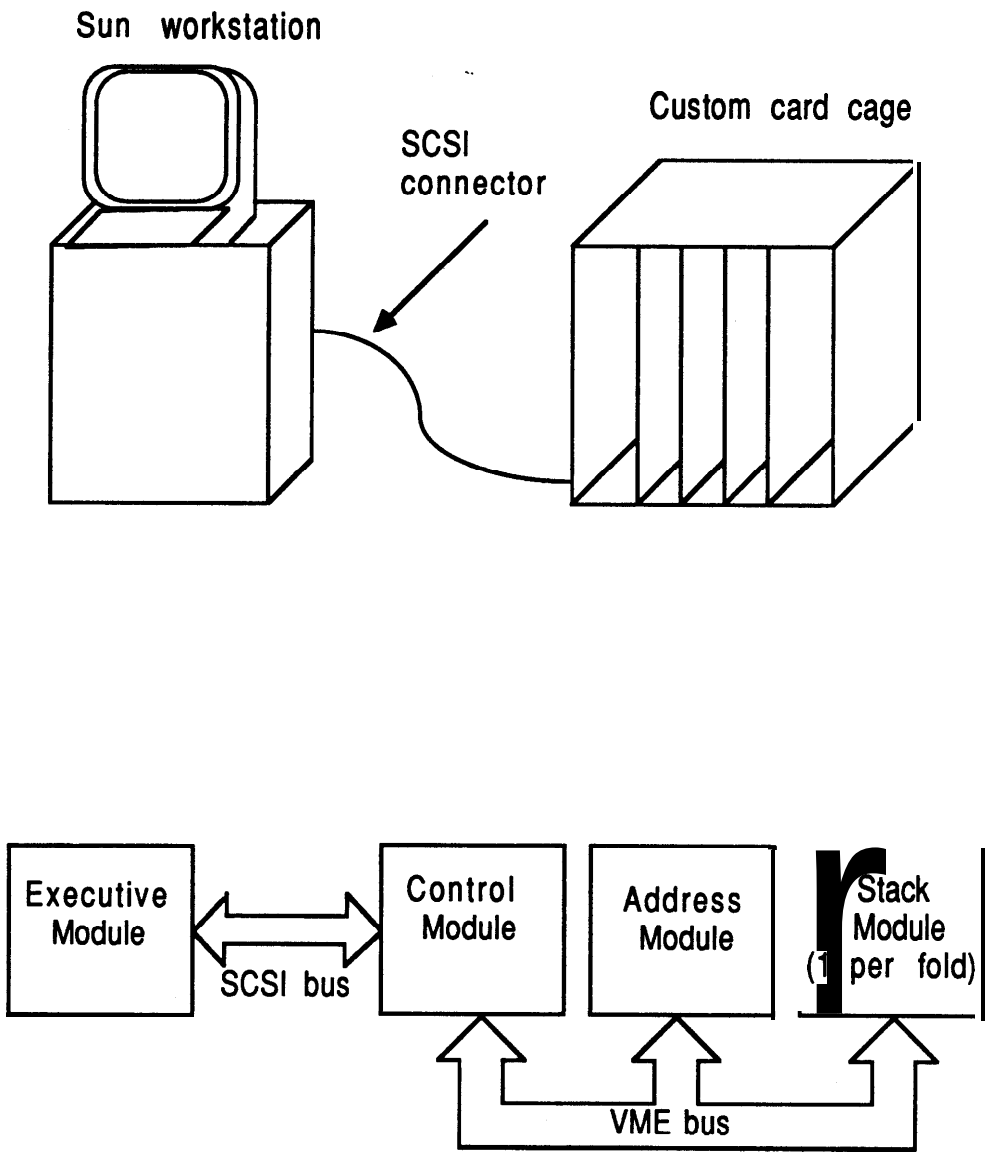


Figure **2.1:** Physical and block diagrams of SDM prototype.

tational intensity of this task, the Address Module is custom designed and implemented on a wire-wrap board. The Address Module is the only custom designed piece of hardware in the entire system.

The Stack Module holds the contents of the folds. Each fold is implemented on a Plessey Microsystems Motorola 68000 based single-board computer with 4MB of random-access memory. Each counter is implemented as a byte (8 bits) in the memory space of the microprocessor, with the task of writing into the counter or reading from the counter being performed by the processor. Since the contents of a location consist of 256 counters, we use 256 sequential bytes to implement the 256 counters associated with each location.

2.2 Functional description:

2.2.1 How the Address and Stack modules implement SDM locations: The concept of "Tags"

In a Sparse Distributed Memory, each Hard Address has associated with it a set of counters in which to store words that are written into that location. In our implementation, the Hard Addresses are stored on one board (the Address Module) while the counters associated with them are stored on a separate board (the Stack Module). The one-to-one association is maintained via a 13-bit "Tag" that associates a particular Hard Address on the Address Module with a particular set of 256 byte-sized counters on the Stack Module. The conceptual arrangement is shown in Figure 2.2.

Thus, when the Address Module determines that a particular Hard Address is within the cut-off Hamming distance of the Reference Address, it simply passes the 13-bit Tag associated with that Hard Address to the Control Module. The Control Module in turn passes the Tag to the Stack Module, which uses the Tag to uniquely identify a set of 256 bytes that hold the contents of the location associated with the Hard Address in question. Physically, the Address Module stores the Hard Addresses in a set of 32 static RAMs, each of which is 8K $\times$ 8 bits. Since these RAMs are operated in parallel, they behave like a set of locations that is 256 bits wide and 8192 deep, which is what we need to implement 8192 Hard Addresses. The arrangement is shown in Figure 2.3.

As described earlier, the Stack Module implements the 256 counters associated with a location with 256 consecutive bytes in the address space of the M68000 processor. Thus, the counter values associated with the Hard Address with Tag 0 would be stored in bytes 0 to 255. For an arbitrary Tag N, ($0 \leq N \leq 8191$), the associated bytes would be at addresses (N * 256) to (N * 256 + 255). (The exact mapping is conceptually

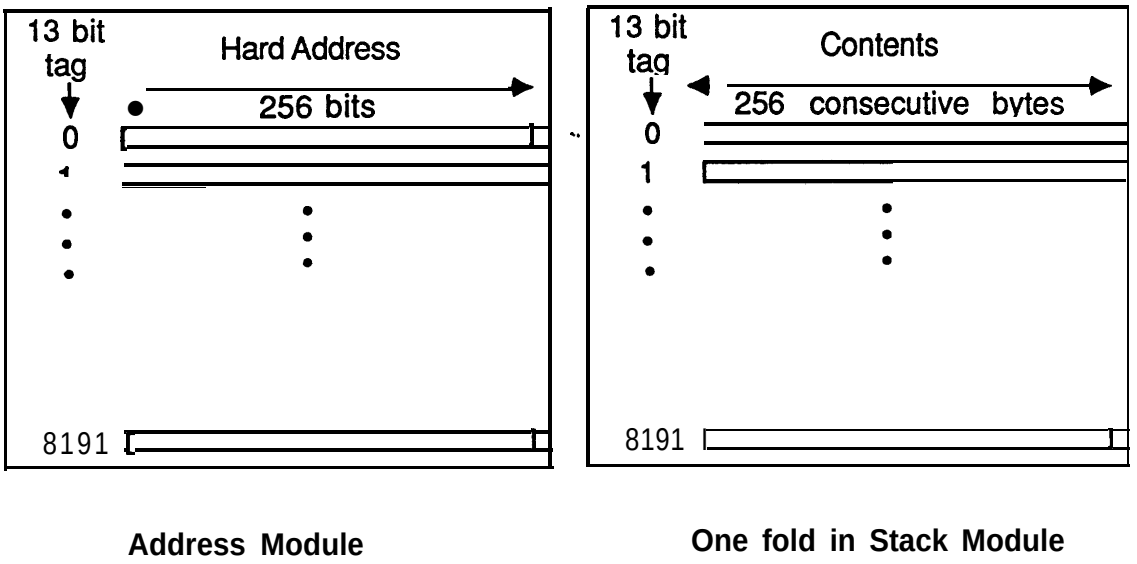


Figure 2.2: Relation between Address and Stack modules.

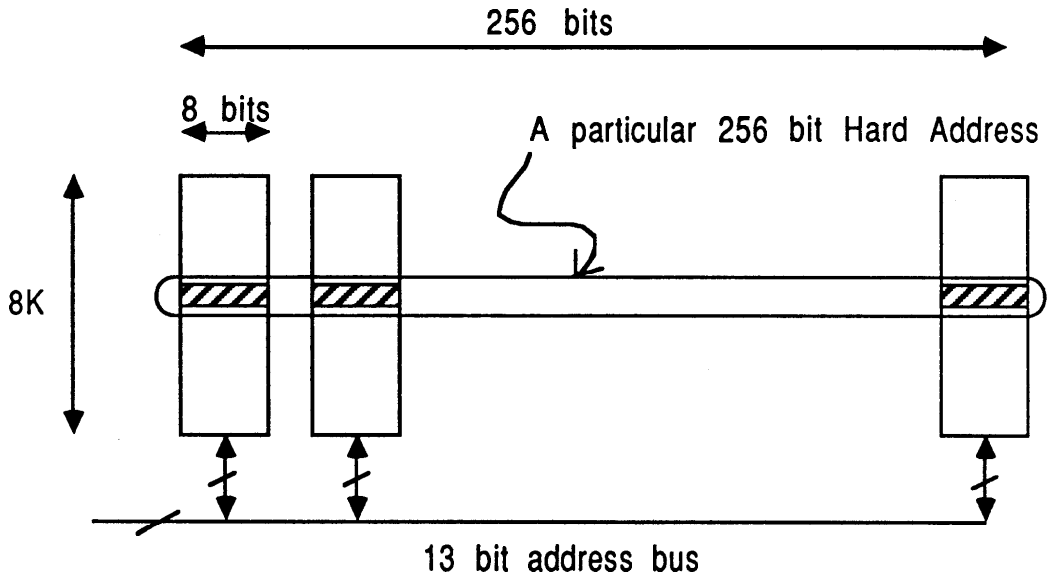


Figure 2.3: Physical arrangement of hard addresses in Address module.

identical, though the locations in memory space are a little different).

2.3 Operational description

In order to explain how the prototype works, the description is divided into a set-up and operating phase. Each of these is explained below.

2.3.1 Set-up

The set-up phase consists of loading a set of up to 8192 Hard Addresses into the Address Module. The user determines what Hard Addresses to use for a particular application. For example, a set of 8192 randomly distributed Hard Addresses could be generated by rolling a die 256 x 8192 times and converting the results into a sequence of 0s and 1s by viewing any roll of 3 or less as a 0 and any roll of 4 or greater as a 1. The results of every set of 256 consecutive rolls could be used as a Hard Address.

Once the user has determined the Hard Addresses to be used, the Executive Module passes each Hard Address and an associated Tag to the Control Module, which in turn passes them to the Address Module, which writes the Hard Address into the physical location identified by the Tag. (If one were starting a fresh application, there would be no reason to not load Hard Addresses into sequential locations in the Address Module, and passing the Tag from the Executive Module would be redundant. However, this ability to load a Hard Address at a particular Tag location is useful for testing performance characteristics and for debugging. For example, it allows one to change a particular Hard Address by simply storing a new one into the same Tag location.)

2.3.2 Operation

During its operating mode, the SDM system either reads or writes. How each module works to achieve this is explained below, first for a write and then for a read.

(a) SDM Write

1. The Executive Module passes the 256 bit Reference Address and the 256 bit "data" word to the Control Module, as well as the cut-off Hamming distance to use.

2. The Control Module passes the Reference Address and cut-off Hamming distance to the Address Module, and the data-word to the Stack Module.

3. The Address Module sequentially calculates the Hamming distance between each Hard Address and the Reference Address and compares it to the cut-off Hamming distance. Whenever it finds a distance less than the cut-off, it passes to the Control Module both the Tag of that Hard Address and the Hamming distance it calculated.

4. Whenever the Control Module receives a Tag from the Address Module, it passes the Tag to the Stack Module and both the Tag and Hamming distance to the Executive Module. (The latter is to study performance characteristics of applications) .

5. Whenever the Stack Module receives a Tag from the Control Module, it performs 256 integer adds. Each add consists of adding either +1 or -1 into sequential bytes in memory, depending on whether the associated bit in the data-word is 1 or 0. A code segment to explain what happens is shown in Figure **2.4.**

(b) SDM read

1. The Executive Module passes a Reference Address and a cut-off Hamming distance to the Control Module.

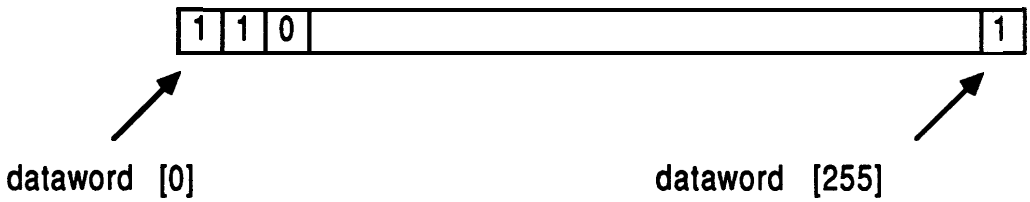
2. The Control Module passes both of these to the Address Module.

3. The Address Module performs exactly the same operations that it does during a write. Namely, it passes back to the Control Module the Tag and Hamming distance for every Hard Address Hamming distance from the Reference Address is less than the cut-off Hamming distance.

4. The Control Module performs the same operations as during a write. Received Tags and Hamming distances are passed to the Executive Module, while the Tags alone are also passed to the Stack Module.

5. The Stack Module establishes a 256 element integer array to hold the results of its operations. Every time it receives a Tag from the Control Module, it performs 256 integer adds; each add consists of adding a byte-sized counter value into an array element. The operation is shown in Figure 2.5.

6. When the Address Module has gone through all 8192 Hard Addresses and the Stack Module has performed its accumulation task for every selected Tag, the Stack Module sends to the Control Module the results of its accumulations (i.e., Result[O] to Result[255] from Figure 2.5).




```
For each tag received from Control Module do :

begin

  StartingByteAddress := Tag . 256 ;
  for J := 0 to 255 do
      begin
      M := StartingByteAddress + J ;
      byte [M] := byte [M] + ( 2 * dataword [J] - 1 ) ;
     end   ;

end ;
```

The J^{th} bit of the data word is accumulated into the J^{th} byte of each selected location. A '1' bit in the **dataword** causes the count to go up by 1, while a '0' bit causes the count to go down by 1.

Figure 2.4: What the Stack module does during a write operation.

```
Result :  Array [0..255] of Integer (* an array of integers *)


For each tag received from Control Module do :

begin

  StartingByteAddress := Tag * 256 ;
  for J := 0 to 255 do
      begin
      M := StartingByteAddress + J ;
      Result [J] := Result [J] + byte [M] ;
     end   ;

end ;
```

Figure 2.5: How the Stack module accumulates the contents of locations selected by the Address module during a read operation. The M^{th} counter of each selected location is accumulated into the M^{th} element of the array "Result" for J from 0 to 255.

7. The Control Module thresholds each result to either 0 or 1, depending on whether it is negative or positive. It then constructs a 256 bit data-word from the thresholded results. This data-word is returned to the Executive Module as the result of the read operation.

2.4 Summary

The prototype has been designed to provide a significant improvement in performance over software simulations of sparse distributed memory systems, while maintaining a high degree of flexibility. Those goals have been achieved by dividing the system into four modules, and by using standard subsystems (e.g., single-board computers) and software based implementations wherever feasible. One module contains a custom-hardware-based design. This was necessary in order to achieve the desired speed in the critical task of calculating and comparing Hamming distances.

The next chapter describes the hardware, and in particular the custom-designed Address Module, in more detail. Chapter 4 provides more details on the software.

Chapter 3

Hardware Description

This chapter describes the details of the hardware portions of the SDM system in the following order:

- The executive model
- The control module
- The stack module
- The address module

The address module is described in the most detail since it uses custom hardware required to meet the performance specifications.

3.1 The Executive module

The primary requirements for the executive module are:

- Provide a user interface to the SDM;
- Implement a high-speed communications protocol to the other portions of the SDM;
- Provide a common programmer interface;
- Handle the computation requirements for program modules that don't use the SDM.

The equipment that best fits the above requirements is the workstation. In particular, workstations provide an excellent user interface, have a programming interface that most researchers are comfortable with, and handle the computation requirements. Coincidentally, SDM simulators have been written for workstations; researchers who are familiar with these can easily assimilate themselves into this implementation.

One component that is of critical importance is the communications interface between the executive module and the SDM. The communications system must support the following operations:

1. Debugging. The entire contents of the SDM, including the Stack, Address, and Control Module memory, should be transferrable in a reasonable amount of time.
2. Setup. The contents of the Hard Address Memory, to be located on the Address Module, should be quickly downloadable.
3. Operation. Given the operational requirement of 50 read/write operations per second (designed with a safety margin of ×2), the system should support a burst transfer rate that will allow transfer of the 256-bit reference address, an operator, and up to 100 16-bit tags (13-bit tags are passed as 16-bit words) and 256 16-bit sums per fold back to the executive module 100 times per second.

Of these, requirements 1 and 2 are of convenience; clearly, we wish to keep waiting while debugging and setting up to a minimum. However, requirement 3 places a real bound on the communications subsystem.

Assuming the worst case for a single-fold system, we must transfer 50 reference addresses, 50 instructions, 50 * 100 16-bit tags, and 50 * 256 16-bit sums per second. This translates into:

$$C = 50 * (256 + 8 + 100 * 16 + 256 * 16) = 298{,}000 \text{ bits/second}$$

Again, assuming worst case, we allow for a protocol overhead of 50%:

$$N = 2C = 596{,}000 \text{ bits/second}$$

for a single-fold system. Additional folds add 409,000 bits/second/fold.

One communications system that would easily meet this requirement has already been mentioned: a bus-to-bus mapping switch. Clearly, the transfer rate of such a device greatly exceeds our requirement. However, the use of a bus map introduces an enormous amount of inflexibility into the SDM; in particular, preferential addresses will probably be already taken by the workstation. The use of a bus map would also require

that the bus used by the executive module be the same as that used by SDM. This limits flexibility of choice of the executive module, and may place severe structural constraints on the SDM.

A solution to this problem is to use an 'existing communications protocol that is simple and provides the required bit rate. In particular, it helps to view the SDM as a smart disk drive that is attached to the executive module. Most vendor-independent disk protocols provide read and write operations and support some level of control-message handling. Moreover, most workstations support some form of vendor-independent disk protocol.

For our implementation, we chose the Small Computer Systems Interface protocol, or SCSI, to provide communications between the Executive and the other modules of the SDM. SCSI supports a burst rate of 1.5 Mbps and provides a reasonable number of control functions. Because of the large number of workstations that support SCSI, this implementation of the SDM system may be attached to almost any workstation, mainframe, or even a personal computer.

Thus, the requirements for the Executive module indicate a powerful workstation that supports the SCSI protocol. Because of our familiarity with the products of Sun Microsystems, the current implementation uses a Sun 3/60 color workstation, with the SDM attached to one of the SCSI ports. We emphasize, however, that ***any system that supports the SCSI protocol* is *capable of using the SDM as an attached processor.***

3.2 The Control module

The Control Module, or CM, acts as the interface between the executive module and the rest of the SDM. It manages the operation of the Address Module and transfers tags from the AM to the appropriate Fold in the stack module.

The choice of a processor for the CM was influenced heavily by two considerations:

1. The need for a large (greater than 24 MB) address space;

2. The availability of software tools.

The large address space mandates the use of the VME-bus standard interconnect, which in turn favors the Motorola 68000 family of processors. In addition, we had software tools available for this processor family. The memory-to-memory transfer bandwidth mandates the use of a 32-bit processor; the MC68020 fills all of these requirements.

Thus, the requirements for the CM are as follows:

1. Motorola 68020, 15 MHz or greater clock speed;
2. A large quantity (about 4MB) of dynamic RAM, to hold reference tables and Hard-Address-to-Tag translation maps;
3. Dual-Ported memory, to support asynchronous transfers;
4. An SCSI port, to interface back to the executive module.

A vendor search indicated that Plessey Microsystems would be able to deliver a board with these specifications, in addition to one that matched the requirements for the folds, which will be discussed later.

The actual functionality of the CM is implemented in software; however, there are a few features of the CM hardware that particularly facilitate the SDM:

1. Address Module simulation: Because of the large quantity of memory available, a significant number of simulated AM operations can be stored on the CM for debugging purposes.
2. Fold Downloading: The entire state of a single fold may be stored on the CM; this allows us to directly manipulate the counters, and set up each fold in a predetermined state.

3.3 The Stack module

The Stack Module consists of a number of submodules, known as Folds. Each fold is independent; thus, the design of the SM is simplified by considering it as made up of a number of similar submodules. The requirements for each fold are:

1. Enough memory to hold an eight-bit count (i.e., one byte) for each bit in every word of the Hard Address Memory. Practically, this implies a need for 256 * 8,192 = 2MB of memory;
2. The memory must be dual ported for debugging and initialization purposes;
3. A processing system that will be able to perform integer byte adds quickly enough to satisfy the 50 operations/second requirement.

These requirements fall within the capabilities of a single-board computer, as long as the processor on the board is fast enough to perform the necessary integer adds.

Because the machine deals largely with byte adds and 16 bit references (for translating Tags into counter addresses), a fast 16-bit processor will suffice.

To this end, considering the implementation decisions for the Control Module, we again selected a processor card based on the Motorola MC68000 architecture. Plessey Microsystems was able to provide a board with a 16 MHz MC68000, and 4MB of dual ported dynamic RAM, that was less expensive than their MC68020 product. The additional memory can be used to implement the Fold software, or it could be used to implement 16 bit counters (the Fold software would then be contained within two 8KB RAM/ROM positions which are open on the card).

3.4 The Address module

The Address Module, or AM, is the only custom component of the SDM. It will be described in considerably more detail than the other components. As mentioned in the architecture section, more than one AM may be used within the SDM if the performance increase is required. The reason for a custom implementation lies in the fundamental operation that the AM performs, which is a 256-bit Hamming distance calculation.

It should be stressed that a 256-bit word is very large. In order to fully appreciate the difficulty of designing a processor with 256-bit internal data paths, consider the following:

1. If one designs the data-path elements with byte-wide, commercially available products, each element in the data path requires 32 chips.

2. Each link in the data path requires 256 connections; a four-element data path will easily require in excess of 1,000 wire wraps.

These two numbers (components and connections) will quickly outstrip the capacity of most boards and development systems; our system will fit only on a 400 mm high by 366 mm wide 9U sized VME bus card, which is the largest one available.

3.4.1 Overview and board floorplan of Address module

The Address Module appears to the control module as a set of sequential memory locations on the VME bus. These addresses are defined as follows:

Register	Longname	[width]	R/W	#32-bit Regs	R#
TR	Tag Register	[16]	W	1	0
Associates a 16 bit "Tag" with a hard address, by setting the address in the Hard Address Memory in which a Hard Address is stored.					
HAR	Address Register	[256]	W	8	1-8
Enters a Hard Address into the Hard Address Memory.					
MR	Mask Register	[256]	W	8	9-16
Sets the Hamming AND mask.					
RAR	Ref. Address Register	[256]	W	8	16-24
Sets the Reference Address-the Address in Question.					
LR	Limit Register	[8]	W	1	25
Defines the radius of the Hamming Sphere. Any tag with Hamming distance less than or equal to this will be written to TCO.					
SR	State Register	[8]	R/W	8	26
The bits in this register are wired to the state flipflops of the machine.					
TCO --.	Tag Cache Output	[24]	R	1	27
256-word cache that holds the accepted tags.					

The bits in the state register are defined as follows:

Run Modes:	Bit1	Bit0	
	0	0	Reset
	0	1	Running
	1	0	Hit
	1	1	Wait
Reset Control:	Bit2		
	0		Run (or Hit or Wait)
	1		Reset
Cache Indicator:	Bit3		
	0		TCO empty
	1		TCO not empty
Mode:	Bit4		
	0		Uncomplemented Addresses
	1		Complemented Addresses

Bits in the tag cache output are defined as follows:

Bit7 – Bit0 Hamming Distance
Bit23 – Bit8 Tag ID

Figure 3.1 is the board floorplan for the AM. It also names the six submodules of

the AM, namely:

1. Clock and Sequencer: This submodule contains the master state machine and the master clock.

2. Hard Address Memory. All 8,192 256-bit addresses are contained in this memory, which consists of 32 8KB static RAMs.

3. Arithmetic Logic Unit: This submodule performs the 256-bit exclusive-OR and mask operations. The Reference Address Register and Mask Register, each consisting of 32 74ALS373 8-bit latches, are combined here with a Hard Address by a logic unit, which consists of 32 programmable logic arrays (PLAs). The ALU's result is a 256-bit quantity with 1s in those positions where the Reference Address differs from the Hard Address, in those positions where the Mask bit is 1. The ***number*** of 1s in this result is the Hamming distance between the Reference and Hard Address. (The Mask is a 256-bit user-specified pattern to restrict the Hamming distance calculation to a subset of the 256 bits, if desired.)

4. The Bit Counter. The bit counter calculates the Hamming distance by essentially adding 256 one-bit quantities, and then compares the result with the Limit Register. If the result is less than the Limit Register, we have a "hit," and both the result and the Tag associated with the particular Hard Address are written into the Tag Cache.

5. Tag Cache. During a read or write cycle, when the Address Module finds a Hard Address whose Hamming distance from the Reference Address is within the limit prescribed by the Limit Register, it is supposed to send the Tag associated with that Hard Address, as well as the Hamming distance, to the Control Module. The Tag Cache allows this operation to occur without the need for synchronization between the Address Module and the Control Module.

   When the Bit Counter discovers a Hit, it writes the Tag and its associated distance into the Tag Cache. The Control Module reads this information out of the Tag Cache. Thus, the Tag Cache acts as a First In-First Out buffer between the Address Module and the Control Module. It has space for holding up to 256 Tags and their associated Hamming distances.

   Mechanisms described later prevent the Tag Cache from overflowing. Essentially, when the Tag Cache is full, the ALU and Bit Counter are halted until the Control Module has cleared some space in it. This condition is rarely expected to occur since the average number of hits per read or write operation is expected to be less than 100.

6. Bus Interface. Finally, the bus interface provides. address mapping, power, and control information to the AM from the VME bus.

The signals common to most of the submodules are referred to as global signals and are detailed in Figure 3.2. Note that there **are only** two 256-bit global signal groups; these constitute the main data path of the AM.

Figure 3.3 details the state transitions of the Address Module. The four processor states are as follows:

0. **Reset.** This is the "normal" power-up state of the AM. The AM may be reset and held in this state by setting the reset bit in the Status Register.

1. Run. The AM enters this state from a reset when the reset bit in the Status Register is zeroed. It continues to loop within this state until a hit (Hard Address "close" to Reference) occurs or until the Hard Address Memory is exhausted.

2. **Store.** When a hit occurs, the AM must transfer the Tag associated with the Hard Address, and the distance, to the Tag Cache.

3. Wait. This state is entered only if the Control Module allows the Tag Cache to fill up. For the most part, the total number of hits per operation will be much less than the size of the tag cache. This state is provided as a safety measure and should be useful for experimentation and debugging.

The timing diagram in Figure 3.4 indicates the expected sequence of events during a machine cycle in each of the AM states. The longest state to execute is a Run state, which takes about 750 ns. Variable clock-cycle states were not implemented, since the AM spends most of its time (10,000 cycles out of 10,100, or 99%) doing Run cycles. This is well within the required cycle time of 1.0 microsecond, and a 1.0 MHz clock is used in the current version.

We will now examine each submodule of the AM in detail.

3.4.2 The clock/sequencer

The clock/sequencer implements the state machine described above. It consists of a pair of J-K flip-flops, logic to implement the state transitions, and a 1.0 MHz clock. It also contains the Reset flip-flop.

All of the flip-flops used in this submodule are both readable and writeable from the CM, via the Status Register. This allows the state of the AM to be determined, for

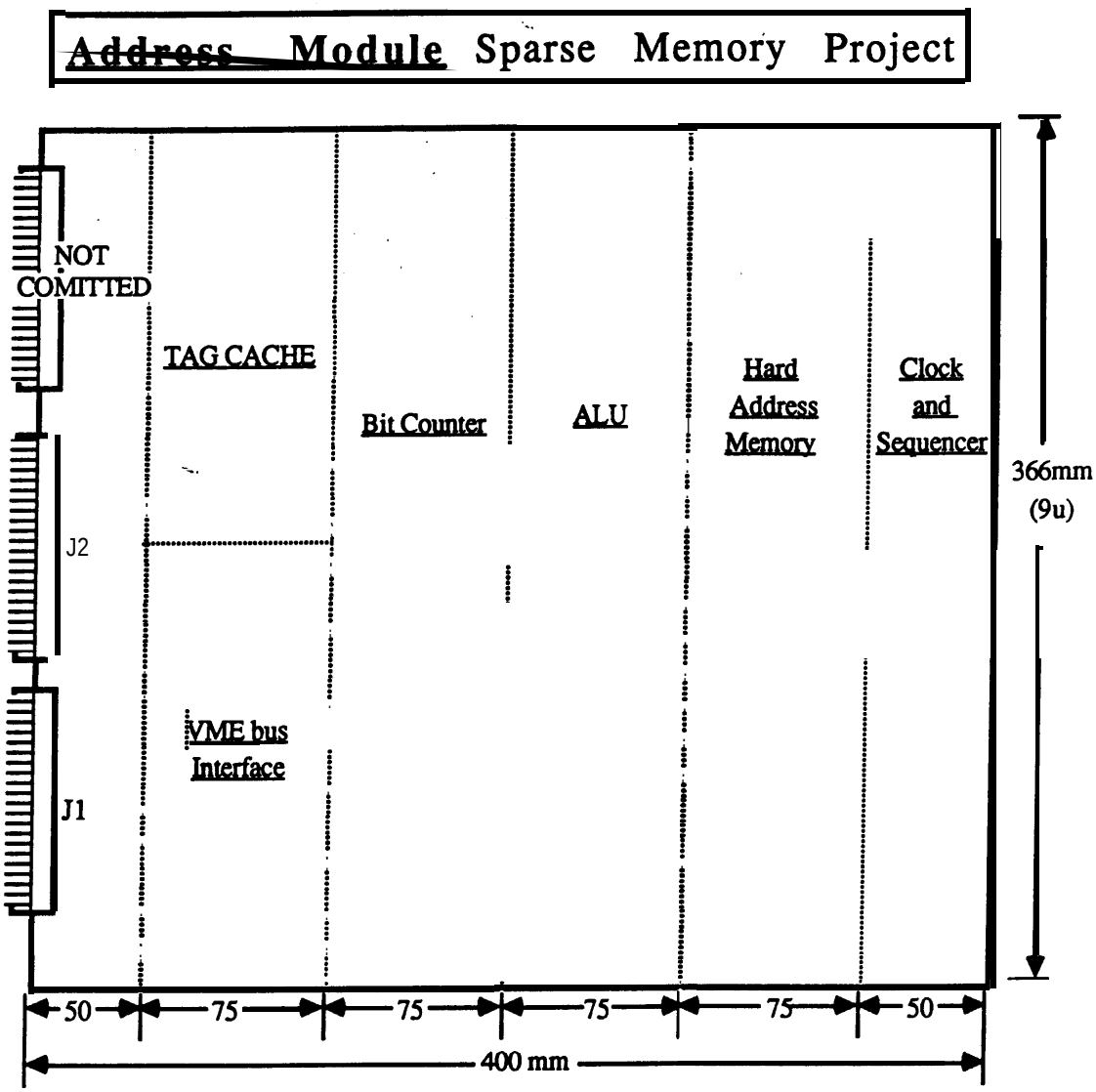
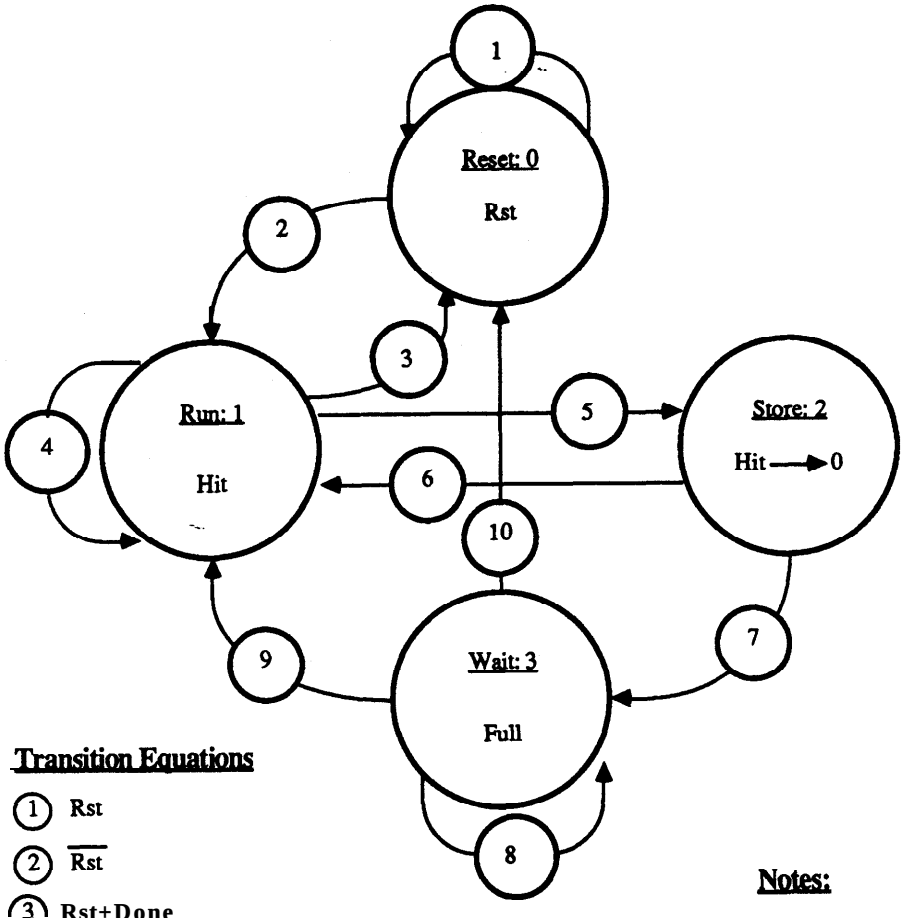


Figure 3.1: **Floorplan.**

Bus Name	Bus Definition
Reg[00]-Reg[1F]	Register Address Lines
	Reg 00 :: TR
	Reg 01-Reg 08 :: HAR
	Reg **09-Reg** 10 :: **RAR**
	Reg 11-Reg 18 :: **MR**
	Reg 19 :: LR
	Reg **1A** :: SR
	Reg **1B** :: TCO
	Reg **1C-Reg 1F**:: Undefined
Xd[OO]-**Xd[1F]**	32 bit bidirectional Data Bus
Yd[00]- Yd[FF]	256 bit HAM to ALU Data Bus
Zd[00] - Zd[FF]	**256** bit HAM to BC Data Bus
D[0] - D[7]	8 bit **Haming** Distance Data Bus
T[0] - T[F]	16 bit Tag on Data Bus
S[0] - S[3]	**4** bit State Bus

Flag Name	Flae Definition
Hit	HAR within LR of RAR
Rst	**Global Reset**
Not	**Use complemented HAR**
Full	**TC"full"**
Hrw	**HAM Read / $\overline{\text{Write}}$**
Tr	**Tag Cache Read**
Tw	**Tag Cache Write**

Figure 3.2: Global signals of the Address module.



Transition Equations

1. Rst
2. $\overline{Rst}$
3. Rst+Done
4. $\overline{Hit * Rst * Done}$
5. Hit*Rst
6. Full
7. $\overline{Full}$
8. Full * $\overline{Rst}$
9. $\overline{Full}$ * $\overline{Rst}$
10. Rst

Notes:

A All stste transitions occur on the positive clock edge.

B Hit & Full inhibit tag counter incremented

Figure 3.3: State-transition diagram of Address module.

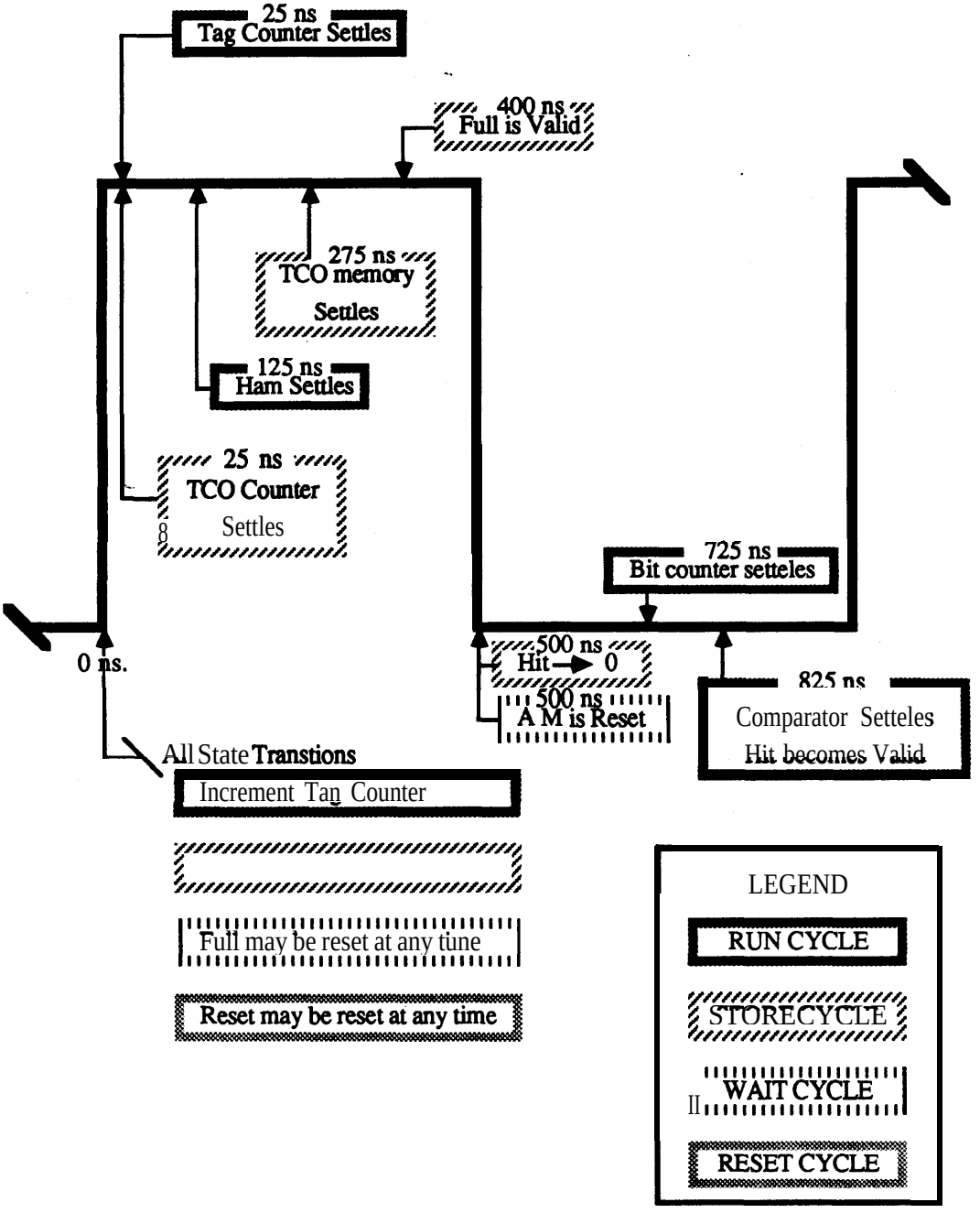


Figure 3.4: Timing diagram of Address module.

debugging purposes. In effect, the state machine delivers a hardwired "instruction" to the rest of the AM; note that the four states correspond to programming statements, while the state-transition logic defines the transfer of control in the microprogram.

The clock/sequencer submodule also contains a 13-bit cyclical counter, which is used to address the Hard Address Memory during a Run cycle.

3.4.3 The hard address memory

The Hard Address Memory stores all 8,192 256 bit hard addresses, which define the memory distribution. It consists of 32 6264LP-10 8k-by-8 bit static CMOS RAMs, along with the associated internal bus-switching logic, which demultiplexes the input/output of the RAMs. We also looked at implementing 16,384 hard addresses, but that would have added 32 RAMs to the board, exceeding the available board space. An earlier goal of 10,000 hard addresses was given up as unfit for available chip sizes.

Associated with each 256-bit Hard Address is the 13-bit address in which it is stored in the Hard Address Memory. This 13-bit quantity is referred to as a Tag. To load the Hard Address Memory, the Tag Register (implemented with two 74ALS373 8-bit latches) is set to the desired Tag, and the Hard Address associated with the Tag is loaded into the Hard Address Register (which is directly written into the RAMs).

3.4.4 The ALU

The ALU consists of a Reference Address Register, a Mask Register, and a logic array to perform a 256-bit XOR-AND operation. The registers consist of 32 74ALS373 8-bit latches each, and the logic array is implemented in 64 programmable logic arrays. In addition, the ALU submodule houses the "Complement Mode" flip-flop bit. This allows the complementary address space to be searched without reloading the Hard Address Memory.

The resulting 256-bit word is then passed to the Bit Counter. The number of bits in this word that are '1' is the Hamming distance between the Reference Address and the Hard Address in question.

3.4.5 The bit counter

The Bit Counter implements a 256-bit wide one-bit adder in four stages. Each stage was constructed from several 27256 32KB EPROMs. The first stage consists of 17 such EPROMs, with each of the 15 address lines connected to one of 256 one-bit quantities.

The resulting four-bit numbers are then added in the next stage, and so forth. At the end of the fourth stage, the bit count is represented by a single eight-bit number (with 256 mapped to 255).

This eight-bit result is then compared with the Limit Register (a single **74ALS373**); if the result is less than the limit, the Hit flip-flop is set. The reset line for the Hit flip-flop is made available to the Clock/Sequencer submodule, so that it may be complemented once a Hit cycle is completed.

3.4.6 The tag cache

The Tag Cache is activated whenever the Bit Counter detects a Hit or whenever the CM wishes to read its contents. The central component of the Cache is a trio of dual-ported 256 byte static RAMs. One RAM stores the distance, the other two store the Tag.

In operation, whenever the AM is ready to write a Tag, a write counter internal to the Tag Cache submodule is incremented; this counter points to the address in the DPRAM where the Tag and distance will be stored. The Tag and distance are then written to the DPRAM. Whenever the counter reaches 255, the Full flip-flop is set.

When the CM issues a read request from the Tag Cache Output register, a read counter is incremented. The contents of the DPRAM at that address is then sent back to the CM via the VME bus. When the read counter reaches 255, the Full flip-flop is reset. In normal operation, the Full flip-flop is never set because the read and write counters are reset at the start of an SDM read or write and a normal SDM read or write results in fewer than 255 hits.

3.4.7 The bus interface

Finally, the Bus Interface submodule handles all of the VME-bus addressing for the AM, and internal to external bus translations. It consists of a set of bus transceivers, along with a demultiplexor to generate all 28 register address lines, and supplies the necessary power to the board.

3.4.8 Operation of the Address module

The AM operates in two distinct modes. Before an actual application is run on the SDM system, specific Hard Addresses must be loaded into the AM. This in accomplished in a “set-up” mode. During this mode the AM is in the Reset-state shown in the State Transition Diagram, Figure 3.3.

Once set-up is complete, applications can be run on the system. This occurs in "operating" mode, during which the AM can be in any one of three states, namely Run-state, Store-state, and Wait-state, depicted in Figure 3.3.

A single read or write for an application causes the AM to:

1. Loop through the Run-state 8192 times. During each loop, the Reference Address is compared to one of the Hard Addresses, and the Hamming distance between them is calculated and compared to the value in the Limit Register.
2. Visit the Store-state once for every hit encountered during a Run-state.
3. Visit the Wait-state whenever the Tag cache gets filled. As described above, this will rarely happen, since the Tag Cache can hold the results of 256 individual Hits.

The details of what happens in the two modes are described below.

Set-up mode:

1. The CM sets the Reset bit in the Status Register; this effectively sets the Reset flip-flop.
2. The CM loads the Hard Address Memory. This is accomplished by first loading the Tag Register with the desired Tag, and then loading the Hard Address Register. The Tag Register controls the address lines of the 8KB RAMs; each 32-bit bus transfer loads four of the 32 RAMs.

Operating Mode:

1. The CM resets the Reset bit in the Status Register, and the state machine transitions into the Run state.
2. At the beginning of each new Run cycle, the Tag counter is incremented. The output of the counter is placed on the address lines of the Hard Address Memory. The contents of the Memory are then transferred to the ALU, to be XORed with the Reference Address. The result is then ANDed with the mask, and passed to the Bit Counter. The result propagates through all five stages, and is then compared to the Limit Register. If the result is less than the Limit, the Hit flip-flop is set.
3. If the Hit flip-flop is set, the next cycle becomes a Hit cycle and the AM enters the Store-state. The resulting Hamming Distance is sent to the Tag Cache, along with the contents of the Tag counter. (The Tag counter contains the Tag of the Hard Address that just scored a Hit.) The address in the Cache is then incremented; if the result reaches 255, the Full flip flop is set. The Hit flip-flop is then reset.

4. If the Full flip-flop is set, the next few cycles will be Waits, until the Cache Read counter reaches 255. The Full flip-flop will then be reset.

5. If neither the Hit nor Full flip-flops are set, the next cycle will be a Run cycle. This process continues until all of the Hard Addresses have been checked, at which time the Address Module enters the Reset state.

A complete block diagram of the Address Module may be found in Figure 3.5.

3.4.9 Additional hardware

Beyond the basic board set and workstation, a VME bus card cage and SCSI cable are also needed. The card cage must be able to accommodate at least two 400mm 9U sized cards and more than eleven 6U 200mm cards. The cable required depends on the workstation connectors.

3.5 Summary

A Sparse Distributed Memory system consists of an Executive Module, together with a separate set of modules designated as the SDM. The SDM, in turn, consists of a Control Module, an Address Module, and a Stack Module.

The Executive Module consists of a standard workstation that supports the SCSI communications protocol. The Executive Module is connected to the Control Module via the SCSI port. The CM consists of a fast single-board computer, with a large quantity of dual-ported memory. The CM controls the Address and Stack Modules via a common interface, the VME bus. The Address Module is a custom-made board that performs the Hamming distance calculations for the SDM. The Stack Module consists of several Fold submodules. Each Fold consists of a 16-bit single-board computer, with enough memory to hold 2,097,152 8-bit counters.

This implementation of the SDM should be capable of 50 read/write operations per second. It implements 8,192 hard addresses, each with 256 8-bit counters.

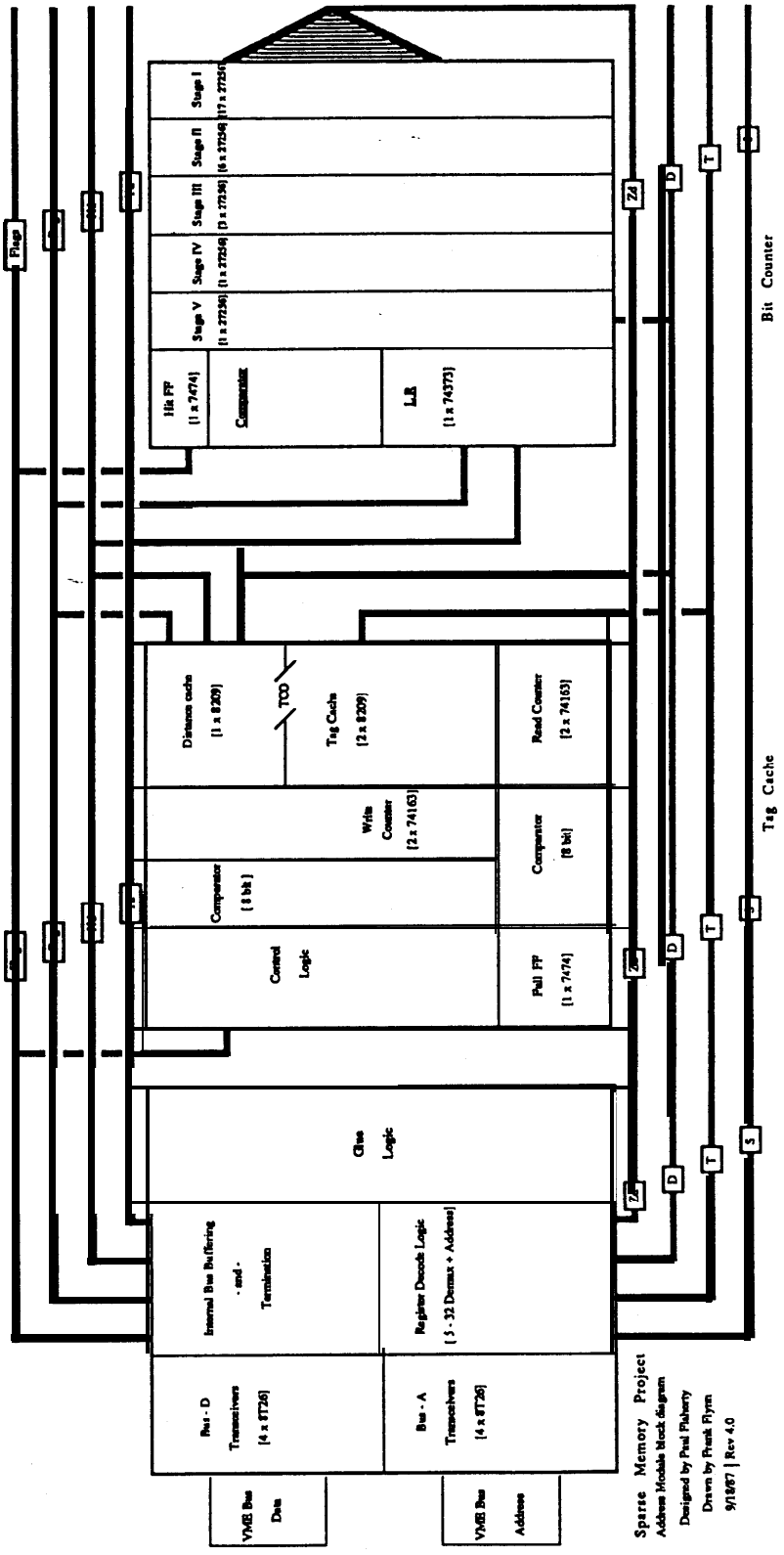


Figure 3.5: Address module block.

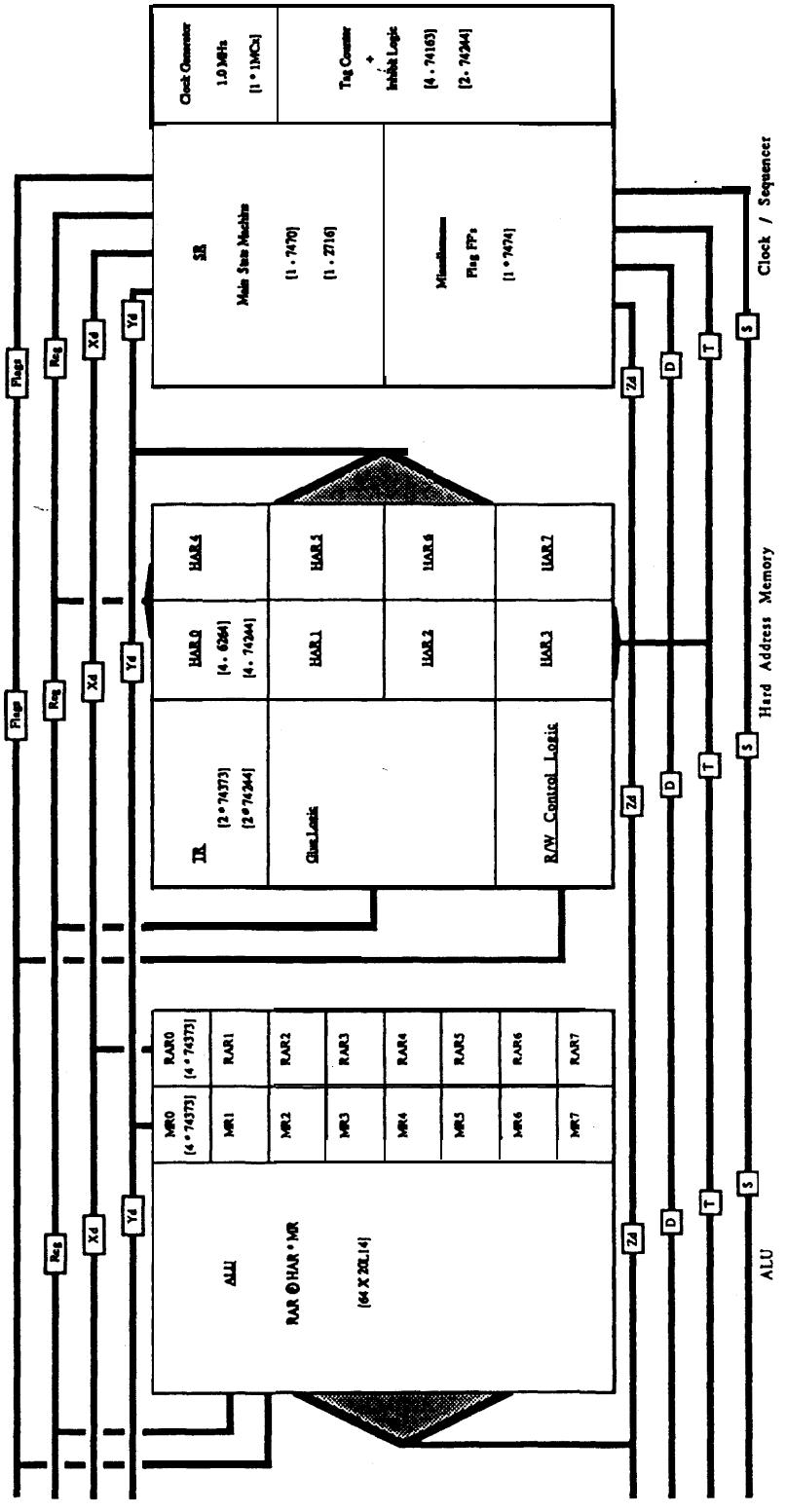


Figure 3.5: Address module block, cont'd.

Chapter 4

Software Design

4.1 Introduction

This chapter describes the software systems that operate in conjunction with the hardware described in Chapter 3. The prototype system is broken down into four modules:

- Executive Module
- Control Module
- Address Module
- Stack Module

The next section describes each module briefly, and then gives a description of how the system works as a unit, including how communications are made between the different modules.

The rest of the chapter takes each module in turn and gives a detailed description of its operation. The user interface is included in the section on the executive module.

4.2 Overview

This section reviews the basic structure of the four modules, as described earlier in Chapter 2. Figure 4.1 shows how the modules are arranged and the communications paths involved.

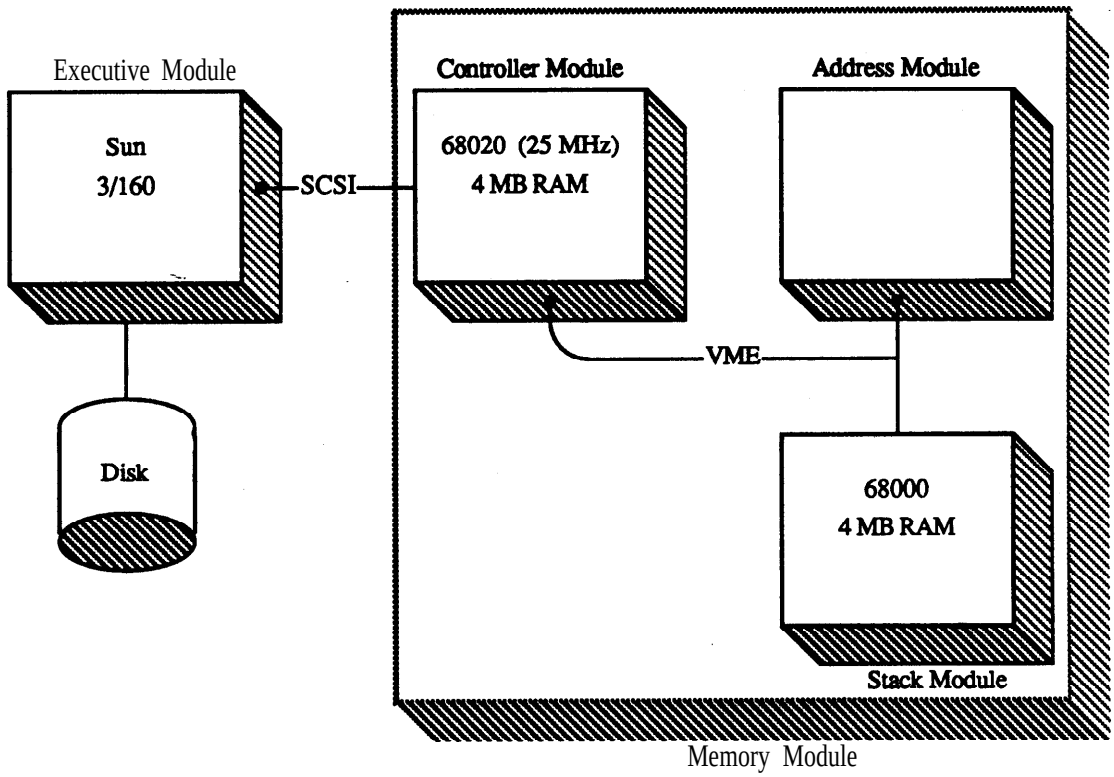


Figure 4.1: Module arrangement.

4.2.1 Executive module

This is the user interface to the SDM. The software is being developed for a Sun 3/160 workstation. The interface is a mouse-oriented interface with display windows and control panels.

The executive software is written in C. It communicates with the memory module via a SCSI port. The software is designed to be portable (i.e., programs should run on any workstation with a SCSI port).

The basic task of the executive is to translate user commands into directives to the memory system. The executive allows the user to read and write SDM locations and save memory images, among other things.

4.2.2 Control module

The control module, or controller for short, is the link between the executive and the address and stack modules (see Figure 4.1). It communicates with the executive via a SCSI bus and with the address/stack over a VME bus. The controller consists of a 25MHz M68020 processor with 4MB of RAM. The basic task of this module is to provide system power-up, initialization, diagnostics, and to act as a memory controller.

4.2.3 Address module

The address module decodes sparse memory addresses-i.e., it selects addresses that are within the Hamming distance of the read/write address.

4.2.4 Stack module

The stack module stores the information written in the SDM. It is handed selected addresses by the controller. For reading, it retrieves the data from these selected addresses and constructs a data word which is sent back to the controller. For writing, it simply updates the selected locations with the data given to it by the controller.

The stack module has a 68000 processor with 4 MB of RAM.

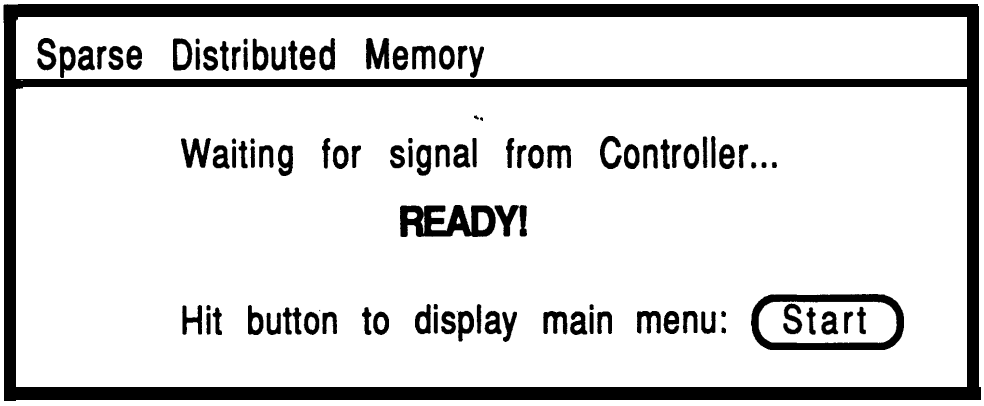


Figure 4.2: Start-up window.

4.3 Software operation

The first step is to run the file sdm.c on the Sun workstation. The memory module must then be powered up. Figure 4.3 is a schematic of the software operation and communication paths between the different modules. Table 4.1 explains the terms used in Figure 4.3. Figure 4.4 shows the software breakdown in the executive module. Note that each module in the system has a command register which indicates the status of that module. System operation begins with the control module. When booted, it will send a "ready" signal to the executive. It then waits for the executive to respond (Figure 4.2).

The executive, after receiving the "ready" signal, displays a normal-mode window and related control panels (Table 4.4 and Figure 4.5). The executive then waits for input. There are two alternative modes of operation, debug mode or normal mode. Debug mode simply allows the user to read any location in the memory module. In normal mode, the user can choose a Hamming radius, select an address mask and an SDM location, and decide to either read or write that location. Selecting "read" or "write" from the control panel initiates communication with the control module. Figure 4.6 shows how the executive module handles user input. Every time the user selects a command, the executive command register (XCR) is updated, see Table 4.2.

If the user has selected "read" or "write" in normal mode, the executive sends a copy of its XCR to the control module. It also sends copies of the mask address, Hamming radius, SDM address, and SDM data if it is a write operation (Figures 4.3 and 4.5). The controller uses the XCR to update the control module command register (CCR), the stack module command register (SCR), and the address module command register

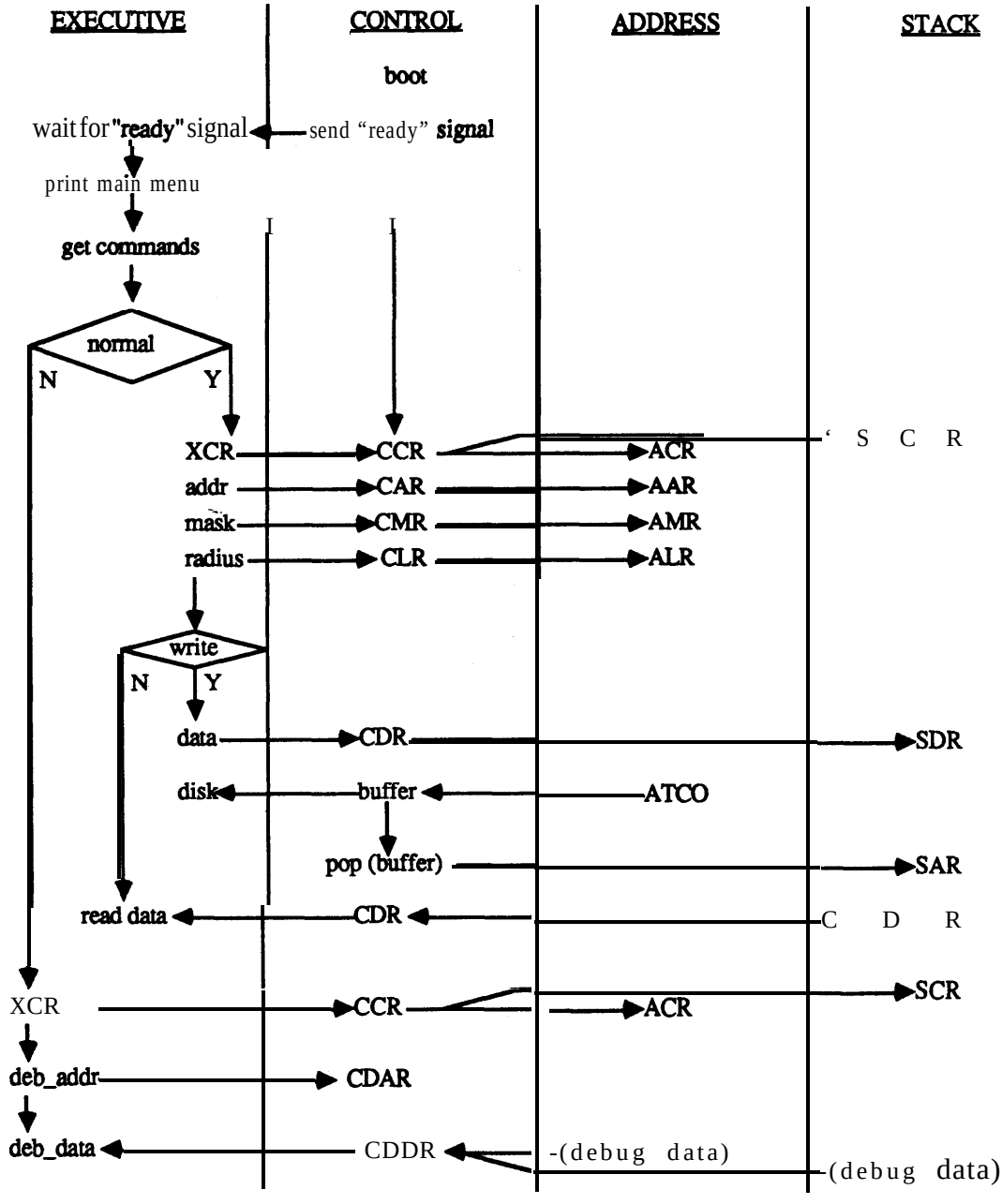


Figure 4.3: Software operation and module communication.

Table 4.1: Explanation of terms.

XCR	(Executive Module Command Register). This register details the state of the executive module. i.e. whether it is currently in a read or a write cycle and whether it is in debug or normal mode (see Table 4.2).
addr	The SDM address to read/write.
mask	The address mask.
radius	The Hamming radius chosen by the user.
data	The data to read/write at addr.
disk	Where the selected SDM addresses are to be dumped.
deb,addr	The memory module address/register the user wishes to read while in debug mode.
deb,data	The data read at deb,addr.
CCR	(Control Module Command Register--see Table 4.2).
CAR	(Control Module Address Register). This contains the SDM address the user wishes to read/write.
CDR	(Control Module Data Register). This contains the data the user wishes to write or which has been read at CAR.
CMR	(Control Module Mask Register). This contains the address mask.
CLR	(Control Module Limit Register). This contains the Hamming radius chosen by the user.
CDAR	(Control Module Debug Address Register). This contains the memory module address/register the user wishes to read in debug mode.
CDDR	(Control Module Debug Data Register). This contains the data read at CDAR.
ACR	(Address Module Command Register—see Table 4.2).
AAR	(Address Module Address Register). This contains the SDM address the user wishes to read/write.
AMR	(Address Module Mask Register). This contains the address mask.
ALR	(Address Module Limit Register). This contains the Hamming radius chosen by the user.
ATCO	(Address Module Tag Cache Output). This is a store for the selected SDM addresses.
buffer	A temporary store for selected SDM addresses.
debug data	The memory module data read during debug mode.
SCR	(Stack Module Command Register-see Table 4.2).
SAR	(Stack Module Address Register). This contains the currently selected SDM address.
SDR	(Stack Module Data Register). This contains the data the user wishes to write or which has been read at the chosen address.

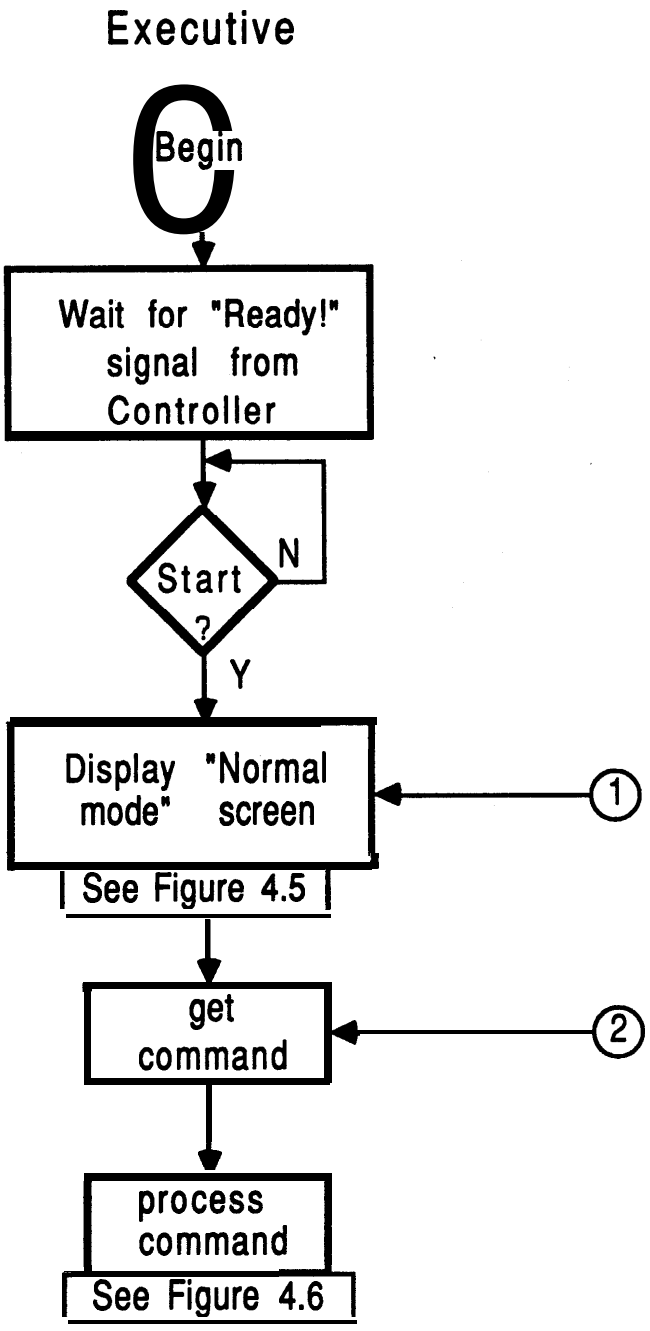


Figure 4.4: Executive software breakdown.

(ACR). Copies of the SDM address, address mask, and Hamming radius are sent to the address module. If the operation is a write, the controller forwards the SDM data also. The address module now has enough information to begin decoding SDM addresses. When the addresses are decoded the address module updates the ACR. The controller continually polls the ACR until it indicates that the selected SDM addresses are ready (Figures ***4.7, 4.8).*** The **controller** then reads the addresses from the address module's tag cache output register (ATCO) into a **buffer** and passes them one by one to the stack address register (SAR). (The selected addresses and their associated Hamming distances are also dumped to the executive module). Processing is now concentrated in the stack module (Figure 4.9). If the SCR indicates a read, the stack will add the contents of each selected SDM location to an accumulator until the control module buffer is empty. The contents of the accumulator are then forwarded as SDM data to the controller which passes them to the executive for observation by the user. For a write operation, the stack will update the selected SDM locations with the SDM data it has been sent previously.

If, **however, the** user has selected debug mode, a debug mode window is displayed (Figures 4.10, 4.11) and operation proceeds **as** follows (see Figures 4.3, 4.12, and 4.13). First the user is given the option to read either an SDM location or a register in the memory module. When "read" is selected, the XCR is updated and sent to the controller, along with the debug address (address/register to be read). The controller updates its CCR and the stack and address module command registers (SCR, ACR). The stack and address modules, when in debug mode, simply wait for the controller to read the required address or register. The controller reads the data at this address and forwards it to the executive.

Table 4.2: Command register descriptions.

Executive Module Command Register. (XCR)

	1	0
XCR [0]	error	
XCR [1]	debug mode	normal mode
XCR [2]	read operation	
XCR [3]	write operation	-
XCR [4–7]	*- reserved -*	

Control Module Command Register (CCR)

	1	0
CCR [0]	error	
CCR [1]	debug mode	normal mode
CCR [2]	read operation	
CCR [3]	write operation	
CCR [4]	operation completed	operation not completed
CCR [5–7]	*- reserved -*	

Address Module Command Register (ACR)

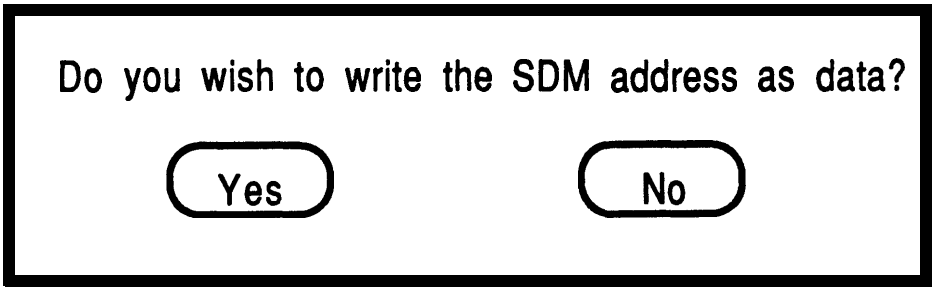
ACR [O-l]	run modes	0	0	reset
		0	1	running
		1	0	hit
		1	1	wait
ACR [2]	reset control	0		run (or hit or wait)
		1		reset
ACR [3]	cache indicator	0		TCO empty
		1		TCO not empty
ACR [4]	mode	0		uncomplicated addresses
		1		complicated addresses
ACR [5–7]				*- reserved -*

Stack Module Command Register (SCR)

	0	1
SCR [0]	error	
SCR [1]	debug	normal
SCR [2]	read	
SCR [3]	write	
SCR [4]	operation completed	operation not completed
SCR [5]	beginning of address	not beginning of address
SCR [6]	next address ready	next address not ready
SCR [7]	*- reserved -*	

Table 4.3: Normal mode screen functions.

The Keypad	Hex digits that are entered at the cursor location in either the primary window (normal mode) or the secondary window (debug mode).
Pack	Move the cursor back one hex digit.
Digit	Move the cursor forward one hex digit.
Word	Move the cursor to the first digit of the next word.
Clear	Zero the entire field that the cursor is currently in.
Next	Move the cursor to the next field (i.e., either address, mask, or data fields).
Read	Read the selected memory address in normal (associative) mode.
Write	The "write" control panel will appear in the secondary window:



Yes the "memory address" is copied to 'data' and the SDM write proceeds normally.

No SDM write proceeds normally.

Debug	Setup debug-mode screen configuration (see Figures 4.10, 4.11) and enter debug mode.
Quit	Exit the program. Confirmation with a left click is required.

Table 4.4: Debug mode screen functions.

Memory debug mode	
Keypad	Same as for normal mode but the only field that can be typed into is "debug memory address."
Read	Read memory location in debug (non-associative mode).
Register	Switch to register debug mode (Figure 4.11).
Normal	Switch to normal mode (Figure 4.5).
Quit	Exit the program. Confirmation with a left click is required.
Register debug mode	
	The lower left control panel allows the user to select the register to be read.
Read	Read the register value.
Memory	Switch to memory debug mode (Figure 4.10).
Normal	Switch to normal mode (Figure 4.5).
Quit	Exit the program. Confirmation with a left click is required.

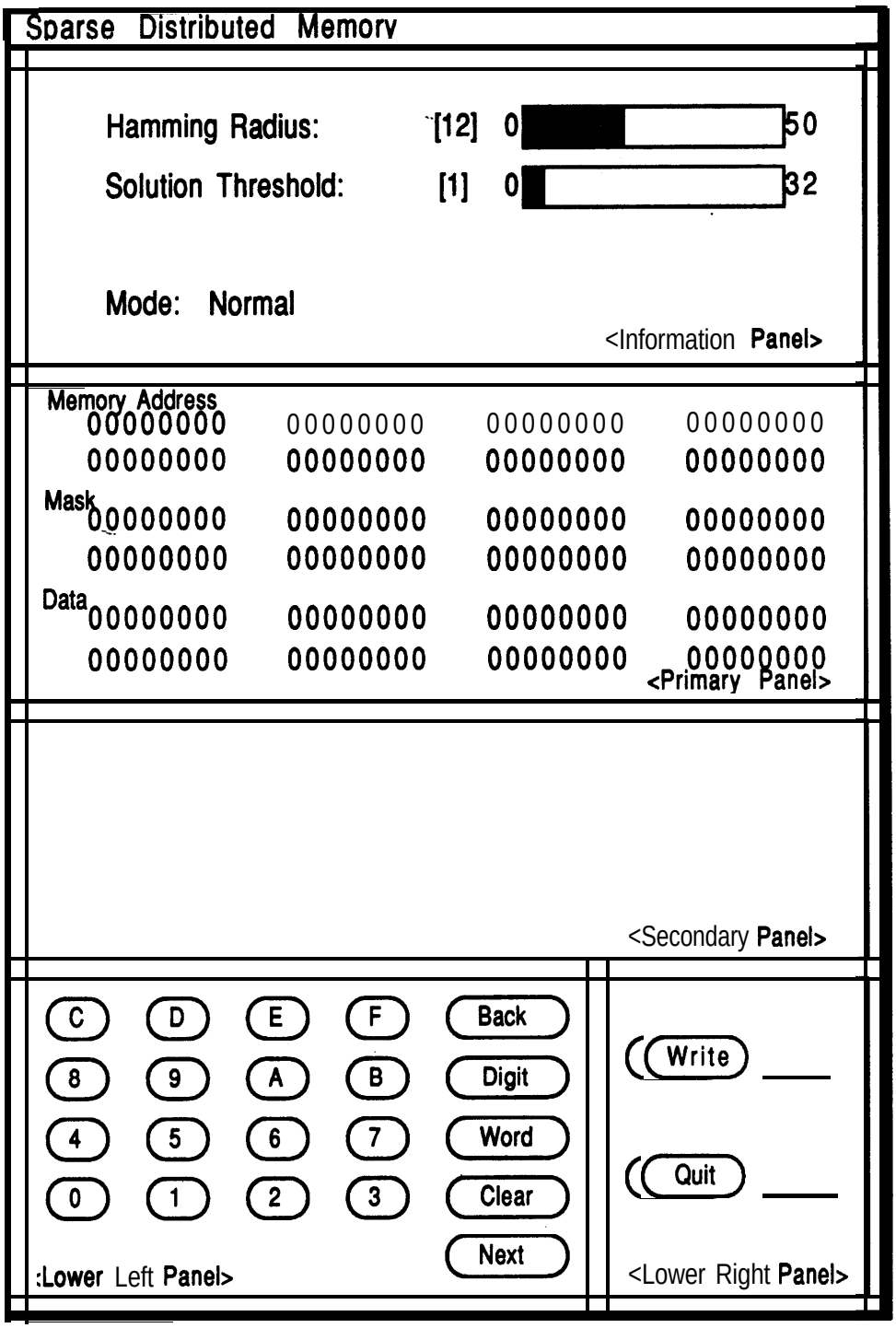


Figure 4.5: Normal mode screen.

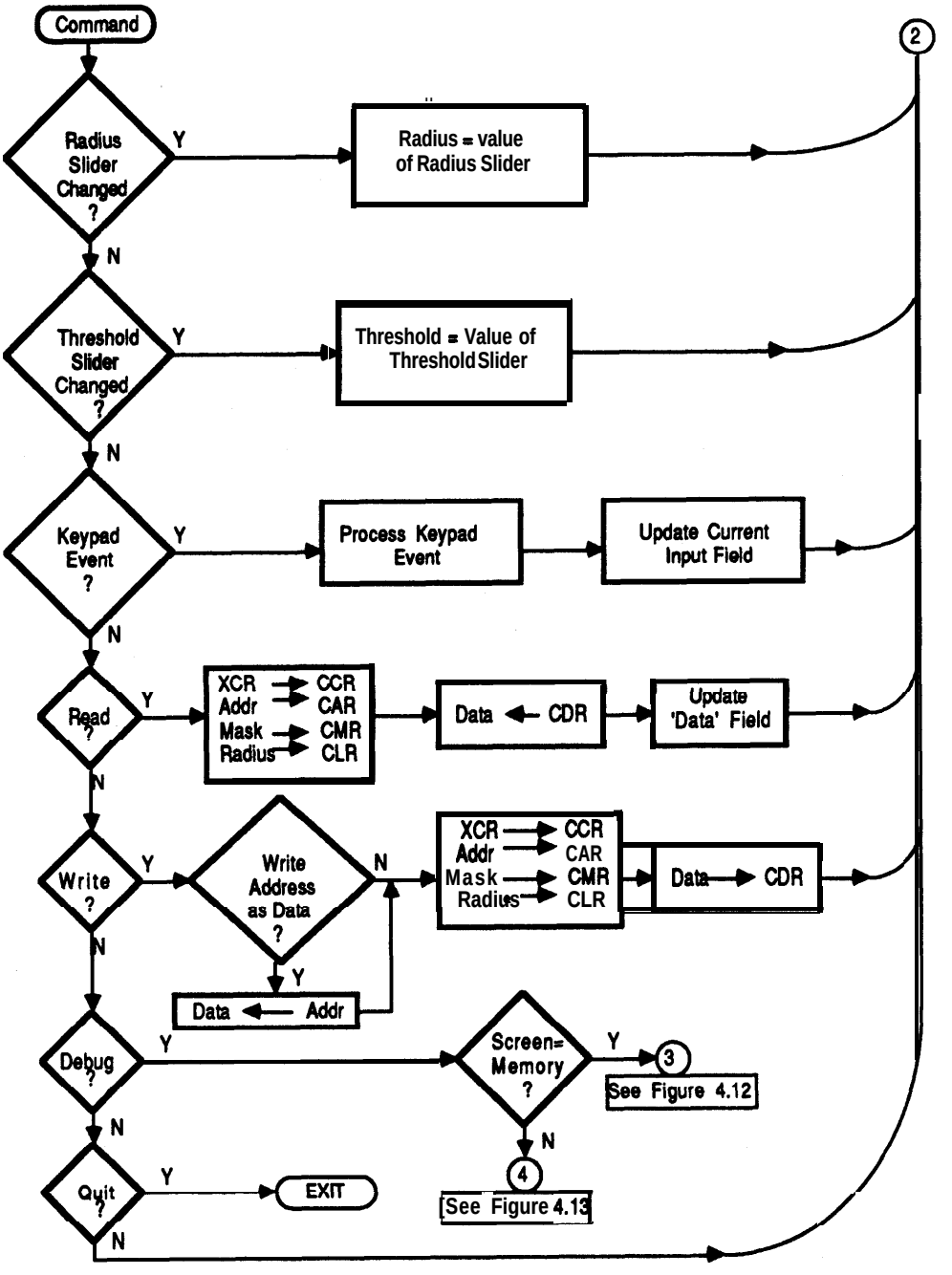


Figure 4.6: Process normal mode command.

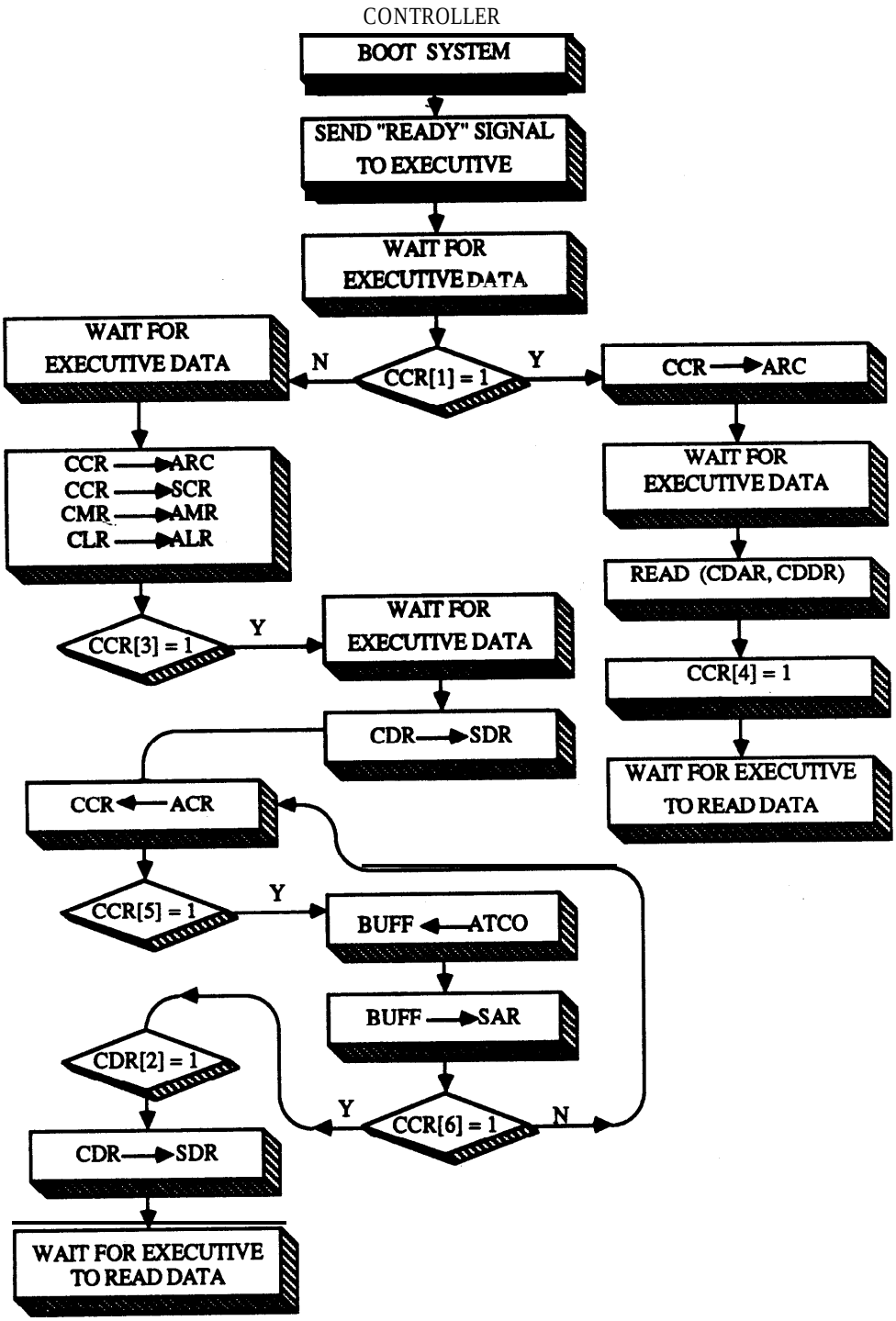


Figure 4.7: Controller module operation.

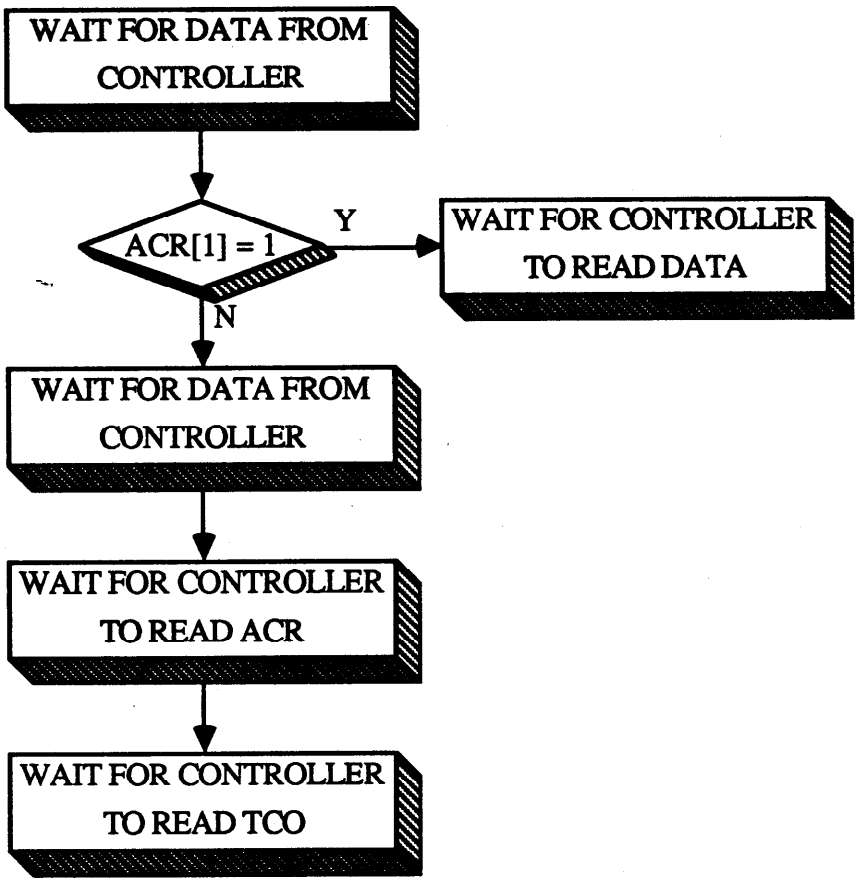


Figure 4.8: Address module operation.

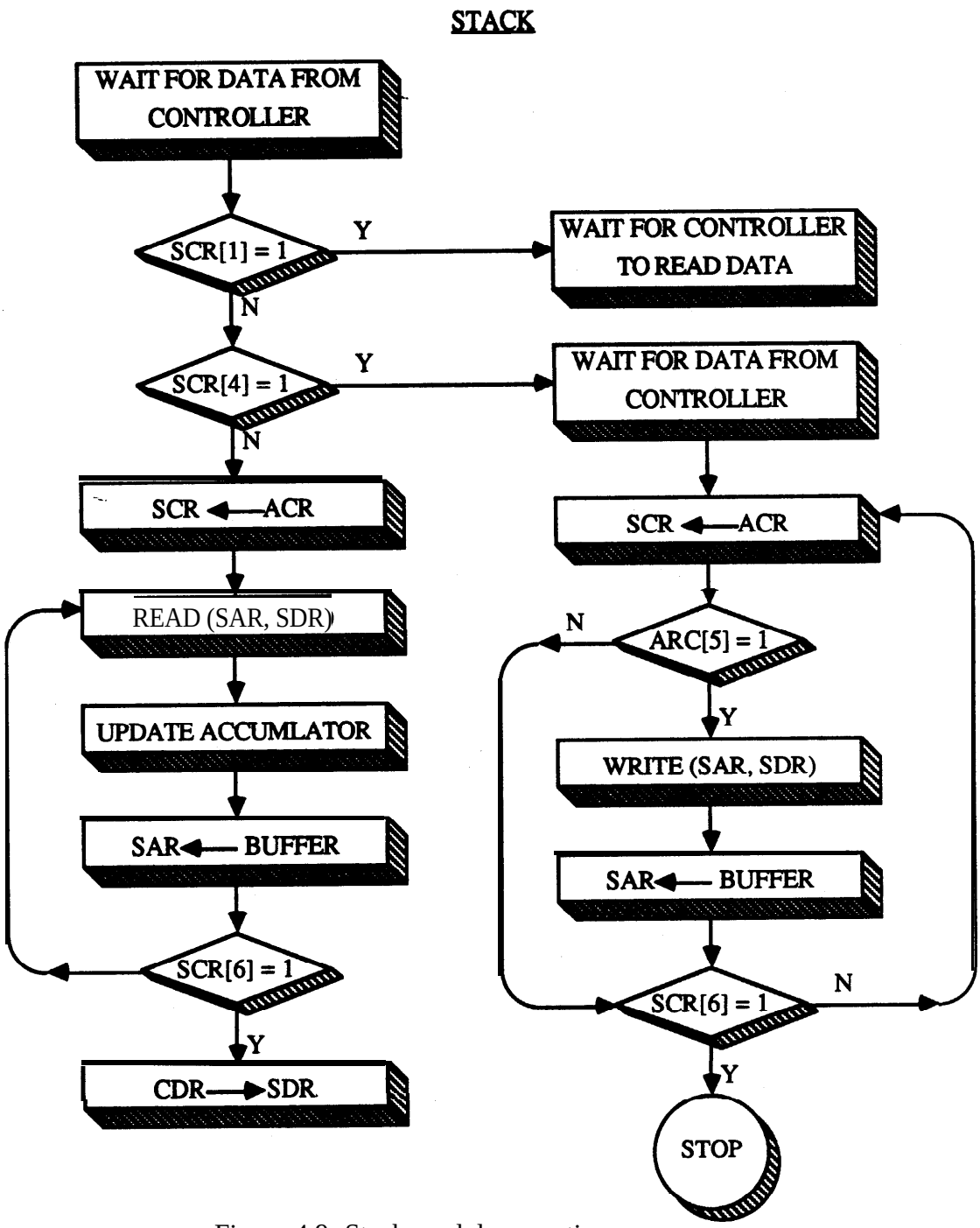


Figure 4.9: Stack module operation.

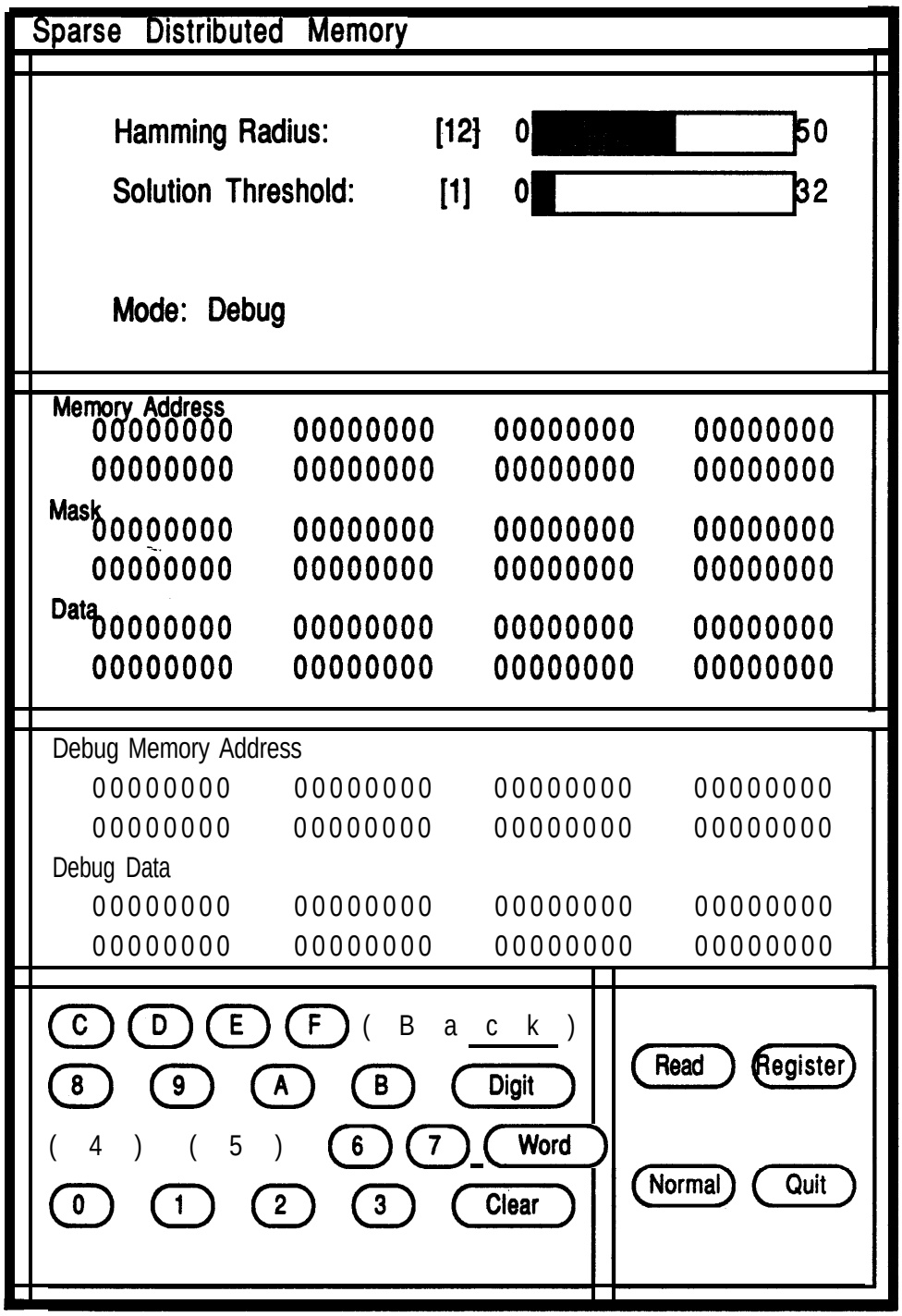


Figure 4.10: Memory debug mode screen.

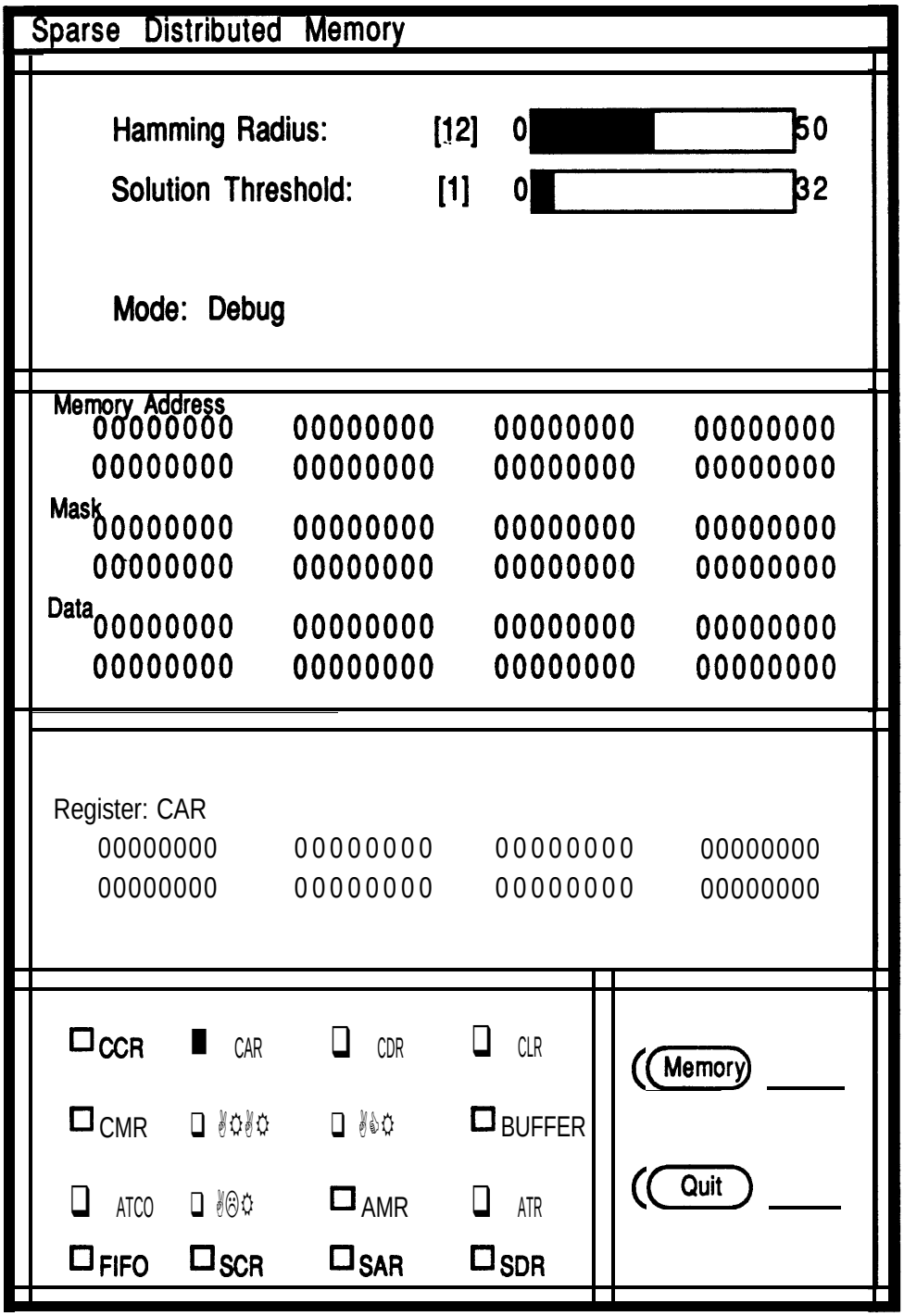


Figure 4.11: Register debug mode screen.

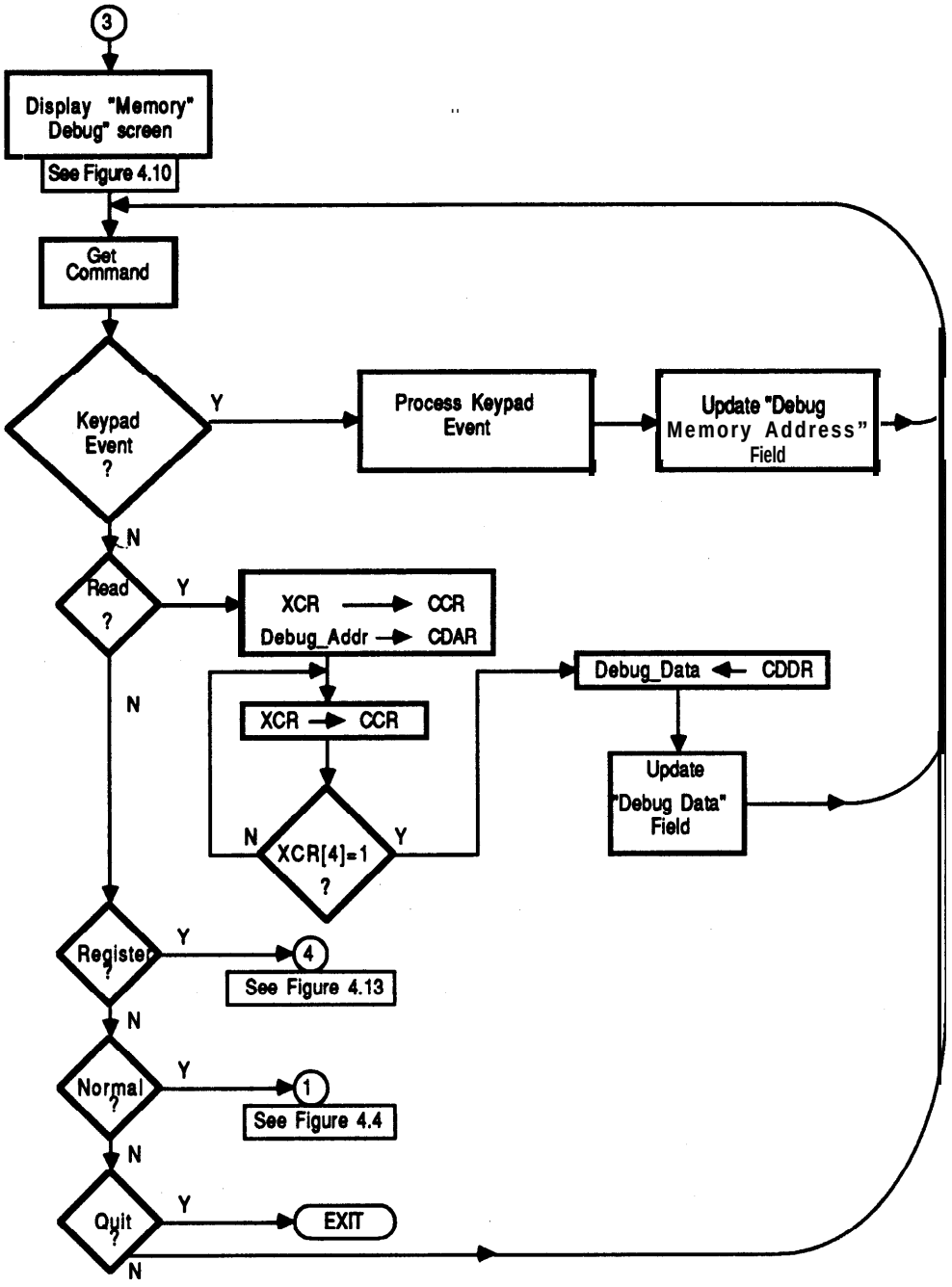


Figure 4.12: Process memory debug command.

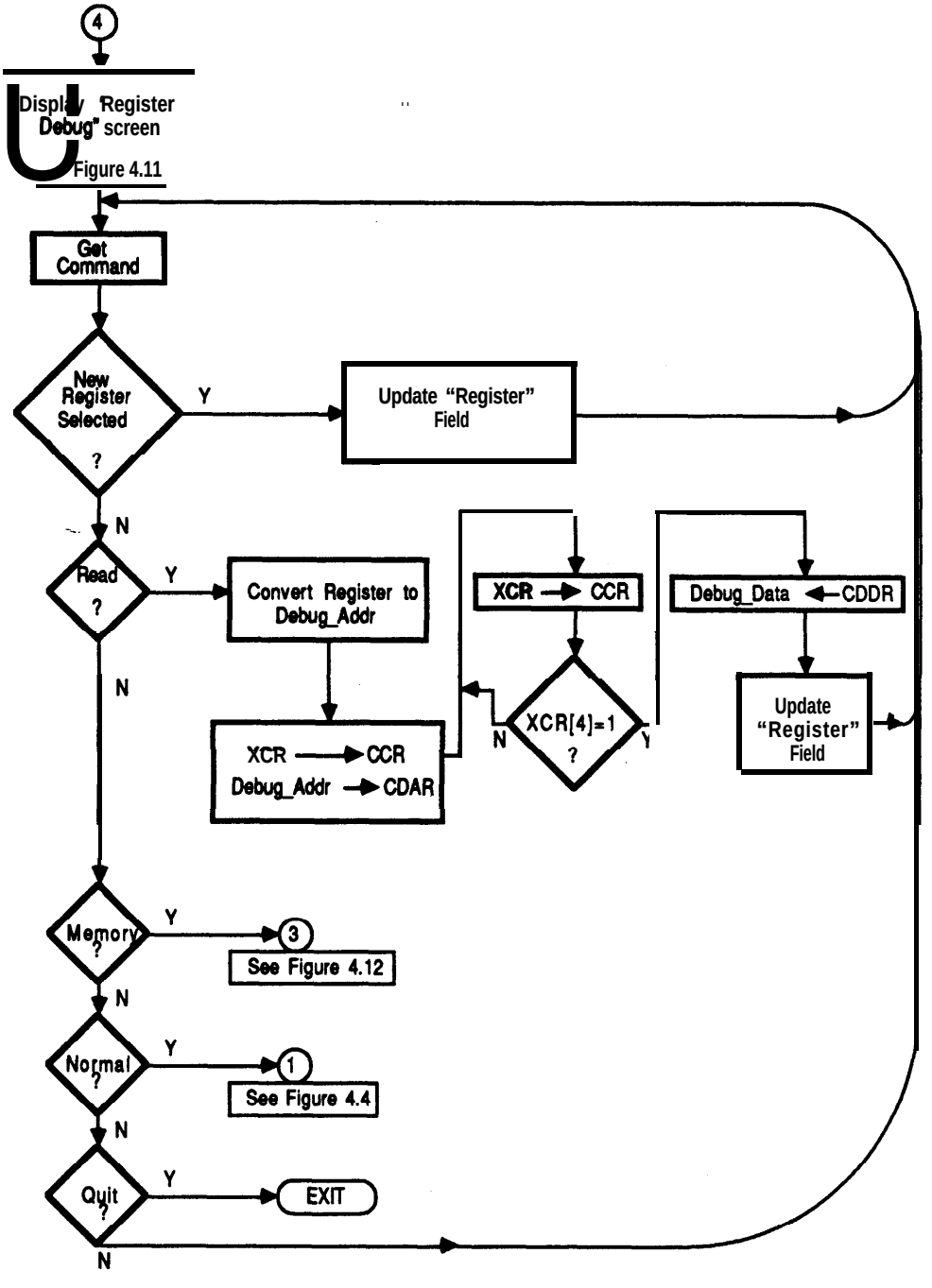


Figure 4.13: Process register debug command.

4.4 Summary

. The bulk of the operations of the SDM are governed by software. The executive software, written in C, runs on a Sun workstation and provides an interface to users.

The internal operation of the control and stack modules are governed by software written in Motorola 68000 family assembly language.

Though not as fast as a custom designed hardware implementation would be, the software implementation is easy to modify and meets the performance specifications of 50 operations (reads or writes) per second.

Appendix A

Stack and Controller memory maps and assembly language code

This appendix gives the breakdown of the Stack and Controller memory maps, plus a listing and explanation of the assembly language code for each module.

The Stack module is implemented using a Plessey 68-12 board (68000 processor), and the Controller module is implemented using a Plessey 68-22 board (68020 processor).

A.1 Stack Module

A.1.1 Stack memory map

This section explains in detail the breakdown of the 4MB of Stack DPR for software and data, the use of SRAM for storage of local data, and the addresses used in the I/O address space (see Figure A.l).

DPR (Dual Ported Ram)

The DPR houses the folds (SDM locations), the SDR (Stack Data Register), the SAR (Stack Address Register), the Accumulator register, and the Stack software. See Figure A.2.

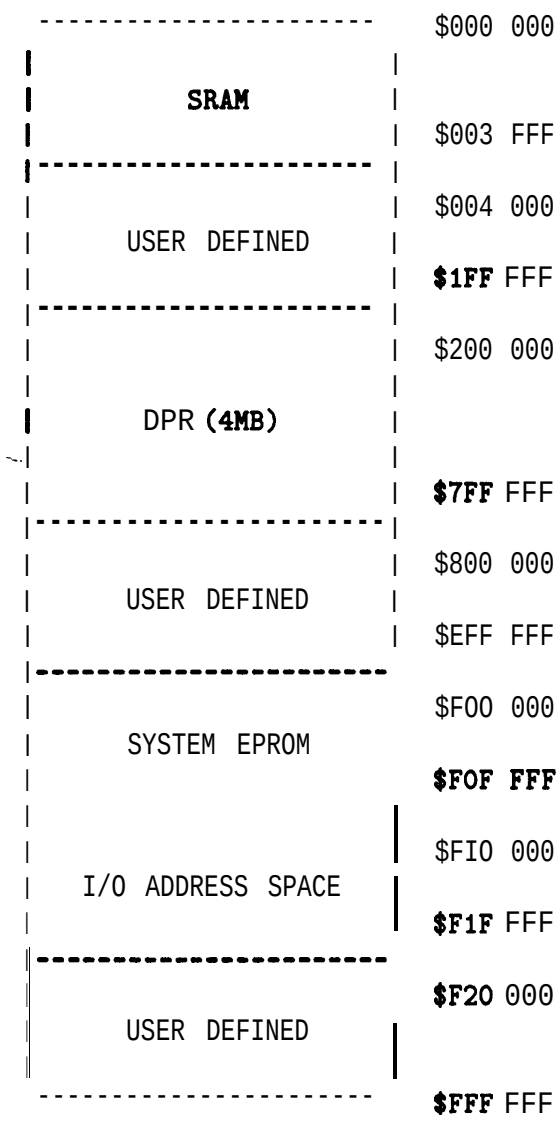


Figure A.l: Stack module memory map.

1. Folds:

   2MB fold = 8,000 locations (8,000 * 256 * 8 = 2MB)

   Each location = 64 * 32-bit memory slots = 256 bytes

   Fold begins: $400 000, fold ends: $600 000

2. SDR (Stack Data Register):

   SDR = 64 * 32-bit slots = 256 bytes

   Fold begins: $600 100, fold ends: $600 200

3. SAR (Stack Address Register):

   SAR = 32 bits = 4 Bytes

   Fold begins: $600 210, fold ends: $600 214

4. ACCUMULATOR:

   ACCUM = 128 * 32-bit slots = 512 bytes

   Fold begins: $600 300, fold ends: $600 500

5. PROGRAM:

   Software:

   Fold begins: $601 000, fold ends: $7FF FFF

SRAM

The SRAM is used to store frequently used routines and local variables. At present it is used for storage of the Stack module Control Register:

$0 = copy of Control Register

(The Control Register itself is located at $F12 001 and is read-only).

I/O Address Space

At the moment, the stack software only accesses the 68-12 Control Register (at location $F12 001).

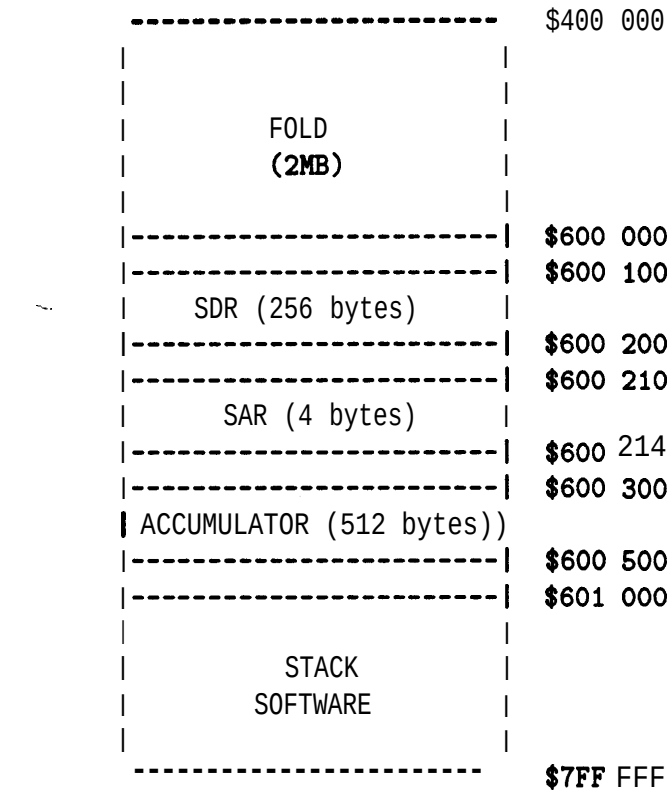


Figure A.2: Memory map for the Stack module DPR.

A.1.2 Stack module code

```
1.        MOVE.B #$FF, $0          clear SCR
2.        MOVE.B $0, $F12001       ($0 = copy of SCR = $F12001)
3.  (1)   BTST.B #6, $0            READY flag
4.        BNE.B (1)

5.        BTST.B #1, $0            debug/normal?
6.        BNE.B (1)

7.  (2)   BSET.B #4, $0            reset op. completed flag
8.        MOVE.B $0, $F12001
9.        MOVE.L $600210, DO       get SDM address
10.       MULU DO, $40             multiply by 64
11.       MOVE.L DO, $600210
12.       BTST.B #2, $0            READ flag
13.       BNE.B (6)                WRITE
14.       BTST.B #5, $0            start of read block?
15.       BNE.B (4)                No

16.       CLR.L D1
17.       MOVE #128, D1            ACCUM. = 128 * 32-bits
18.       MOVEA.L #$600300, A0     get accumulator address
19.  (3)  CLR.L (AO)+              clear accumulator
20.       SUBQ #1, D1
21.       BNE.B (3)

22.  (4)  CLR.L D2
23.       MOVE #128, D2
24.       MOVEA.L $600210, A2      SAR
25.       MOVEA.L #$600300, A3     ACCUMULATOR
26.  (5)  CLR.L D3
27.       ADD.W (A2)+, D3          add data
28.       ADD.L D3, (A3)+          update accumulator
29.       SUBQ #1, D2
30.       BNE.B (5)

31.       BCLR.B #4, $0
32.       MOVE.B $0, $F12001       op. finished
33.  (6)  BTST.B #6, $0
34.       BEQ.B (1)                READ op. completed
35.       BTST.B #4, $0
36.       BEQ.B (2)                next address ready!
37.       BRA (6)                  wait for next address

38.       CLR.L D4                 (WRITE)
39.       MOVE #64, D4             SDR = $64 * 4 * 8$ bits
40.       MOVEA.L $600210, A3      get SDM address
41.       MOVEA.L #$600200, A4     get SDR address
```

```
42. (7) CLR.L D4
43.       ADD.L (A4)+, D4
44.       ADD.L D4, (A3)+
45.       SUBQ #1, D4
46.       BNE (7)
47.       BCLR.B #4, $0          op. complete
48.       MOVE.B $0, $F12001
49. (8) BTST.B #6, $0
50.       BEQ.B (1)              WRITE op. complete!
51.       BTST.B #4, $0
52.       BEQ.B (2)              next address ready!
53.       BRA (8)                wait for next address
```

A.1.3 Explanation of code

Line #	Explanation
1, 2	SCR is initialized to $FF (all bits = 1);
3, 4	wait for READY signal from Controller; (the Controller signals the Stack by clearing SCR[6])
5, 6	find out if the operation is normal or debug; (check SCR[1]; if set → debug (wait) if clear → normal (continue))
7, 8	reset op. completed flag (SCR[4] = 1);
9	get SDM address (14-bit);
10, 11	convert it to a 20-bit address
12, 13	find out if op. is a READ (test SCR[2]d; if set → WRITE if clear → continue)
14, 15	start of read block? (test SCR[5]; if set → normal READ if clear → clear accumulator)
16-21	clear accumulator;
22-30	SDM READ (read one SDM location and add it to the accumulator; one location = 128 * 16-bit words)
31, 32	declare read operation finished (clear SCR[4])
33, 34	is READ completed? i.e., *all* selected SDM locations read? (test SCR[6]; if clear → finished → wait for next Controller command if set → still more locations to be read → continue)
35-37	is next address ready? (test SCR[4]; if clear → next address ready → fetch it if set → address not ready → wait for it)
38-46	SDM WRITE (write contents of SDR to ONE SDM location; one data location = 64 * 32-bit words)
47, 48	declare read operation finished (clear SCR[4])
49, 50	is WRITE completed? i.e., *all* selected SDM locations written? (test SCR[6]; ifc1ear → finished → wait for next Controller command if set → more locations to be written → continue)
51-53	is next address ready? (test SCR[4]; if clear → next address ready → fetch it if set → address not ready yet → wait for it)

A.1.4 Use of 68000 registers

Data:	DO	holds SDM address
	D1	counter for the accumulator clearing function
	D2	counter for adding SDM data to accumulator locations (READ)
	D3	holds SDM data temporarily during an SDM READ
	D4	counter for writing data from SDR to the folds
Address:	A0	holds accumulator addresses while accumulator is being cleared
	A1	holds current address of fold during a READ
	A2	holds current accumulator address during a READ
	A3	holds current address of fold during a WRITE
	A4	holds current SDR address during a WRITE

A.2 Controller Module

A.2.1 Controller memory map

This section explains the breakdown of the Controller module (68-22) address space. Figure A.3 shows the communications paths between the Controller and the other modules. Figure A.4 shows the complete 68-22 memory map. The map can be broken down into three sections: VME address space, DPR, and I/O.

VME address space

Figure A.5 is a breakdown of the VME address space (4GB); sections of the Address module and Stack module memories are contained within the VME address space:

Stack module:	fold	2MB
	SDR (Stack Data Register)	256 bytes
	SAR (Stack Address Register)	4 bytes
	Accumulator	512 bytes
Address module:	28 * 32-bit registers	112 bytes

DPR

The 4MB DPR will contain the Controller software and data. Figure A.6 outlines the breakdown of the Controller DPR.

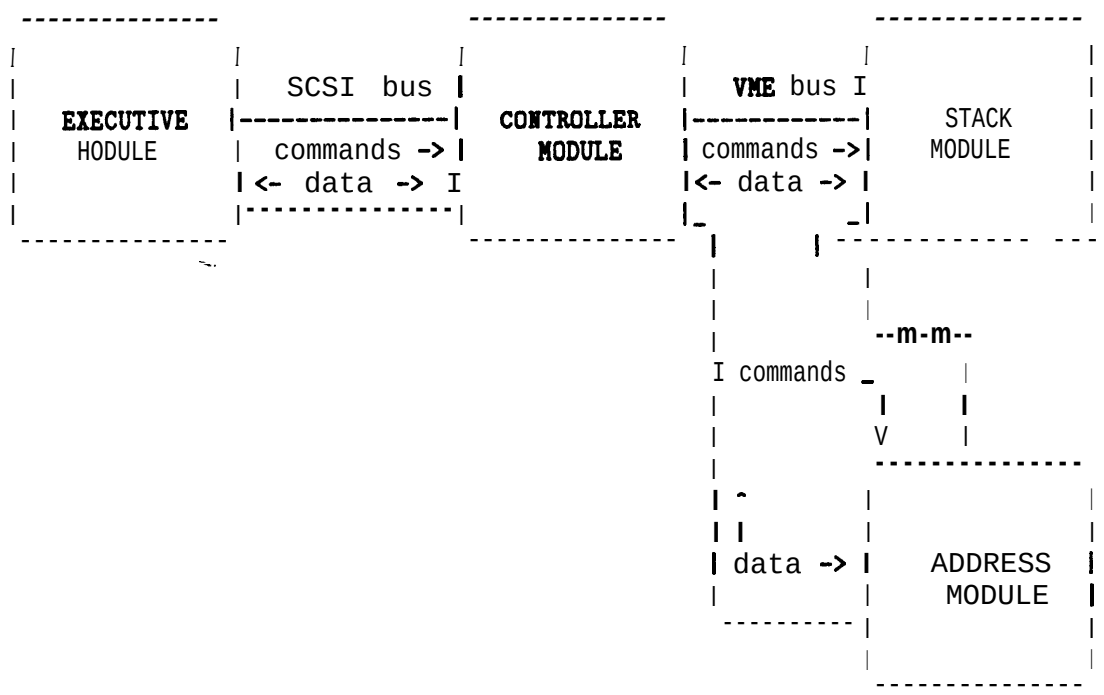


Figure A.3: Communications paths between Control module and the rest of the system.

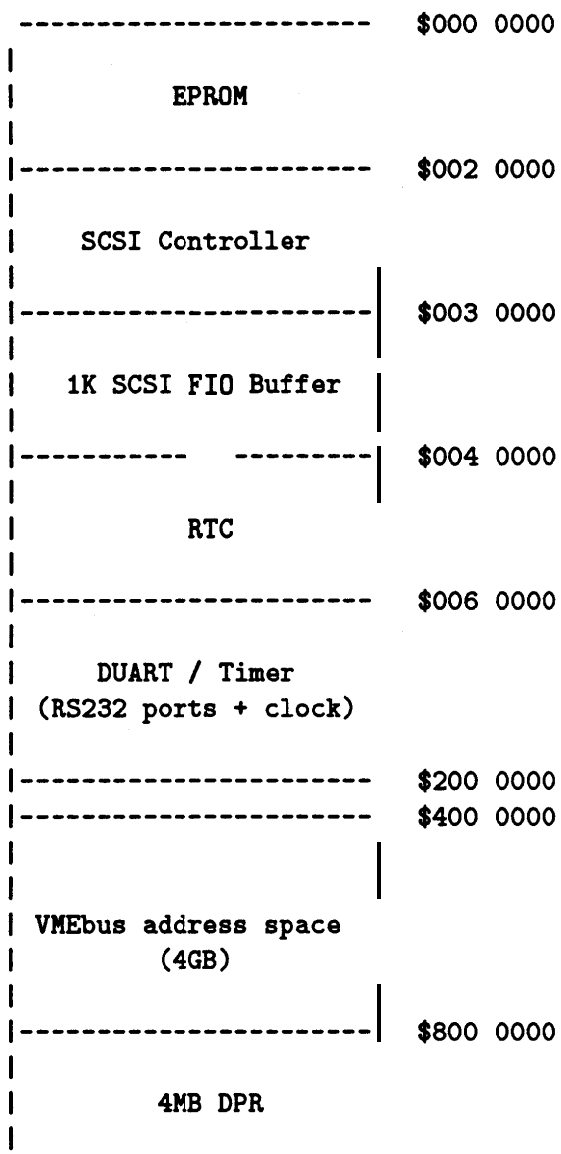


Figure A.4: Controller module memory map.

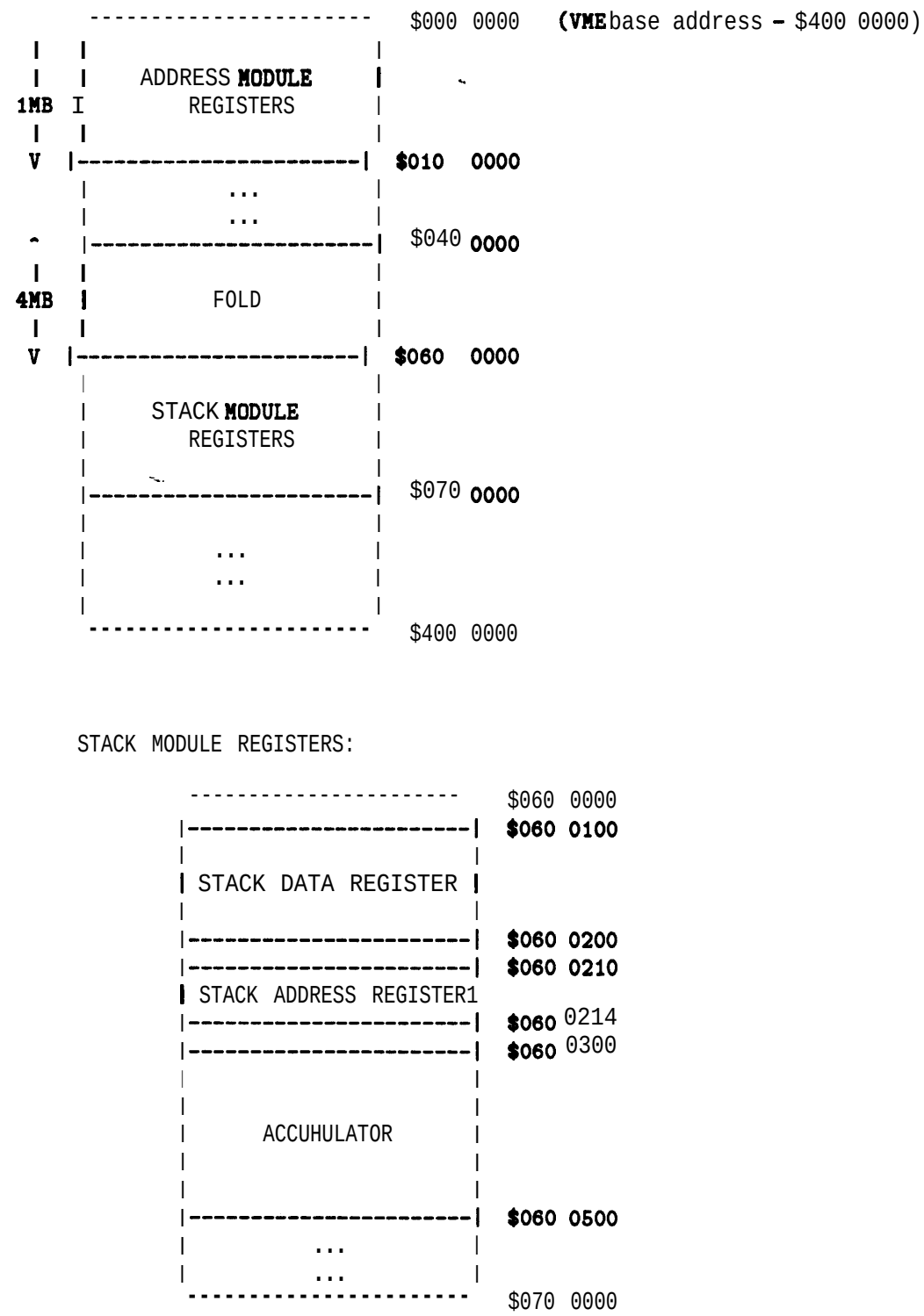


Figure A.5: Breakdown of the VME Address Space (4GB).

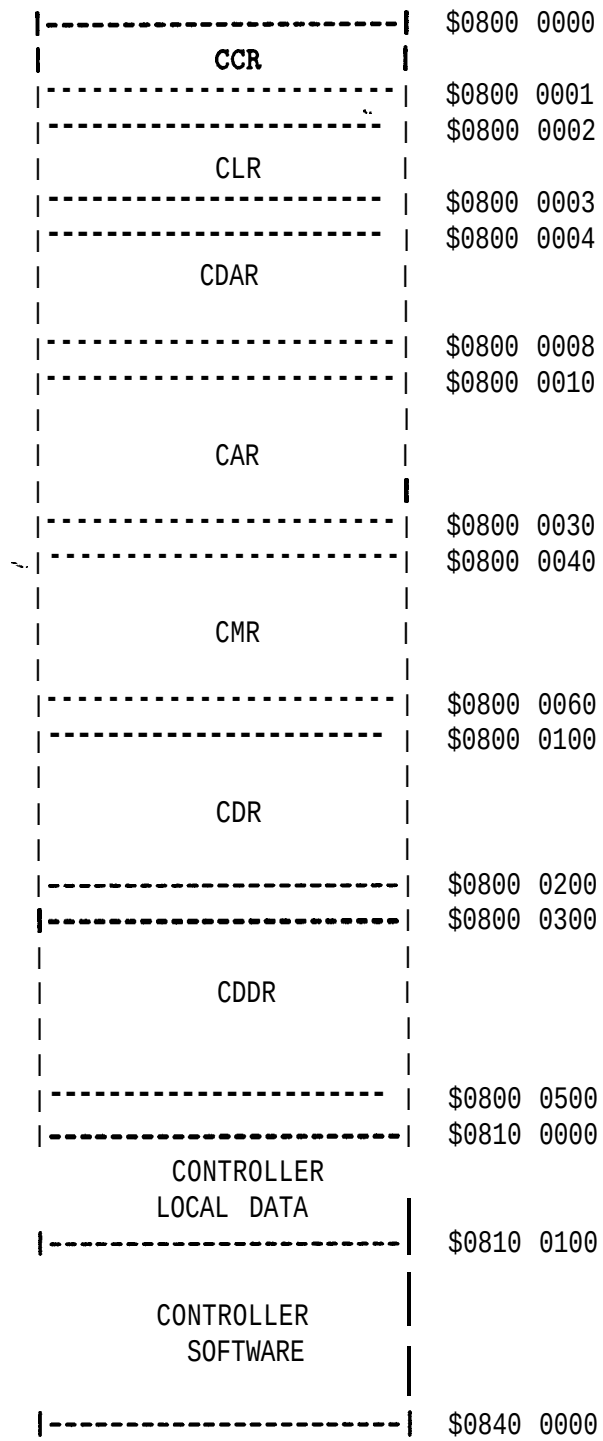


Figure A.6: Breakdown of Controller DPR.

I/O

The VME bus has been explained above; there remains the SCSI interface to describe. The SCSI is controlled by reading and writing the registers listed in Figure A.7.

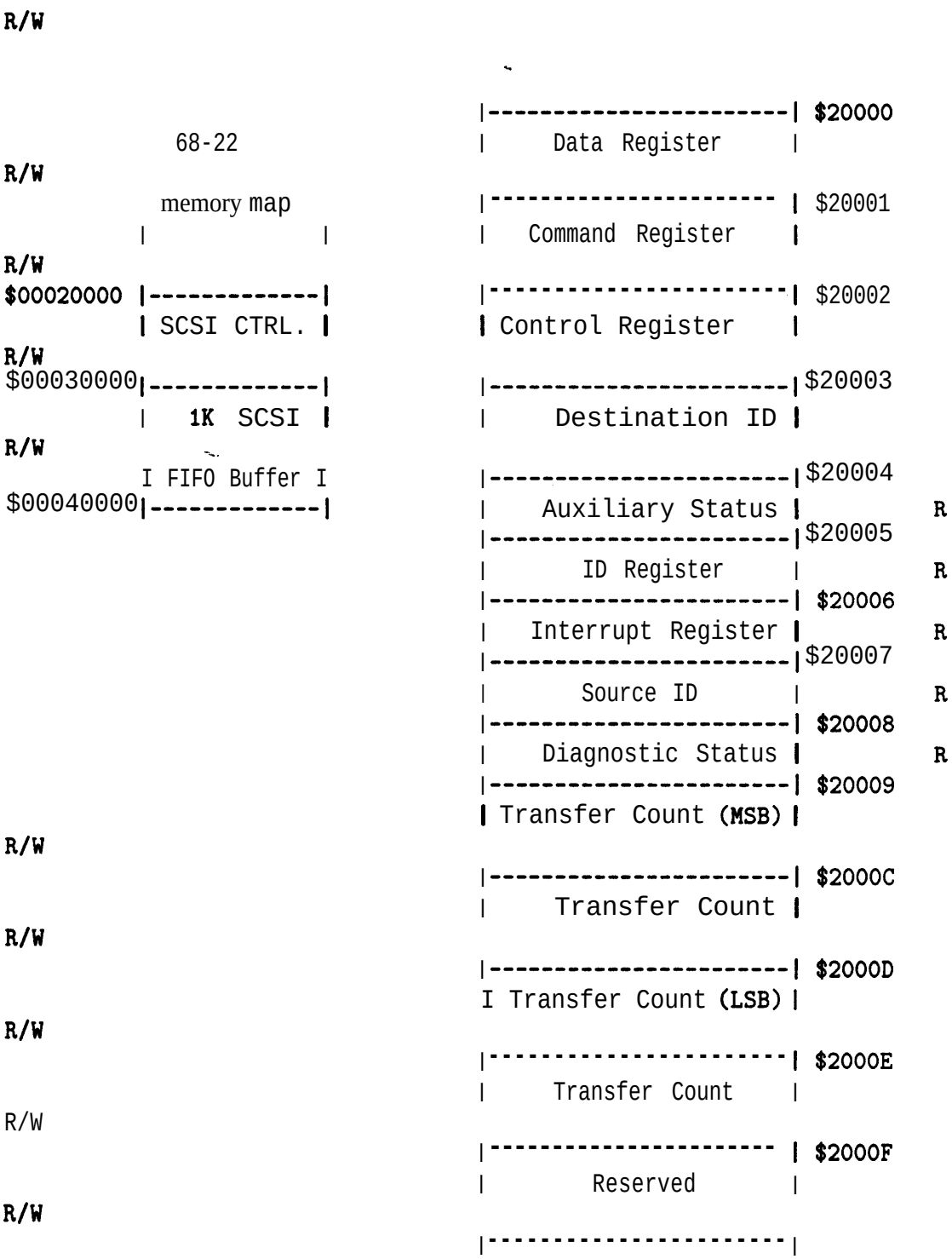


Figure A.7: SCSI Registers.

A.2.2 Controller module code

```
                SCSI-WRITE      ; "READY" signal → Executive
LOOP            SCSI-READ       ;CCR ← XCR
                COPY (CCR, SR)  ; copy into 68-22 status register
                CMP (SR[1], 1)  ; debug or normal mode?
                BEQ DEBUG
NORMAL          SCSI-READ       ;CCR ← XCR
                                ; CAR ← ADDR
                                ;CMR ← MASK
                                ;CLR ← RADIUS
                VME,WRITE       ;CCR → ACR
                                ; CAR → AAR
                                ; CMR → AMR
                                ; CLR → ALR
                                ;CCR → SCR
                CMP (SR[3], 1)  ; SDM WRITE?
                BNEQ ADDRESS    ;fetch addresses from ADDRESS module
WRITE           SCSI-READ       ; CDR ← DATA
                VME,WRITE       ; CDR → SDR
ADDRESS         VME,READ        ; CCR ← ACR
                COPY (CCR, SR)
                CMP (SR[5], 1)  ;TCO not empty?
                BNEQ ADDRESS    ; not yet!
                VME,READ        ;BUFF ← ATCO
                VME,WRITE       ;BUFF → SAR
                CMP (SCR[6], 1) ; address buffer empty?
                BNEQ ADDRESS
FINISD          CMP (SCR[2], 1) ; SDM READ?
                BNEQ EXIT
                VME,READ        ;CDR ← SDR
                SCSI-WRITE      ; CDR → DATA
                BRA EXIT

DEBUG           VME,WRITE       ;CCR → ACR
                                ; CCR → ACR
                SCSI-READ       ; CDAR ← D,ADDR
                VME,READ        ;read (CDAR, CDDR)
                MOVE (#1, SR[4]) ; debug data is ready
                COPY (SR, CCR)
                SCSI-WRITE      ; CDDR → D-DATA
EXIT            BRA LOOP
```

Bibliography

[1] P. A. Chou. The capacity of the Kanerva associative memory. August 1987. Submitted for possible publication to ***IEEE Transactions on Information Theory.***

[2] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. ***Proceedings of the National Academy of Sciences, USA,*** 79:2554–2558, April 1982.

[3] P. Kanerva. ***Self-propagating search: A unified theory of memory.*** Technical Report CSLI-84-7, Stanford Center for the Study of Language and Information, March 1984.

[4] J. D. Keeler. ***Comparison between Sparsely Distributed Memory and Hopfield-type Neural Network Models.*** Technical Report RIACS TR 86.31, NASA Research Institute for Advanced Computer Science, Mountain View, CA, December 1986.