General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by the NASA Center for Aerospace Information (CASI)
# Table of Contents

1. Introduction 1

2. Design of the Image Labeling System 2
   2.1 Algorithm Description 3
   2.2 Circuit Description 6
   2.2.1 CAM1 Chip 6
   2.2.2 CAM2 Chip 7
   2.2.3 System Description 10

3. DGM Description of System 12
   3.1 Description of DGM 13
   3.2 Using DGM to Construct A Data Flow Graph 13
   3.3 Modeling the Region Labeling System Using DGM 15

4. Evaluation

5. Conclusion 16
1. Introduction

The purpose of this project was to evaluate the applicability of the Directed Graph Methodology (DGM) to the design and analysis of special purpose image and signal processing hardware. To this end, a special purpose image processing system was designed and described using DGM. The design, suitable for VLSI, implements an innovative region labeling technique. The utility of DGM was evaluated using this design.

Two chips were designed, both using NMOS technology, as well as a functional system utilizing those things to perform real-time region labeling. The system was described in terms of DGM primitives.

As a result of this effort, it was concluded that DGM, as it is currently implemented, is inappropriate for describing synchronous, tightly coupled, special purpose systems. Instead, the nature of the DGM formalism lends itself much more readily to modeling of networks of general-purpose processors. Section 2 of this report describes the image labeling system, including the two custom chips which were designed.

Section 3 provides an overview of DGM, and then shows how the special purpose design may be described using DGM.

Section 4 describes and justifies the conclusion that DGM is inappropriate for describing special purpose signal processing systems.

Details are contained in the appendices.
2. **Design of the Image Labeling System**

DGM was evaluated in the design of a hardware system for region labeling. The purpose of this circuit is to partition an image into a set of meaningful regions, and to do so "on the fly" with a single pass over the data. These partitioned regions are composed of all pixels that have similar attributes and have a four-neighborhood connectivity.

One technique for assigning pixels to regions is known as "region growing." The region growing technique is initiated by choosing a pixel which meets some criteria (e.g. grey level above threshold) for inclusion in a region. The algorithm then proceeds by examining all adjacent neighbors of the pixel and comparing that pixel with the neighbor in question. Typical measures of similarity include the magnitude of the neighboring pixel's grey level or the relative contrast between the pixel and its neighbor under consideration for inclusion in the region. This process is repeated recursively for all newly accepted pixels until no new pixels can be added to the region. Since the region-growing technique always results in closed regions, this technique is often preferable to other techniques which are based on edge detection or line fitting.

The algorithm for region labeling incorporated into the system architecture described in this report differs from traditional region growing in that it performs the assignment of pixels in a sequential, raster-scan fashion rather than using a recursion. For this reason, it is potentially orders of magnitude faster than recursive region growing. It is a technique based on the concept of equivalence relationships between pixels of the image. The regions are labeled in a single pass over the image by utilizing a content-addressable memory. Appendix 1 provides the theoretical foundation for the algorithm described herein.
2.1 Algorithm Description

Two pixels a and b are defined to be equivalent (designated R(a,b)) if they belong to the same region of an image. This relationship can be shown to be reflexive (R(a,a)), symmetric (R(a,b) -> R(b,a)) and transitive (R(a,b) AND R(b,c) -> R(a,c)).

The transitive property enables all pixels in a region to be determined by considering only local adjacency properties. In this algorithm, each pixel will be compared with each adjacent pixel in a left-to-right, top-to-bottom raster scan fashion. Pixels in a simple binary image are labeled in raster scan order.

The system in this report assigns labels to pixels maintained in a table of equivalence relationships. Figure 2 shows that this hardware resides between the image memory and a host computer.

If two pixels meet some criterion, in the case of a binary image, both pixels are at logic 1, and they are adjacent, then they are in the same region. By definition, if two pixels are in the same region, the R(a,b) holds.

That is

\[ \text{ADJACENT} \left( \langle x, y \rangle, \langle x', y' \rangle \right) \land |I(x,y) - I(x',y')| < T \Leftrightarrow R(\langle x, y \rangle, \langle x', y' \rangle) \].

The transitive property of R cannot be used to infer

\[ R(\langle x, y \rangle, \langle x', y' \rangle) \Rightarrow |I(x,y) - I(x',y')| < T \]

without also considering the adjacency property.

As the region partitioning proceeds in real-time (i.e. synchronously with the raster scan), two activities must be performed. First, the M memory must be loaded with the region label number of each pixel under consideration, and second, the CAM memory must be updated with all equivalence relationships discovered. For example, if region 4 is actually identical to region 2, then
both CAM(2) and CAM(4) will contain 2 (the lower numbered region label takes precedence. Hence, when the host computer interrogates pixel \((x, y)\) of the memory, the interface/processor interprets \(M(x, y)\) in terms of the CAM memory and returns CAM\((M(x, y))\) to the computer. Whenever an equivalence relationship is detected, all locations in the K memory containing the larger region label number are loaded with the smaller region label number. While the execution of this step in real time is not within the capabilities of conventional random access memories, it is within the capability of the content-addressable memories.

The architecture used to implement the algorithm is shown in figure 1. The architecture contains four major components: image \(I\), region label memory \(M\), equivalence CAM memory, and an interface/processor. The region labels assigned to individual pixels are contained in the region label memory. However, the contents of the \(M\) memory also include all intermediate region labels for which equivalence labels were determined.

The \(M\) memory is a conventional random access memory. However, the equivalence memory has two modes of operation. It may be used as a conventional RAM where the address in corresponds to the region table, and data out is the equivalent table. In the associative memory mode, it is used to update that table. In this mode, two activities occur in synchronism with a 2-phase clock:

Phase 1--all memory cells whose contents match the contents of the data bus, set their corresponding enable flip-flops. (see figure 5)

Phase 2--all memory cells whose enable flip-flops are set, read the contents of the data bus.

This operation effectively updates the equivalence table in parallel during the scan.
Thus when two regions are found to be identical (step 6 below), all locations in the CAM memory containing the larger region number are changed to the smaller region label number, thus allowing regions to be grown in a single pass over the image.

Algorithm: Region Growing

C - current pixel
N - previous pixel to C on current scan line
A - pixel from previous scan line which is "topographically" above current pixel
P - previous pixel to A on previous scan line

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>C</td>
</tr>
</tbody>
</table>

Square template for region growing

Let the initial label number, K=1

Scan the image from left to right and top to bottom. f(i) refers to the image brightness at point i. In this description only binary-valued images are considered. The extension to grey-valued images is straightforward.

1. If f(C) = 0
   then label (C) = 0
   else
   begin
   2. If f(N) = f(C) = 1 and f(P) = f(A) = x
      then label (C) = label (N)
      comment: X 0
   3. If f(P) = f(A) = f(N) = f(C) =1
      then label (C) = label (N)
      comment: 1 1
   4. If f(A) = f(C)=1, and f(P) = x, and f(M) = 0
      then label (C) = label (A)
      comment: X 1
   5. If f(C) = 1 and f(A) = f(N) = 0 and f(P) = x
      then label (C) = K; CAM(K) = K
      comment: X 0
K = K+1 ; A new region

6. If \( f(C) = f(\&N) = f(N) = 1 \) and \( f(P) = 0 \)
then

7. If label (A) < label (N)
then

\[
\begin{align*}
\text{label (C)} & = \text{label (A)} \\
\text{CAM(N)} & = \text{CAM(A)} \quad \text{(update)}
\end{align*}
\]

Else

\[
\begin{align*}
\text{label (C)} & = \text{label (N)} \\
\text{CAM(A)} & = \text{CAM(N)} \quad \text{(update)}
\end{align*}
\]

Continue till finished

END

2.2 Circuit Description

2.2.1 CAM1 Chip

This content-addressable memory contains the equivalencies between regions and has two modes of operation. In the first mode, it behaves like a conventional RAM and is used in this mode when a new region is encountered. The first pixel in a new region cannot be equivalent to any other region. Therefore, each cell in the CAM is initialized to contain its own address. This is illustrated in step 5 of the algorithm. CAM(i) refers to the contents of address i in the CAM. Thus, initially, CAM(i)=i.

In the associative memory mode, the CAM updates the equivalencies. When the chip is in this mode, two functions occur in synchronism. The word to be updated is placed on the data bus of the CAM. All memory cells whose contents match this word set their flip-flops. Next, the replace word is placed on the data bus and all memory cells whose flip-flops were set are now changed to the replace word. This operation has now merged all regions which were found to be equivalent. An individual cell in the CAM may be found at different times to be equivalent to many different regions and be updated several times as a result.
Figure 2 shows a block diagram of the CAM 1 chip, illustrating the use of the common data bus and enable flip flop. Appendix 2 contains a complete description of the CAM 1 chip, as well as simulation and performance analysis results.

2.2.2 CAM2 Chip

The purpose of the CAM2 chip is to update the current scan line when an equivalence is found; as a result, this will eliminate the time consuming read to CAM1.

In steps 6 and 7 of the algorithm, an equivalence between two regions is found. Here, CAM1 has to be told, for instance, that region 3 is equivalent to region J. That is, at cell 3 in the CAM1, a data 1 needs to be written. Also, before the next pixel can be interrogated, M memory will be written with the smallest of these two labels. (In this case, a 1 is written into M.)

If all pixels on the current scan line that have been labeled as region 3 have not already been changed to region label 1, a read to CAM1 will be necessary to find out if region 3 is equivalent to any other region. Instead of having to read cell 3 of CAM1 (a slow process), CAM2 was designed to change all region labels that were labeled as a 3 to region label 1 on the current scan line. The CAM2 chip needs only to hold one raster scan line of labeled regions to perform this function. Figure 3 shows a block diagram of the CAM2 chip, and figure 4 shows the circuit layout.

The CAM2 chip (Figure 3), consists of eight input pins called VL_n, and two more sets of eight input pins called Replace and Compare. The chip has an output port called VL_A and three control lines, latch, replace, and VL_A enable. The chip behaves as a regular shift register except when it is given a replace control signal.
When the replace signal is high, every word on the previous Raster Scan line is bit by bit compared with the eight bit compare register. Every word which is "true" to this compare operation will at the trailing edge of \( t + \Delta t \) be replaced by the contents of the replace register. If the replace control line was not high the words are not clocked again. The inputs replace and compare are not latched by the CAM2 package and are assumed to be valid throughout the duration of the replace command.

2.2.3 System Description

The form pixels (binary valued) to be tested by the hardware are defined as follows:

<table>
<thead>
<tr>
<th>Previous Line</th>
<th>P</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Line</td>
<td>N</td>
<td>C</td>
</tr>
</tbody>
</table>

C - Current pixel [ any pixel to the right of C is currently undefined ]
N - Previous pixel to C on current scan line
A - pixel from previous scan line which is "topographically" above current pixel
P - previous pixel to A on previous scan line

The following six test conditions satisfy all possible logical combinations for a four neighbor connectivity and serve as appropriate control signals.

\[
\overline{C} \quad CN\overline{A} \quad ACNP \quad CA\overline{N} \quad \overline{ACN} \quad ACNP
\]

Only one condition will be true at any given pixel evaluation.

Refer to figure 5 for the system block diagram.

---

Case 1:

\[
\begin{array}{cc}
\overline{C} & X \\
X & X \\
0 & X \\
\end{array}
\]

Whenever the current pixel isn't a logical 1, that pixel is to be unconditionally labeled as a zero.
Bus Connections for Case 1:

1. Zeros are placed on the data bus, VL_N, and VL_N-1.
2. Latch signals are sent to VL_N, VL_N-1, and a write signal is sent to M-memory.
3. The address counter to M-memory is incremented.

Case 2: CNA 1 1

This condition arises when the current and previous pixel are at logic 1. The current pixel is to be labeled identically as the previous pixel.

Bus Connections for Case 2:

1. The contents of VLn is gated onto VL_N and the data bus.
2. A latch signal is sent to VLn and a write signal is sent to M-memory.
3. The address counter to M-memory is incremented.

Case 3: ACNP 1 1

Here, all four of the test pixels are at logic 1. The current pixel is to be labeled identically as the previous pixel.

Bus Connections for Case 3:

Same as for CNA.

Case 4: CAN \overline{X} 1

Here the current pixel and the above pixel are at logic 1. Current pixel is to be labeled identically to its above pixel.
Figure 3: Organization of CAM2
Bus Connections for Case 4:

1. Wait for VLa to propagate through CAM 2 package.
2. The contents of VLA is gated to the data bus, VLN-1, and VLN.
3. A latch signal is sent to VLN-1, VLN, and a write signal is sent to M-memory.
4. The address counter for M-memory is incremented.

Case 5:

The current pixel is at logic 1, but none of its test pixels are true. This condition shows the appearance of a new label region. The label counter is to be incremented and the current pixel is labeled from the incremented label counter.

Bus Connections for Case 5:

1. The label counter is gated onto the data bus, CAM buses, VL1-1, and VLN.
2. A write signal is sent to M-memory and to the CAM.
3. The address counter to the M-memory is incremented.

Case 6:

The current, previous, and above pixels are at logic 1, while the previous pixel to A is at logic 0. The contents of VLA contain the above label and VLN-1 holds the previous label. These two labels are compared and the current pixel is labeled from the smallest of the two. The CAM and the CAM 2 chip are updated accordingly.
Bus Connections for Case 6:

1. Wait for \( VLA \) to propagate through CAM 2 package.
2. The contents of \( VLA \) is gated onto the comparator inputs and latched for future access.
3. The contents of \( VLN_{-1} \) is gated onto the comparator.
4. The comparator is enabled.

When \( VLA < VLN_{-1} \)

a. The contents of \( VLN_{-1} \) is gated onto the CAM 2 compare inputs, and onto the CAM address bus.

b. The contents of \( VLA \) is gated onto the CAM 2 replace inputs.

c. A replace signal is sent to the CAM 2 package and a union signal to the CAM.

d. After one CAM delay, the contents of \( VLA \) is gated to CAM data inputs.

e. \( VLA \) is placed onto the data bus and latch signals are sent to \( VLN_{-1} \), \( VLN \) and a write signal is sent to M-memory.

When \( VLA > VLN_{-1} \)

a. The contents of \( VLN_{-1} \) is gated onto the CAM 2 compare inputs, and onto the CAM address bus.

b. The contents of \( VLN_{-1} \) is gated onto the CAM 2 replace inputs.

c. A replace signal is sent to the CAM 2 package and a union signal to the CAM.

d. After one CAM delay, the contents of \( VLN_{-1} \) is gated to CAM data inputs.

e. \( VLN_{-1} \) is placed onto the data bus and latch signals are sent to \( VLN \) and a write signal is sent to M-memory.

3. DCM Description of System

In this section, an overview of DCM is provided, followed by a description of this system in DCM format, and a discussion of the effectiveness of the representation.
3.1 Description of DGM

The DGM software, as supplied, consists of two parts: a directed graph editor (DGMED) and an ADA package library manager (DGMLM). Both are written in VAX (VMS) Pascal.

DGM is intended to be a hierarchical system design and analysis tool. A system is represented as a directed graph. Each vertex in the graph represents a system function and arcs designate data flows between vertices. Arcs have attributes such as produce, consume, threshold and capacity. These attributes are related to the amount of data at a node input that must be present before a node can fire, and to the amount of data that is produced and consumed when a node does fire.

Vertex functions are implemented by ADA packages assigned to the vertices from a library of packages. A set of processor assignments can be specified for each package as an aid in mapping the flow graph onto an architecture.

The methodology supports a top down design strategy. A design is refined by expanding higher level nodes into more detailed subgraphs until the desired level of refinement is reached. Each node in the graph has an ADA package assigned which performs the node function. The use of flow graphs at all levels of the hierarchy provides a uniform, consistent representation of the system and can provide a convenient mechanism for moving up and down the hierarchy.

3.2 Using DGM To Construct A Data Flow Graph

The process of constructing a flow graph begins by using DGMLM, the library manager, to enter the ADA package definitions of vertex functions into the package library. DGMLM maintains a library of functions, so only new functions need to be entered.
Information required for a package is its name and the specification of its inputs, outputs and data types. Produce, consume and threshold attributes can also be specified for each package. Only ADA package header information is kept by the library manager. The actual code bodies would be included when the graph description was compiled.

DGMLM itself is a menu driven program which allows for addition, deletion, modification and display of package definitions. The most serious shortcoming of DGMLM is that although a list of packages currently in the library is available, it is difficult to tell what function a particular package performs. The package name and inputs and output data descriptions are available, but there is no provision for a text description of what the package does. Clearly a package name can provide some indication of function as can knowledge of the inputs and outputs, but this is not sufficient. A text description capability would be a useful addition.

This makes the use of package definitions already in the library very difficult, and requires the entry of new definitions and much external bookkeeping to keep track of what each package does for each new flow graph. The next step is the entry of the graph description using DGMED. DGMED, also a menu driven program, allows for the creation and modification of flow graphs. Vertex name and function definitions are entered as well as the connectivity and attribute information provided by the arcs. ADA package assignments are also made to each node.

The major shortcoming of DGMED is its lack of a graphic data entry and poor display capability. While the menu driven approach is simple to use, it makes verification of the correct construction of a flow graph difficult. Verification must be done by examining a text description of the graph and comparing it to a mental picture or a hand drawn prototype. The graphic display capability provided is very primitive and not very useful.
DGMED also makes it difficult to maintain more than one graph at a time in the same directory. The creation of a new graph destroys the old graph, since the same files are used for the graph description. To maintain different graphs requires renaming files or moving files to another directory and starting over. This must be done by the user.

3.2 Modeling The Region Labeling System Using DGM

A data flow graph of the system is shown in figure 6 and a block diagram is shown in figure 5. Appendix 3 contains a tabular summary of the circuit flow graph. Appendix 4 contains the ADA package definitions and Appendix 5 contains a description of the graph in DGM notation.

4. Evaluation

The basic thrust of DGM, that of representing a system as a data flow graph, has significant potential as a design tool. However, the utility of a design aid is directly related to the information that can be extracted from the design representation. The DGM software, as it exists at NCSU, is primarily for the entry and maintenance of data flow graphs and the package library. Few graph analysis tools currently exist.

The ability to obtain information from the graph at all levels of the hierarchy is important. This information can be then used to analyze and improve the design. The information required can change at different stages of the design.

In the initial stages of a design, functional correctness will be important. Later stages may put the emphasis on other considerations such as performance. These differing requirements mandate a variety of analysis tools.
The ability to assign ADA packages to graph nodes and the existence of graph control variables implies that some type of functional simulator is planned, but it is currently not available. This capability would be very useful in establishing functional correctness of a design and for generating test data.

DGM, as it currently stands, seems to be primarily concerned with software system design. Support for ADA software packages and processor assignments is provided, as is the ability to create new data types. In addition, data flow graphs are inherently asynchronous, while hardware systems are usually considered to be synchronous.

In the early stages of a hardware system design, a functional simulation based on software function modules could be useful. However, at some point in the design, this is no longer adequate. Hardware notions such as clocks, registers and propagation delays are probably better represented in a hardware description language and simulator than in a general purpose language such as ADA. Thus the ability to assign both hardware and software function modules to graph nodes would be an important addition to DGM.

5. Conclusion

Our basic conclusion is that DGM has the potential to be a valuable design tool for both hardware and software system design. Flow graphs can provide a convenient and useful representation of a system hierarchy. However, the asynchronous nature of data flow graphs does not well model tightly coupled, synchronous hardware systems.

The ultimate utility of any design aid depends on the information it provides the designer. In the case of DGM, this requires the further development of tools which can extract such information from the flow graph representation.
A similar design system, based on many of the ideas of DGM, is under development at the Research Triangle Institute in North Carolina. This system has a color graphics data input and display, and a variety of analysis tools. These include a dynamic graph simulator, an analyzer based on a Petri net model of a graph and a hardware description language interface.
Figure 1
Figure 2: Organization of the K Memory
Appendix I.

Content-Addressable Read/Write Memories for Image Analysis

by
Wealey E. Snyder
Carla D. Savage

Appendix 2

Design of a Content-Addressable RAM

by

Robert Tyszczenko
1. Introduction

This chapter has two major components: 1) a decoder and 2) a memory cell with attached logic. These two components have been designed and, to some extent, tested. Figure 1 shows what one word of memory looks like at its highest level.

The three major operations consist of two that are fairly straightforward, the Read & Write of a memory cell. The third, Union, requires the extra logic in the "smart memory." Because of the variety of operations being performed, a 4-phase clock is used, rather than pipelining. Before an operation begins, the previous operation is completely over.

To complete the chip, some logic and pass transistors need to be designed to regulate the flow of data & addresses from pads to their destination. In particular, the fact that input and output is done with the same pad and drivers causes a problem on and between the Read & Union operations. A solution is proposed later in this report.

The basic operation of the circuit is best understood by reading the "Timing Conventions" data, and the "Mixed Notation" illustration in conjunction with the following explanation.

Since this circuit uses mostly nor logic, inputs to indicate a Read, Write, or Union, are active when low. Note also that the decoder which selects a given data word requires two phases for operation. For a Read or Write, a memory location is specified by the decoder. Dropping the appropriate control (Read, Write) line completes the operation. The Union operation is not done with decoder assistance. It occurs because a "flag" was set (by xor logic) to indicate that one or more memory locations match a data registers contents. All cells that have their "flag" set will be rewritten with the new data placed in the data register on $2.$
In what follows, in a filename such as xor.ab, the .ab tells ABCD that the file contains ABCD text. Wires are frequently labelled with something like: wire-N at the top and: wire_s at the bottom. This facilitates simulation because qrs assumes that they are one node. Labels are required whenever a wire at the periphery of a cell is to connect to another cell or to a wire outside of the present cell.

2. Description of Cells

2.1 mcull.ab (fig. 6)

This is the memory itself. This design was chosen because of the simple refresh control, performed by clocking a pass transistor on φ₁, and the requirement that both the true and complement form of the memory cell be available at all times.

Notice that reading is controlled by ren_e/ren_w. The signal on this line is generated by a read enable logic cell called renccell.ab. Writing to memory is more complicated since it can occur as: 1) a simple RAM write, 2) a Union operation write. Writing is controlled by a signal on union_e/union_w (from uenable.ab) or by a signal on ram_e/ram_w (from wencell.ab).

2.2 xor.ab (fig. 7)

Performs the xor function. If the contents of memory match the contents on the data bus then xor_out will go to Vss. Note that the pulldowns (pd.) appear to form two legs—one to the left and the other to the right of the pullup (pu.). Since at most one leg will have a path to Vss:

\[
\begin{align*}
\text{pu.} & : \frac{1}{4} \\
\text{pd.} & : \frac{1}{2}
\end{align*}
\]

and pass transistors are avoided.
2.3 pulldn.ab (fig. 8)

This cell is essential for the Union operation. The wire labelled 
pwr_w/pwr_e is precharged on 1. Assume the contents of memory match the 
contents of a data register to which it is compared. The cell xor.ab does the 
compare. Since the two are equal, xin_n is at \( V_{SS} \), and pwr_w/pwr_e stays 
high. This is the "flag" that indicates that a write should occur for this 
memory cell on 3. The logic to generate the enable signal is in uenable.ab. 
The cell otl.ab is affected too.

2.4 slice.ab (fig. 9)

The constituents of this cell are 1) moell.ab; 2) xor.ab and 3) 
pulldn.ab.

2.5 connect.ab (fig. 10)

This cell is composed simply of wires. The following wires come from 
off-chip: 1) penable_n/penable_s to otl.ab 
   2) gndenab_s/gndenab_N to mcell.ab 
   3) Vss_n/Vss_e to mcell.ab 
   4) Mbar_n/Mbar_s to uenable.ab 
   5) Wbar_n/Woar_s to cencell.ab 
   6) Rbar_s/Rbar_N to rencell.ab 

The following wires are generated on chip: (actually the signals on them 
are generated on-chip)

renable_w/renable_s - from rencell.ab to mcell.ab 
senable_w/uenable_s - from uenable.ab to mcell.ab 
wenable_w/wenable_s - from wencell.ab to mcell.ab
2.6 \texttt{ctl.ab} (fig. 11)

This cell is used during Union operations. During \( \phi_1 \), the upper pass transistor is on which charges the wire labelled \texttt{pwr_w/pwr_e}. The charge is stored on an inverter attached to \texttt{pwr_e} and resides in \texttt{uenable.ab}. The lower pass transistor is off and means that the charge remains even if the previous state of \texttt{pulldn.ab} would have allowed it to discharge. After the output of \texttt{xor.ab} settles (by \( \phi_2 \) hopefully) the lower pass transistor is turned on by \( \phi_2 \). If the memory cell (all 10 bits) differs from the data that it was compared to, \texttt{pwr_w/pwr_e} and the gate in \texttt{uenable.ab} will discharge.

2.7 \texttt{rencell.ab} (fig. 12) and \texttt{wencell.ab} (fig. 13)

Both cells perform the nor function. Both are used when operating in the RAM mode. Both share an active low input from the decoder. Either \texttt{Wbar_n/Wbar_s} or \texttt{Rbar_n/Rbar_s} can go to \( V_{ss} \) if their respective operations (Write, Read) are being performed. They should not both be low at the same time. Their outputs enable the Read or Write by activating pass transistors in \texttt{mcell.ab}.

2.8 \texttt{uenable.ab} (fig. 14)

Basically an inverter and a nor gate. If the inverter has a low input this implies that a mismatch between the memory cell and the data register occurred causing \texttt{xor.ab} to output a high signal which discharged \texttt{pulldn.ab} and the gate of this inverter. Despite the fact that \texttt{Mbar_n/Mbar_s} may be at \( V_{ss} \) (for Union operation) nothing will happen. Similar reasoning will reveal that the Union operation will occur if the memory contents match the data register contents.
2.9 Decoder: in general

The decoder was designed such that it dissipates no static power, which justifies its larger size.

This decoder can have 256 outputs and yet be built with little more than a proper arrangement of:

1) dec00.ab
2) dec01.ab
3) dec11.ab and
4) decout.ab attached to provide the outputs.

For example, let us look at how to arrive at the arrangement in figure 3.

We want 4 outputs.

Count in binary:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This is easily extended (but tedious).

I allow for 10 inputs even though \( \log_2 256 \) seem sufficient because the 2 high order bits can, effectively, act as chip select inputs. (Recall that 4 chips each with 256 locations are expected in the final configuration)

3. Timing Conventions

To write:

*1: Latch data. Latch address to decoder. Refresh memory.*

*2: Let decoder select a word.*

*3: Drop Write control line.*

*4: Raise Write control line.*
To Read:

\*1: Latch address to decoder. Refresh memory. Precharge data lines if desired by placing Vdd on I/P pads.

\*2: Let decoder select a word. Drop Read control line.

\*3: Latch \( \overline{A/p} \) to pads.

\*4: Raise Read control line.

To Union

\*1: Precharge pulldn.ab. Refresh memory. Latch I/P data.

\*2: Enable ground in pulldn.ab and otl.ab cells

\*3: Latch new data. Lower Mode control line.

\*4: Raise Mode control line.

4. Testing

4.1 Decoder Test: (figs. 2&3)

\textit{Dectest.ab} (not capitalized) represents the decoder that was tested (fig. 2). As above, \textit{ats} required that I create a file called \textit{decoid.ab}. In either case, what was tested could be called a low-going 1-of-4 decoder. Even though the pu/pd ratio was about 2 instead of 4, a successful simulation is depicted in figure 3.

For \textit{gds}: the spicefile is : spfiledec

the clockfile is : clkfiledec

5. Pincount and Estimate of Transistor Count

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Pins} & \textbf{Count} \\
\hline
\textit{AO} - \textit{A9} & 10 \\
\textit{DO} - \textit{D9} & 10 \\
\textit{Vdd\&Vss} & 2 \\
\textit{4-phase clk} & 4 \\
\textit{MODE} & 1 \\
\textit{WRITE} & 1 \\
\textit{READ} & 1 \\
\textit{penable} & 1 \\
\textit{genable} & +1 \\
\hline
\end{tabular}
\end{center}
Transistor Count:

\[
mcell.ab, xor.ab, pulldn.ab : 14/\text{slice} \Rightarrow 140/\text{word} \\
total \text{ control logic} : + 13/\text{word} \\
\]

\[
152 \times 256 = 39,168 \\
+ 5,120 \quad \text{(Decoder, Cmos type)} \\
\]

\[
44,288 \quad \text{(Decoder, Cmos type)} \\
\]

Solution to problem posed in introduction

\( \phi_3 \) conflict occurs between action for Read and for Union

To Read: we need something like this:

```
Read \quad \phi_3 \quad \text{latch output}
```

To Union:

```
Union \quad \phi_3 \quad \text{latch I/P}
```
FIGURE 7

mcell
xor
pulldn
FIGURE 9
FIGURE 1:
FIGURE 12
Appendix 3

Tabular Representation of a Data Flow Graph
Summary of graph CAMCHIP

<table>
<thead>
<tr>
<th>QUEUE</th>
<th>THRESHOLD</th>
<th>READ</th>
<th>CONSUME</th>
<th>CAPACITY</th>
<th>PRODUCE</th>
<th>DATA-TYPE</th>
<th>INIT</th>
<th>SOURCE</th>
<th>SINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLN</td>
</tr>
<tr>
<td>L2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>CAM</td>
</tr>
<tr>
<td>L3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>MEMM</td>
</tr>
<tr>
<td>L4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLSI</td>
</tr>
<tr>
<td>Z1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>ZERO</td>
</tr>
<tr>
<td>Z2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLN</td>
</tr>
<tr>
<td>Z3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLSI</td>
</tr>
<tr>
<td>A1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>ADDCNT</td>
</tr>
<tr>
<td>V1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>CAM</td>
</tr>
<tr>
<td>V2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLN</td>
</tr>
<tr>
<td>V3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLSI</td>
</tr>
<tr>
<td>V1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLSI</td>
</tr>
<tr>
<td>V2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLSI</td>
</tr>
<tr>
<td>V3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLSI</td>
</tr>
<tr>
<td>VL4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>MEMM</td>
</tr>
<tr>
<td>VL5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>VLSI</td>
</tr>
<tr>
<td>MEMOU</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>MEMM</td>
</tr>
<tr>
<td>CAMOU</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>F</td>
<td>CAN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NODE</th>
<th>PACKAGE</th>
<th>1ST PROCESSOR</th>
<th>2ND PROCESSOR EXCLUDES</th>
<th>SHARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABELCNTR</td>
<td>LABELCNTR</td>
<td>1</td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>COMPARE</td>
<td>COMPARE</td>
<td>3</td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>VLSI</td>
<td>VLSI</td>
<td>4</td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>ZERO</td>
<td>ZERO</td>
<td>5</td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>VLN</td>
<td>VLN</td>
<td>6</td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>ADDCNTR</td>
<td>ADDCNTR</td>
<td>7</td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>MEMMEM</td>
<td>MEMMEM</td>
<td>10</td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>CAM</td>
<td>CAM</td>
<td>VL8</td>
<td></td>
<td>FALSE</td>
</tr>
</tbody>
</table>

End of graph CAMCHIP
Appendix 4

ADA Package Definitions
package LABELCNTR is

  procedure GO_LABELCNTR ( )

    -- output queues in package
    OUT_QUEUE_1: out array(1..1) of INTEGER ;
    OUT_QUEUE_2: out array(1..1) of INTEGER ;
    OUT_QUEUE_3: out array(1..1) of INTEGER ;
    OUT_QUEUE_4: out array(1..1) of INTEGER ;

  )

end LABELCNTR ;

package CAM is

  procedure GO_CAM ( )

    -- input queues in package
    IN_QUEUE_1: in array(1..1) of INTEGER ;
    IN_QUEUE_2: in array(1..1) of INTEGER ;
    IN_QUEUE_3: in array(1..1) of INTEGER ;

    -- output queues in package
    OUT_QUEUE_1: out array(1..1) of INTEGER ;

  )

end CAM ;

package ADDCNTR is

  procedure GO_ADDCNTR ( )

    -- output queues in package
    OUT_QUEUE_1: out array(1..1) of INTEGER ;

  )

end ADDCNTR ;
package MMEM is

procedure GO_MMEM ()

-- input queues in package
IN_QUEUE_1: in array(1..1) OF INTEGER ;
IN_QUEUE_2: in array(1..1) OF INTEGER ;
IN_QUEUE_3: in array(1..1) OF INTEGER ;
IN_QUEUE_4: in array(1..1) OF INTEGER ;

-- output queues in package
OUT_QUEUE_1: out array(1..1) OF INTEGER ;
);

end MMEM ;

package ZERO is

procedure GO_ZERO ()

-- output queues in package
OUT_QUEUE_1: out array(1..1) of INTEGER ;
OUT_QUEUE_2: out array(1..1) of INTEGER ;
OUT_QUEUE_3: out array(1..1) of INTEGER ;
);

end ZERO ;

package VLSI is

procedure GO_VLSI ()

-- input queues in package
IN_QUEUE_1: in array(1..1) OF INTEGER ;
IN_QUEUE_2: in array(1..1) OF INTEGER ;
IN_QUEUE_3: in array(1..1) OF INTEGER ;

-- output queues in package
OUT_QUEUE_1: out array(1..1) OF INTEGER ;
);
-- output queues in package

OUT_QUEUE_1: out array(1..1) of INTEGER;
OUT_QUEUE_2: out array(1..1) of INTEGER;
OUT_QUEUE_3: out array(1..1) of INTEGER;
OUT_QUEUE_4: out array(1..1) of INTEGER;

end VLSI;

package COMPARE is

procedure GOCOMPARE ()

-- input queues in package

IN_QUEUE_1: in array(1..1) OF INTEGER;
IN_QUEUE_2: in array(1..1) OF INTEGER;

end COMPARE;

package VLN is

procedure GOVLN ()

-- input queues in package

IN_QUEUE_1: in array(1..1) OF INTEGER;
IN_QUEUE_2: in array(1..1) OF INTEGER;
IN_QUEUE_3: in array(1..1) OF INTEGER;

-- output queues in package

OUT_QUEUE_1: out array(1..1) OF INTEGER;
OUT_QUEUE_1: out array(1..1) OF INTEGER;
OUT_QUEUE_2: out array(1..1) OF INTEGER;
OUT_QUEUE_3: out array(1..1) OF INTEGER;

end VLN;
Appendix 5

Data Flow Graph in DGM Notation
graph CAMCHIP contains;

package LABELCNTR has

    output =

L1

    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

L2

    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

L3

    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

L4

    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

package CAM has

    input =

VL3

    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

L2 threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER
V1
threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER

output =
CAMOUT

threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER

package PDCN-1:

has

output =
A1

threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER

package MEM:

has

input =
A2

threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER

L3
threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER

VL4
threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER
Z1
threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER

output = MEMOUNT
threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER

package ZERO
输出 =
Z1
threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER
Z2
threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER
Z3
threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER
package VLSI
  has
    input =
      V3
        threshold = 1
        read = 1
        consume = 1
        capacity = 1
        produce = 1
        data_type = INTEGER
      L4
        threshold = 1
        read = 1
        consume = 1
        capacity = 1
        produce = 1
        data_type = INTEGER
      Z3
        threshold = 1
        read = 1
        consume = 1
        capacity = 1
        produce = 1
        data_type = INTEGER
    output =
      VL1
        threshold = 1
        read = 1
        consume = 1
        capacity = 1
        produce = 1
        data_type = INTEGER
      VL2
        threshold = 1
        read = 1
        consume = 1
        capacity = 1
        produce = 1
        data_type = INTEGER
      VL3
      VL4
package COMPARE has

input =

V1
    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

VL2
    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

package VL.SN has

input =

L1
    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

VL1
    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

Z2
    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER

output =

V3
    threshold = 1
    read = 1
    consume = 1
    capacity = 1
    produce = 1
    data_type = INTEGER
V2

threshold = 1
read = 1
consume = 1
capacity = 1
produce = 1
data_type = INTEGER

queue L1 has type = DATA := 0 ;
queue L2 has type = DATA := 0 ;
queue L3 has type = DATA := 0 ;
queue L4 has type = DATA := 0 ;
queue Z1 has type = DATA := 0 ;
queue Z2 has type = DATA := 0 ;
queue Z3 has type = DATA := 0 ;
queue A1 has type = DATA := 0 ;
queue V1 has type = DATA := 0 ;
queue V2 has type = DATA := 0 ;
queue V3 has type = DATA := 0 ;
queue VL1 has type = DATA := 0 ;
queue VL2 has type = DATA := 0 ;
queue CL3 has type = DATA := 0 ;
queue CL4 has type = DATA := 0 ;
queue MEMOUT has type = DATA := 0 ;
queue CAMOUT has type = DATA := 0 ;

node LABELCNTR has package LABELCNTR with
processor = 1
priority = 1
shareable = FALSE
output = L1, L2, L3, L4
node COMPARE has package COMPARE with
processor = 3
sharable = FALSE
output = V2 , VL2 , L3 , L4

node VLSI has package VLSI with
processor = 4
priority = FALSE
sharable = ZERO
output = VL1 , VL2 , VL3 , VL4

node ZERO has package ZERO with
processor = 5
sharable = FALSE
output = Z1 , Z2 , Z3 , L4

node VLN has package VLN with
processor = 6
priority = FALSE
sharable = L1
output = V1 , V2 , V3 , L4

node ADDCNTR has package ADDCNTR with
processor = 7
sharable = FALSE
output = A1 , L2 , L3 , L4

node MEMMEM has package MMEM with
processor = 10
sharable = FALSE
input = L1
output = MEMOUT , A1 , Z1 , VL4

node CAM has package CAM with
processor = VL8
sharable = FALSE
INPUT = VL3
output = CAMOUT , V1 , L2

endgraph CAMCHIP