Purdue University
Purdue e-Pubs

Department of Electrical and Computer Engineering Technical Reports

Department of Electrical and Computer Engineering

12-1-1984

# Parallel Algorithms for Isolated and Connected Word Recognition

Mark Alan Yoder
*Purdue University*

Leah H. Jamieson
*Purdue University*

Follow this and additional works at: https://docs.lib.purdue.edu/ecetr

Yoder, Mark Alan and Jamieson, Leah H., "Parallel Algorithms for Isolated and Connected Word Recognition" (1984). *Department of Electrical and Computer Engineering Technical Reports.* Paper 531.
https://docs.lib.purdue.edu/ecetr/531

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

# Parallel Algorithms for Isolated and Connected Word Recognition

Mark Alan Yoder
Leah Jamieson

TR-EE 84-47
December 1984

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

# PARALLEL ALGORITHMS FOR ISOLATED AND CONNECTED WORD RECOGNITION

Mark A. Yoder

Leah H. Jamieson

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Purdue University
TR-EE 84-47
December 1984

This work was supported by National Science Foundation grants ECS-7909016 and ECS-8120896.

## ACKNOWLEDGMENTS

The authors would like to thank *Sharon Katz* for the excellent job she did preparing the figures, and for the use of her equipment in assembling the final version. We also would like to thank *Sarah K. Yoder* for the time she spent in proof reading and correcting the next to final draft.

Thanks go also to *Steven J. Holmes* for the hours he spent explaining the internals of the Poker System to me, and to *James T. Kuehn* for his help with the SIMD machine simulations.

This work was supported by National Science Foundation grants ECS-7909016 and ECS-8120896.

## TABLE OF CONTENTS

Page

Page

APPENDICES

## LIST OF TABLES

# LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Appendix
Figure

Appendix
Figure Page

Appendix
Figure — Page

# ABSTRACT

Mark Alan Yoder, Ph.D., Purdue University. December 1984. Parallel Algorithms for Isolated and Connected Word Recognition. Major Professor: Leah H. Jamieson.

For years researchers have worked toward finding a way to allow people to talk to machines in the same manner a person communicates to another person. This verbal man to machine interface, called speech recognition, can be grouped into three types: isolated word recognition, connected word recognition, and continuous speech recognition. *Isolated word recognizers* recognize single words with distinctive pauses before and after them. *Continuous speech recognizers* recognize speech spoken as one person speaks to another, continuously without pauses. *Connected word recognition* is an extension of isolated word recognition which recognizes groups of words spoken continuously. A group of words must have distinctive pauses before and after it, and the number of words in a group is limited to some small value (typically less than six).

If these types of recognition systems are to be successful in the real world, they must be speaker independent and support a large vocabulary. They also must be able to recognize the speech input accurately and in real time. Currently there is no system which can meet *all* of these criteria because a vast amount of computations are needed.

This report examines the use of *parallel processing* to reduce the computation time for speech recognition. Two different types of parallel architectures are considered here, the *S*ingle *I*nstruction stream - *M*ultiple *D*ata (SIMD) machine and the VLSI processor array. The SIMD machine is chosen for its flexibility, which makes it a good candidate for testing new speech recognition algorithms. The VLSI processor array is selected as being good for a dedicated recognition system because of its simple processors and fixed interconnections.

This report involves designing SIMD systems and VLSI processor arrays for both isolated and connected word recognition systems. These architectures are evaluated and contrasted in terms of the number of processors needed, the interprocessor connections required, and the "power" each processor needs to achieve real time recognition.

The results show that an SIMD machine using 100 processors, each with an MC68000 processor, can recognize isolated words in real time using a 20 KHz sampling rate and a 1,000 word vocabulary.

# 1. INTRODUCTION

Voice input to machines is one of the most natural forms of man-machine communication. For years researchers have worked toward finding a way to allow a person to talk to machines in the same manner a person communicates to another person. This verbal man to machine interface, called speech recognition, can be grouped into two major types, continuous speech recognition and isolated word recognition. The following describes what each type entails.

The computer's role in *continuous speech recognition* is analogous to the role of a secretary taking dictation in that the machine would take the voice input and transcribe it into the words that were spoken.

In *isolated word recognition* there is a distinctive pause (of about 100 ms) between each utterance. Isolated word recognition is the more likely of the two types of recognition to be found on an assembly line taking orders to do a given task. Here single words or short phrases are given to control a machine. The distinctive pauses before and after the utterance make it easier to find where the utterance begins and ends. Continuous speech may not have pauses around each utterance, which makes finding word boundaries within continuous speech more difficult than isolated speech. This is one reason why isolated word recognition is easier to perform than continuous speech recognition.

A third type of recognition is *connected word recognition*. Connected word recognition is an extension of isolated word recognition which allows recognition of groups of words spoken continuously. A group of words must have distinctive pauses before and after it, and the number of words in a group is limited to some small value (typically less than six). The presence of distinctive pauses, and the knowledge that there is only a small number of words in a group makes connected speech recognition easier to perform than continuous speech recognition. Since connected word recognition is an extension of isolated word recognition, it is not considered a major type.

For any of the types of speech recognition to be successful in general usage, they must meet the following criteria.

1) *Speaker Independence:* Many recognition systems are trained to a small group of speakers. A system is called speaker independent if it can recognize speakers not in the training group. To do this it must be able to handle different dialects, accents, speaking rates, and pitches.

2) *Large Vocabulary:* The typical adult may know 100,000 words or more [LeLi81]. Although an isolated word recognizer controlling a machine may only need to recognize a few command words, the use of continuous speech recognition to take dictation requires a large vocabulary.

3) *Accurate Recognition:* Recognition accuracy is a common standard used to compare different recognition systems. Certainly the machine should accurately recognize all utterances in order to avoid having the user repeat words, or worse yet, have the machine misrecognize words.

4) *Real-Time Response:* The response time is the time needed to decide what was spoken. Real-time response is needed so that the speaker does not grow tired waiting for an answer. In a continuous speech recognition system, real time response is needed so processing does not accumulate. This has not been achieved by a system which also met the other three characteristics.

An example of a continuous speech recognition system in the literature is the HWIM [BBN76] system that is able to understand continuous speech from three cooperative male general American speakers. It can recognize a 1,097 word vocabulary with a 56% error rate while operating at 1,350 times real time on a PDP-10.

The level building dynamic time warping algorithm by Myers and Rabiner [MyRa81b] is an example of a connected word recognition system. The system can recognize up to five words in a connected utterance. The basic operation performed by the system is a form of dynamic programming, known as a time warp, to compare the input utterance to stored templates representing the vocabulary. (Time warping will be discussed in detail in later chapters. For now, it is the complexity of the time warp process which is of interest.) With a vocabulary size of 10 words, it requires 50 basic time warps. On a Data General Eclipse S230 minicomputer, Myers et al. [MRR80] states that a basic time warp requires 289 to 454 milliseconds.* This means a vocabulary of 10 words requires 14.45 to 22.7 seconds, while a vocabulary of 1,000 words needs 24 to 38 minutes just for the dynamic time warping. Therefore, the level building method cannot run in real time with a large vocabulary on a conventional processor.

Neither of the above two systems is speaker independent, nor could they meet the real time response constraint. Currently these two constraints are met by using a simpler type of recognition, i.e., isolated word recognition. Systems are commercially available which recognize isolated words in real time [Dodd81]. Generally these systems are speaker dependent with small (10-20 word) vocabularies. Even though the real-time response is possible, it is at the expense of a small vocabulary and small speaker population.

This report investigates the use of parallel processing to reduce the computation time for speech recognition. This will be done by writing ***parallel processing algorithms*** for the component algorithms that make up the speech recognition systems.

Two different parallel architectures are considered here, the *s*ingle *i*nstruction stream - *m*ultiple *d*ata stream (SIMD) [Flyn66] computer and the VLSI processor array. In the SIMD machine many processors execute the same instructions simultaneously on different data. The instructions are broadcast from a control unit, and the processors are able to pass data between each other by a general interconnection network. The VLSI processor array, on the other hand, is a multidimensional pipeline consisting of many cells, with the output(s) of one cell connected to the input(s) of other cell(s). Although most cells will be executing the same instructions on different data, it is possible some "special" cells will be executing different instructions. The VLSI processor array can be thought of as a super systolic array [Kung80]. Both arrays are the same in that they both use a fixed interconnection network. They differ since each cell of the systolic array performs simple instructions like addition and multiplication and has a small fixed number of registers (as few as three), while each cell of the VLSI processor array can be as powerful as a

* The figures Myers gives are 57.8 to 90.8 ms for combinatorics with local distance measures requiring 80% of the computation time.

microprocessor with its own addressable memory. The systems examined are programmable parallel systems. Since speech recognition is a research area in which new methods are likely to be proposed, ***special purpose hardware*** devices (e.g. [LMMB84]) are not considered.

Chapter 2 presents the SIMD machine model and a language for writing parallel algorithms for it. Chapter 3 discusses the VLSI processor array model and gives examples of how it works. Chapter 4 describes the word template matching approach to isolated word recognition. Chapter 5 is a survey of parallel speech processing algorithms. Chapter 6 describes the new parallel speech processing algorithms developed for this report. Chapter 7 presents the results of simulating the SIMD algorithms and Chapter 8 presents the VLSI processor array simulation results. Chapter 9 discusses connected word recognition and presents a parallel algorithm for a level building dynamic time warp. And finally, Chapter 10 gives the conclusions of this research effort.

# 2. THE SIMD MACHINE MODEL

With the advent of VLSI technology, large-scale processing systems with as many as $2^{14}$ processors have become feasible [Ba79,Pe77,SBK77]. One approach to using a large number of processors is the *s*ingle *i*nstruction stream - *m*ultiple *d*ata stream (SIMD) machine. An SIMD machine typically consists of a control unit (CU), a set of $N = 2^n$ processing elements (PEs), and an interconnection network as shown in Figure 2.1 [Sieg81a]. A PE consists of a processor with its own memory, fast access general purpose registers, an address register (ADDR), and two *d*ata *t*ransfer *r*egisters (DTRin and DTRout) as shown in Figure 2.2. The PEs are addressed (numbered) from 0 to N−1 in a machine of size N. The register ADDR in PE i contains the integer i, for $0 \leq i < N$. The two data transfer registers allow each PE to access the interconnection network which in turn allow each PE to send and receive data from the other PEs [Si79]. The CU broadcasts instructions to all PEs, and each active PE executes each of these instructions on the data in its own memory. All active PEs execute each instruction simultaneously. It is possible to enable and disable PEs so all N PEs may not be active.

## 2.1. Flock Algol – Introduction

A tool called Flock Algol has been developed by Siegel et al. [Sieg81b] to aid in writing and describing parallel algorithms. Flock Algol is used here because it incorporates ways to express SIMD processing in an algorithm description language. The following summarizes Flock Algol and focuses on the constructs it uses to express and control parallel execution. Finally an example of a Flock Algol algorithm is given.

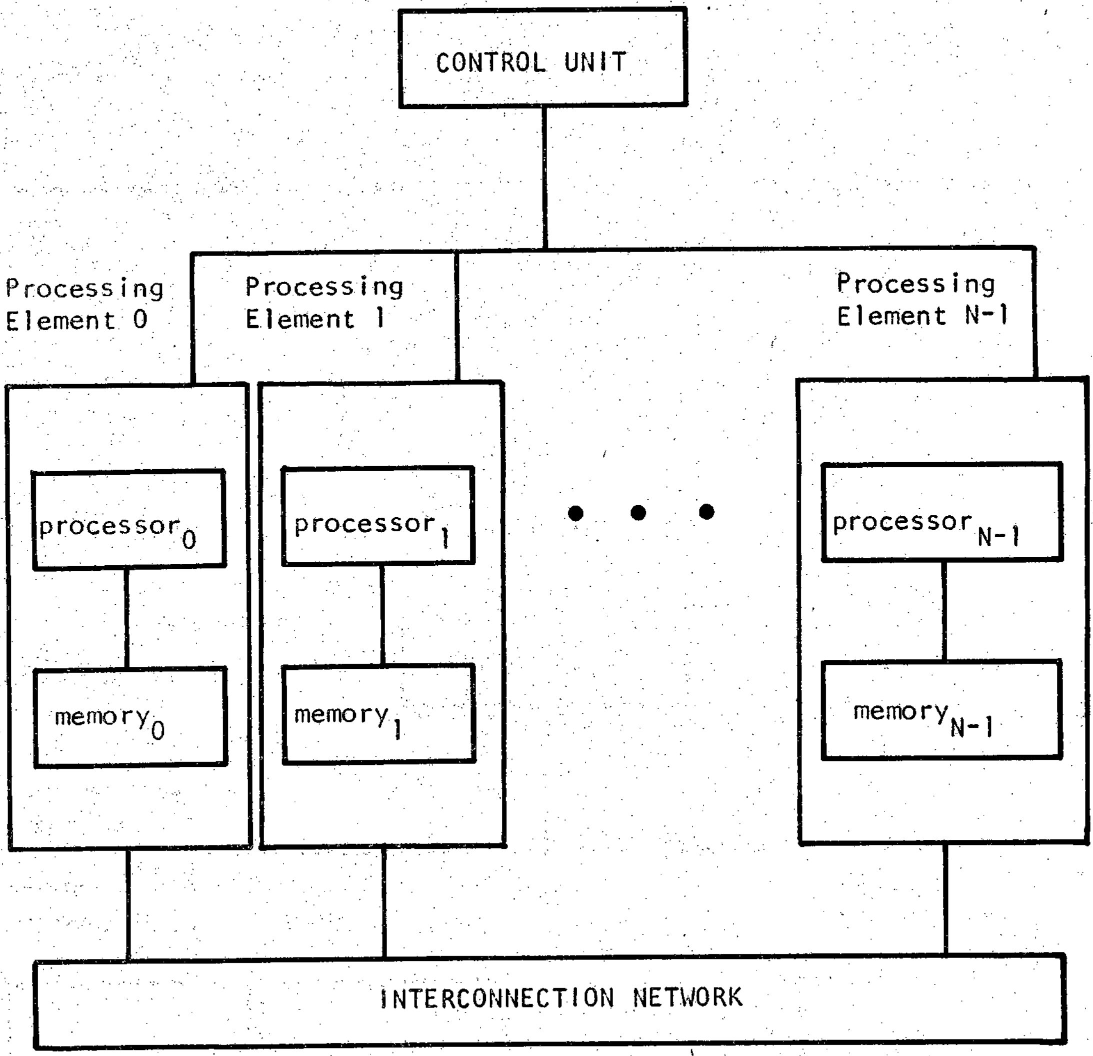


Figure 2.1. SIMD machine organization.

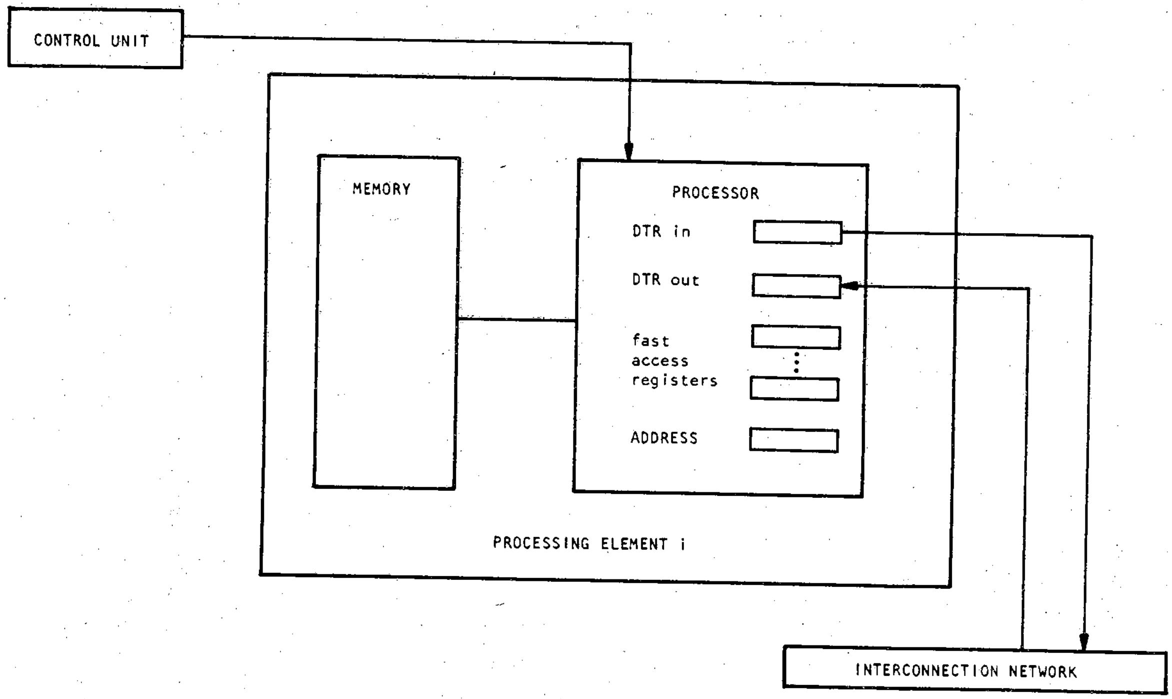


Figure 2.2. Model of an SIMD processing element (PE).

## 2.2. Summary of Flock Algol

Flock Algol uses traditional mathematical and programming language constructs, after Pidgin Algol [AHU74]. It also contains parallel-specific constructs extending its Pidgin Algol origin to accommodate parallel algorithms. As in Pidgin Algol, any statement with a clear meaning is allowed.

A Backus-Naur form (BNF) specification is used here to describe Flock Algol. A BNF statement has the form

<non–terminal> ::= sequence of terminals and/or non–terminals.

Terminals are elements of the set of language symbols. For Flock Algol the keywords include IF, THEN, ELSE, FOR, STEP, BEGIN, END, PROCEDURE, ENABLE, DISABLE, TRANSFER, BROADCAST, USE, etc. To aid the reader, Flock Algol keywords are shown in all capital letters. However, case is unimportant when expressing algorithms in Flock Algol. Nonterminals are symbols delimited by < > such as <program>, <statement>, <variable>, <expression>, <condition>, <initial value>, <step size>, <final value>, <procedure name>, <parameter list>, etc.

The BNF specification consists of a set of "rewriting rules," where each rewriting rule specifies the ways in which a given non-terminal can be rewritten. In the BNF specification, a vertical bar ( | ) separates alternative ways of rewriting a given non-terminal. Braces ( { } ) denote optional replication, and are used to indicate that the contents between the braces may be employed zero or more times.

Flock Algol includes a core of constructs drawn from Pidgin Algol [AHU74], Pascal [JeWi74], and C [KeRi78] which is shown in Figure 2.3. Figure 2.4 shows the BNF specification of the extensions to Pidgin Algol incorporate SIMD parallelism. The statements are of three general types:

1) mask statements, to allow subsets of PEs to be enabled (active) for execution of a statement or set of statements (and implicitly, to disable other PEs);
2) transfer statements, to specify the transfer of data between PEs; and
3) broadcast statements, to allow the dissemination of a single data item to a specified set of PEs.

The following gives a synopsis of each of these statement types.

```
<program> ::= <procedure definition>

<procedure definition> ::= PROCEDURE <procedure name> (<parameter list>)
        {<procedure definition>} <block>
<block> ::= <statement> | <declaration part> <statement>

<statement> ::=
        1.  <variable> ← <expression> |
        2a. IF <condition> THEN <statement> |
         b. IF <condition> THEN <statement> ELSE <statement> |
        3.  FOR <variable> ← <initial value> TO <final value>
                    DO <statement> |
        4.  BREAK |
        5.  BEGIN <statement> {  <statement> } END |
        6a. <procedure name> ( <argument list> ) |
         b. <variable> ← <procedure name> ( <argument list>)
         c. RETURN | RETURN <expression> |
        7.  miscellaneous statements |
        8.  <null statement>
```

Figure 2.3. Pidgin Algol core for Flock Algol.

```
<statement> ::= <mask statement> | <transfer statement> |
                        <broadcast statement> | <set network>
   1. <mask statement> ::= [<mask specification>] <statement> |
                                 <data conditional mask>
        a. <mask specification> ::= ENABLE <well defined set of PEs> |
                                          DISABLE <well defined set of PEs>
        b. <data conditional mask> ::=
                WHERE <condition> DO <statement> ENDWHERE |
          WHERE <condition> DO <statement> ELSEWHERE <statement> ENDWHERE
   2. <transfer statement> ::= TRANSFER {<source specification>
              {TO <destination specification>}}
        <source specification> ::= <variable>
        <destination specification> ::= <variable>
   3. <broadcast statement> ::= BROADCAST <broadcast specification> |
        <broadcast specification> ::= <source specification>
                                        FROM PE <PE source>
                                        TO <destination specification>
        <PE source> ::= <constant with value between 0 and N-1> |
        <variable with value between 0 and N-1>
   4. <set network> ::= USE <interconnection function>
```

Figure 2.4. Flock Algol statements to express parallelism.

## 2.3. Mask Statements

A mask statement will have the effect of specifying a subset of the N PEs in the SIMD system. Masks provide the system user with a method to control the active/inactive status of the PEs of the system. Siegel [Si77] gives details of the various types of masking schemes. Flock Algol includes two mask formats.

### 2.3.1. ENABLE and DISABLE

In the first format, the statement of type 1a consists of the keyword ENABLE or DISABLE, followed by an unambiguous specification of a set of PEs. The PEs enabled as a result of the mask specification execute the statement following the mask specification. If no mask accompanies a statement, all PEs are assumed to be active. The speech processing algorithms presented here use PE address masks [Si77] to specify which PEs to enable or disable. The PE address masks are n-position (where $n=\log_2 N$) masks that specify which of the N PEs are active for each instruction. Each mask position contains a 0, 1, or X ("don't care") and only those active PEs whose address (in binary representation) matches the mask are enabled (or disabled). An "X" matches either a 1 or a 0. Superscripts are repetition factors i.e., $[X^5] = [XXXXX]$. Square brackets denote a mask. For example ENABLE $[X^{n-1}1]$ activates all odd numbered PEs and DISABLE $[x^{n-1}0]$ disables all even PEs. If no mask accompanies an instruction, all PEs are active.

### 2.3.2. WHERE ... ELSEWHERE

The second format for mask statements is a data conditional statement, defined in statement type 1b. Data conditional masks are the implicit result of performing a conditional branch dependent on local data in an SIMD machine environment, where the result of different PEs' evaluations may differ. As a result of a conditional WHERE statement of the form

```
WHERE <condition> DO
    <statement>
ELSEWHERE
    <statement>
ENDWHERE
```

each PE will be active for the statement following for either the DO or the ELSEWHERE, but not both. The execution of the ELSEWHERE statement must follow the DO statement; i.e., the DO and ELSEWHERE statements cannot be executed simultaneously. For example, as a result of executing the statement

```
WHERE A > B DO
    C ← A
ELSEWHERE
    C ← B
ENDWHERE
```

each PE will assign to C the maximum of its A and B values, i.e., some PEs will execute "C ← A," and then the rest will execute "C ← B." Machines such as the Illiac IV [Barn68] and PEPE [Cran72] use this type of masking. Nesting data conditional mask statements is possible, the implementation can be accomplished using a run-time control stack, as discussed in [SiMu78].

From an implementation point of view, data conditional masks allow the specification of the mask condition to depend on PE data. The subset of PEs to enable is determined at execution time. The time to execute a "WHERE ... ELSEWHERE" statement will be the sum of the times to execute the statements following the DO and the ELSEWHERE.

The "IF-THEN-ELSE" and "WHERE-DO-ELSEWHERE" statements correspond to two different actions on an SIMD machine. An "IF-THEN-ELSE" is a control flow statement executed by the CU to determine which of two sets of code should be executed. The expression specifying the condition in an IF-THEN-ELSE STATEMENT will contain only constants and CU variables. If the code to be executed includes PE instructions, all active PEs will execute that code. A "WHERE-DO-ELSEWHERE" statement divides the PEs in the system into two sets, and instructs the two sets to execute different code. In this case, both sets of code are executed one after the other, but by different PEs. An "IF-THEN-ELSE" format could be used to specify data conditional mask statements. However, since the basic function of the two types of statements is different, it seems clearer to use different keywords to identify the two types of actions.

## 2.4. TRANSFER and USE Statements

The purpose of the TRANSFER statement (type 2 in Figure 2.4) is to allow inter-PE communications. The USE statement (type 4 in figure 2.4) specifies the type of interconnection function to use, and the interconnection functions specify the type of transfer to perform. Formally, an interconnection function is a bijection on the set of PE addresses. When an interconnection function, f, is executed, the contents of the source variable in PE j are transferred to the destination variable of PE f(j). This occurs for all j simultaneously, for $0 \leq j < N$ and PE j active.

The PEs interface to the interconnection network via the DTRin and DTRout registers. If the DTRin and DTRout register names are used in the algorithm, the "<source specification> TO <destination specification>" in the transfer statement syntax can be omitted. In this case, the source is assumed to be the DTRin, and the destination is the DTRout. The DTRin acts as the standard input to the network, and the DTRout acts as the standard output from the network. If the "<source specification>" is given without the "<destination specification>" the destination is the same as the source.

The following are interconnection functions used in the speech processing algorithms presented in Sections 5 and 6.

### 2.4.1. The Cube Interconnection Function

The *Cube* [SiMc81b] interconnection function is defined by letting $P = p_{n-1} \cdots p_1 p_0$ be the binary representation of the address of an arbitrary PE. The *n* cube interconnection functions are:

$$\text{Cube}(i)[p_{n-1} \cdots p_i \cdots p_0] = p_{n-1} \cdots \overline{p_i} \cdots p_0,$$

where $0 \leq i < n$, $0 \leq P < N$, and $\overline{p_i}$ is the complement of $p_i$. This means the cube(i) interconnection function connects PE P to cube(i) [P] where cube(i) [P] is the same address as P with the *i*th bit complemented.

### 2.4.2. The Permutation Interconnection Function

The *Perm*utation[Si81] interconnection function is defined as:

$$\text{Perm}_i(j) = \begin{cases} i-j & \text{where } 1 \leq j < i \\ j & \text{elsewhere} \end{cases}$$

$\text{Perm}_5(j)$ would switch data between PEs 0 and 5, PEs 1 and 4 and, PEs 2 and 3.

### 2.4.3. The Shift Interconnection Function

The *Shift* interconnection function is defined as:

$$\text{Shift } +n\ (j) = j+n \bmod N$$
$$\text{Shift } -n\ (j) = j-n \bmod N$$

where N is the number of PEs. Therefore Shift +1 (j) would send data from PE 0 to PE 1, PE 1 to PE 2, and so on.

## 2.5. Broadcast Statements

The purpose of broadcast statements (type 3 in Figure 2.4) is to allow the dissemination of a value from one PE to all PEs. The <PE source> is the PE containing the value to be broadcast. If the PE source is not given, the value is broadcast from the CU. The value is broadcast to all PEs.

## 2.6. An Example of a Flock Algol Algorithm

The following is an example of a Flock Algol algorithm. It performs a computation similar to that given in [Ston80]. Suppose the vector a[] is given, and the vector y[] is to be found such that

$$\begin{aligned} y[0] &= a[0] \\ y[i] &= y[i-1] + a[i] \qquad 1 \le i < N \end{aligned} \tag{2.1}$$

therefore y[i] is the sum of a[0] + a[1] + ... + a[i]. On a serial machine y[] is found by:

```
y[0] ← a[0]
FOR i ← 1 TO N−1 DO
    y[i] ← y[i−1] + a[i]
```

This algorithm appears to be serial since y[i−1] is computed before y[i]. Since the last statement is executed N−1 times, the time complexity is O(N). An SIMD machine with N PEs can find y[] in O(log N) time by using the method diagrammed in Figure 2.5. The figure is for N=8 PEs, where the nodes with an open circle do nothing, while the nodes with filled in circles form the sum of the two operands. The following SIMD algorithm to find y[], assumes element i of vector a[] is stored in PE i for $0 \le i < N$. After the algorithm, y[i] is stored in PE i.

```
1    y ← a
2    FOR j ← 0 TO $\log_2 N-1$ Do
3            TRANSFER y TO DTRout USING Shift $+2^j$   (2.2)
4            DISABLE $[0^{n-j}X^j]$
5                    y ← y + DTRout
```

Each step does the following:

1) Store a[i] in y[i] for $0 \le i < N$. This is done in all PEs simultaneously.

2) Execute statements [3]-[5] $\log_2 N$ times.

3) Transfer the data in *y* in PE i to DTRout in PE $(i+2^j)$ mod N. On the first loop, the data in *y* in PE 1 will transfer to DTRout in PE 2, and PE 2's data will transfer to PE 3, and so on. PE N−1 will transfer its *y* value to PE 0. When j=1, the data in *y* in PE 1 will transfer to DTRout in PE 3 etc.

4) Turn off some PEs. The first time through, the mask will be $[0^n]$ (where

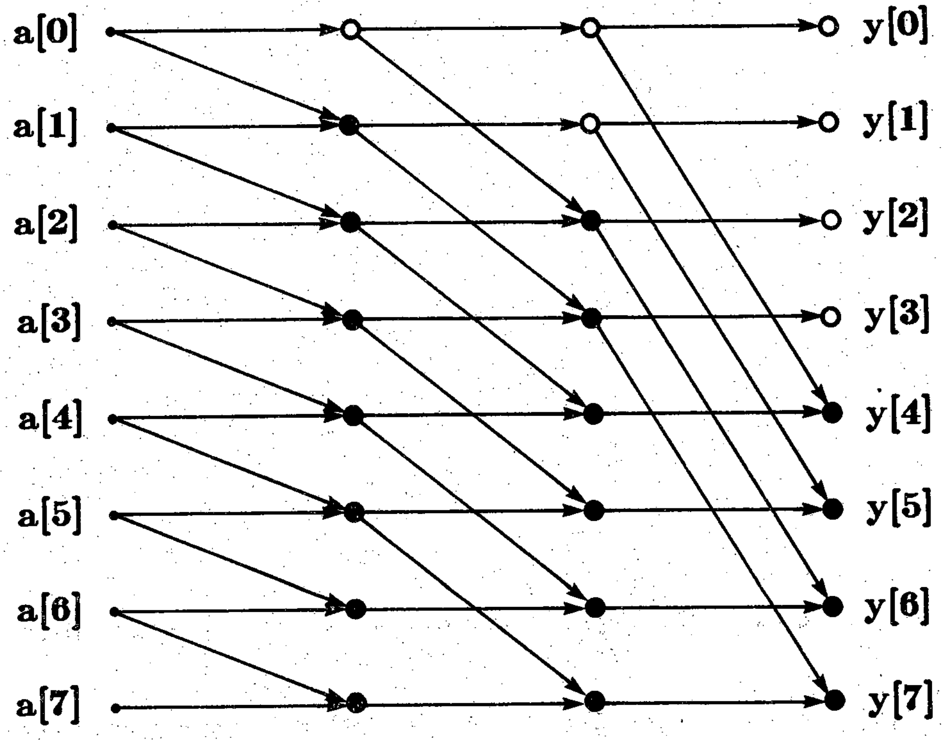


Figure 2.5. Parallel calculation of y[i] = y[i-1] + a[i].

$n = \log_2 N$) which matches only PE 0, so PE 0 will be disabled. This is indicated by a circle at node 0 in Figure 2.5. The second time through the loop, $j = 1$, so the mask is [$0^{n-1}X$] which matches PEs 0 and 1, so they are disabled. The DISABLE instruction only disables the PEs during the indented instruction(s) below it, therefore on subsequent times through the loop, all PEs will execute steps [2]-[4].

5) The new data transferred into DTRout is added to y[] only in the enabled PEs.

Figure 2.6 shows the intermediate values for this algorithm. Kogge and Stone[KoSt73] call this technique of shifting and summing *recursive doubling.* The time complexity is clearly O(log N) since the body of the loop in lines [2]-[5] of algorithm (2.2) is executed $\log_2 N$ times.

## 2.7. Summary

Real-time recognition of speaker independent isolated or connected speech using a large vocabulary requires more processor throughput than current serial machines can deliver. The SIMD machine is one possible way to organize a large number of processors to do the recognition in real time.

Flock Algol provides a high level algorithm description language for SIMD algorithms. It is based on a general model of an SIMD machine, and is intended to separate the structure of the parallel algorithm from architecture-specific issues such as the physical interconnection network or the actual mechanisms used to implement data broadcasts and the enabling/disabling of PEs.

The time complexity in the example algorithms above is reduced from O(N) on the serial machine to O(log N) on the SIMD machine. This shows that the parallelism of the SIMD machine can reduce the execution time of some algorithms. The following chapters will show how the SIMD machine can reduce the time complexity of various speech processing algorithms.

| | | Shift +0 | Mask | | Shift +1 | Mask | | Shift +2 | Mask | |
|---|---|---|---|---|---|---|---|---|---|---|
| PE | | TRANSFER | [000] | Sum | TRANSFER | [00X] | Sum | TRANSFER | [0XX] | Sum |
| | y a | DTRout | | y | DTRout | | y | DTRout | | y |
| 0 | y(0,0)=a[0] | y(7,7) | 0 | y(0,0) | y(5,6) | 0 | y(0,0) | y(1,4) | 0 | y(0,0) |
| 1 | y(1,1)=a[1] | y(0,0) | 1 | y(0,1) | y(6,7) | 0 | y(0,1) | y(2,5) | 0 | y(0,1) |
| 2 | y(2,2)=a[2] | y(1,1) | 1 | y(1,2) | y(0,0) | 1 | y(0,2) | y(3,6) | 0 | y(0,2) |
| 3 | y(3,3)=a[3] | y(2,2) | 1 | y(2,3) | y(0,1) | 1 | y(0,3) | y(4,7) | 0 | y(0,3) |
| 4 | y(4,4)=a[4] | y(3,3) | 1 | y(3,4) | y(1,2) | 1 | y(1,4) | y(0,0) | 1 | y(0,4) |
| 5 | y(5,5)=a[5] | y(4,4) | 1 | y(4,5) | y(2,3) | 1 | y(2,5) | y(0,1) | 1 | y(0,5) |
| 6 | y(6,6)=a[6] | y(5,5) | 1 | y(5,6) | y(3,4) | 1 | y(3,6) | y(0,2) | 1 | y(0,6) |
| 7 | y(7,7)=a[7] | y(6,6) | 1 | y(6,7) | y(4,5) | 1 | y(4,7) | y(0,3) | 1 | y(0,7) |

Figure 2.6. Intermediate values for recursive-doubling algorithm.

Where: y(i,j) denotes $\sum_{k=i}^{k=j} a(k)$,

and a 0 mask means the PE is disabled,
and a 1 mask means it is enabled.

# 3. VLSI PROCESSOR ARRAY MODEL

Very large scale integration technology has shown that simple regular interconnections are easy to implement, and give high densities. The VLSI processor arrays are so named because they are designed to have simple regular interconnections which exploit the capabilities of VLSI technology. A VLSI processor array is a network of specialized processing elements *(cells**) that circulate data in a regular fashion. The network configuration for a VLSI processor array is particular to the algorithm (or class of algorithms) being implemented. In general, the data flow can be viewed as a multidimensional pipeline. The VLSI processor array is a generalization of the systolic array [Kung80]. Both arrays have fixed interconnection networks. They differ in that systolic cells are assumed to be very simple, whereas VLSI processor array cells may be complex. For example, Figure 3.1 shows a systolic array presented by Kung[KuLe] for matrix multiplication. Without going into the details of how it works, notice each cell has only three registers (a,b,c) and the cell only does the operations shown in the lower right corner of Figure 3.1. Figure 3.2 shows a VLSI array for dynamic time warping. (Details of the array will be discussed in Section 6.4.2.1.) All the cells are connected by a fixed interconnection network as with the systolic array, but each cell has several registers, some of which contain vectors. Each cell does all the instructions shown in the lower right side. Figures 3.1 and 3.2 are only examples of one systolic array and one VLSI processor array. Both arrays can have different interconnections and perform different operations. This example shows that the cells in the VLSI processor array are more complex than those in the systolic array.

* Since the processing elements in the SIMD machine are different from those in the VLSI processor array, they will be called "PEs" in the SIMD machine and "cells" in the VLSI processor array.

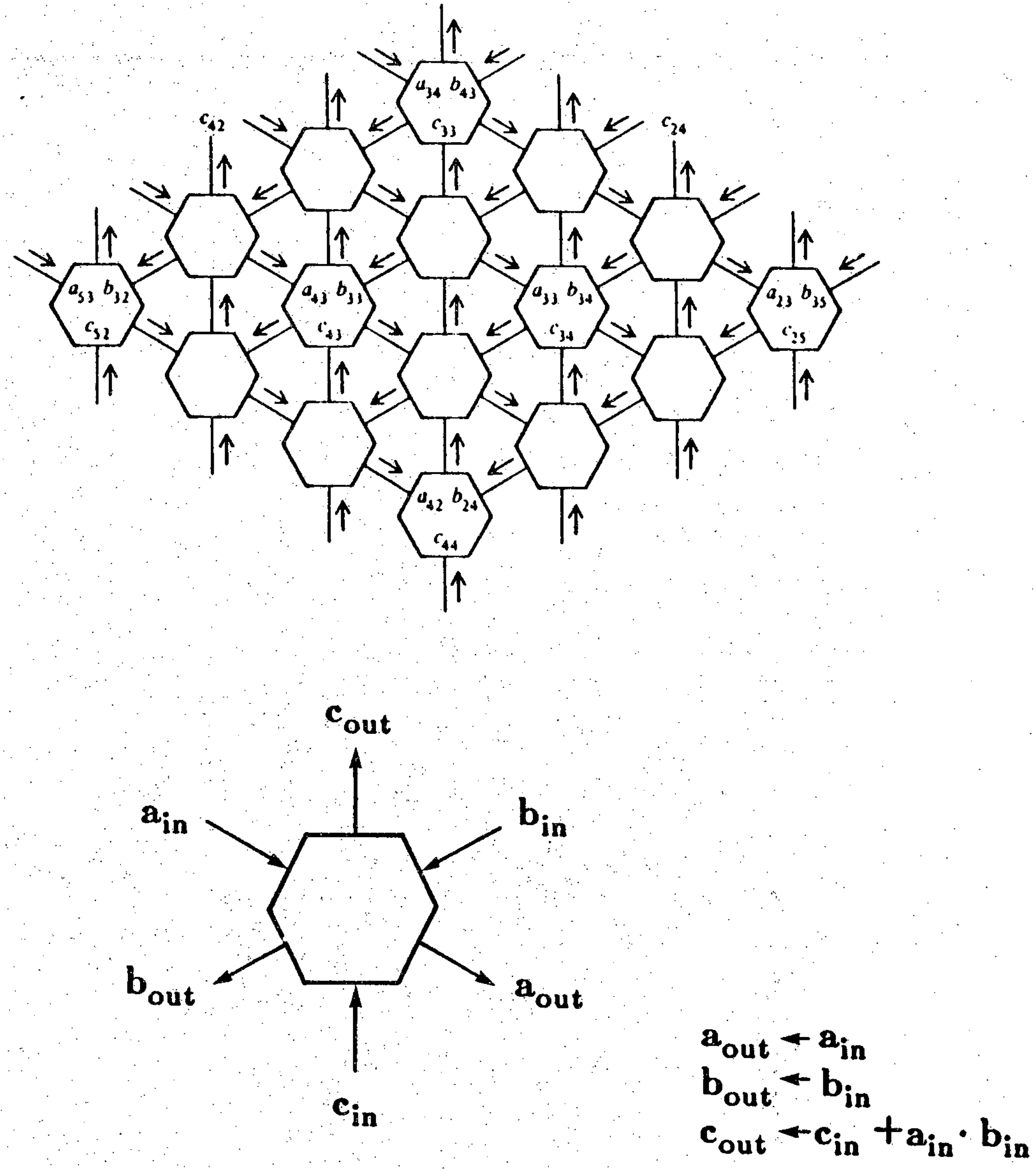
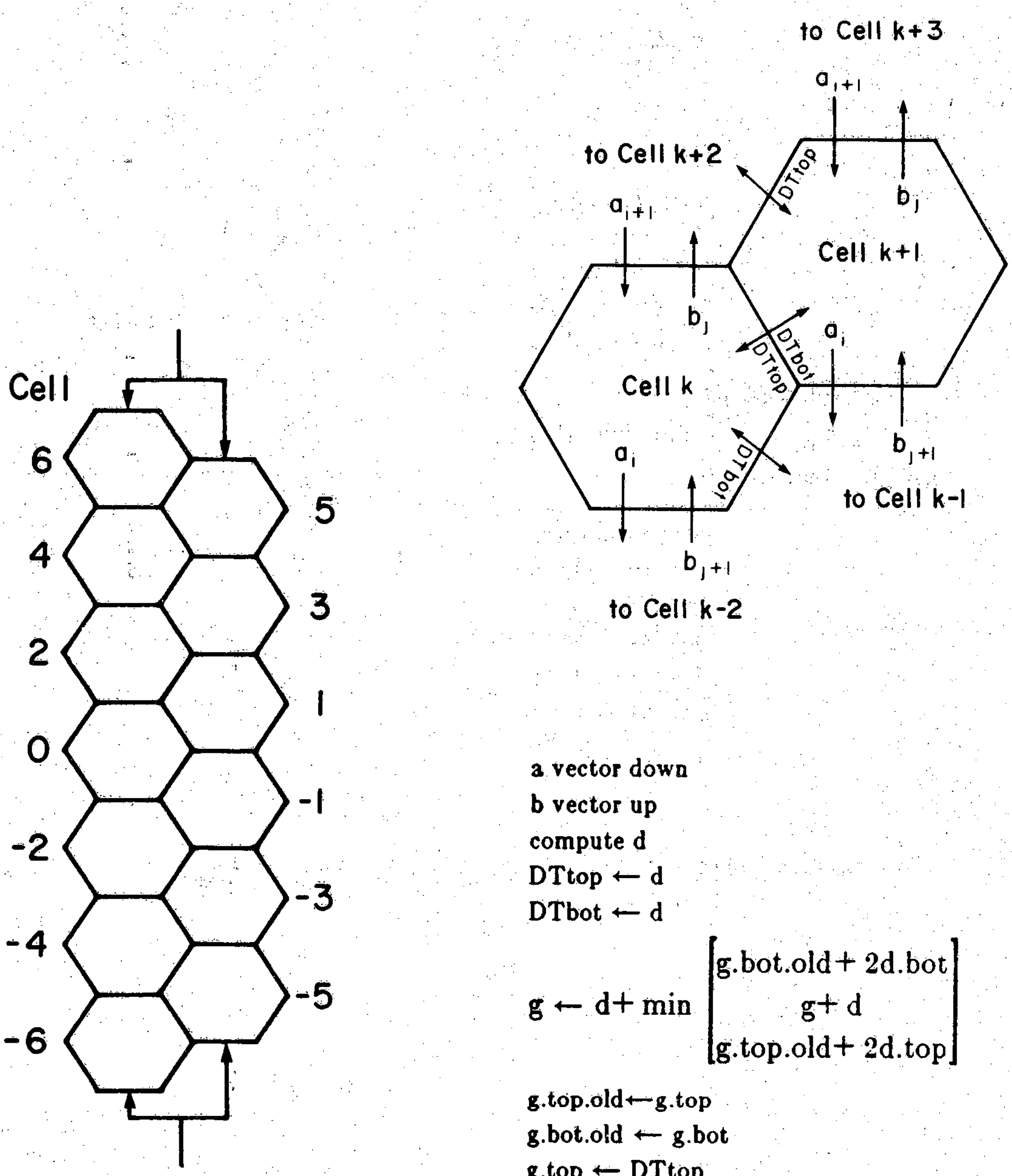


Figure 3.1. An example of a systolic array.



```
a vector down
b vector up
compute d
DTtop ← d
DTbot ← d
```

$$g \leftarrow d + \min \begin{bmatrix} g.bot.old + 2d.bot \\ g + d \\ g.top.old + 2d.top \end{bmatrix}$$

```
g.top.old←g.top
g.bot.old ← g.bot
g.top ← DTtop
g.bot ← DTbot
d.bot ← DTbot
d.top ← DTtop
DTtop ← g
DTbot ← g
```

Figure 3.2. An example of a VLSI processor array.

Both VLSI processor arrays and SIMD machines are forms of synchronous large scale parallel processing systems. VLSI processor arrays represent algorithm-specific systems with fixed interconnections between cells, specialized processors and a small set of registers for memory. SIMD machines are more complex, having a large memory in each cell and a general interconnection network between cells, making the system more flexible. The VLSI processor array algorithms are specified by giving the fixed interconnections between cells, and the instructions executed by each cell.

### 3.1. A Sample VLSI Processor Array Algorithm – Filtering

An example of a linear VLSI processor array is the finite impulse response (FIR) filter presented by Kung[Kung80]. The output $y_m$ of a FIR filter is given by:

$$y_m = \sum_{k=0}^{q} b_k x_{m-k} \qquad q \le m \le M \tag{3.1}$$

where $x_m$ is the input to the filter, the $b_k$'s are the filter coefficients, and M is the number of samples in the signal to be filtered.

Kung's FIR filter algorithm computes a (q+1)-tap FIR filter using a linear array of q+1 systolic cells. It solves the equation in which $y_m$ is computed using the summation in equation (3.1). The output $y_m$ can be computed by the following recurrence relation, where $y_m^{(k)}$ is the partial result in the computation of $y_m$ after k steps in the recurrence.

$$\begin{aligned} y_m^{(0)} &= 0 \\ y_m^{(k+1)} &= y_m^{(k)} + b_{q-k} x_{m-q+k} \qquad 0 \le k \le q \\ y_m &= y_m^{(q+1)} \end{aligned} \tag{3.2}$$

The above recurrences can be evaluated by pipelining the $x_m$ and $y_m^{(k)}$ values through q+1 linearly connected processors as shown in Figure 3.3. Each processor has three registers, $R_b$,$R_x$, and $R_y$, which hold b, x, and y values

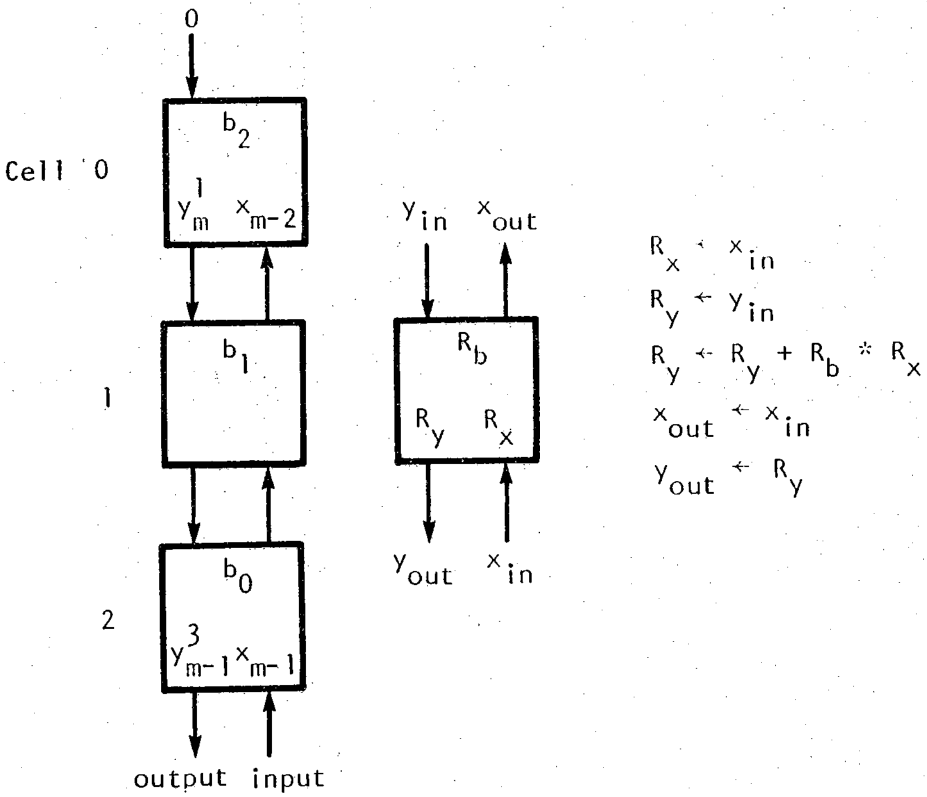


**Figure 3.3.** VLSI processor array to compute FIR filter for q=2.

respectively. Initially, all $R_x$ and $R_y$ registers contain zeros, and the $R_b$ register in processor i contains $b_{q-i}$. Each cycle of the array consists of the steps shown in Figure 3.3. $y_m^{(k)}$ is computed in cell k−1, and the output is produced in cell q. The data flowing up (the $x_m$ values) must be synchronized with the data flowing down (the $y_m^{(k+1)}$ values) so that they meet in the correct cell with the correct coefficient. Therefore during odd numbered cycles, only even numbered cells contain valid data, and during even numbered cycles only the odd cells contain valid data. Thus only half of the cells are active during a given cycle. One output value is therefore computed every two cycles of the systolic array, where during each cycle, the operations performed are the simultaneous transfer of data in the two pipes, plus the one addition, one multiplication, and one assignment shown in equation (3.2).

Figure 3.4 is the data flow diagram for the linear array. Each column of the data flow diagram represents the contents of each register in each cell after a given cycle. Moving from left to right shows how the data changes from one cycle to the next. The arrows show where $R_x$ and $R_y$ will be transferred on the next cycle.

This linear array uses q+1 cells and produces a new *y* value every two cycles. Ignoring the startup and stop time (i.e., the time required to pipe $y_0$ from cell 0 to cell q and to pipe $y_{M-1}$ from cell 0 to cell q) the VLSI processor array is (q+1)/2 times faster than a serial machine. This is because there are q+1 cells, half of which are doing computations on valid data at a given time.

## 3.2. Summary

Although the SIMD machine may have the computing power needed to recognize speech in real time, its general nature (a general purpose processor in each PE and a general interconnection network) may make it too expensive for a dedicated application. The VLSI processor array, on the other hand, with its fixed interconnection network and independently operating cells may be able to perform the task with less hardware.

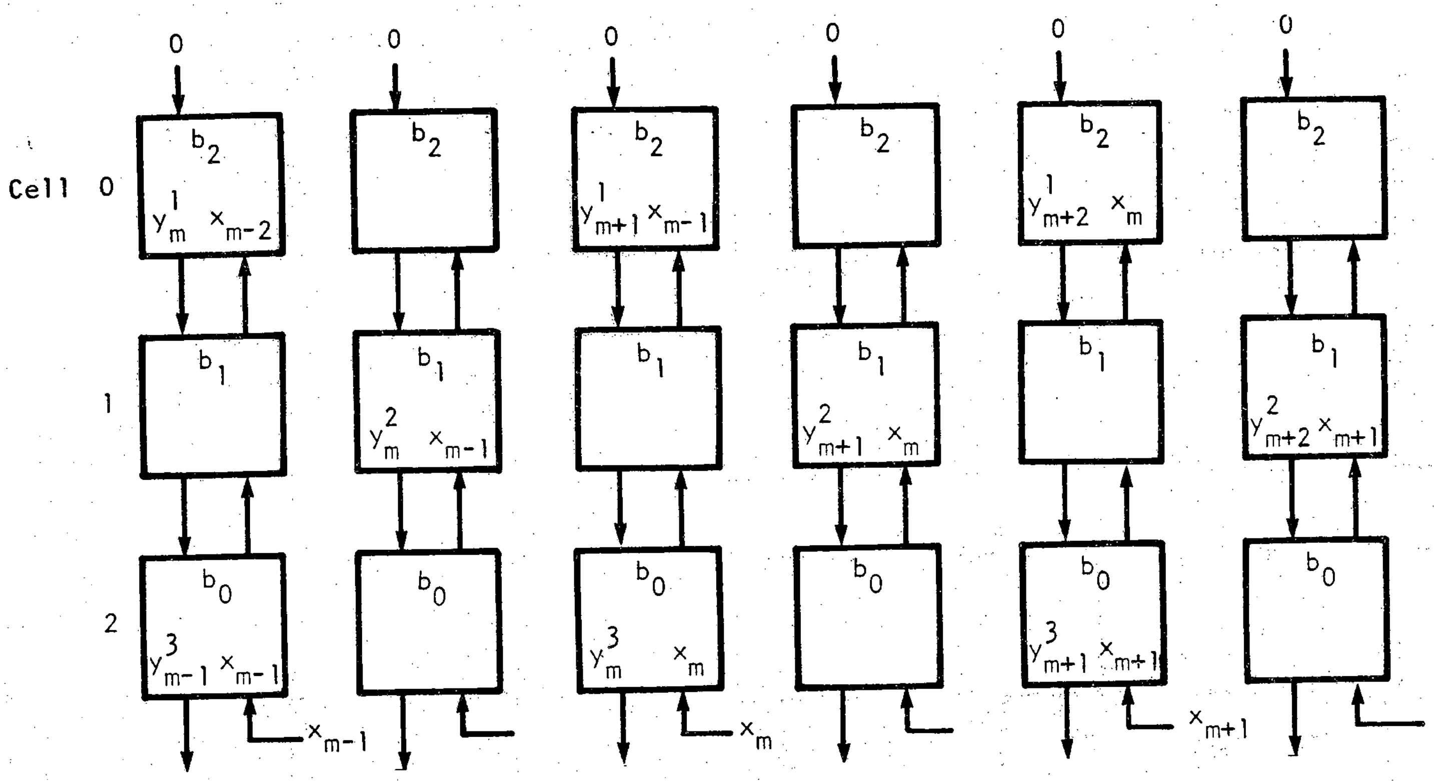


Figure 3.4. Data flow diagram for Figure 3.3.

This chapter presented a VLSI processor array model along with an example of how a linear array of q+1 cells could achieve a speed up of (q+1)/2 over a serial algorithm. The VLSI processor array is a generalization of Kung's systolic array. The generalization adds a more powerful processor in each cell along with more memory and broadcast capability. Chapters 5 and 6 present some parallel speech processing algorithms which use the VLSI processor array and Chapter 8 presents the results of simulating the algorithms.

# 4. AN ISOLATED WORD RECOGNITION SYSTEM

Of the many commercially available speech recognition systems, most perform isolated word recognition [Dodd81] since it is easier than connected word recognition. In isolated speech each utterance is separated from the next by a short pause (>100 ms). These pauses help the system in locating the beginning and end of each utterance. After the unknown utterance is located, many speech recognition systems rely on pattern matching techniques to match the features of an unknown input utterance to previously stored features of known utterances. Figure 4.1 is a block diagram of a typical template matching system for isolated word recognition [RLRW79].

A template matching based system has two modes of operation, training and recognizing. During training, the speech signal is bandpass filtered (to prevent aliasing) and then sampled. After sampling, the speech signal is broken into fixed sized frames that generally contain between 100 and 400 samples. Each frame passes through a preemphasis filter followed by autocorrelation analysis. Next linear predictive coding (LPC) [Makh75,MaGy76] analysis is used to take the autocorrelation coefficients and produce LPC coefficients. The LPC analysis reduces each frame from N samples ($100 \leq N \leq 400$) to p LPC coefficients where p is typically between 6 and 25. Next, endpoint detection finds the first and last frames of the utterance and discards the silent frames before the first frame and after the last frame. The discarded frames are not used in the rest of the processing. At this point an utterance will be represented by approximately 40 frames of 8-14 coefficients each. If the utterance has more or less than 40 frames, a linear time warp (LTW) normalizes, in time, the utterance to 40 frames.

The process above is repeated for each utterance in the vocabulary, and the 40 sets of LPC coefficients for each word are stored for later use. To achieve speaker independence, the same word is spoken by several different

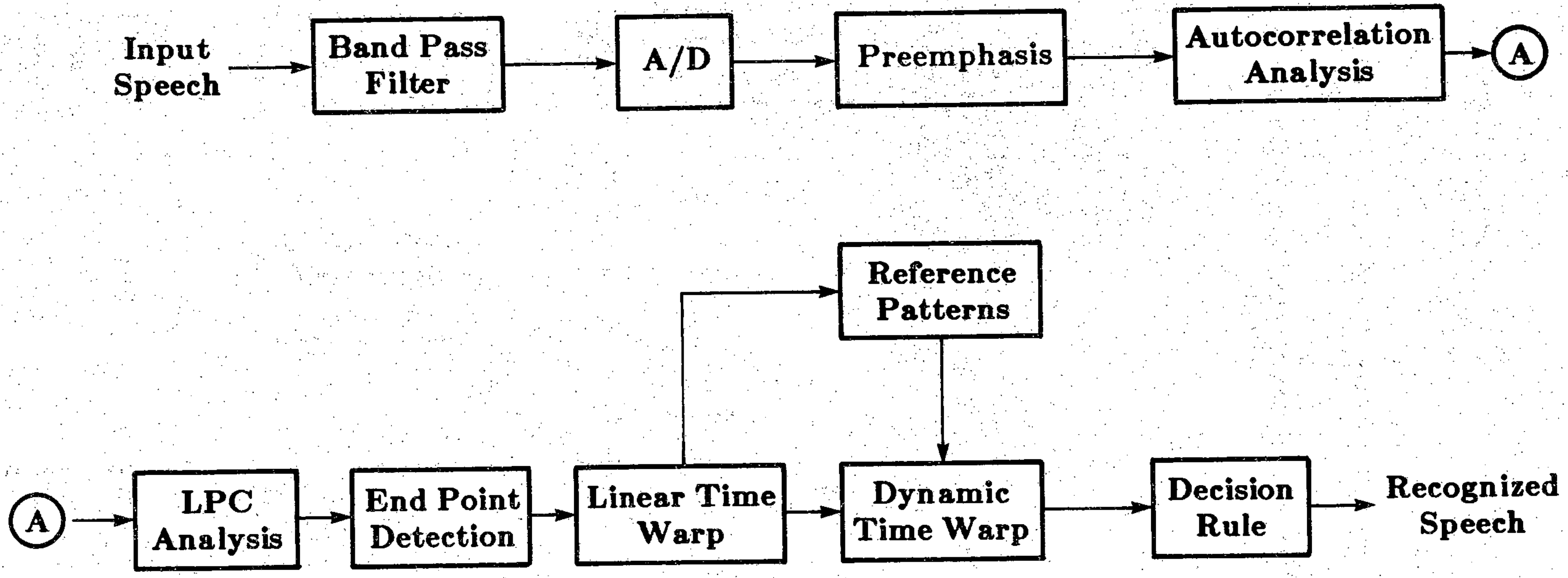


Figure 4.1. Block diagram of an isolated word recognition system.

speakers and all sets of coefficients are stored or clusters are used as discussed in [RLRW79].

During the recognition mode the same steps as in the training are used, except after the linear time warp a dynamic time warp (DTW) compares the word to be recognized (the test template) to the training set (the reference templates). The distance from the input utterance to all the stored utterances is found, and the stored utterance with the shortest distance from the input utterance is picked as the utterance that was spoken.

The following is a detailed description of each block in Figure 4.1.

## 4.1. Filtering and Sampling of Input Signal

The first step in recognizing a word is to filter and sample the input signal. The choice of filtering frequencies and sampling rate depends on the quality of speech available. The input is low pass (or possibly bandpass) filtered at 10 KHz (100 - 10 KHz) and sampled at 15-20 KHz when using "high quality" speech. If the system is to work over the phone lines (telephone quality speech) the input is band pass filtered around 300-3200 Hz and sampled at 6.67 KHz. Systems using both 6.67 KHz sampling [RLRW79] and 20 KHz sampling [BBGI80] have appeared in the literature, along with various other sampling rates in between.

## 4.2. Preemphasis Filtering

Each frame passes through a digital preemphasis filter with a z transform of

$$H(z)=1-az^{-1}$$

where typically $a \simeq 0.95$. Experimental evidence shows that preemphasis serves to reduce the variance of the distance calculation in an LPC based template matching system[RLRW79].

## 4.3. Autocorrelation Analysis

Next, the sampled signal is broken into frames for autocorrelation analysis. The LPC processing that is done later dictates the number of samples per frame. The frame length should be short enough so the vocal tract configuration is constant during the frame, but long enough so the initial condition assumptions (i.e., the values the signal is assumed to have outside of the frame) have a small effect on the coefficients. Frame lengths are usually fixed and contain between 100 and 400 samples, which correspond to 10-20 ms of speech depending on the sampling rate. One common method uses 300 sample frames that begin every 100 samples. This leaves a 200 sample overlap between frames. This overlap tends to reduce the variance in the LPC coefficients between frames containing the same speech sound.

The short term autocorrelation coefficients are found by using:

$$R(i) = \sum_{m=0}^{M-i-1} s(m)s(m+i) \qquad 0 \leq i \leq p \tag{4.1}$$

where M is the frame length and p is determined by the LPC processing and is between 6 and 25. The first autocorrelation coefficient, R(0), is the energy for each frame, while all the coefficients are used in the LPC analysis which follows.

### 4.4. Linear Predictive Coding

Following the autocorrelation analysis is linear predictive coding analysis. LPC models the speech sounds as an all pole filter and an excitation source [MaGy76]. The filter represents the configuration of the vocal tract, i.e., the position of the mouth, nose, and throat. If the sound is voiced, the excitation represents the pitch pulses from the vocal chords. If the sound is unvoiced, the excitation represents the "noise-like" sound of the air being forced past some constriction. The constriction may be the tongue and the avleolar ridge (behind the upper front teeth) as in the sound "s."

LPC assumes that the *m*th sample of the speech signal {s} can be represented by two components:
1) a linear combination of the p previous speech samples, and
2) the excitation, $\delta(m)$, which may differ for each sample s(m).

The sample s(m) is modeled as follows: [AtHa71,Makh75,RaSc78]

$$s(m) = \sum_{k=1}^{p} a(k)s(m-k) + \delta(m) \qquad p \leq m < M \tag{4.2}$$

A common method used to find the LPC coefficients, a(k) for $1 \leq k \leq p$, is to define $\hat{s}(m)$ as the predicted signal (i.e., the linear combination of the p previous samples) and minimize the squared prediction error which is:

$$E^2 = \sum_m [s(m) - \hat{s}(m)]^2 = \sum_m [s(m) - \sum_{k=1}^{p} a(k)s(m-k)]^2 \tag{4.3}$$

To find the a(k)'s, find the k partial derivatives of $E^2$ with respect to a(k) and set them to zero:

$$\frac{\partial E}{\partial a(k)} = 0 \qquad 1 \leq k \leq p$$

This will result in p equations with p unknowns. By assuming the speech signal is zero before and after the frame (i.e., s(m)=0 $m < 0$ and s(m) = 0 $m \geq M$), equation (4.2) can be solved by defining the short-term autocorrelation functions as in equation (4.1) and rewriting equation (4.3) as

$$\sum_{k=1}^{p} a(k)R(|i-k|) = R(i) \qquad 1 \leq i \leq p \tag{4.4}$$

Equation (4.4) can be written in matrix form as:

$$K\vec{a} = \vec{R} \tag{4.5}$$

where $\vec{R}$ and $\vec{a}$ are p element vectors of elements R(i) and a(i) respectively for $1 \leq i \leq p$, and K is a p by p matrix with $K = R(|i-k|)\ 0 \leq i,k < p$. K is a Toeplitz matrix, i.e., it is symmetric with all elements on each diagonal being equal.

Finding the coefficients a(k) takes two steps,
1) Find the p autocorrelation coefficients R(i), and
2) solve equation (4.5) for $\vec{a}$.

$\vec{a}$ could be found from equation (4.5) by finding the matrix inverse of K, but since K is Toeplitz, more efficient methods are available. Figure 4.2 is the serial algorithm for Durbin's method, which is one of the most efficient methods available.

### 4.5. Endpoint Detection

After LPC analysis the endpoints are located. The endpoints of an utterance are the frames where the word begins and ends.

Rabiner [RaSa75] presents a simple but robust method to detect endpoints based on using an upper (UE) and a lower (LE) "energy" threshold, and a zero crossing threshold (ZC). The following are definitions of the terms used in describing the method to find the beginning point. (Reverse all directions when finding the ending point.)

energy: The "energy" for each frame is the first autocorrelation coefficient, R(0). (See equation (4.1).)

zero crossing: The zero crossing rate is defined as the number of times the normalized signal changes sign in one frame.

```
$E^{(0)} = R(0)$;
FOR i ← 1 TO p DO
                                        /* compute k(i) */
    $k(i) \leftarrow 0$;
    FOR j ← 1 TO i-1 DO
        $k(i) \leftarrow k(i) + a_j^{(i-1)} * R(i-j)$;
    $k(i) \leftarrow [R(i) - k(i)] / E^{(i-1)}$;
    $E^{(i)} \leftarrow (1-k(i)^2) * E^{(i-1)}$;
                                        /* compute $a_j$'s for stage i */
    $a_i^{(i)} \leftarrow k(i)$;
    FOR j ← 1 TO i-1 DO
        $a_j^{(i)} \leftarrow a_j^{(i-1)} - k(i) * a_{i-j}^{(i-1)}$;
FOR j ← 1 TO p DO
    $a_j \leftarrow a_j^{(p)}$;
```

Figure 4.2. Durbin's Algorithm to compute LPC coefficients $a_i$ from autocorrelation coefficients $R(i)$, $0 \leq i \leq p$.

frame pointer: The frame pointer points to the frame that is currently being considered as the first (or last) frame of the word.

frame after: If the frame pointer is at frame n, the frame after is frame n+1.

back up: When the frame pointer is backed up, it moves from frame n to n−1 to n−2, etc. until the criterion is met.

Rabiner's method works as follows:

1) The energy and zero crossings are measured for all frames in the utterance.

2) After the thresholds are set (to be discussed later), the frame pointer is used to find the first (or last) frame in the utterance by setting the frame pointer to the first frame to exceed the upper energy threshold.

3) Next the frame pointer is backed up to the frame after the first frame that *does not* exceed the lower energy threshold.

4) If three frames before this frame exceed the zero crossing rate threshold, the frame pointer is backed up until the frame after the first frame that *does not* exceed the zero crossing rate threshold.

After step 4, the frame pointer is pointing to the first frame of the utterance. The same procedure (and thresholds) are used to locate the ending point. Figure 4.3 is an example of how the thresholds are used to find the endpoints. The circled numbers represent the location of the frame pointer after the given step number.

The three thresholds are set by finding the mean ($\mu_{zc}$) and standard deviation ($\sigma_{zc}$) of the zero crossings for the first 10 frames. These frames are assumed to be silent (background noise only). The zero crossing threshold (ZC) is found by:

$$ZC = MIN(FIXED, \mu_{zc} + 2\sigma_{zc})$$

where FIXED is a fixed threshold. A typical value for FIXED is 25 crossings per 10 ms if the sampling rate is 10 KHz. The UE and LE thresholds are found by:

$$LE = \min(0.03*(PEAK-SILENT)+SILENT\ ,\ 4*SILENT)$$

$$UE = 5*LE$$

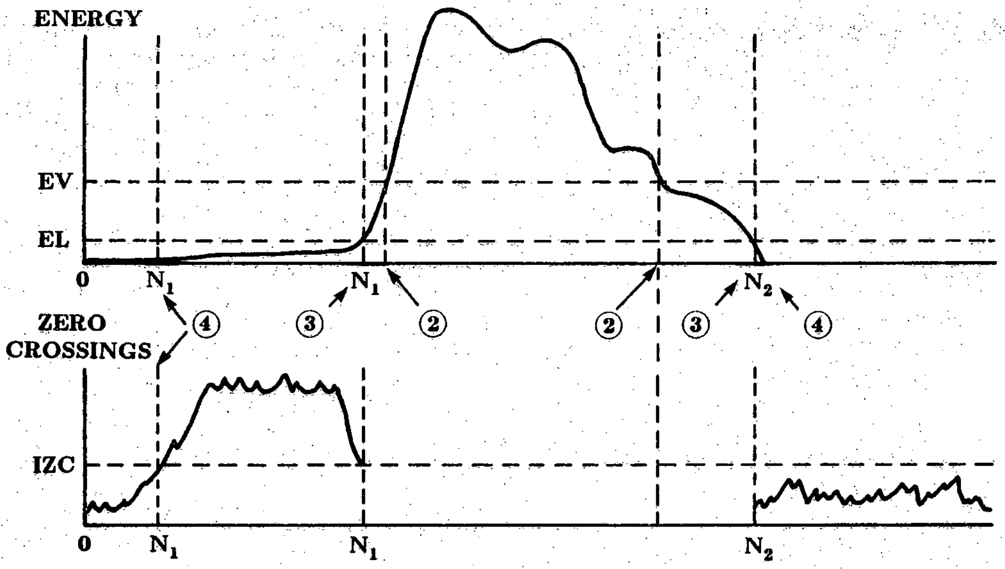


Figure 4.3. An example of how the zero crossings and energy thresholds are used to find the end-points of a word (from [RaSa75]).

where PEAK is the largest energy over all frames, and SILENT is the largest energy of the silent frames (silent frames are assumed to be the first 10 frames).

The double energy threshold is used so that mouth noises (breathing, lip smacking, etc.) that commonly occur before an utterance are not included as part of the utterance. These noises will tend to exceed the lower energy threshold, but not the upper energy threshold. The zero crossing rate is used to detect the beginnings of words starting with a fricative. The energy of a fricative is generally not enough to exceed the upper energy threshold, so the zero crossing rate is used to detect the high frequencies which are commonly present in fricatives. Lamel [LRRW81] states that the use of zero crossing rate is not effective in detecting words starting with a fricative for telephone quality recognition since telephone speech is band limited to 3200 Hz.

## 4.6. Time Warping

Dynamic time warping (DTW) is widely used in word and speech recognition to eliminate the effects of nonlinear time fluctuations in speech patterns. The function of DTW is to find the minimum time-normalized distance between two templates A and B where A and B are sequences of features vectors $\underline{a}_i$ and $\underline{b}_j$ for $1 \leq i \leq I$, $1 \leq j \leq J$. Each $\underline{a}_i$ and $\underline{b}_j$ is a vector of features for a segment of speech. In the template matching system discussed here, the feature vector contains the p LPC coefficients. It is generally easier to compare two templates of equal length with dynamic time warping, so linear time warping is used before dynamic time warping to normalize the length (i.e., the number of frames) of the templates. The following two sections describe the linear and dynamic time warping.

### 4.6.1 Linear Time Warping

The following linearly warps a template of speech of length M to length N.

$$T(n) = (1-s)*R(m) + s*R(m+1), \quad n=1,...,N \qquad (4.6)$$

where R(m) for $1 \leq m \leq M$ are the M frames of the input templates, and T(n) for $1 \leq n \leq N$ are the N frames of the output template and:

$$m = \left\lfloor (j-1)\,\frac{(M-1)}{(N-1)} + 1 \right\rfloor$$

$$s = (n-1)\,\frac{(M-1)}{(N-1)} + 1-m$$

where $\lfloor x \rfloor$ is the greatest integer less than or equal to x. For a time signal, the simple linear interpolation used in equation (4.6) is adequate as long as M and N do not differ greatly [Myer80]. Words are typically 40 frames long, so N = 40.

### 4.6.2 Dynamic Time Warping

Following the linear time warp is a dynamic time warp. This is done, as shown in Figure 4.4, by finding a path connecting (1,1) to (I,J) such that the accumulated distance is a minimum. Figure 4.5 is an example of how an input signal is warped to match a reference signal. The accumulated distance is a weighted sum of the local distances d(i,j) between the feature vectors $\underline{a}_i$ and $\underline{b}_j$. An exhaustive search of all possible paths is computationally infeasible, so dynamic programming (DP) theory is used to reduce the number of paths searched. DP theory states that if the point (i,j) is on the optimum path, then the path from (1,1) to (i,j) is locally optimum. One method to find the accumulated distance, g(i,j), restricts the possible paths leading to a given point to those shown in Figure 4.6. Using these restrictions*, g(i,j) is recursively defined as,

*Myers [MRR80] would describe these restrictions as Type I local constraints with an unsmoothed Type d weighting function.

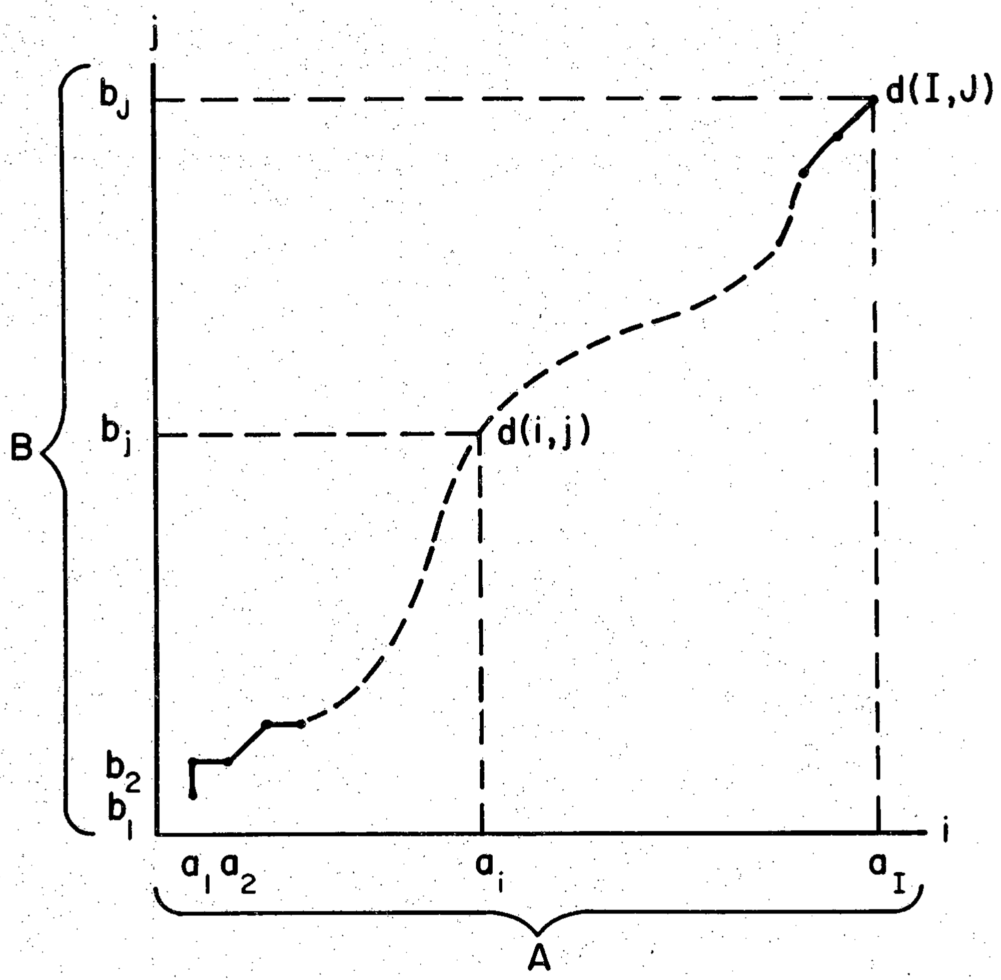


Figure 4.4. Dynamic time warping paths.

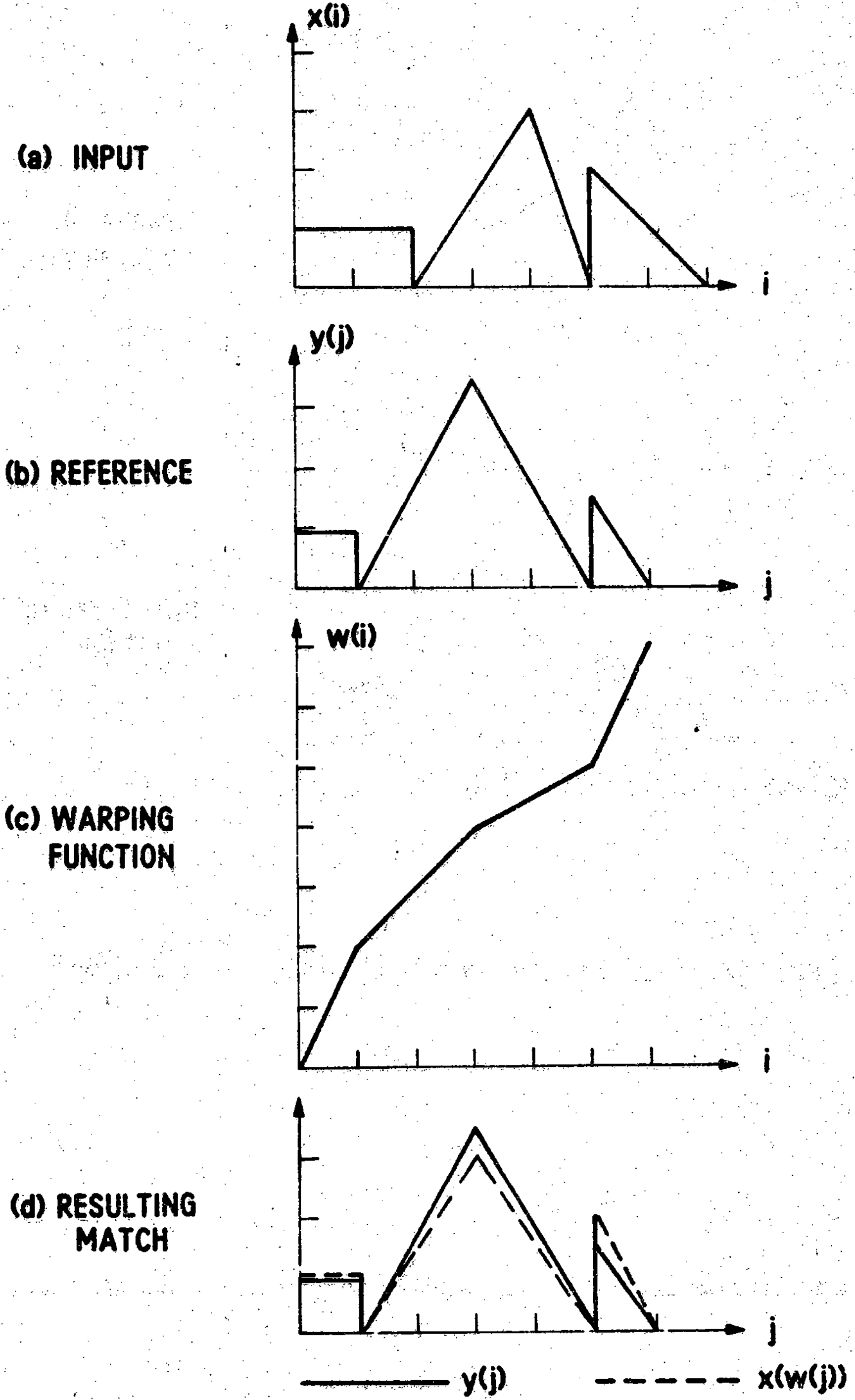


Figure 4.5. An example of time warping (from [Myer80]).

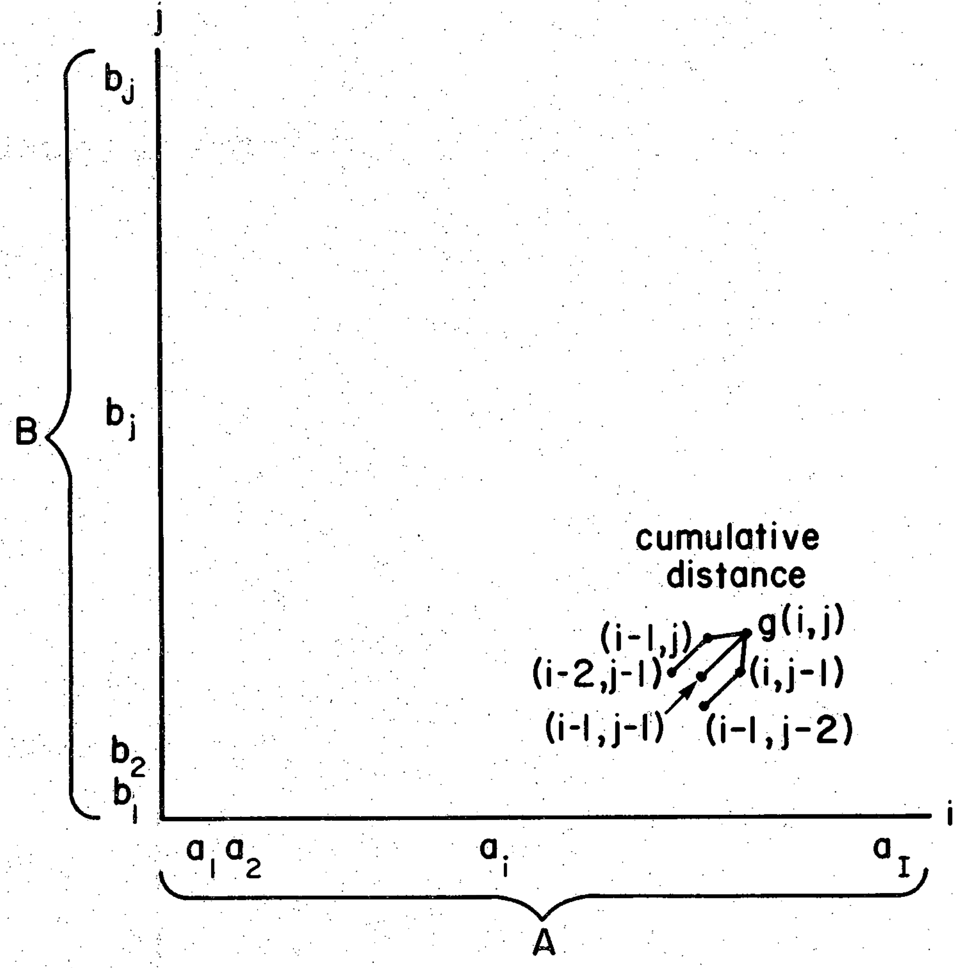


Figure 4.6. Possible paths to a point.

$$g(i,j) = d(i,j) + \min \begin{bmatrix} g(i-1,j-2)+2d(i,j-1) \\ g(i-1,j-1)+d(i,j) \\ g(i-2,j-1)+2d(i-1,j) \end{bmatrix} \tag{4.7}$$

$$g(1,1) = 2d(1,1)$$

Once g(I,J) is found, the normalized distance D(A,B) can be found by dividing g(I,J) by I+J.

Two methods that can be used to reduce the computation time are an adjustment window and pruning. The adjustment window[SaCh71], *r,* reduces the number of local distance calculations by restricting the domain of the time warp to those g(i,j) for which $|i-j| \leq r$, as shown by the two diagonal lines in Figure 4.7. Pruning compares the g(i,j) values at each point in the time warp to a threshold, and if the threshold is exceeded, the DTW is stopped and DTW on the next reference template is started. This reduces the DTW time by aborting comparisons that will definitely not yield the minimum distance.

The steps needed to compute one g(i,j) are:

1) computing the local distance d(i,j);

2) the two multiplications and four additions in equation (4.7); and

3) two comparisons to find the minimum of three values.

These three steps are defined as one *loop* and will be used as a basis to compare the time complexities of different dynamic time warping algorithms. The serial algorithm in Figure 4.8 must execute one loop for every (i,j) pair in Figure 4.4. Using no adjustment window, the total time is $I^2$ loops*. However, if the adjustment window is used, the number of loops is $I^2 - 2\sum_{i=1}^{I-r} i = 2Ir - I - r^2 + r$.

*A linear time warp is commonly used on both the test and reference patterns to make them the same length, allowing the assumption that I=J.

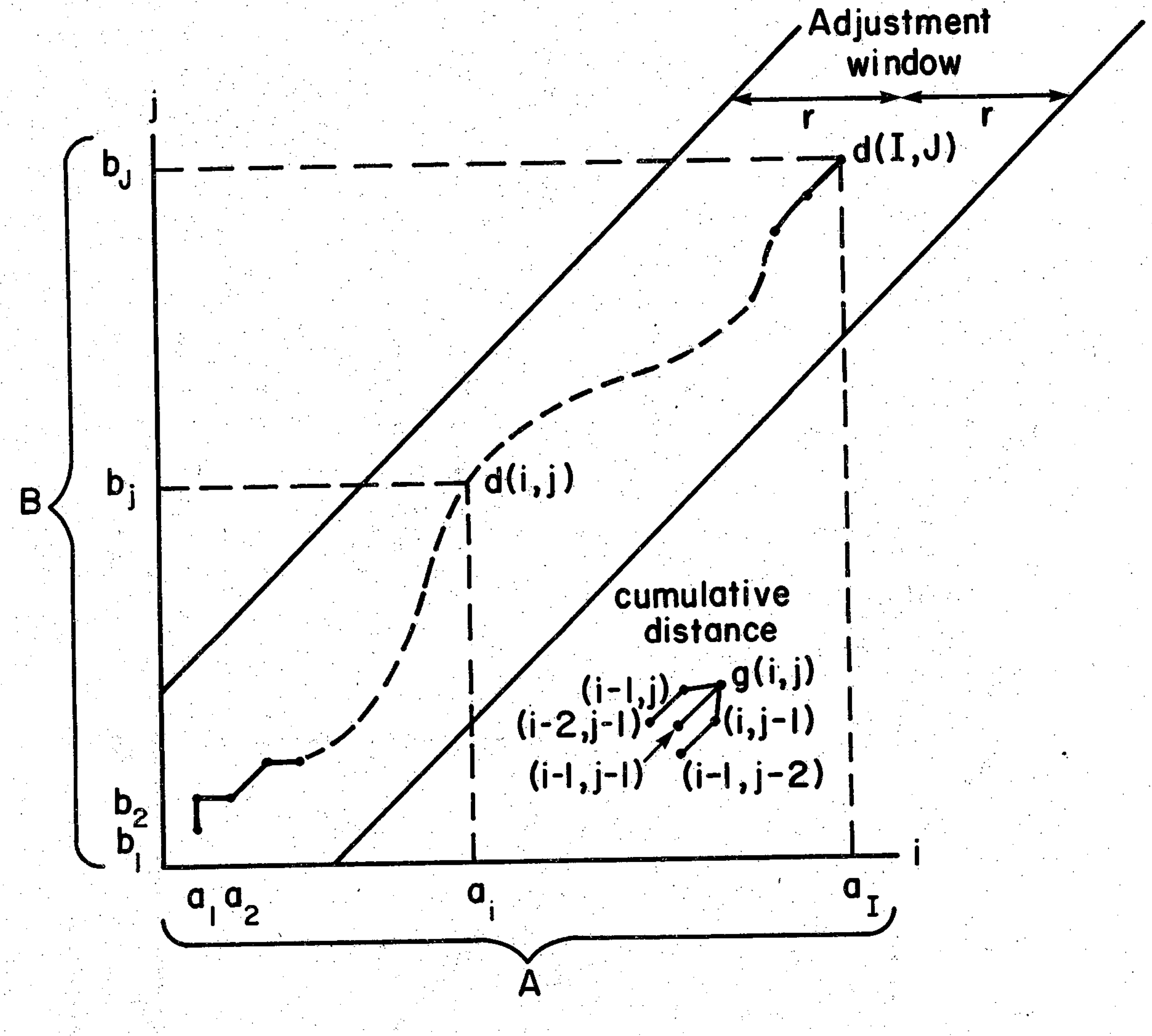


Figure 4.7. Adjustment window of width r.

```
/*
                    Serial program for dynamic time warping.

                    I                number of test vectors
                    J                number of refence vectors
                    r                adjustment window
                    known[x][i]      contains coefficient i of
                                     vector x of the known utterance.
                    unknown[y][i]    contains coefficient i of
                                     vector y of the unknown utterance.
                    d[x][y]          contains the local distance between
                                     the x known vector and the y unknown vector.
                    g[x][y]          contains the accumulated distance up to
                                     the x known vector and the y unknown vector.
*/

Line  Time in μs
 1           PROCEDURE DTW
 2    9      FOR y ← 0 TO I-1                 /* For each frame in the
                                                       unknown utterance*/
 3    4          FOR x ← -r TO r              /* For each frame in the
                                                       warping path*/
 4    5              IF (y+x ≥ 0) AND (y+x ≥ 2I-2)
 5           /*
 6                          Compute the local distance.
 7           */
 8    .5                    sum ← 0;
 9    2.75                  FOR i ← 0 TO p-1
10    11.25                     sum ← sum+(known[x][i]-unknown[y][i])^2;
11    2                     d[x][y] ← sum;
12           /*
13                          Check initial conditions
14           */
15    5                     IF Y = 0 AND X=0
16    8.25                      g[x][y] ← 2 * d[x][y];
17                          ELSE
18                              IF Y = 0        /* Check left edge*/
19    3.25                          min ← 2 * d[x][y-1];
20                              ELSE IF X = 0 /* Check bottom edge*/
21    3.75                          min ← 2 * d[x-1][y];
22                              ELSE
23           /*
24                                  Compute possible paths.
25           */
26    4                             A ← g[x-1][y-2] + 2d[x][y-1];
27    3                             B ← g[x-2][y-1] + 2d[x-1][y];
28    2                             C ← g[x-1][y-1] + 2d[x][y];
29           /*
```

Figure 4.8. Serial DTW program. Execution times assume an 8 MHz MC68000. (See Section 7.6)

```
                                      Find minimum path.
          */

                                      min ← A
6.5                                   WHERE B < A
.5                                            min ← B;
2                                     ENDHWERE
6.5                                   WHERE C < min
.5                                            min ← C;
2                                     ENDWHERE

2.5                               g[x][y] ← d[x][y] + min;
          /*
                        If g[x][y] is ≥ ∞ set of ∞, otherwise
                        repeated doubling might cause it to wrap around
                        to −∞
          */
6.5                               WHERE g[x][y] ≥ ∞
.5                                        g[x][y] ← ∞;
2                                 ENDWHERE

2.5       RETURN g[I][I];
```

Figure 4.8. (Continued)

### 4.7. Summary

This section has described an isolated word recognition system that uses template matching. This system was chosen to be implemented on an SIMD machine and VLSI processor array for the following reasons:

1) It has speaker independent accuracies as high as 98.2% [RLRW79].

2) It and systems like it have appeared many time in the literature, therefore there is interest in such a system.

3) The system currently cannot run in real time on a serial processor.

As the vocabulary size increases, this system will take more time to do the pattern matching. If a vocabulary of 1,000 words is used, a conventional processor cannot compare the input templates to all the test templates in real time. The following chapters present algorithms for SIMD machine and VLSI processor arrays to do each step in the recognition system. When the SIMD and VLSI processor array speech algorithms are combined into one system, (either as all SIMD or all VLSI processor array) it should be able to run in real time with a large vocabulary. If so, this system will meet three of the four criteria given in Section 1; namely, real time response, large vocabulary, and speaker independent. The only criterion not met will be continuous speech recognition, which is a topic of future research.

# 5. SURVEY OF PARALLEL SPEECH PROCESSING ALGORITHMS

The following is a survey of some of the highly parallel speech processing algorithms in the literature. The algorithms examined are those needed for the recognition systems considered here. The major topics are LPC coding (including autocorrelation algorithms), dynamic time warping, and digital filtering. Each section presents an algorithm and then discusses the machine requirements and speed up obtained by the algorithm.

## 5.1. Autocorrelation

Autocorrelation has many uses in speech processing. The template matching recognition system often uses it as an intermediate step to finding LPC coefficients (See Section 4.4). The short term autocorrelation function, R, is defined as:

$$R(i) = \sum_{m=0}^{M-i-1} s(m)s(m+i) \qquad 0 \leq i \leq p$$

Three methods to find the autocorrelation coefficients are discussed here. The first method (AUTO1) uses M PEs to multiply the M−i−1 s(m)s(m+i) terms in parallel. The second method (AUTO2) uses M PEs to compute R(i) for $0 \leq i < M$ using two FFTs. The third method (AUTO3) uses p+1 PEs to sum the terms in each R(i) in parallel.

### 5.1.1. Autocorrelation Using M PEs – AUTO1

Siegel [Si80a] gives a SIMD algorithm to compute the autocorrelation coefficients $R(i)$, $0 \leq i \leq p$ for an M-point signal $s(m)$, $0 \leq m < M$. Her algorithm, listed in Figure 5.1, is referred to here as AUTO1. It uses N PEs where $2^{n-1} < M \leq 2^n = N$. The signal $s(m)$ is initially distributed among the PEs so that $s(j)$ is stored in variable s in PE j for $0 \leq j < M$, and 0 is stored in variable s in PE j for $M \leq j < N$. Each element $R(i)$ is computed simultaneously by transferring $s(m+i)$ in PE $m+i$ to PE m, and then computing $s(m)s(m+i)$ in PE m for $0 \leq m < M-i$. These products are summed up using a recursive doubling technique (see Section 2.6). Figure 5.2 shows the pattern of data transfers used to compute the product terms. Figure 5.3 shows the data transfers used in recursive doubling with a Cube transfer function. Using the Cube transfer function allows the sum of the products to appear in the first L PEs, i.e., on completion of the algorithm PEs 0 through L will contain $R(i)$, $0 \leq i \leq p$. This is done so that the data is in place for the LPC algorithm which follows autocorrelation. The LPC algorithm needs $R(i)$, $0 \leq i \leq p$ to be stored in PE i, $0 \leq i < p$.

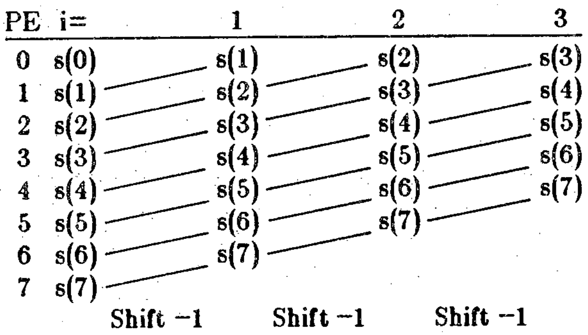
Assume that $M-p \leq L$ and M is a power of two, then the total number of parallel multiplications performed in the algorithm is $p+1$. For each $R(i)$, the recursive doubling requires at most $\lceil \log M \rceil$ parallel additions, so the total number of addition steps is $(p+1)\lceil \log M \rceil$. The number of Shift $-1$ transfers performed is p, and the number of Cube transfer functions is at most $(p+1)\lceil \log M \rceil$. The total number of transfer steps is at most $p+(p+1)\lceil \log M \rceil$. The asymptotic complexity is reduced from $O(Mp)$ for the serial algorithm to $O(p \log M)$ for the SIMD algorithm.

### 5.1.2. Autocorrelation Using Two FFTs – AUTO2

Another parallel method to find the autocorrelation coefficients presented by Siegel[Si80a] is to take the fast Fourier transform (FFT) of the magnitude squared of the FFT of the signal $s(m)$ padded with zeros to a length of 2M. This method, referred to as AUTO2 is not practical on a serial machine, especially when only small number of coefficients are needed, since it requires so much computation time. However, on a parallel machine, certain values of M and p make this method practical.

```
        /*      Algorithm Name:   auto
                Section:          5.1.1
                Machine:          SIMD
                Function:         This program finds the autocorrelation
                                  coefficients of input speech data.
                Number of PEs:    N
                Transfers:        Shift(-1), Cube
                Masking:          Data Conditional
                Parameters:       autocoef, The number of coefs. to find.
                                  N, The number of PEs in use.
                                  NetD, The interconnection network
                                          delay time in cycles.
                Input:            The input data is stored in PEs 0 through N-1
                                  with PE i containing sample i for 0 ≤ i ≤N.
                Output:                   The autocorrelation coefficients, R(i),
                                  for 0 ≤ i ≤autocoef-1 appear in PE i
                                  for 0 ≤ i ≤N (i.e. each PE contains
                                  every coefficient).
                Cycles:           autocoef[136+NetD + (54+2NetD)logN] - 12 - NetD
                Typical Time:     1,757 μs for autocoefs=9, NetD=18, and logN=7.
                Variable Usage: (* means set by calling routine)
                        ADDR:     Address of PE (e.g. ADDR = 0 in PE 0).
                        L:        on completion, PEs 0-L will contains R(i).
                        partsum:  temporary  variable holding a partial sum.
                        R():      autocorrelatin coefficients.
                        sig:      input signal
                        slast:    after stage i: "slast" in PE m holds sig(m+i).
                */
Line  Time in μs
 1    1.5       slast ← sig       /* After stage I, "slast" in
                                  PE m holds sig(m+i) */
 2
 3    5         FOR i ← 0  TO p DO
 4    1.5               IF i ≠ 0 THEN
 5    3                           USE Shift(-1)
 6    1.5                         DTRin ← slast
 7    4.5                         TRANSFER
 8    1.5                         slast ← DTRout
 9    0.5               partsum ← 0
10    6.5               WHERE ADDR < M-i DO
11    9.25                        partsum ← slast * sig
12    2                 ENDWHERE
13    2.25              FOR j ← 0 TO max(⌈log(M-i)⌉-1,log(L-1)) DO
14    3                           USE Cube(j)
15    12.5                        TRANSFER partsum TO tmp
16    0.75                        partsum ← tmp + partsum
17    1.5               R(i) ← partsum
```

Figure 5.1. Algorithm for autocorrelation using N PEs. The execution times assume an 8 MHz MC68000. (See Section 7.3.)



| PE | i= | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | s(0) | s(1) | s(2) | s(3) |
| 1 | s(1) | s(2) | s(3) | s(4) |
| 2 | s(2) | s(3) | s(4) | s(5) |
| 3 | s(3) | s(4) | s(5) | s(6) |
| 4 | s(4) | s(5) | s(6) | s(7) |
| 5 | s(5) | s(6) | s(7) | |
| 6 | s(6) | s(7) | | |
| 7 | s(7) | | | |
| | | Shift −1 | Shift −1 | Shift −1 |

Figure 5.2. Data transfers to move s(m+i) to PE m to compute s(m)*s(m+i) terms for R(i), $0 < i \leq p$. shown for N=M=8, p=3.

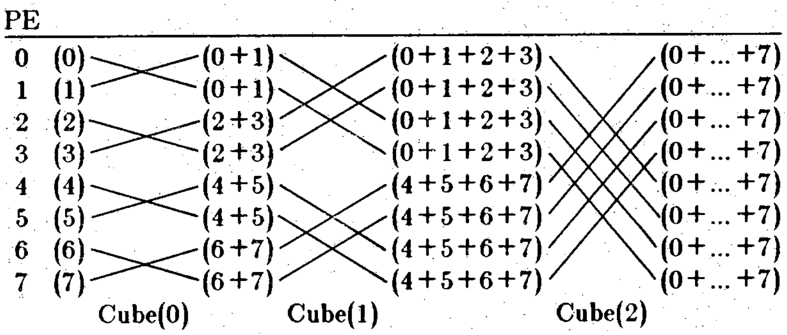


Figure 5.3. Performing sum of elements in N PEs using recursive doubling for N=8.

Siegel et al. [Si81,SMS79,MSS80], present an algorithm to compute the FFT using the decimation-in-frequency approach on an SIMD machine. This algorithm uses M PEs to compute the FFT of a 2M point signal where PE i initially contains s(i) and s(i+M), $0 \leq i < M$. Using this SIMD algorithm, each 2M-point DFT, M a power of 2, is computed in M PEs at a cost of log M+1 parallel complex multiplications, 2(log M+1) parallel complex additions, and log M parallel data transfers. Finding the magnitude squared of each of the 2M-points that are distributed over M PEs requires 2 complex additions and 2 complex multiplications. After the second FFT, p+1 broadcasts are needed to move the R(i)'s from the M PEs so that all R(i)'s appear in each of the first L PEs. Table 5.1 is a summary comparing the two methods.

### 5.1.3. Autocorrelation Using p+1 PEs – AUTO3

Ashajayanthi [ASV79] also presents an algorithm to find autocorrelation coefficients. It is rewritten in Flock Algol and is listed in Figure 5.4 and referred to as AUTO3. AUTO3 uses p+1 PEs and the signal s(m), $0 \leq m < M$, is stored in PE 0 (or PE 0 reads s(m) from some input device). Lines 1-10 input each new s(m) and shift it from PE i to PE i+1 until PE i contains s(p−i) for $0 \leq i \leq p$. Figure 5.5a shows the data allocation after line 14 for p=3 and M=4. Lines 17-19 broadcast Q from PE p which is the oldest of the p+1 stored samples to all other PEs. Each PE multiplies this value times its current Q value and adds it to its own variable sum. Then lines 19-21 read in a new s(m) and shift the old samples from PE i to PE i+1 as shown in Figure 5.5b. After M loops, R(i) will be in PE p−i, $0 \leq i \leq p$. Lines 24-26 use p+1 broadcasts to send all R(i) values to all PEs. The computation times listed in Table 5.1 do not count lines 1-14 since the other SIMD algorithms all assumed the data was already in each PE.

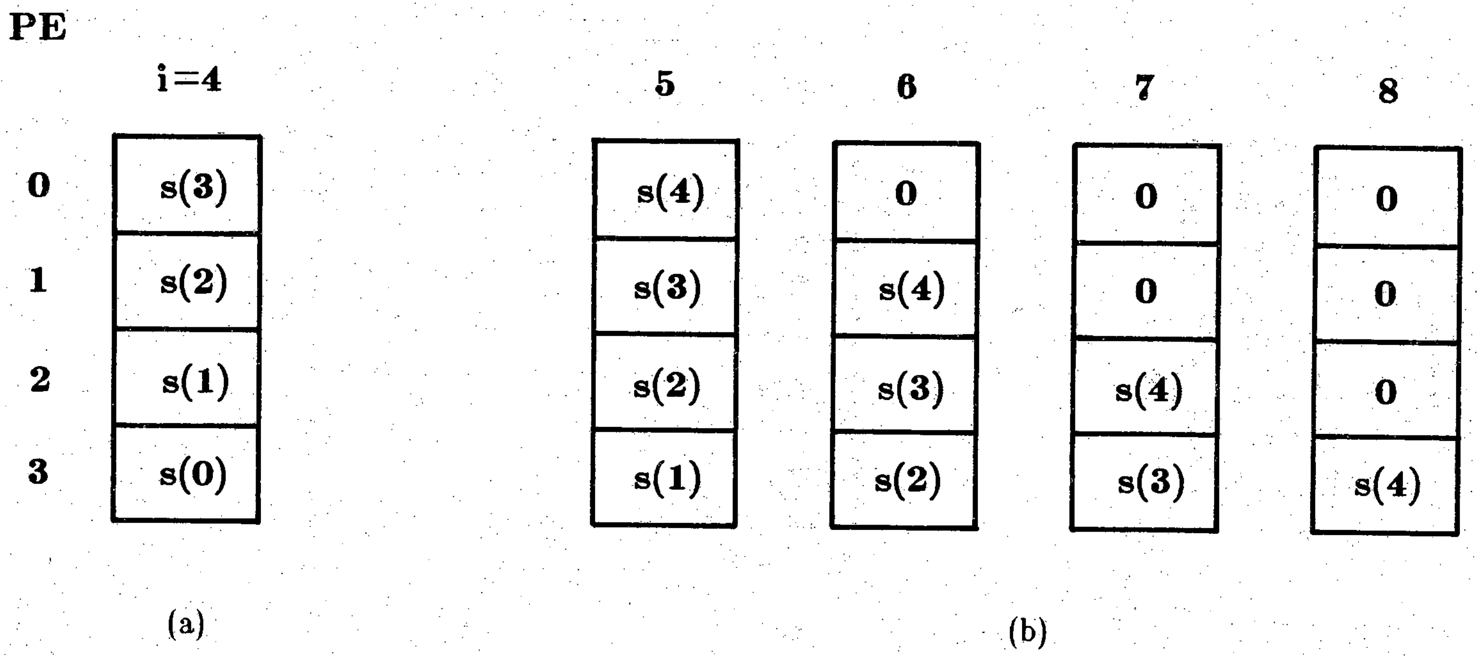
Table 5.2 shows the time complexity for each method with M=128 and p=8. There is no clear best method. If p is small compared to M, straight computation with M PEs (AUTO1) will require the least time. If p is close to M in value, FFT (AUTO2) is the fastest approach.

Table 5.1. Summary of the methods to compute autocorrelation coefficients.

| | PEs | additions | multiplications | transfers | broadcasts |
|---|---|---|---|---|---|
| Serial | 1 | M(p+1)-p(p+1)/2 | M(p+1)-p(p+1)/2 | | |
| AUTO1 | M | (p+1)log M | p+1 | p+(p+1)log M | |
| AUTO2 | M | 4log M+6<br>(complex) | 2log M+4<br>(complex) | 2log M | p+1 |
| AUTO3 | p+1 | M | M | M | M+p+1 |

```
/*
    sum: sum of all coefficients in each PE.
    p       : address of last PE.
    Q       : register used to hold values being shifted between PEs.
    R(i): autocorrelation coefficients. (output)
    s(i)    : input signal, enters in PE 0.
*/

sum ← 0                          /* Initialize autocorrelation functions sum to 0 */
USE Shift +1
FOR i ← 0 TO p-1 DO
            WHERE ADDR = 0 DO                  /* Shift in first p+1 samples into */
                  DTRin ← s(i)           /* PEs 0 through p */
            ELSEWHERE
                  DTRin ← Q
            ENDWHERE
            TRANSFER
            Q ← DTRout

WHERE ADDR = 0 DO                        /* Shift in new input sample */
            Q ← s(p)
ENDWHERE

FOR i ← p TO M-1 DO              /* Broadcast Q from PE p to all PEs */
            BROADCAST Q FROM PE p
            sum ← sum + Q * DTRout   /* Muliply Q times value from PE p */
            TRANSFER Q
            WHERE ADDR = 0 DO                  /* Input new sample into PE 0 */
                  Q ← s(i)
            ENDWHERE

FOR i ← 0 TO p DO                /* Store all coefficients in all PEs */
            BROADCAST sum FROM PE i
            R(i) ← DTRout
```

Figure 5.4. SIMD algorithm (AUTO3) to compute autocorrelation coefficients $R(i)$, $0 \leq i \leq p$, for an M-point signal, using $p+1$ PEs.



| PE | i=4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|
| 0 | s(3) | s(4) | 0 | 0 | 0 |
| 1 | s(2) | s(3) | s(4) | 0 | 0 |
| 2 | s(1) | s(2) | s(3) | s(4) | 0 |
| 3 | s(0) | s(1) | s(2) | s(3) | s(4) |

Figure 5.5. Contents of variable P in each PE at the start of line 16 for p=3, M=5.

Table 5.2. Time complexities for computing autocorrelation coefficients for $M=128$ and $p=8$.

| | PEs | additions | multiplications | transfers | broadcasts |
|---|---|---|---|---|---|
| Serial | 1 | 1116 | 1116 | | |
| AUTO1 | M | 54 | 9 | 62 | |
| AUTO2 | M | 30 (complex) | 16 (complex) | 12 | 9 |
| AUTO3 | p+1 | 128 | 128 | 128 | 137 |

## 5.2. Linear Prediction of Speech

Linear prediction is a popular method used in speech recognition and speech compression. Parallel algorithms for both coding speech into linear prediction coefficients and reconstructing speech from LPC coefficients are presented in the literature. The following sections discuss parallel algorithms for computing LPC coefficients using both autocorrelation and covariance methods. It also discusses a parallel algorithm for synthesis using LPC coefficients.

### 5.2.1. Parallel LPC Using the Autocorrelation Method

Siegel [Si80a,Si80b,Si81] presents an SIMD algorithm for linear predictive coding using Durbin's method [Makh75,RaSc78]. The serial algorithm is in Figure 4.2. The SIMD algorithm achieves its speedup over the serial algorithm by computing the k(i)'s in line 6 in parallel and the $a_j$'s in line 11 in parallel.

The SIMD algorithm uses P PEs to solve the p pole linear predictor, where $2^{m-1} < p \leq 2^m = P$. Initially, each PE contains all R(i)'s for $0 \leq i \leq p$. After stage i in the iteration, the predictor coefficient, $a_j^{(i)}$, is in the variable a of PE j mod N, for $1 \leq j \leq i$ (i.e., if $p < N$, PE j will contains $a_j$ for $1 \leq j \leq p$; if p=N, PE j will contain $a_j$ for $1 \leq j \leq p$, and PE 0 will contain $a_p$). At the completion of the algorithm, logical PE j will contain $a_j$ for $1 \leq j \leq p$.

The two parts of Durbin's method are:

1) computation of the k(i)'s from the R(i)'s and,

2) the iterative computation of the predictor coefficients ($a_j^{(i)}$'s) for an order i predictor from the k(i)'s and the predictor coefficients from the previous iteration.

The SIMD computation of the k(i)'s uses recursive doubling. For each iteration i, the $a_j^{(i)}$'s are computed by transferring data so that $a_j^{(i-1)}$ and $a_{i-j}^{(i-1)}$ are in the same PE, and then executing the operations of line 11 in the serial algorithm in parallel for all values of j, $0 \leq j \leq i$. Figure 5.6 shows the transfers needed for a 4-th order predictor computed in 4 PEs. No transfers are needed for i=1 and i=2. Stage i of Durbin's algorithm requires pairing elements $a_j^{(i-1)}$ and $a_{i-j}^{(i-1)}$, for $1 \leq j < i$, which is done with the $\mathrm{Perm}_i$

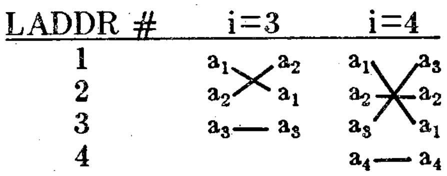


Figure 5.6. Data transfers for computation of $a_j$'s for p=4 in four PEs.

interconnection function. See Section 2.4.2 for more details on the Perm function.

The serial algorithm requires $p^2+p$ additions and multiplications, and p divisions to compute the $a_j$'s. Siegel's algorithm, shown in Figure 5.7, requires p multiplication steps to compute the k's (lines 4-7), and (p+1)log N additions and (p+1)log N data transfers (lines 11-15). Computing E (in lines 17-18) requires 2p multiplications and additions, and p divisions. Computing the $a_j$'s requires p−1 multiplications and divisions with p−1 data transfers. Table 5.3 summarizes these results. The parallel algorithm reduces the asymptotic time complexity from $O(p^2)$ to O(p log N).

### 5.2.2. Parallel LPC Coding Using the Covariance Method

The covariance method [RaSc78] is another method used to find the LPC coefficients of a speech waveform. This method involves solving:

$$\sum_{k=1}^{p} a_k \phi(i,k) = \phi(i,0) \qquad 1 \leq i \leq p$$

where $a_k$, $1 \leq k \leq p$, are the LPC coefficients and the covariance matrix, $\phi(i,k)$, is defined as:

$$\phi(i,k) = \sum_{m=-k}^{M-k-1} s(m)s(m+k-i) \qquad 1 \leq i \leq p\,,\, 0 \leq k \leq p \qquad (5.3)$$

This equation looks something like equation (4.1) which was used for the autocorrelation method, but the samples s(m), $-p \leq m < M$, are used where equation (4.1) used only s(m), $0 \leq m < M$. Equation (5.3) can be written as:

$$K\vec{a} = \vec{R}$$

where $\vec{R}$ and $\vec{a}$ are p element vectors of elements $\phi(i,0)$ and a(i) respectively for $1 \leq i \leq p$ and K is a p by p matrix with $K = \phi(i,k)$, $1 \leq i, k \leq p$. This is the same as the autocorrelation analysis equation (4.1) except K is symmetric, and *not* Toeplitz. Durbin's method cannot be used to solve for *a*; instead the Cholesky decomposition [RaSc78] can be used.

Siegel et al. [Si80b], presents a parallel SIMD algorithm to compute the covariance coefficients. This algorithm uses M PEs and requires p+1

```
        /*
                        LADDR:          logical address of PE (e.g. LADDR = i+1 in PE i).
                        a:              LPC coefficients (output)
                        E:              prediction error
                        k:              temporary variable
                        R():            Autocorrelation coefficients (input)
        */
Line    Time in μs
1       2.5             E ← R(0)
2       0.5             a ← 0
3       5.4             FOR i ← 1 TO p DO            /* Compute k(i)*/
4       0.75                k ← 0
5       6.5                 WHERE LADDR < i DO
6       12.75                   k ← a * R(i–LADDR)
7       2                   ENDWHERE
8
9                       /*  Sum k's in all PEs so all PEs have E     */
10
11      2.75                FOR j ← 0 TO logN – 1 DO
12      3                       USE Cube(j)
13      0                       DTRin ← k
14      17                      TRANSFER
15      0.75                    k ← k + DTRout
16
17      28                  k ← [R(i) – k] / E
18      31.25               E ← (1 – $k^2$) * E
19
20                      /*  Compute $a_j$'s for stage i */
22      3                   USE $\text{Perm}_{\text{LADDR}}$(i)
22      8.5                 WHERE LADDR = i DO
23                              a ← k  /* $a_i^{(i)}$←k(i)      */
24      2                   ELSEWHERE
25      6.5                     WHERE LADDR < i DO
26      1.5                         DTRin ← a
27      4.5                         TRANSFER
28      14.75                       a ← a – k * DTRout
29      2                       ENDWHERE
30      2                   ENDWHERE
```

Figure 5.7. SIMD algorithm using Durbin's method to solve for p predictor coefficients using p PEs. Executions times are based on an 8 MHz MC68000. (See Section 7.4.)

Table 5.3. Summary of parallel and serial LPC analysis algorithms.

| | | Additions | Multiplications | Divisions | Data Transfers |
|---|---|---|---|---|---|
| Serial | k's | $p(p+1)/2$ | $p(p+1)/2$ | | |
| | E | $p$ | $p$ | $p$ | |
| | $a_i$'s | $p(p-1)/2$ | $p(p-1)/2$ | | |
| | Total | $p^2+p$ | $p^2+p$ | $p$ | |
| Parallel | k's | $(p+1)\log N$ | $p$ | | $(p+1)\log N$ |
| | E | $2p$ | $2p$ | $p$ | |
| | $a_i$'s | $p-1$ | $p-1$ | | $p-2$ |
| | Total | $(p+1)\log N$ | $4p-1$ | $p$ | $(p+1)\log N+p-1$ |

multiplications, (p+1)(log M+1) additions, and log M(p+1)+3p+1 transfers. A serial covariance algorithm requires $Mp+p^2-p$ additions and multiplications. The parallel algorithm has reduced the time complexity from 0(pM) to O(p log M).

Safranek [Saf82] presents a parallel SIMD algorithm to solve equation (5.3) for *a*. This algorithm uses p PEs and consists of three parts: decompose, transpose, and solve. The decomposition part assumes $\phi(i,k)$, $1 \leq i, k \leq p$ will be stored in $\phi[j]$ in PE i. Table 5.4 shows the computation requirements for the decomposition. The decomposition results in a matrix which must be transposed. The transposition requires p+1 additions, and p transfers. Following the transposition, the predictor coefficients are then computed. Table 5.4 shows the operations used for solving for the predictor coefficients. Table 5.4 also shows the number of operations used by a serial algorithm for each of the three parts of the Cholesky decomposition. The time complexity of the serial algorithm is $O(p^3)$. The parallel algorithm, on the other hand, uses p PEs and has a time complexity of $O(p^2)$. Thus this method provides an ideal asymptotic speed up.

### 5.3. Digital Filtering

Digital filtering is frequently used in speech and signal processing. The following discusses four parallel algorithms for recursive digital filtering. The basic operations in recursive filters are the computation of the sum of product terms, with output $y_m$ given by:

$$y_m = \sum_{k=1}^{p} a_k y_{m-k} \tag{5.5}$$

where p is the order of the filter and $a_k, 1 \leq k \leq p$, are the filter coefficients and $y_i=0$ for $i < 0$. All four parallel algorithms solve equation (5.5) by breaking it down into the following recurrence relations.

$$y_m^{(0)} = 0 \tag{5.6a}$$

$$y_m^{(k+1)} = y_m^{(k)} + a_n y_{m-n} \qquad 0 \leq k \leq p-1 \text{ , } n=p-k \tag{5.6b}$$

Table 5.4. Operations needed for Choleksy decomposition.

| | Decomposition | | Transpose | | Solve | |
|---|---|---|---|---|---|---|
| | Parallel | Serial | Parallel | Serial | Parallel | Serial |
| Add/Sub. | $2p(p+1)$ | $2p^2(p+1)$ | $p+1$ | 0 | $4p$ | $4p^2$ |
| Multiply | $p(p+1)$ | $p^2(p+1)$ | 0 | 0 | $2p$ | $2p^2$ |
| Divide | $p^2+1$ | $p^3+1$ | 0 | 0 | $p+1$ | $p^2+1$ |
| Transfer | $p(p+1)$ | 0 | $p$ | 0 | $p+2$ | 0 |

$$y_m = y_m^{(p)} \tag{5.6c}$$

Kung's method (FIL1) is for a VLSI processor array, while Kogge's (FIL2) and Kuck's (FIL3, FIL4) methods are for SIMD machines.

### 5.3.1. Recursive Filtering for the VLSI Processor Array (FIL1)

Kung [KuLe,Kung80] has given systolic arrays to do both recursive and non-recursive filtering and has shown that these arrays are useful for both types of digital filtering. (In digital signal processing terminology, "recursive" filter typically refers to any filter that includes a recursive dependence of the output on previous outputs. A non-recursive filter is a filter whose output does not depend on previous outputs.) The non-recursive array was given as an example of a VLSI array in Section 3. The following is a description of a VLSI array to do recursive filtering.

Kung's recursive filter algorithm computes $y_m$ by using one cell for each of the p recurrence equations of equation (5.6b). Figure 5.8 shows the linear array of p+1 cells used to perform the computations. Each PE is the same as in the non-recursive filter algorithm, except that PE p is a dummy PE that reads the $R_y$ data from PE p−1 and routes this same data to $R_x$ in PE p−1. Figure 5.9 shows the data flow for the array in Figure 5.8.

Each cycle of the array consists of multiplying $R_y$ times $R_a$ and adding the product to $R_x$. This array can produce one $y_m$ every two cycles for a total of 2M cycles to produce all $y_m$'s for $p < m < M$.

### 5.3.2. SIMD Digital Recurrence Filter – Kogge (FIL2)

Kogge and Stone [KoSt73] have formulated an SIMD method for solving recurrence relations using recursive doubling. In this approach, the computation of M terms of equation (5.5) are found by rewriting the p equations of (5.6b) as:

$$Y_i = AY_{i-1}$$

where

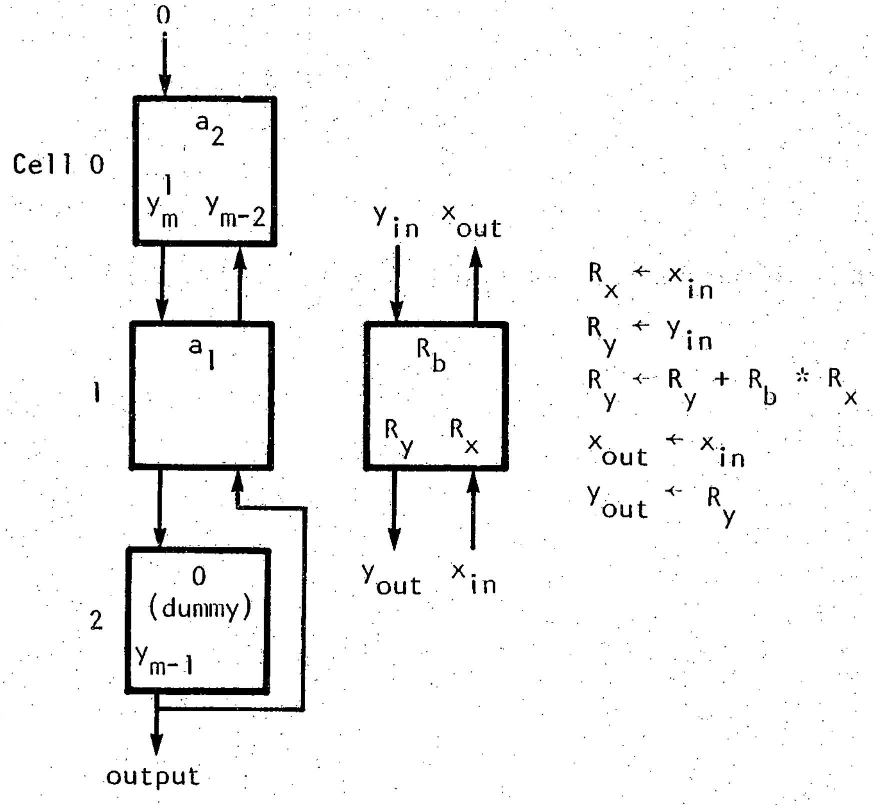


**Figure 5.8.** Systolic array to compute recursive filter for p=2.

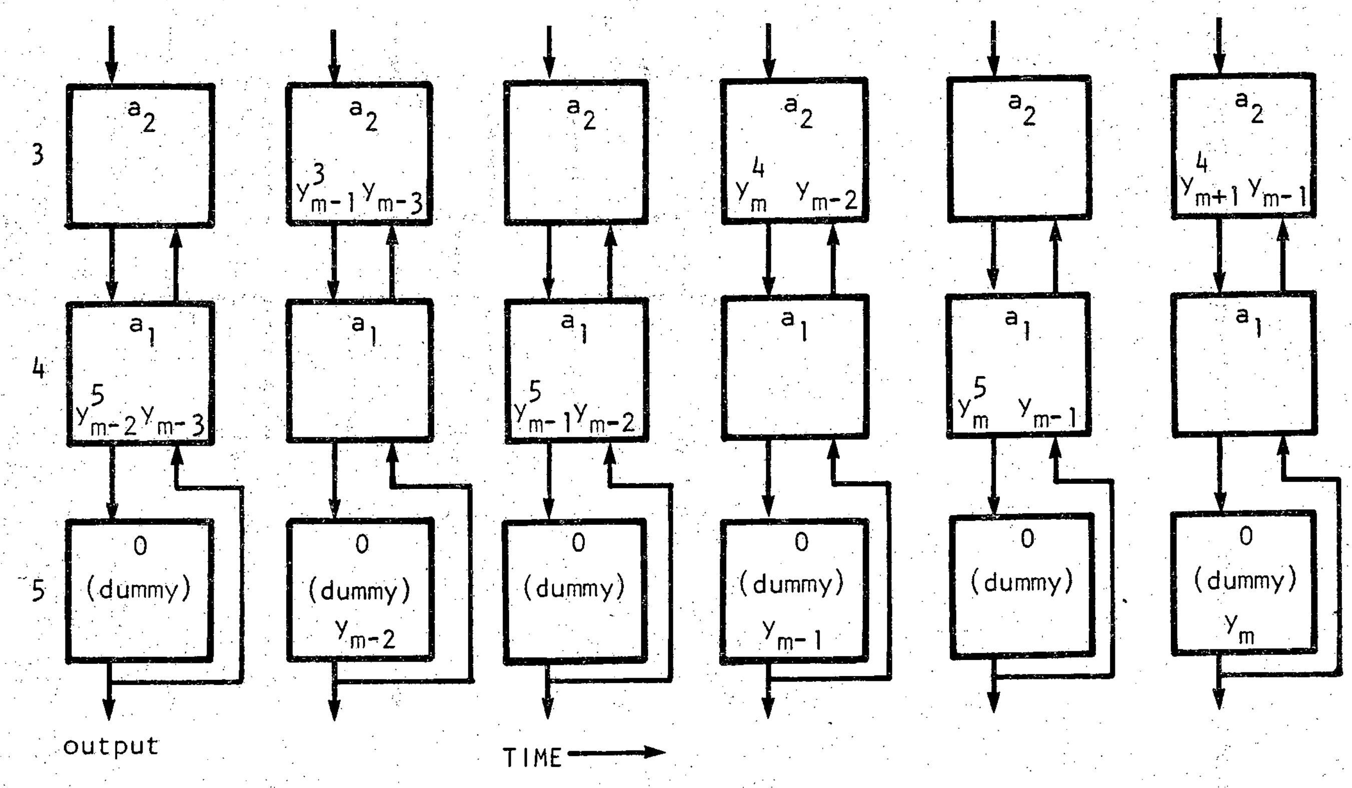


Figure 5.9. Data flow for array in Figure 5.8.

$$Y_i = \begin{bmatrix} y_{i-1} \\ . \\ . \\ . \\ y_{i-m} \end{bmatrix} \qquad A = \begin{bmatrix} a_1 & a_2 & . & . & . & a_p \\ 1 & 0 & . & . & . & 0 \\ 0 & 1 & . & . & . & 0 \\ . & . & . & . & . & . \\ 0 & . & . & 0 & 1 & 0 \end{bmatrix}$$

Therefore, A is a p by p matrix, and Y is a p by 1 vector. This approach uses M/p PEs and requires an initialization process plus $\lceil \log(M/p) \rceil$ steps. Each step, however, consists of multiplication of a p by p matrix by a p by 1 matrix and the transfer of the resulting p by p matrix to a different PE. This method is efficient when p is small and when M/p PEs are available.

### 5.3.3. SIMD Digital Recurrence Filter – Kuck

#### *5.3.3.1. Column Sweep Method (FIL3)*

Kuck [Kuck77] presents two algorithms to solve equation (5.5). The first is the column sweep method. It requires M−1 PEs, one for each $y_i, 1 \leq i \leq M-1$ that is to be computed. Initially, $y_0$ is known. In step 1, $y_0$ is broadcast to all PEs. Each PE multiplies $y_0$ by the correct $a_k$ and adds it to SUM. SUM is a variable in each PE which contains the intermediate $y_m^{(k)}$ terms from equation (5.6b) and $a_k$, $0 \leq k \leq p$, are the filter coefficients which have been precomputed and stored in each PE. After step 1, SUM in PE 0 contains $y_1$. Then $y_1$ is broadcast and the same is done for $y_1$ as was done with $y_0$. This continues until $y_m$ is found. This method requires M−1 steps. Each step consists of an addition, a multiplication, and a broadcast. This method is efficient when $p \simeq M$ and M PEs are available.

#### *5.3.3.2. Product-Form Recurrence Method (FIL4)*

Kuck's second method [Kuck77] to solve equation (5.5) is the fastest method known for computing recurrences. The method requires at most $(2+\log p)\log M - \frac{1}{2}(\log^2 p + \log p)$ steps. Each step consists of an addition and multiplication. The number of PEs used is at most $p^2M/2 + O(pM)$ for $p<<M$. For large p, the number of PEs used is quite large. The following section compares the four parallel recursive filtering algorithms.

### 5.3.4. Summary of Parallel Recursive Filtering Algorithms

Table 5.5 is a summary of the four algorithms. Consider the problem of a signal with M=128 samples and a p=16 pole filter. Table 5.6 shows how many PEs (cells) and steps are needed by each algorithm. FIL3 and FIL4 are designed for recurrences where $p \simeq M$. For digital filtering, $p<<M$, which makes FIL3 and FIL4 impractical for filtering applications. The number of PEs per steps required by FIL4 are both upper bounds, therefore these numbers could be much smaller. FIL2 uses the least number of PEs and steps, but each step requires a 16 by 16 matrix multiply, and a 16 by 16 matrix transfer. The matrix multiplication alone uses 256 scalar multiplications and 240 scalar additions. Therefore, FIL2 may be the slowest of the four.

FIL1 is the only algorithm whose number of PEs does not depend on M. It is also the only algorithm that can filter an arbitrary length signal. This is a desirable property for real time processing.

## 5.4. Dynamic Time Warping

As discussed in Section 4.6.2, dynamic time warping (DTW) is a common but time consuming method used in speech recognition. Its purpose is to compare each known utterance in the vocabulary to the unknown input utterance. The result of each comparison is a distance score, the lower the score, the better the match. Myers et al. [Myer80], reports that dynamic time warping uses from 50 to 90% of the computation time in word recognition on a serial computer. About 80% of the dynamic time warp calculation time is spent computing the local distances between feature vectors. This makes dynamic time warping a prime target when trying to reduce the total recognition time. One system mentioned in the literature to do dynamic time warping on a

Table 5.5. Summary of parallel recursive filtering algorithms.

| | PEs<br>(cells) | Operations<br>Per Cycle | Cycles to Compute<br>$y_m$, $p < m < M$ |
|---|---|---|---|
| FIL 1 | $p+1$ | 1 scalar add<br>1 scalar mult<br>2 shifts | 2M |
| FIL 2 | $M/p$ | 1 p by p matrix<br>mult.<br>1 p by p matrix<br>transfer | $\lceil \log_2(M/p) \rceil$<br>+ overhead |
| FIL 3 | M−1 | 1 scalar add<br>1 scalar mult<br>1 broadcast | M−1 |
| FIL 4 | $\leq p^2M/2 + O(pM)$<br>$p << M$ | 1 scalar add<br>1 scalar mult | $\leq (2+\log p)\log M-$<br>$(\log^2 p + \log p)/2$ |

Table 5.6. PEs and cycles needed to filter a M=128 sample signal with a p=8 pole recursive filter.

| | PEs<br>(cells) | Operations<br>Per Cycle | Cycles to Compute<br>$y_m$, $p < m < M$ |
|---|---|---|---|
| FIL1 | 17 | 1 scalar add<br>1 scalar mult<br>2 shifts | 256 |
| FIL2 | 8 | 1 p by p matrix mult.<br>1 p by p matrix transfer | 3 + overhead |
| FIL3 | 127 | 1 scalar add<br>1 scalar mult<br>1 broadcast | 127 |
| FIL 4 | $\leq$ 16,384<br>+ O(2,048) | 1 scalar add<br>1 scalar mult | $\leq$ 32 |

VLSI processor array is the high speed array computer (HSAC) by Burr et al. [BAW81,WBA83,BAW84]. The following section discusses the HSAC which uses a full I by I grid of cells where I is the number of frames in each utterance. The section after that presents a reduced array which requires fewer cells, but still exploits the parallelism of the DTW task.

### 5.4.1. High Speed Array Computer – Full Array

The HSAC presented in [BAW81] uses an I by I grid of cells to compare several vocabulary templates to the input template simultaneously. Figure 5.10 shows a typical cell which has two serial input lines and two serial output lines. The reference feature vector $\underline{a}_i$ enters the cell from the "bottom" in a bit serial manner as the test feature vector $\underline{b}_i$ enters from the "left" side. The cell calculates the local distance $d$ between them, and outputs $\underline{a}_i$ bit serially out of the top of the cell to the cell "above" it, while it outputs $\underline{b}_i$ to the cell to the right. The calculation of the accumulated distance, $g$, overlaps with the transfer of $\underline{a}_i$ and $\underline{b}_i$. Following the calculation of $g$, $g$ and $d$ are moved bit serially to both the cells above and to the right over the same lines that transferred the feature vectors. Overlapping the transfers with the calculations helps reduce the overhead of the bit serial transfers. All cells on an $x+y=k$ (for k equal to some constant) diagonal execute the same instructions at the same time, for example, cells (3,1), (2,2), and (1,3) perform the same instructions simultaneously; at the same time cells (4,1), (3,2), (2,3), and (1,4) execute the same instructions, which are possibly different from the (3,1), (2,2), (1,3) instructions. This allows one diagonal of cells to compute their accumulated distances, while an adjacent diagonal is receiving new feature vectors, thus overlapping transfers and calculations. Figure 5.11 shows an example of how sixteen of the HSAC cells are connected in a four by four grid. The unknown feature vectors enter the grid on the left, pass from cell to cell unchanged and emerge on the right. The reference vectors enter from the bottom and pass to the top.

To compare reference template $A=\{\underline{a}_1,\underline{a}_2,....\underline{a}_M\}$ to test template $B=\{\underline{b}_1,\underline{b}_2,....\underline{b}_M\}$, $\underline{a}_1$ enters cell (1,1) via $R_1$ while $\underline{b}_1$ enters via $U_1$. While finding the local distance, $\underline{a}_1$ is shifted to cell (1,2) while $\underline{b}_1$ is shifted to cell

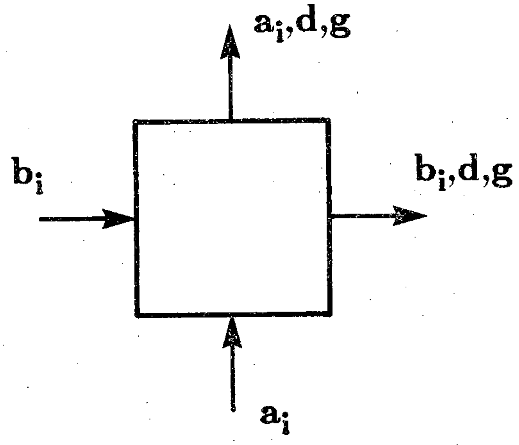


Figure 5.10. One cell in HSAC.

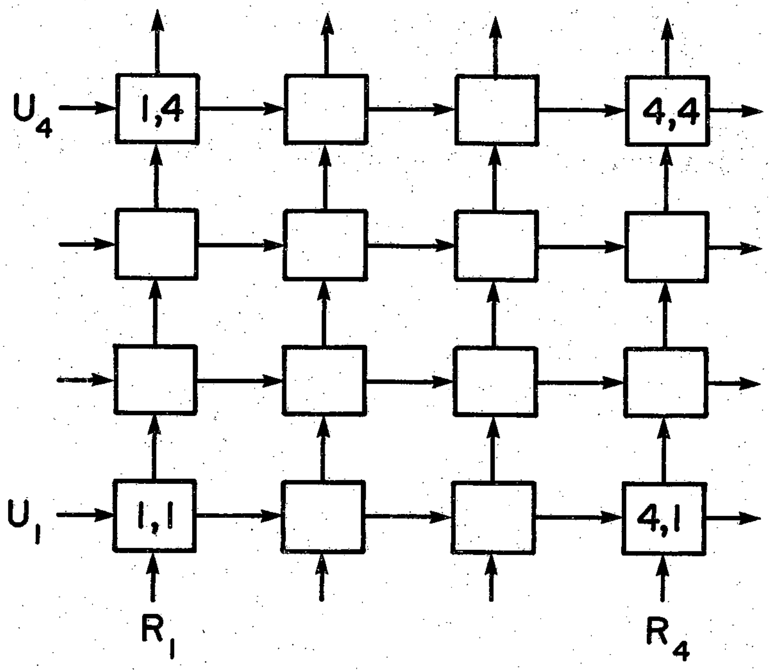


Figure 5.11. High Speed Array Computer used to compute dynamic time warp.

(2,1). $\underline{a}_2$ is enters into cell (2,1) via $R_2$ and $\underline{b}_2$ enters into cell (1,2) via $U_2$ at the same time. All cells on this diagonal find the local distance between $\underline{a}_1$,$\underline{b}_2$ and $\underline{a}_2$,$\underline{b}_1$ in parallel while shifting $\underline{a}_1$ and $\underline{a}_2$ to cells (1,3) and (2,2) respectively and shifting $\underline{b}_1$ and $\underline{b}_2$ to cells (3,1) and (2,2). This continues until cell (I,I) computes g(I,I) from vectors $\underline{a}_I$ and $\underline{b}_I$. g(I,I) is the optimal distance for the templates A and B. Figure 5.12 shows the data flow for I=4. In general $\underline{a}_i$ ($\underline{b}_i$) enters at $R_i$ ($U_i$) one loop* after $\underline{a}_{i-1}$ ($\underline{b}_{i-1}$) enters at $R_{i-1}$ ($U_{i-1}$). Cell (i,j) computes d(i,j) and g(i,j), with computation progressing on a diagonal wave from the lower left to the upper right of the array. For a W-word vocabulary, the comparison of words X and Y can start one loop after words X−1 and Y−1 are started by entering $\underline{a}_i^X$ ($\underline{b}_i^Y$) in $R_i$ ($U_i$) one loop after $\underline{a}_i^{X-1}$ $\underline{b}_i^{Y-1}$ enters $R_i$ ($U_i$). For a W word vocabulary with I frames per word, the HSAC requires 2I−1 loops to compute the first comparison, and one loop for each subsequent comparison, for a total time of 2I+W−2 loops. HSAC needs $I^2$ cells if an adjustment window is not used. If an adjustment window is used, the cells in the upper left and lower right corners can be omitted leaving an r cell wide "warping path" from cell (1,1) to cell (I,I). Only $2Ir-I-r^2+r$ cells are needed, but the same number of loops are required. For I=40, the HSAC requires 1600 cells if no adjustment window is used; if an adjustment window of r=8 is used, 544 cells.

554 is a large number of cells. The next section discusses reduced arrays, which can use fewer cells.

### 5.4.2. High Speech Array Computer – Reduced Arrays

Implementing the HSAC with a full array of cells require a large number (>500) of cells and is dependent on the problem size since the array must have as many rows and columns as unknown frames in the utterance. West, Ackland, and Burr [WBA83,BAW84] present the "reduced" array which overcomes these problems. The reduced array uses enough cells to compute an integral number of diagonals in parallel. Figure 5.13 shows a reduced array with three

*A loop, as used here, is defined as the time after vector $\underline{a}_x$ enters the grid and before vector $\underline{a}_{x+1}$ enters.

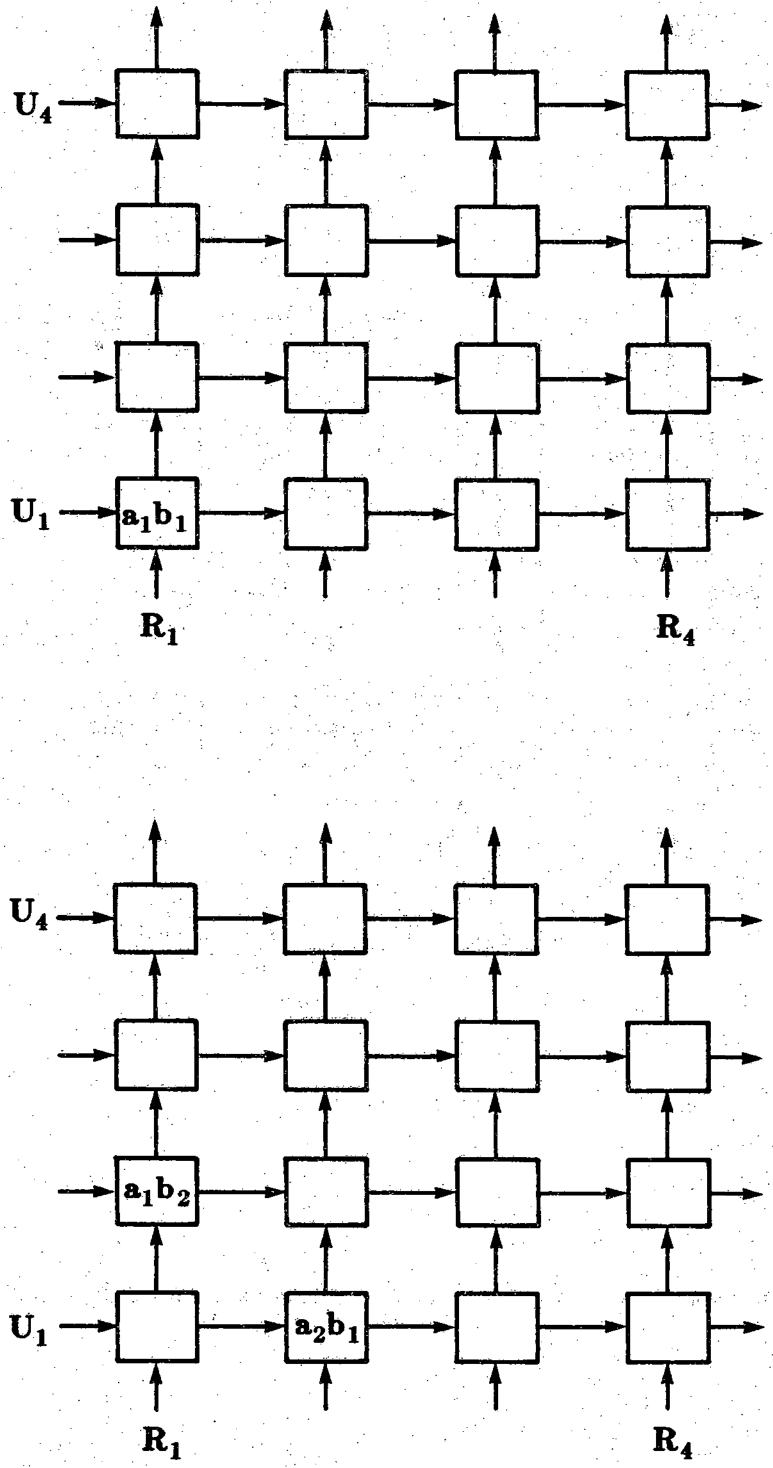


Figure 5.12. Data flow for HSAC.

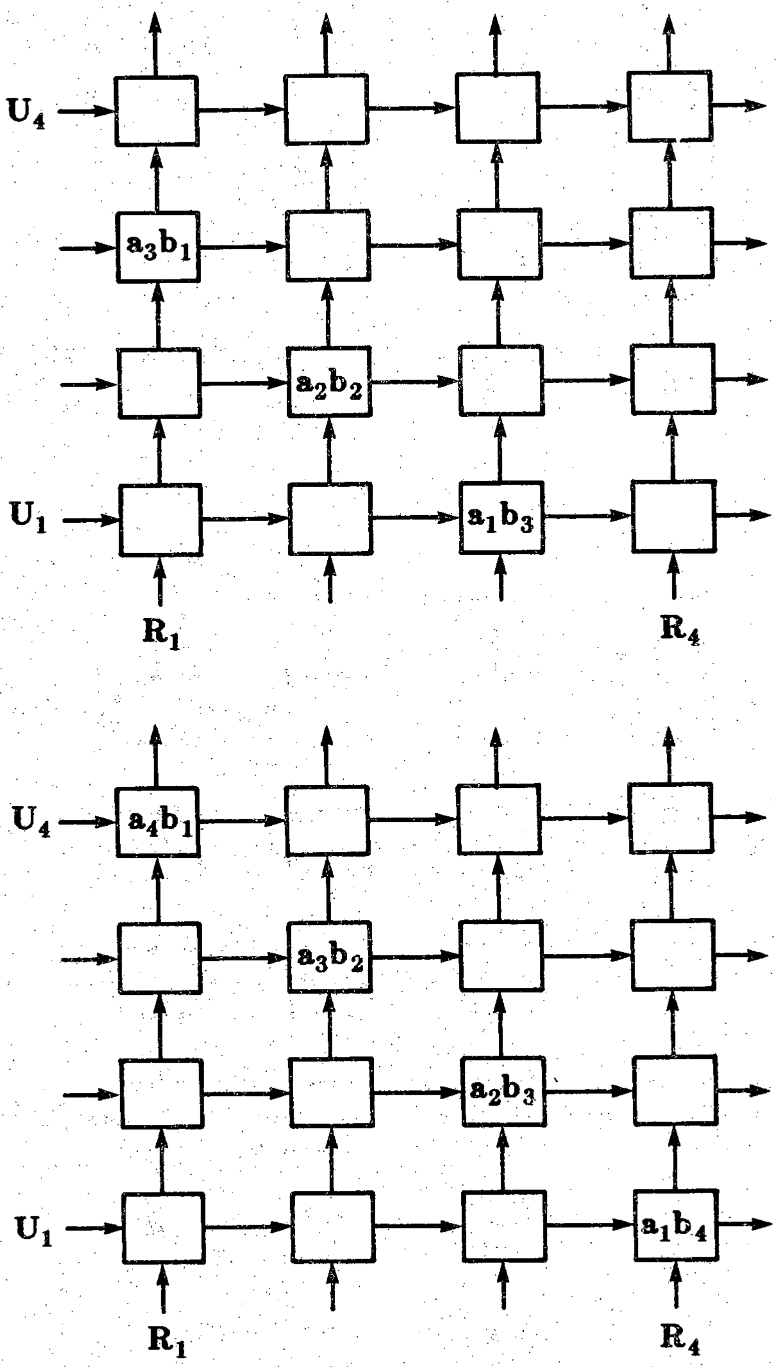
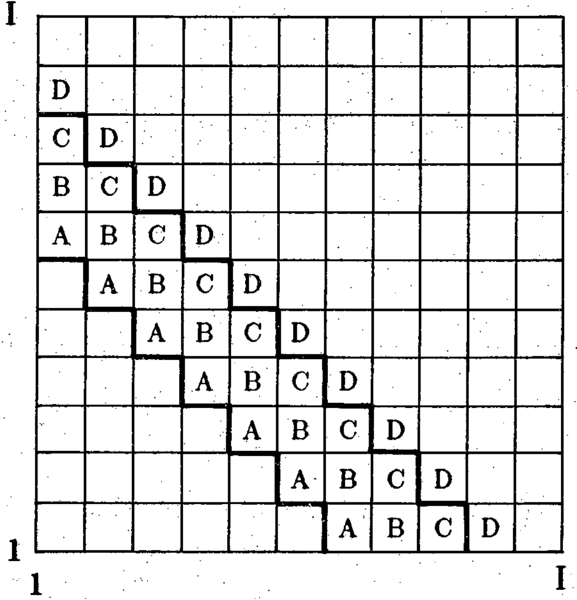


Figure 5.12. (Continued)



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | |
| D | | | | | | | | | | |
| C | D | | | | | | | | | |
| B | C | D | | | | | | | | |
| A | B | C | D | | | | | | | |
| | A | B | C | D | | | | | | |
| | | A | B | C | D | | | | | |
| | | | A | B | C | D | | | | |
| | | | | A | B | C | D | | | |
| | | | | | A | B | C | D | | |
| | | | | | | A | B | C | D | |

Figure 5.13. Virtual movement of reduced array through I by I grid.

diagonals. The large square represents the I by I grid of a full array. Three pairs of vectors are being compared simultaneously. The diagonals labeled A, B, and C are the three diagonals of the reduced array which are doing the comparison. When the computations for the current diagonal are complete, the A diagonal will move to the B diagonal, and the B diagonal will move to the C diagonal. The C diagonal would move to the D diagonal in a full array, but there is no D diagonal in the reduced array. Instead, the C diagonal moves to the A diagonal in the reduced array.

The reduced array is therefore sweeping the matrix space of the I by I grid as shown in Figure 5.14. The advantages of the reduced array are:

1) Fewer cells are used.

2) The number of diagonals used is independent of the problem size.

3) The number of cells can be traded off for performance.

The disadvantages are:

1) Some cells are idle during the computation as shown in Figure 5.14.

2) Slightly more complex hardware is needed to recirculate the data from the right edge of the reduced array to the left.

3) Fewer pairs of utterances can be compared at a time.

The smallest size a reduced array can be is one diagonal. If no adjustment window is used, the diagonal will have I cells. If an adjustment window is used, r cells are needed. The one diagonal reduced array can compute one comparison in 2I loops.

This HSAC can not use pruning since pruning aborts a comparison if at some time during the comparison it is apparent the current comparison will not be the closest match. Once this array starts a comparison it is difficult to abort it without affecting the other comparisons that are occurring in parallel [WBA83].

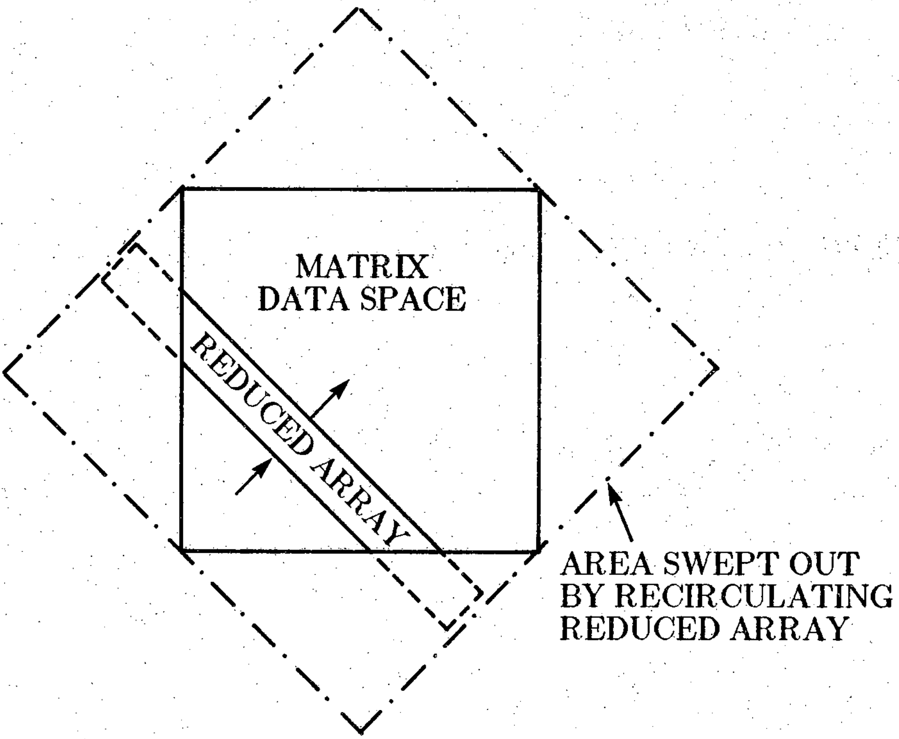


Figure 5.14. Virtual propagation of diagonal reduced array (from [BAW84]).

### 5.5. Summary

This section presented parallel algorithms for autocorrelation, LPC analysis, dynamic time warping, and digital filtering. One of the filtering algorithms, the three autocorrelation algorithms, and the LPC algorithm are for the SIMD machine. Three of the dynamic time warping algorithms and the rest of the digital filtering algorithms are for VLSI arrays. Several new algorithms for speech processing for both SIMD machines and VLSI processor arrays are presented in the next chapter.

# 6. NEW PARALLEL ALGORITHMS FOR SPEECH PROCESSING

The following are several new algorithms for speech processing on SIMD machines and VLSI processor arrays. This chapter presents four parallel algorithms for digital filtering, one for autocorrelation analysis, two for linear time warping, along with three algorithms for dynamic time warping. Each section presents an algorithm and then discusses the machine requirements and speed up obtained by the algorithm.

## 6.1. Digital Filtering

The basic operations in digital filtering are the computation of sum of products terms, with output $y_m$ given by

$$y_m = \sum_{k=1}^{p} a_k y_{m-k} + \sum_{k=0}^{q} b_k x_{m-k} \qquad p \leq m < M \tag{6.1}$$

where $x_m$ is the input to the filter at sample m, the $a_k$'s and $b_k$'s are the filter coefficients, and M is the number of samples in the signal to be filtered. The first sum in (6.1) represents a recursive filter. (In digital signal processing terminology, "recursive" filter typically refers to any filter that includes a recursive dependence of the output on previous outputs, so the filter in (6.1) is a recursive filter. To make a distinction between the recursive and non-recursive portions of the computation, we will refer to (6.1) as a "generalized" recursive filter, and will use the term recursive filter to refer to a filter having only a recursive dependence.) In the recursive filter, the dependence of output $y_m$ on the previous $y_{m-k}$ values, $1 \leq k \leq p$, takes the form of a linear recurrence relation. The second sum in (6.1) represents a non-recursive filter, in which the current output value depends only on the current and q previous input values. In digital filtering applications, non-recursive filters are used to realize finite impulse response (FIR) filters, and it is common for q to be as large as 250 [RaGo75]. In digital filtering applications, generalized recursive filters are used to realize infinite impulse response (IIR) filters (e.g., Butterworth or Chebyshev filters, or filters for linear prediction [Makh75,Si80b]), with $p \leq 20$ [RaGo75].

Real-time applications often use digital filtering as a single processing step in tasks requiring other extensive computations. It is therefore desirable to consider fast implementations. The computations required for digital filtering are also characteristic of the general class of problems involving linear systems and linear recurrences. Some work in the use of parallel systems for solution of such problems has been reported. Kung [Kung80] presents systolic array algorithms to implement the two basic types of filters, non-recursive and recursive, that were described in Sections 3 and 5.3. Because of the recursive nature of the computation, the systolic array appears to be a natural structure for implementing the digital filter. Kogge and Stone [KoSt73] have formulated an SIMD method for solving recurrence relations using recursive doubling. Kuck [Kuck77] presented two SIMD algorithms for solving recurrence relations. The first used the column sweep method, and the second used a product-form recurrence method. All these approaches were discussed in Section 5.3.

This section presents five parallel algorithms to perform digital filtering. Four of these algorithms originally appeared in [YoSi81]. The first (VLSI1) is a simple extension of Kung's systolic array algorithms, showing how the non-recursive and recursive systolic arrays can be combined in a straightforward way. The second (SIMD1) is an SIMD algorithm derived from the VLSI1 approach. The third (VLSI2) is an VLSI processor array algorithm derived from the SIMD1 algorithm. The fourth (SIMD2) is an SIMD algorithm that assumes more powerful processors and more flexible inter-PE communications than the VLSI-based algorithms. The fifth algorithm presented (SIMD3) is an extension of the fourth algorithm to allow problems of varying sizes (number of coefficients) to be run on a fixed number of PEs. Together, the fourth and fifth algorithms provide a general method for dealing with recurrence relations in an SIMD system.

### 6.1.1. VLSI Processor Array Algorithm – VLSI1

The first VLSI processor algorithm presented here combines the nonrecursive (FIR) and recursive filter systolic algorithms covered in Sections 3 and 5.3.1 into a generalized recursive filter algorithm. It is based on a linear array of cells, with each cell holding one filter coefficient and data flowing in opposite directions in two pipelines. One pipe circulates the input data ($x_m$ values) while the other passes partial results in the $y_m$ computations. The generalized digital filter of equation (6.1) can be computed by combining the two algorithms discussed in Sections 3 and 5.3.1. The recurrence relations used for the generalized digital filter are:

$$\begin{aligned}
&y_m^{(0)} = 0 \\
&y_m^{(k+1)} = y_m^{(k)} + b_{q-k}x_{m-q+k} && 0 \le k \le q \\
&y_m^{(k+1)} = y_m^{(k)} + a_n\, y_{m-n} && q+1 \le k \le p+q \\
&y_m = y_m^{(p+q+1)}
\end{aligned} \tag{6.2}$$

Figure 6.1a shows that the recurrences can be evaluated by pipelining the $x_m$ and $y_m^{(k)}$ values through $p+q+2$ linearly connected cells. Input $x_m$ feeds into $R_x$ in cell q and output $y_m$ appears in $R_y$ in cell $p+q$. Figure 6.1b is the data flow diagram for the linear array. Each column of the data flow diagram represents the contents of each register in each cell after a given cycle. Moving from left to right shows how the data changes from one cycle to the next. The arrows show where $R_x$ and $R_y$ are transferred on the next cycle. As in the component algorithms, only half of the cells are active during a given cycle. Before the first cycle, the correct coefficient is loaded into each cell, and the $R_x$ and $R_y$ registers are set to zero. The first q cycles shift $x_0$ from the input line in cell q to $R_x$ in cell 0, $x_1$ from cell q to $R_x$ in cell 2, ..., and $x_{\lfloor q/2 \rfloor}$ to $R_x$ in cell $2\lfloor q/2 \rfloor$ (i.e., initializing the array by placing the first $\lfloor q/2 \rfloor + 1$ input values in every other cell, starting with $x_0$ in cell 0). After $p+q+1$ more cycles, every two cycles of the VLSI array compute one output value, where during each cycle, the operations performed are the simultaneous transfer of data in the two pipes, one addition, one multiplication, and one assignment.

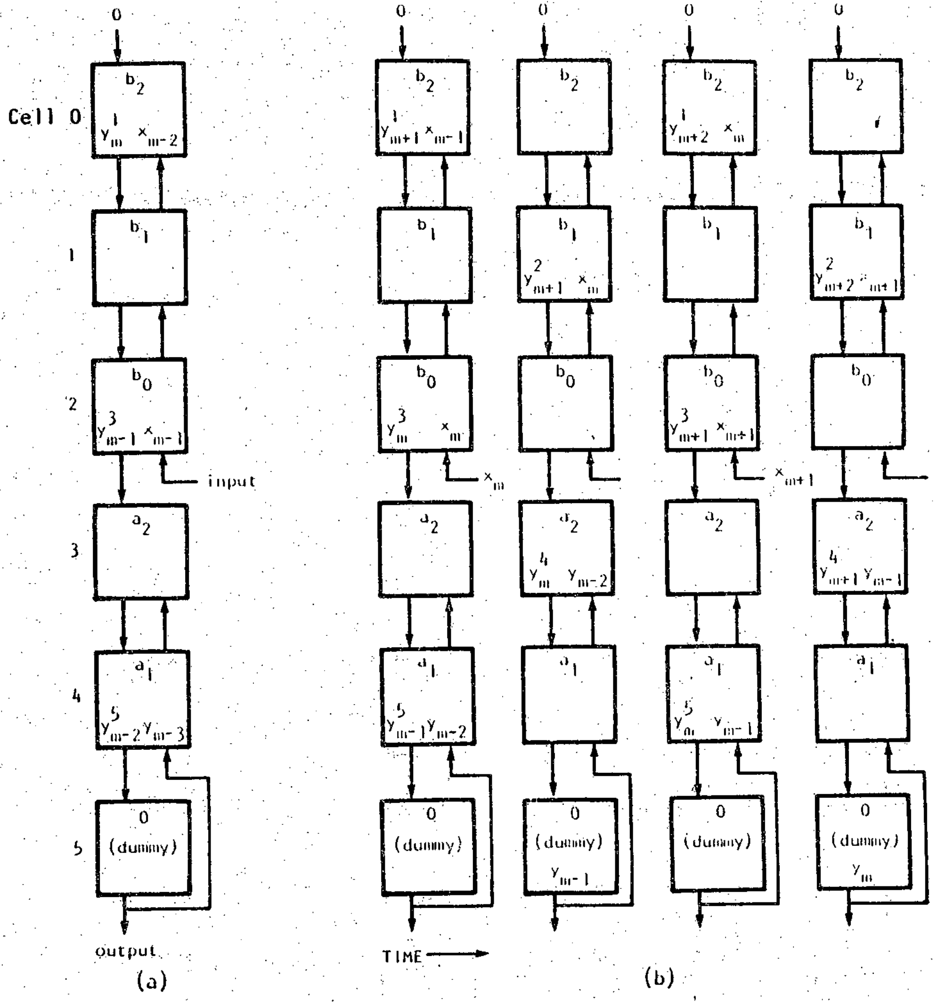


Figure 6.1. a) VLSI processor array to compute generalized digital filter p=2, q=2. b) Data flow diagram for (a).

### 6.1.2. An Improved Parallel Filtering Algorithm – SIMD1 and VLSI2

A major drawback to the VLSI processor array algorithm is that only half of the cells are active during a given cycle, so that a new $y_m$ value is computed every two cycles of the VLSI array. This problem can be overcome on the SIMD machine by using a data broadcast. A broadcast sends a data item in one PE to a specified set of PEs. A broadcast may be implemented either by having the control unit broadcast the data item to all the desired PEs (e.g., Illiac IV [Barn68,Bouk72]), or by using the interconnection network to transfer the data item to the desired PEs (e.g., Cube [SiMc81b] or ADM [SiMc81a] networks). See Section 2.5 for more information on broadcasts.

In the first SIMD algorithm (SIMD1), each PE holds one filter coefficient, as in the generalized VLSI array algorithm. The upward flowing pipeline from the VLSI processor array structure, which was used to disseminate the input x values and the completely computed y values, is replaced by two broadcasts of data. One broadcasts the current x value, and the other, the newly computed y value. By making this replacement, every PE is active during every cycle. Figure 6.2 shows the data flow diagram using this technique. As each partial y shifts into a given PE, the correct coefficient and x (in PEs 0 through p) or y (in PEs $p+1$ through $p+q+1$) are there to meet it. Moreover, if a given PE receives an x as a result of the broadcast, it will not receive a y as a result of the broadcast, and vice versa. If the interconnection network (rather than the control unit) performs the broadcasts, it may be possible to do the two broadcasts to disjoint sets of PEs simultaneously [SiMc81a,SiMc81b]. Whether this is possible will depend on factors such as the type of interconnection network used, the actual sets to which the data items are being broadcast, and the way in which the x values enter the system. The data flow of the partial results (the $y_m^{(k+1)}$ values) and the placement of one coefficient per PE is the same as the VLSI processor array algorithm. The replacement of the upward pipe by two broadcasts simplifies the synchronization problems, and allows all PEs to be active at every step. Every cycle of the SIMD1 algorithm produces one output value; the operations performed in one cycle are the two possibly simultaneous broadcasts, one data transfer of partial results (the remaining pipe), one addition, one multiplication, and one assignment. This cycle is clearly longer than the cycle in the VLSI array; however, an output is produced every cycle instead of every two cycles.

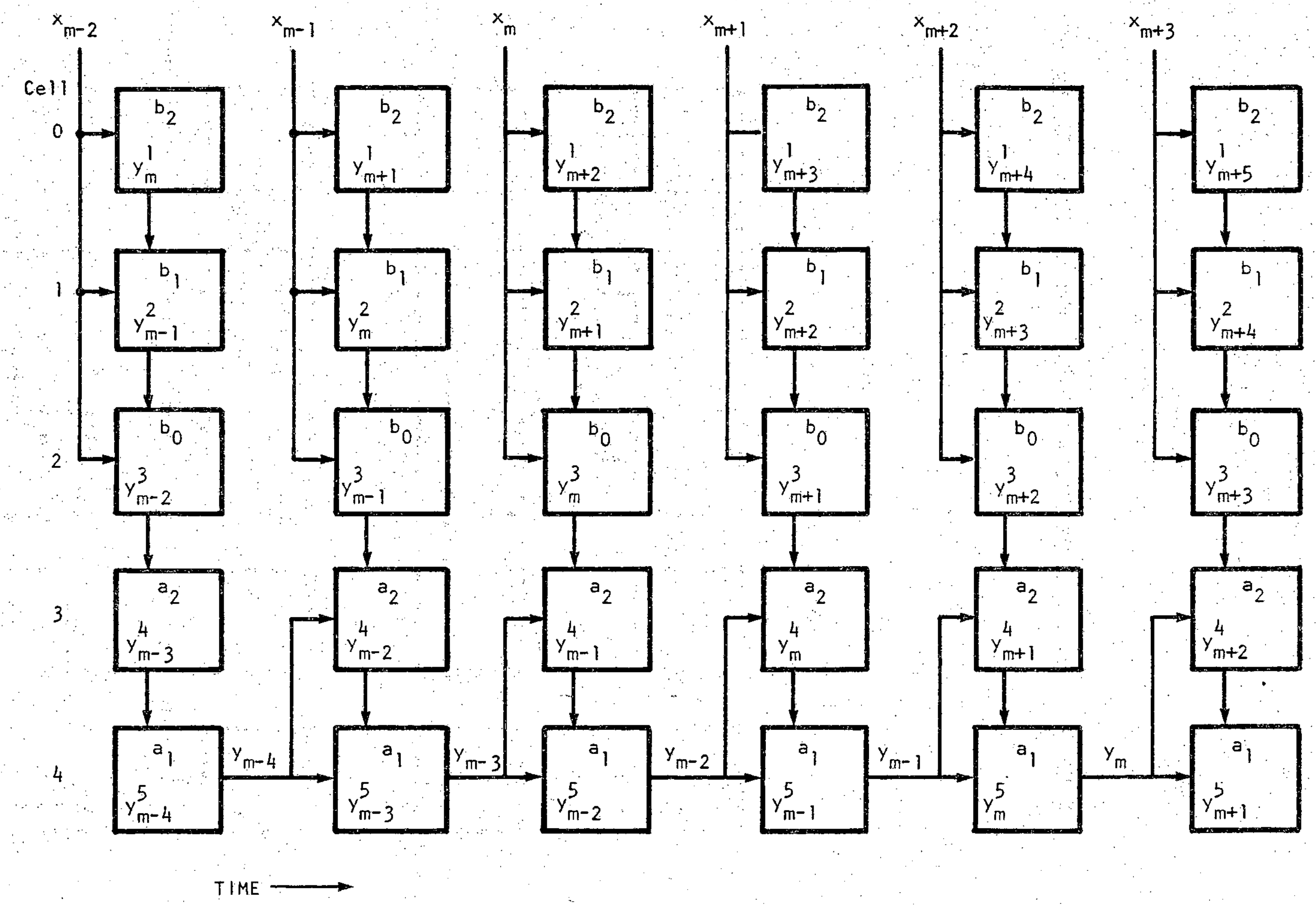


Figure 6.2. Data flow diagram for SIMD1 generalized digital filtering algorithm for p=2, q=2.

The principal attributes of the SIMD1 algorithm above are that

1) each PE holds one filter coefficient and always computes the same term (i.e., superscript k) of the recurrence for the $y_m$ values,
2) a pipeline similar to the VLSI array pipeline passes partial results from one PE to the next, and
3) broadcasts are used to disseminate new x and y values to the PEs in which they are needed.

The use of broadcasts is the only architectural difference between the SIMD1 algorithm and the VLSI1 algorithm. If the VLSI processor array can broadcast data, it can execute the same SIMD1 filtering algorithm as the SIMD machine. Therefore the second VLSI algorithm (VLSI2) is the same as the SIMD1 algorithm. The major differences are:

1) The broadcasts in the VLSI2 algorithm will occur simultaneously with the shifts, while the SIMD1 broadcasts and shifts must be performed sequentially.
2) The broadcast time in the VLSI2 algorithm should be much shorter than the SIMD1 algorithm since the VLSI2 algorithm uses a fixed interconnection network.

Section 6.1.5 will compare these algorithms.

### 6.1.3. An Improved SIMD Algorithm – SIMD2

The SIMD1 algorithm can be improved by arranging the data so that partial results ($y_m^{(k+1)}$ values) do not have to be shifted from one PE to another. In the SIMD2 algorithm, the same PE performs all the steps needed to compute a given $y_m$, as shown in Figure 6.3. Each PE holds all of the filter coefficients, and uses an indexing operation to select which coefficient to use at a given step of the algorithm. Partial results accumulate within the PEs, rather than being pipelined through them. The data transfers required are two (possibly simultaneous) broadcasts, one of the current input signal value, and one of a completed output value. All PEs are always active, each cycle of the algorithm completes one output, where the computations during a cycle are one indexing operation to select a filter coefficient, two broadcasts, one addition, one

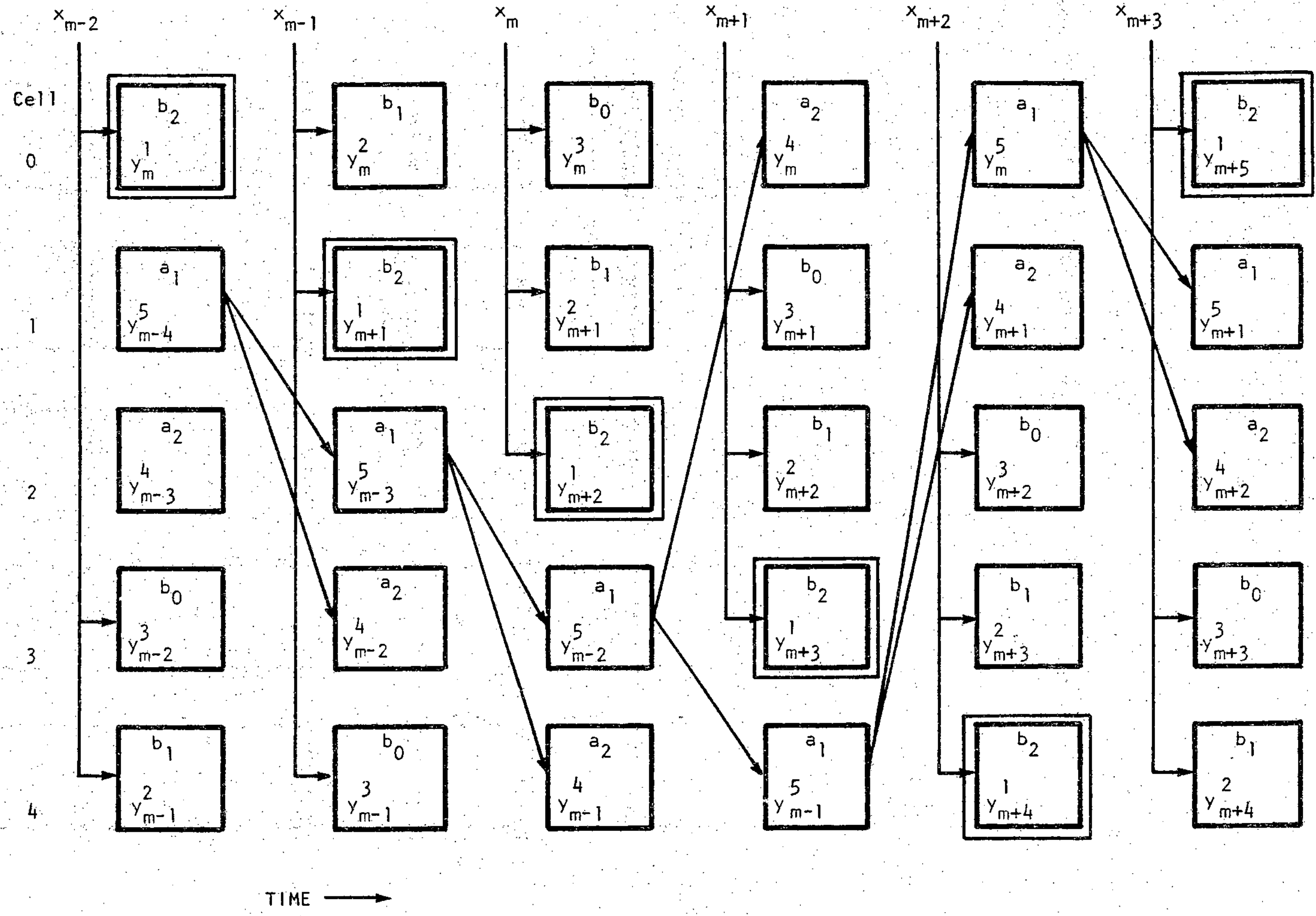


Figure 6.3. Data flow diagram for improved SIMD2 generalized digital filtering algorithm for p=2, q=2. The double boxes indicate the start of a new $y_m$ computation.

multiplication, and one assignment. Figure 6.4 shows that for this SIMD2 algorithm, the coefficients are arranged as a vector in each PE. This arrangement allows each PE to use the same index into the vector to access the correct coefficient for the cycle: at cycle m, each PE accesses its COEF [m mod (p+q+1)], where $p+q+1 = N =$ the number of PEs. The SIMD2 algorithm works on the computation of p+q+1 $y_m$'s simultaneously by having each PE at a different stage in the computation of its own $y_m$. The algorithm is again based on the recurrence relations in equation (6.2). For its own $y_m$, each PE is computing $y_m^{(k+1)}$ for a different value of k, $0 \leq k \leq p+q$. The data is arranged so that if PE i completes $y_m$ after cycle t, then PE (i+1) mod (p+q+1) will complete $y_{m+1}$ after cycle t+1. $y_m$ is used in computing $y_{m+j}$ for $1 \leq j \leq p$, so after PE i computes $y_m$, its value is broadcast to PEs (i+j) mod (p+q+1), for $1 \leq j \leq p$. In general, PE m mod (p+q+1) computes output $y_m$, and $y_m$ is completed in cycle m.

Figure 6.5 gives the SIMD2 algorithm that executes simultaneously in all PEs. Each PE will have its own values for the program variables. Initialization is handled by broadcasting 0 for the value $x_m$ during in the first q cycles of the algorithm. Combined with the initialization of SUM to 0, this ensures that $y[m] = 0$ for $m < 0$. At cycle q+1 (i.e., $m = -p$), $x_0$ is broadcast, followed by $x_1$ on the next cycle, etc. The computation of $y_0$ is completed during the cycle when $m = 0$, followed by completion of $y_1$ when $m = 1$, etc. The algorithm assumes that during each cycle, the current input value x  is broadcast as variable x from the control unit, and the interconnection network broadcasts the newly completed y value from the PE in which it was computed. For simplicity, the algorithm is written so that all PEs receive the broadcast y and x values, and each PE selects which one it will use in accumulating the next term in its sum. To perform this selection, each PE holds a vector of flags in which FLAG[i] is set to one if COEF[i] in that PE is an "a" coefficient, and set to zero if it is a "b" coefficient. By determining whether its COEF[m mod (p+q+1)] value for cycle m is an "a" or "b" coefficient, each PE can select whether it is to use the newly received y value (with an "a" coefficient) or the input x value (with a "b" coefficient) for cycle m.

| PE | COEF[0] | COEF[1] | COEF[2] | .... | COEF[p+q−1] | COEF[p+q] |
|---|---|---|---|---|---|---|
| 0 | $a_1$ | $b_q$ | $b_{q-1}$ | .... | $b_{q-2}$ | $a_2$ |
| 1 | $a_2$ | $a_1$ | $b_q$ |  | $b_{q-1}$ | $a_3$ |
| 2 | $a_3$ | $a_2$ | $a_1$ |  | $b_q$ | $a_4$ |
| . | . | . | . |  | . | . |
| . | . | . | . |  | . | . |
| . | . | . | . |  | . | . |
| p | $b_0$ | $a_p$ | $a_{p-1}$ |  | $a_{p-2}$ |  |
| p+1 | $b_1$ | $b_0$ | $a_p$ |  | $a_{p-1}$ | $b_2$ |
| . | . | . | . |  | . | . |
| . | . | . | . |  | . | . |
| . | . | . | . |  | . | . |
| q+p-1 | $b_{q-1}$ | $b_{q-2}$ | $b_{q-3}$ | .... | $b_{q-4}$ | $b_q$ |
| q+p | $b_q$ | $b_{q-1}$ | $b_{q-2}$ | .... | $b_{q-3}$ | $a_1$ |

Figure 6.4. Skewed coefficient storage for SIMD2 algorithm.

```
/*
              ADDR   Address of PE (e.g., ADDR = 0 in PE 0)
              DTRin  Data Transfer Register input to interconnection network
              DTRout        Data Transfer Register output from interconnection network
              coef[]  Vector of coefficients (see Figure 6.6)
              flag[i] Equals 1 if COEF[i] is an "a" coefficient
              sum     Contains partially computed $y_m$
              m       Index of y value to be completed in this cycle
                      (SUM = $y_m$ in PE (m mod (p+q+1))
*/

sum ← 0
FOR m ← −(p+q) TO M−1 DO
                      /* select the PE containing the newly */
                      /* completed y value: $y_{m-1}$. */
    BROADCAST sum FROM PE m−1 mod (p+q+1) TO DTRout
    WHERE ADDR = m−1 mod(p+q+1)
                      SUM ← 0        /* start a new sum in that PE */
    ENDWHERE

    WHERE flag[m mod (p+q+1)] = 1 DO /* In each PE, select to use */
                      tmp ← DTRout                /* either the broadcast y value */
    ELSEWHERE
                      tmp ← x                            /* or the new x value, $x_{m+p}$ */
    ENDWHERE

    sum ← sum + tmp * coef[m mod (p+q+1)]
```

Figure 6.5. SIMD2 generalized digital filtering algorithm.

### 6.1.4. SIMD Solution of General Linear Recurrence Equations

The approach presented in the SIMD2 algorithm for digital filtering can be applied to the solution of general linear recurrence equations of order p, given $y_i$ for $0 \leq i < p$, solve for $y_m$ for $p \leq m < M$, where

$$y_m = \sum_{k=1}^{p} a_{m,k}\, y_{m-k} + B_m \; .$$

The SIMD algorithm to handle the recursive dependence uses N = p PEs, with PE m mod p computing $y_m$. This PE completes computation of $y_m$ at cycle m, then broadcasts its completed $y_m$ value to PEs (m+j) mod p, for $1 \leq j \leq p$. PE i, $0 \leq i < N$, will hold the coefficient sets ($a_{m,k}$'s) for all m for which i = m mod p. The coefficient sets are skewed in a manner analogous to that in Figure 6.4. In particular, let z be such that z mod p = 0 (i.e., PE 0 computes $y_z$). Figure 6.6 shows that the coefficient sets $a_{(z+j),k}$ for $0 \leq j < p$ are stored. At cycle m of the computation, each PE will access its COEF[m mod p]. For example at cycle m, PE m mod p is completing computation of $y_m$. From Figure 6.6, this PE accesses $a_{m,1}$, which is the coefficient used with $y_{m-1}$, and which is the last term in the recurrence to be accumulated in computing $y_m$ in the SIMD algorithm. At the same time, each other PE is accessing the appropriate coefficient for its computation. Depending on the form of the $B_m$'s, it may be desirable and possible to use additional PEs to compute these terms. (This is the case in the digital filtering algorithm, when $B_m$ is considered to be the (q+1)-term non-recursive sum in each $y_m$.) This general method will reduce the number of multiplications and additions in solving an order p, M-point recurrence from p(M−p) in the serial algorithm to M+p in the p-PE SIMD method. The overhead in the SIMD algorithm is M+p broadcasts. The broadcast-based algorithm for digital filtering therefore provides an efficient general method for solving linear recurrence equations on an SIMD machine.

| PE | COEF[0] | COEF[1] | COEF[2] | ... | COEF[p−1] |
|---|---|---|---|---|---|
| 0 | $a_{z,1}$ | $a_{z,p}$ | $a_{z,p-1}$ | $\cdots$ | $a_{z,2}$ |
| 1 | $a_{z+1,2}$ | $a_{z+1,1}$ | $a_{z+1,p}$ | $\cdots$ | $a_{z+1,3}$ |
| 2 | $a_{z+1,3}$ | $a_{z+2,2}$ | $a_{z+2,1}$ | $\cdots$ | $a_{z+2,4}$ |
| . | . | . | . | | . |
| . | . | . | . | | . |
| . | . | . | . | | . |
| p−2 | $a_{z+p-2,p-1}$ | $a_{z+p-2,p-2}$ | $a_{z+p-2,p-3}$ | $\cdots$ | $a_{z+p-2,p}$ |
| p−1 | $a_{z+p-1,p}$ | $a_{z+p-1,p-1}$ | $a_{z+p-1,p-2}$ | $\cdots$ | $a_{z+p-1,1}$ |

Figure 6.6. Skewed coefficient storage for solution of general linear recurrence equations.

### 6.1.5. Comparison of VLSI Processor Array and SIMD Algorithms

Table 6.1 shows the times for the serial and parallel generalized digital filtering algorithms. (The "Preem" entry will discussed later in Section 6.1.8.) The parallel algorithms can be compared in three ways:

1) total time to compute one $y_m$,

2) number of $y_m$'s computed per unit time (throughput) (the throughput can also be considered by measuring the time between successive $y_m$'s), and

3) speedup over the corresponding serial algorithm.

The times considered are for the steady state operation of the algorithms. Although the algorithms require some initialization steps (for example, to distribute the first $\lfloor q/2 \rfloor + 1$ x's in the VLSI processor array algorithm), most of the processing is in the steady state operation.

The time to compute one $y_m$ value is the time from the beginning of the computation of $y_m$ until the time that $y_m$ is available as an output. In the VLSI processor array algorithms and SIMD1 algorithm, computation of each $y_m$ starts with the calculation of the $b_q x_{m-q}$ term in PE/cell 0 and completes on the inclusion of the $a_1 y_{m-1}$ term in the sum of PE/cell $p+q$. (In the VLSI1 algorithm, $y_m$ is available at this point, or access to $y_m$ may be delayed by one array cycle, until $y_m$ arrives at the output line in the dummy cell.) For all of these algorithms, the time to compute $y_m$ is the time to move, via the algorithm, from PE/cell 0 to PE/cell $p+q$, comprising $p+q+1$ algorithm cycles. The number of arithmetic steps to compute one y is therefore the same as in the serial algorithm. The VLSI processor array algorithms have an overhead of $p+q+1$ shifts and the SIMD1 algorithm has an overhead of $p+q+1$ shifts and $2(p+q+1)$ broadcasts. (This section assumes that the two SIMD broadcasts do not occur simultaneously.) The VLSI2 algorithm has $p+q+1$ shifts and broadcasts, assuming the two broadcasts can occur simultaneously. In the SIMD2 algorithm, the time to compute one y is the time for a single PE to perform the arithmetic operations (i.e., the serial time) plus the time for $p+q+1$ broadcasts of x values and $p+q+1$ broadcasts of completed y values. As in the SIMD1 algorithm, broadcasts of $x_{m+1}$ through $x_{m+q}$ and of $y_{m-q-p}$ through $y_{m-q-1}$ contribute to the time to compute $y_m$, even though they are not used in the $y_m$ calculations.

Table 6.1. Execution times for serial, VLSI, and SIMD digital filtering algorithms.

| | Output(s) | Additions | Multiplications | Shifts | Broadcasts | Speedup |
|---|---|---|---|---|---|---|
| Serial | 1 | $p+q+1$ | $p+q+1$ | 0 | 0 | |
| | M | $M(p+q+1)$ | $M(p+q+1)$ | 0 | 0 | |
| VLSI1 | 1 | $p+q+1$ | $p+q+1$ | $p+q+1$ | 0 | |
| | M | $2(M-1)+p+q+1$ | $2(M-1)+p+q+1$ | $2(M-1)+p+q+1$ | 0 | $(p+q+1)/3$ |
| VLSI2 | 1 | $p+q+1$ | $p+q+1$ | $p+q+1$ | $p+q+1$ | |
| | M | $M+p+q$ | $M+p+q$ | $M+p+q$ | $M+p+q$ | $2(p+q+1)/3$ |
| SIMD1 | 1 | $p+q+1$ | $p+q+1$ | $p+q+1$ | $2(p+q+1)$ | |
| | M | $M+p+q$ | $M+p+q$ | $M+p+q$ | $2(M+p+q)$ | $2(p+q+1)/(2+3t)$ |
| SIMD2 | 1 | $p+q+1$ | $p+q+1$ | 0 | $2(p+q+1)$ | |
| | M | $M+p+q$ | $M+p+q$ | 0 | $2(M+p+q)$ | $(p+q+1)/(1+t)$ |
| Preem | M | 1 | 1 | 1 | 0 | M |

For all the parallel algorithms, the time to compute M output values is the time to compute one y value plus the time to compute (M−1) subsequent y values. The latter time is obtained by considering the time between successive y values. The time between successive y's in the VLSI1 array is two additions, multiplications, and shifts since one y is computed every two cycles. The VLSI2 algorithm takes only one addition, multiplication, and shift/broadcast. The SIMD algorithms on the other hand do one addition, one multiplication, and either two broadcasts and one shift for SIMD1 or two broadcasts for SIMD2 between successive y values, since they compute one y every cycle. Depending on the SIMD broadcast versus VLSI processor array shift time, the second SIMD algorithm may have a greater complete throughput.

The speedup of an algorithm is (serial time/parallel time) [Kuck77]. Assume that additions and multiplications require one time unit on all machines, and data transfers (shifts or broadcasts) require one time unit on the VLSI processor array and *t* units on the SIMD machine. Also assume shifts and broadcasts occur simultaneously on the VLSI processor array and sequentially on the SIMD machine. The value of *t* will depend on a number of factors, including implementation details of the VLSI and SIMD machines. Table 6.1 shows the speed ups for the parallel algorithms, assuming that $M >> p+q$. If $t=2$, SIMD2 will have the same speed up as VLSI1. If $t=1/3$, SIMD2 will match the VLSI2 algorithm. If a multistage interconnection network such as the multistage Cube [SiMc81b] or Augmented Data Manipulator [SiMc81a] performs the broadcasts, it is unlikely that $t \leq 2$. Unless the broadcasts can be performed simultaneously, the speed up for the systolic array is significantly greater than for the SIMD algorithm. However, smaller values for *t* may be feasible. If the control unit performs the broadcasts, then the systolic and SIMD algorithms may have comparable speed ups.

### 6.1.6. Varying the Problem Size on an SIMD Machine

The VLSI processor array and SIMD algorithms can also be compared with respect to the ease with which the machine-size/problem-size relationship can be changed. In particular, assume the above techniques have been used to implement an order $p+q$ digital filter. Consider the impact of deciding to use a higher order filter. Let the new filter have $p'+q'+1$ coefficients, where $p'+q'+1 > p+q+1$. With some modifications, the SIMD2 algorithm can implement a filter having $p'+q'+1$ coefficients with fewer than $p'+q'+1$ PEs. Figures 6.7 and 6.8 show the data allocation diagrams for two different problem sizes. Case A is for $N = p+q+1$ and case B is for $N=p'+q'+1 < p+q+1$. Each rectangle in the diagram represents the cycles during which a given PE is computing a certain y. In each rectangle are the x and y values the PE needs during each cycle of the computation. In the original algorithm (case A, Figure 6.7) PE m mod $(p+q+1)$ computes output $y_m$. Since each $y_m$ computation required $p+q+1$ cycles, as soon as PE m mod $(p+q+1)$ completed computation of $y_m$, computation of $y_{m+p+q+1}$ was about to be started. The computations were skewed so all recurrences that required a given $x_m$ (or $y_m$) as input were computed during the same cycle. Case B (Figure 6.8) shows the data allocation needed to implement a $p'+q'+1$ coefficient filter with $N < p'+q'+1$ PEs. Each PE again performs all the computations for a given output, with $y_m$ computed in PE m mod N. However, since the number of cycles to compute $y_m$ is greater than the number of PEs, computation of $y_{m+N}$ does not begin until $y_m$ is completed. Cycles are classified into two types:

1) *transient cycles,* defined to be cycles in which any PE starts to compute a new y value, and

2) *steady state cycles,* cycles that are not transient.

Following every set of N transient cycles there are $p'+q'+1-N$ steady state cycles. Also, following every set of N cycles during which y values are completed, there are $p'+q'+1-N$ cycles during which no new y values are completed. During the set of N transient cycles, each PE can be placed into one of two classes:

1) PEs that have started computing a new y value since the beginning of the set of transient cycles, and

2) PEs that have not started computing a new y value.

At the start of the set of transient cycles, all PEs are in class 2. After each transient cycle, one PE completes its y value and therefore moves to class 1. At the end of the set of transient cycles, all PEs have moved to class 1. During the steady state cycles, the computations are skewed as in the Case A

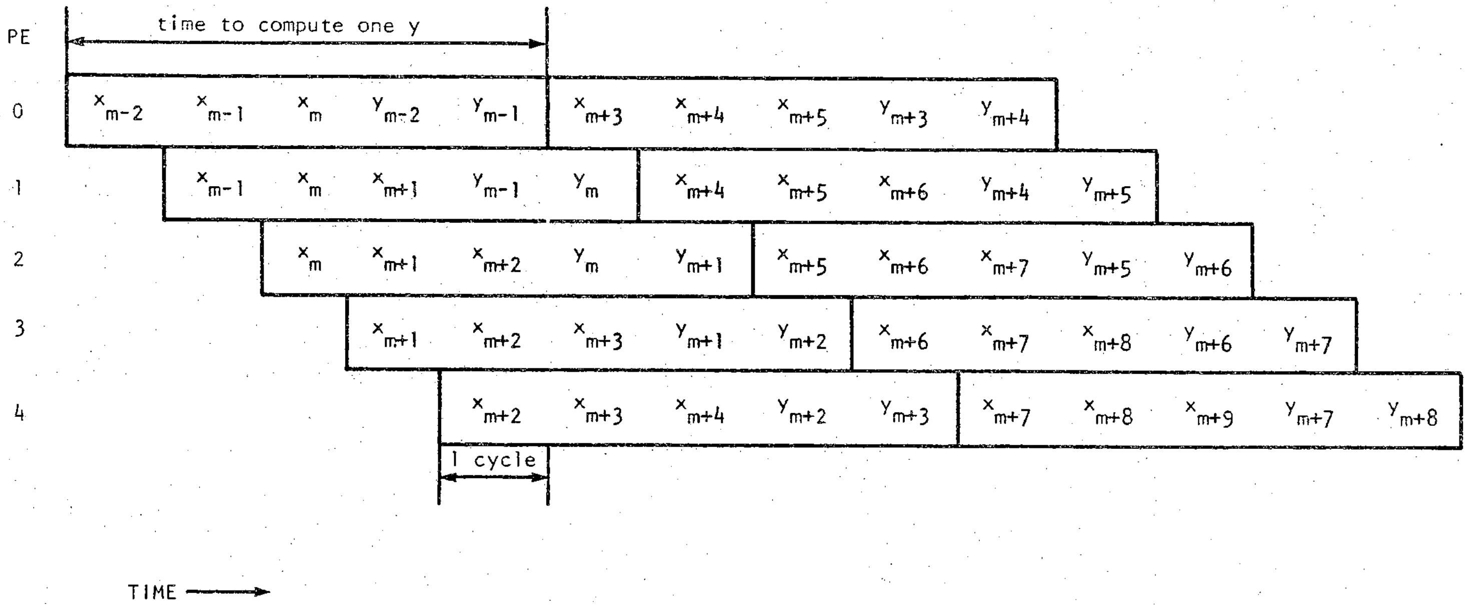

PE
time to compute one y
0
$x_{m-2}$ $x_{m-1}$ $x_m$ $y_{m-2}$ $y_{m-1}$ $x_{m+3}$ $x_{m+4}$ $x_{m+5}$ $y_{m+3}$ $y_{m+4}$
1
$x_{m-1}$ $x_m$ $x_{m+1}$ $y_{m-1}$ $y_m$ $x_{m+4}$ $x_{m+5}$ $x_{m+6}$ $y_{m+4}$ $y_{m+5}$
2
$x_m$ $x_{m+1}$ $x_{m+2}$ $y_m$ $y_{m+1}$ $x_{m+5}$ $x_{m+6}$ $x_{m+7}$ $y_{m+5}$ $y_{m+6}$
3
$x_{m+1}$ $x_{m+2}$ $x_{m+3}$ $y_{m+1}$ $y_{m+2}$ $x_{m+6}$ $x_{m+7}$ $x_{m+8}$ $y_{m+6}$ $y_{m+7}$
4
$x_{m+2}$ $x_{m+3}$ $x_{m+4}$ $y_{m+2}$ $y_{m+3}$ $x_{m+7}$ $x_{m+8}$ $x_{m+9}$ $y_{m+7}$ $y_{m+8}$
1 cycle
TIME

Figure 6.7. Data allocation for SIMD machine algorithm with N = p+q+1 PEs, shown for p=2, q=2.

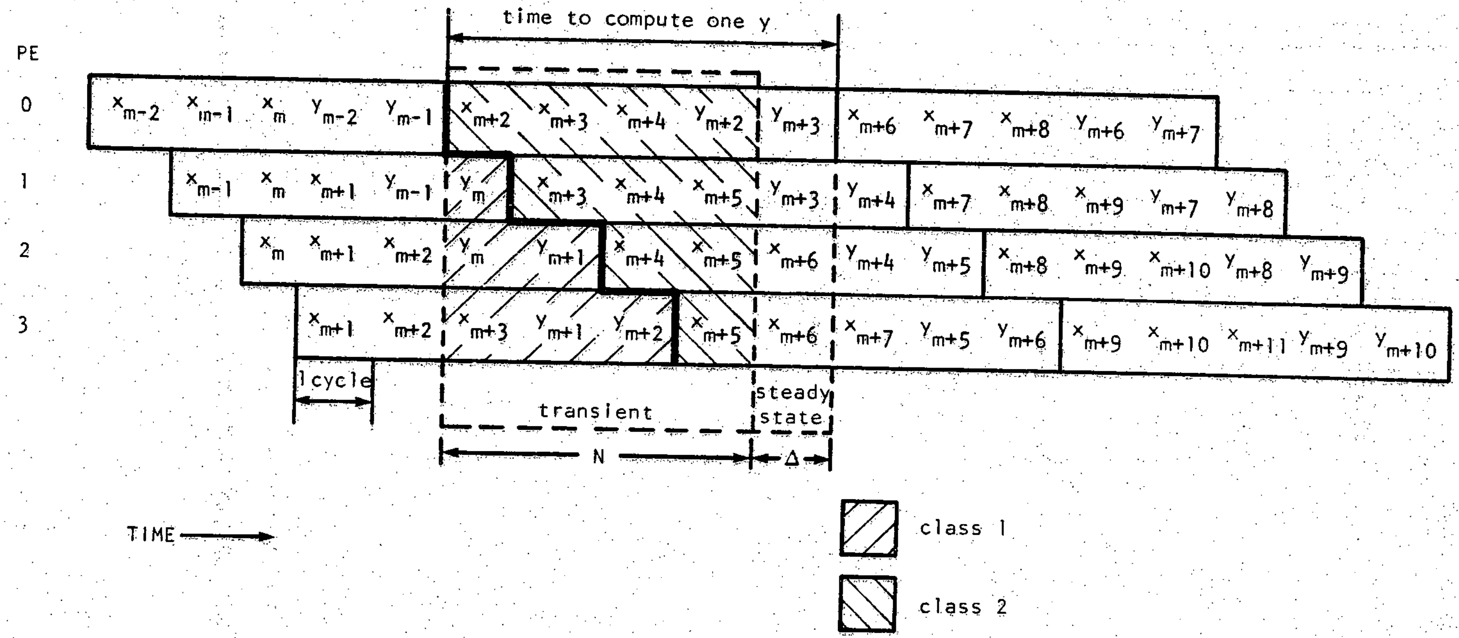

time to compute one y
PE
0
1
2
3
$x_{m-2}$ $x_{m-1}$ $x_m$ $y_{m-2}$ $y_{m-1}$ $x_{m+2}$ $x_{m+3}$ $x_{m+4}$ $y_{m+2}$ $y_{m+3}$ $x_{m+6}$ $x_{m+7}$ $x_{m+8}$ $y_{m+6}$ $y_{m+7}$
$x_{m-1}$ $x_m$ $x_{m+1}$ $y_{m-1}$ $y_m$ $x_{m+3}$ $x_{m+4}$ $x_{m+5}$ $y_{m+3}$ $y_{m+4}$ $x_{m+7}$ $x_{m+8}$ $x_{m+9}$ $y_{m+7}$ $y_{m+8}$
$x_m$ $x_{m+1}$ $x_{m+2}$ $y_m$ $y_{m+1}$ $x_{m+4}$ $x_{m+5}$ $x_{m+6}$ $y_{m+4}$ $y_{m+5}$ $x_{m+8}$ $x_{m+9}$ $x_{m+10}$ $y_{m+8}$ $y_{m+9}$
$x_{m+1}$ $x_{m+2}$ $x_{m+3}$ $y_{m+1}$ $y_{m+2}$ $x_{m+5}$ $x_{m+6}$ $x_{m+7}$ $y_{m+5}$ $y_{m+6}$ $x_{m+9}$ $x_{m+10}$ $x_{m+11}$ $y_{m+9}$ $y_{m+10}$
1 cycle
transient
steady state
N
Δ
TIME
class 1
class 2

Figure 6.8. Data allocation for SIMD machine algorithm with $N < p+q+1$, shown for $p=2$, $q=2$, $N=4$.

computation, so all recurrences requiring a given $x_m$ (or $y_m$) as input are computed during the same cycle. However, during the transient cycles, the PEs in class 1 need a different set of x's and y's than the PEs in class 2 (see Figure 6.8).

Figure 6.9 gives an algorithm to implement a filter where the number of PEs is less than the number of filter coefficients. Lines 6-16 compute the steady state cycles, while lines 18-39 handle the transient cycles. Line 25 broadcasts the newly computed y value to all PEs and line 29 stores the newly computed value in the y[ ] vector. The variable *diff* is used to determine whether a PE is in class 1 or 2. If diff=0, the PE is in class 1; otherwise, diff = $\Delta$ is the difference in indices of the x and y vectors between the PEs in class 1 and the PEs in class 2. Execution time is $p'+q'+1$ cycles to compute one y value, and $\lceil M/N \rceil(p'+q'+1)+((M-1)\text{mod}N)$ cycles to compute M y values. For large M, if $N = (p'+q'+1)/r$ for $r > 1$, then the throughput of the N-PE algorithm is reduced by approximately a factor of *r* from that of the $(p'+q'+1)$-PE algorithm. This ability to adapt the SIMD algorithm to different problem sizes means that a fixed set of PEs can be used to implement digital filters. Alternatively, on reconfigurable systems, in which it is possible to vary the number of PEs that act together as a virtual SIMD machine [e.g., Sieg81], it means that for a given digital filter, the virtual machine size can be tailored to the particular application. Fewer PEs may be chosen if speed requirements do not require the use of $p+q+1$ PEs. If, as will most often be the case, the filtering is one processing step in a sequence of algorithms, fewer than $p+q+1$ PEs may be chosen to make the digital filtering algorithm compatible with other SIMD algorithms to be applied as part of the complete task.

This method of adapting the SIMD digital filtering algorithm to fewer PEs also applies to the solution of general linear recurrence equations. The broadcast-based approach therefore provides a general method for using an SIMD system to solve linear recurrence equations of order p using p or fewer PEs.

In contrast to this flexibility in the SIMD implementation, VLSI processor array needs a major hardware modification (adding more registers to add additional coefficients and $y_m$ values) to handle a digital filter of larger size. It is generally easier to add more cells to the array than to modify the existing cells.

```
/*          ADDR      Address of PE (i.e., ADDR = 0 in PE 0)
            Δ         difference in indices between PEs in class 1 and 2
            diff      0 if PE is in class 1, Δ if PE is in class 2
            x[]       Input data (x[m] = 0 for m < 0)
                      (stored in each PE before start of algorithm)
            y[]       Output data (y[m] = 0 for m < 0)
            coef[]    Vector of coefficients (see Figure 6.6)
            flag[i]   Equals 1 if coef[i] is an "a" coefficient
            sum       Contains partially computed $y_m$ */

diff ← 0
c ← p+q+1
Δ ← c−N
sum ← 0
FOR m ← 1−N TO M−1 DO
            IF m mod N = 1 THEN/* Do steady state recurrences */
                                  /* i.e. no new $y_m$ is started */
                  FOR i ← m TO m+DELTA−1 DO
                        WHERE flag[i mod c] = 1 DO
                              tmp ← y[i−1−DELTA]
                        ELSEWHERE
                              tmp ← x[i+p−DELTA]
                        ENDWHERE

                        sum ← sum + tmp * coef[i mod c]

                  diff = Δ

            WHERE ADDR = m−1 mod N DO        /* a new $y_m$ is computed in
                                               PE m mod N */
                  DTRin ← sum              /* Send newly computed y value to */
                                           /* all PEs by placing it in the DTRin. */
                  sum ← 0                  /* Clear SUM for next $y_m$ */
                                           /* value to be computed in this PE. */
                  diff ← 0          /* When diff=0 in a given PE, */
                                           /* the given PE is in class 1 */
                  BROADCAST                /* Broadcast the SUM placed in DTRin */
                                           /* in line  to all PEs. */
            ENDWHERE
            y[m−1] ← DTRout                /* DTRout contains $y_{m-1}$ */
                                           /* in all PEs */
            WHERE flag[m mod c] = 1 DO /* If COEF[]s are "a" values, */
                                           /* load y values into TMP. */
                  tmp ← y[m−1−DELTA+diff]
            ELSEWHERE                      /* if COEF[]s are "b" values, */
                                           /* load x values into TMP. */
                  tmp ← x[m+p−DELTA+diff]
            ENDWHERE
            sum ← sum + tmp * coef[(m + diff) mod c]
```

Figure 6.9. SIMD digital filtering algorithm for $N < p+q+1$ PEs.

Therefore, a VLSI processor array of size p+q+1 cannot easily implement a larger problem size. In terms of flexibility to adapt to changing problem sizes, then, the SIMD system has the capability of handling varying problem sizes under software control. Adapting a VLSI processor array to a problem size different than that for which the array was designed requires hardware modification. For some computing environments, this difference in flexibility may be significant, and would dictate use of the possibly slower but more flexible SIMD system.

### 6.1.7. Summary of General Digital Filtering Algorithms

Synchronous parallel structures for implementing digital filters have been presented. Both VLSI processor arrays and SIMD implementations yield significant speedups over serial processing. The SIMD method provides a general approach to solving linear recurrence equations on an SIMD system. For a given application or environment, the choice of VLSI processor or SIMD structure depends on a number of factors. Although exact timing is implementation dependent, it is most likely that the VLSI processor array approach will be faster than the SIMD algorithms. System cost will also be less for the VLSI processor array. On the other hand, the SIMD system can accommodate changes in the order of the filter, whereas the VLSI processor array requires hardware modification to handle a change in problem size. Moreover, if the filtering is simply one step in a series of operations, no additional hardware is needed in the SIMD system. The data allocation resulting from the SIMD algorithm, where the output data is distributed across the PEs, is a useful allocation for a number of SIMD signal processing algorithms, including computation of autocorrelation and covariance coefficients [Si80b] and FFTs [SMS79]. The ability to run the SIMD algorithm on different machine sizes improves its potential compatibility with other SIMD algorithms which, together with digital filtering, comprise a complete signal processing task. Therefore, for a particular environment, speed requirements, cost, the importance of flexibility, and the context in which the algorithm is to be used may all be factors in selecting a parallel structure for digital filtering.

### 6.1.8. Parallel Preemphasis Filtering

Fortunately, the preemphasis filtering which is used before performing autocorrelation in a speech processing system is much simpler than the general digital filter. Figure 6.10 is the Flock Algol algorithm for implementing

$$H(z) = 1 - 0.95*z^{-1}.$$

The signal is broken up into frames containing N samples each where N is the number of PEs. Before execution, sample i of the input data is in PE i for $0 \leq i < N$. After execution, PE i contains output sample i for $0 \leq i < N$. The number of PEs used need not be equal to the number of samples per LPC frame (M). However, they are often the same since the autocorrelation algorithm which follows uses M sample frames with the same data arrangement as output by the filtering algorithm. Line 1 sets up the interconnection network for a Shift +1 transfer. Line 2 transfers the input data so that PE i contains sample i in *input* and sample i–1 in *tmp* for $1 \leq i < N$. PE 0 however has sample N–1 in *tmp* since the shift transfer wraps around. Lines 4-8 handle the wrap around from PE N–1 to PE 0 by saving the value in *tmp* in PE 0 for later and using the sample from the previous time the algorithm was used. This value was sample N–1 from the previous N samples, which is the value that is needed. The value that wraps around is saved in *oldvalue* until the next time the routine is called.

After line 8, PE i has both sample i and sample i–1, therefore the filter operation is easily performed by the operation in line 10.

The numbers to the right of the line numbers are the approximate execution times in $\mu$s for each statement. These are based on the program presented in Section 7.2. Since there are no loops, the time complexity is O(1).

```
                          /*
                                  Flock Algol program to do preemphasis filtering.
                                  H(z) = 1 - 0.96 * z ** -1

                                  input:          input data
                                  output:         filter output data
                                  tmp,tmp2:       temporary values
                          */

Line   Time in µs
1      3                  USE Shift +1
2      8                  TRANSFER input TO tmp
3
4      7                  WHERE ADDR = 0 DO          /* Get value from previous call */
5      0.5                   tmp2 ← tmp
6      1.5                   tmp  ← oldvalue     /* Switch tmp and oldvalue     */
7      1.5                   oldvalue ← tmp2
8      2                  ENDWHERE
9
10     12.75              output ← input + tmp * 0.95
```

Figure 6.10. Algorithm for preemphasis filtering. Left column is the execution time assuming an 8 MHz MC68000.
(See Section 7.2.)

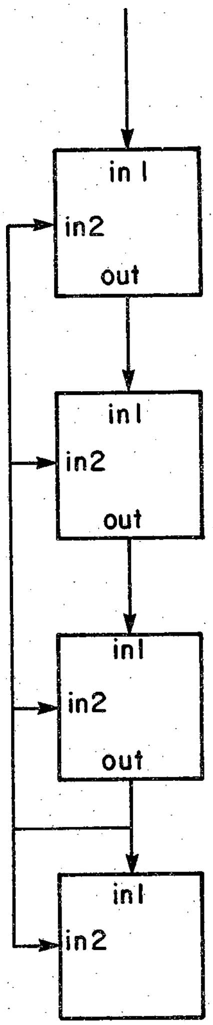
## 6.2. Autocorrelation Algorithms

Section 5.1 presented three SIMD algorithms for computing autocorrelation coefficients. This section presents another algorithm for the same task. It is a variation, with throughput improvement, of Ashajayanthi's [ASV79] SIMD machine autocorrelation algorithm. Ashajayanthi's algorithm (AUTO3) is presented in Figure 5.4 in Section 5.1.3. A direct mapping of it into a VLSI processor array results in the array in Figure 6.11 Each cell performs the operations shown in the figure with all the variables set to zero before the first sample enters cell 0. After sample M−1 enters cell 0, SUM in cell i contains R(p−i−1).

Figure 6.12 shows an improved version of this array (AUTO4). The array differs from AUTO3 in that the data entering *in1* in the top cell is also broadcast to *in2* in all cells. AUTO3, on the other hand, broadcasts the data entering *in1* in the bottom cell to *in2* in all cells. The cells in AUTO3 all do the same operation as the cells in AUTO4, with cell i computing the same operations as in Figure 6.11. All variables are set to zero before sample 0 enters cell 0, and cell i computes R(i) for $0 \leq i < p$. This is an improvement since Figure 6.11 requires p operations to get sample 0 into cell p−1, followed by M−1 operations to compute the coefficients. AUTO4 needs no initialization and requires M operations using the same cells as AUTO3.

### 6.2.1 Summary

Table 6.2 compares Ashajayanthi's algorithm (AUTO3) with the improved algorithm (AUTO4). Initialization times are included in the times in Table 6.2, but were omitted when computing the times in Table 5.2. AUTO4 is a faster algorithm than AUTO3 since it uses the same cells and does not require any initialization steps other than setting R to zero before sample 0 is computed.



$$R \leftarrow R + in1 * in2$$

$$out \leftarrow in1$$

Figure 6.11. Ashajayanthi's SIMD autocorrelation method [ASV79] mapped to a VLSI processor array.

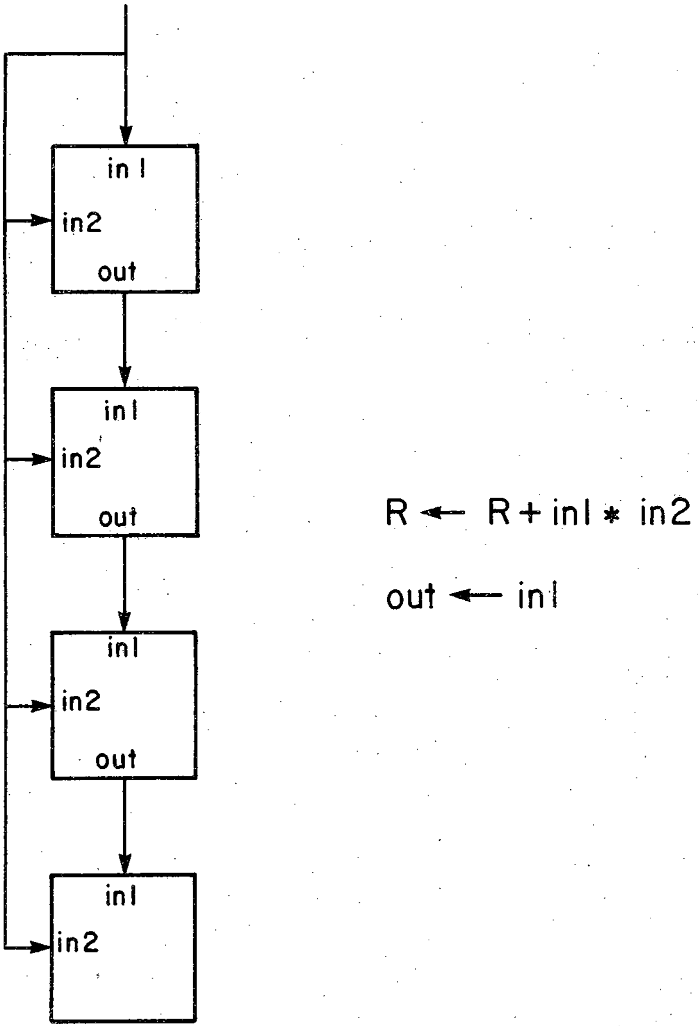


Figure 6.12 Improved VLSI processor array autocorrelation algorithm.

Table 6.2 Comparison between Ashajayanthi's SIMD autocorrelation algorithm (AUTO3) and an improved version (AUTO4).

| | PEs | Additions | Multiplications | Transfers | Broadcasts |
|---|---|---|---|---|---|
| AUTO3 | p+1 | M+p+1 | M+p+1 | M+p+1 | M+2p+2 |
| AUTO4 | p+1 | M | M | M | M+p+1 |

## 6.3. Linear Time Warp

The purpose of linear time warping (LTW) is to take an utterance R(j) for $0 \leq j < J$ and stretch or shrink it to an utterance T(i) for $0 \leq i < I$. Elements of R(j) and T(i) are vectors of LPC coefficients. The following equations show the relationship between R() and T().

$$T(i) = (1-s)*R(j) + s*R(j+1), \quad i=1,...,I \tag{6.3}$$

where

$$j = \left\lfloor (i-1)\,\frac{(J-1)}{(I-1)} + 1 \right\rfloor \tag{6.4}$$

$$s = (i-1)\,\frac{(J-1)}{(I-1)} + 1-j$$

One method to compute T(i) in parallel is to have PE i compute T(i) for $0 \leq i < I$. A second method is to compute the vector/scalar products (1–s)R(j) and sR(j+1) in parallel (i.e. have PE k compute element k of vector T(i)). The following sections discuss each method.

### 6.3.1. Method One

The algorithm in Figure 6.13 does a linear time warp from J frames to I frames on an SIMD machine. It uses equations 6.3 and 6.4 to warp R(j), $0 \leq j < J$, to T(i), $0 \leq i < I$. Each element of R(j) is a feature vector and R(j) for $0 \leq j < J$ is one utterance. The algorithm assumes R(j) is in PE j for $0 \leq j < J$. Method one has three cases, one where J<I, another where J=I, and finally where J>I. The following sections give examples for how the algorithm works when J<I and J>I. The J=I case is a simple copying operation as is not discussed here.

| | |
|---|---|
| Problem: | Take J samples in PEs 0 through J−1 and linearly warp them to I samples in PEs 0 through I−1. |
| Input: | The input frames R are stored with R[j] in PE j.<br>J is equal to the number of input frames.<br>I is equal to the number of output frame. |
| Output: | T[i] will contain the linearly warped output in PEs 0 through I−1. |

```
Line  Time in μs
 1    1.5          IF(I = J) THEN
 2    32.25             T ← R
 3    2                 RETURN
 4
 5    24.5         factor ← (J−1) / (I−1)
 6    26.24        i ← ⌈ADDR/factor⌉
 7
 8                 /*
 9                      If data is being expanded, move input data to
10                      cover all output PEs.
11                 */
12    2            IF(I > J) THEN
13    3                 USE Shift +1
14    3.5               FOR k ← 1 TO I − J
15    6.5                    WHERE(ADDR < i) DO
16    7.5                         TRANSFER i
17    127.5                       TRANSFER R
18    2                      ENDWHERE
19    0.5               i ← ADDR
20
21    11.25        tmp ← i * factor + 1
22    2.5          j ← ⌊tmp⌋
23    1.25         s ← tmp − j
24    3            USE Shift −1
25    96.5         TRANSFER R to R1
26    217          T ← (1−s) * R + s * R1
27
28                 /*
29                      Shift new T's down until only I PEs are occupied
30                 */
31    1.75         IF(I < J) THEN
32    3                 FOR k ← 1 TO J − I
33    7.5                    TRANSFER i TO i_tmp
34    6.5                    WHERE(i_tmp ≤ ADDR) DO
35    92                          TRANSFER T
36    0.5                         i ← i_tmp
37    2                      ENDWHERE
```

Figure 6.13. SIMD algorithm to do linear time warp. Numbers right of line number are the execution times assuming an 8 MHz 68000. (See Section 7.5.)

#### *6.3.1.1. An Example of Expanding J=5 Frames to I=7 Frames*

Suppose J=5 and I=7. Since J < I, the data is being expanded. Using equations (6.3) and (6.4) we have:

$$\begin{aligned}
T(1) &= R(1) \\
T(2) &= \frac{1}{3}R(1) + \frac{2}{3}R(2) \\
T(3) &= \frac{2}{3}R(2) + \frac{1}{3}R(3) \\
T(4) &= R(3) \\
T(5) &= \frac{1}{3}R(3) + \frac{2}{3}R(4) \\
T(6) &= \frac{2}{3}R(4) + \frac{1}{3}R(5) \\
T(7) &= R(5)
\end{aligned} \tag{6.5}$$

Line 6 computes i in each PE based on the PE's address. R(j) can be computed in PE k by using R in PE k and R in PE k+1. Figure 6.14 shows the PEs and their i values. Notice 1 and 4 are missing from the i column. Lines 12-18 shift the data so that T(i) can be computed in PEs 0 through 6. This is done by comparing ADDR to i. If ADDR < i, (as in PEs 2 through 5), i and R(i) are shifted from PE k to PE k+1. This happens I−J times as shown in Figure 6.13. Now i is assigned ADDR in PEs 0 through 6 and R is transferred from PE k to R1 in PE k−1 in line 25. Line 26 then does the computations of the equations in 6.5 in parallel, leaving T(i) in PE i for $0 \leq i < I$.

In general, if J < I, the R(j)'s are then shifted between the PEs until I PEs are used, and PE i contains the two R()'s needed to compute T(i).

#### *6.3.1.2. An Example of Compressing J=7 Frames to I=5 Frames*

Now suppose J=7 and I=5, then the following assignment must be made:

| | | | After one transfer | | After two transfers | | After line 25 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| PE | R | i | i | R | i | R | i | R1 | j | s |
| 0 | R(0) | 0 | 0 | R(0) | 0 | R(0) | 0 | R(0) | 1 | 0 |
| 1 | R(1) | 2 | 0 | R(0) | 0 | R(0) | 1 | R(1) | 1 | 2/3 |
| 2 | R(2) | 3 | 2 | R(1) | 2 | R(1) | 2 | R(2) | 2 | 1/3 |
| 3 | R(3) | 5 | 3 | R(2) | 3 | R(2) | 3 | R(2) | 3 | 0 |
| 4 | R(4) | 6 | 5 | R(3) | 3 | R(2) | 4 | R(3) | 3 | 2/3 |
| 5 | | | 6 | R(4) | 5 | R(3) | 5 | R(4) | 4 | 1/3 |
| 6 | | | | | 6 | R(4) | 6 | R(0) | 5 | 0 |

Figure 6.14. Data flow for LTW for expanding from J=5 to I=7 frames.

$$
\begin{aligned}
T(1) &= R(1) \\
T(2) &= \frac{1}{2}R(2) + \frac{1}{2}R(3) \\
T(3) &= R(4) \\
T(4) &= \frac{1}{2}R(5) + \frac{1}{2}R(6) \\
T(5) &= R(7)
\end{aligned}
\tag{6.6}
$$

This is done by the transfer of lines 24 and 25. Figure 6.15 shows the data in each PE after the transfer. The **boldface** values in the T columns indicate those PEs that are disabled after line 34. Recall that if a PE is disabled, it can pass data to other PEs, but other PEs cannot pass their data to it. Notice the equations in (6.6) can now be computed simultaneously, with PEs 2 and 5 computing values that are not needed ("junk" values). Lines 31-37 then shift the T(i) values so that T(i) is in PE i. Line 33 shifts the i values from PE k to i_tmp in PE k−1, then those PEs with ADDR ≥ i_tmp put i_tmp in i, and R gets the value of R in PE k+1. i is transferred to i_tmp before comparing to ADDR since a disabled PE cannot receive data. This, in effect, shifts good T(i) values over the junk values.

In general, if I > J, PE i computes T(i) and then the data is shifted so PE i contains T(i) for $0 \leq i < I$.

### *6.3.1.3. Time Complexity*

Table 6.3 summarizes the time complexity for the linear time warp algorithm. The total number of PEs required is the maximum of J and I. The 2N products and the I additions in equation (6.3) are all done in parallel by line 26 of Figure 6.13. The rest of the algorithm is for shifting data so that each R(j) and T(i) value is placed in the correct PE. Some of this shifting overhead may be reduced depending on the arrangement of the data in the algorithms before and after the linear time warp algorithm.

| | After Line 26 | | | | | | After first transfer | | | After second transfer | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PE | R | R1 | i | j | s | T | i_tmp | T | i | i_tmp | T | i |
| 0 | R(0) | R(1) | 0 | 1 | 0 | **T(0)** | 1 | **T(0)** | 0 | 1 | **T(0)** | 0 |
| 1 | R(1) | R(2) | 1 | 2 | 1/2 | **T(1)** | 2 | **T(1)** | 1 | 2 | **T(1)** | 1 |
| 2 | R(2) | R(3) | 2 | 4 | 0 | junk | 2 | **T(2)** | 2 | 3 | **T(2)** | 2 |
| 3 | R(3) | R(4) | 2 | 4 | 0 | T(2) | 3 | **T(3)** | 3 | 4 | **T(3)** | 3 |
| 4 | R(4) | R(5) | 3 | 5 | 1/2 | T(3) | 4 | junk | 4 | 4 | **T(4)** | 4 |
| 5 | R(5) | R(6) | 4 | 7 | 0 | junk | 4 | T(4) | 4 | 0 | T(0) | 0 |
| 6 | R(6) | R(0) | 4 | 7 | 0 | T(4) | 0 | T(0) | 0 | 0 | T(0) | 0 |

Figure 6.15. Data flow for compressing J=7 frames to I=5 frames. **Boldface** indicates PEs which are disabled after line 34 of the algorithm in Figure 6.13.

Table 6.3 Time complexities of linear time warping algorithms.

| Method | One | | Two | |
|---|---|---|---|---|
| | Scalar | Vector | Scalar | Vector |
| Number of PEs | max(J,I) | | p | |
| Additions | 5 | 1 | 2I+2 | 0 |
| Multiplications | 1 | 2 | 3I | 0 |
| Divisions | 2 | 0 | 1 | 0 |
| Transfers | $\lvert J-I \rvert$ | $\lvert J-I \rvert +1$ | 0 | 0 |

### 6.3.2. Method Two

The second approach to parallel linear time warping is to have PE k hold coefficient k of frame j for $0 \leq k < p$ and $0 \leq j < \max(J,I)$. Each vector/scalar multiplication is done in parallel. The algorithm is presented in Figure 6.16. The number to the right of the line numbers are the execution times in $\mu$s when implemented on an SIMD machine (see Section 7.5). The number of PEs (cells) used is $p$, the number of coefficients per frame. This algorithm can be implemented on both the SIMD machine and the VLSI processor array (see Section 8.5 for details on the VLSI processor array). The time complexity is summarized in Table 6.3.

### 6.3.3. Summary

Method two is an improvement over method one in that it uses fewer PEs (cells) and does not require vector operations. Method one requires fewer operations overall, and will therefore execute in less time. The final consideration in choosing between these two methods is the arrangement of the data among the PEs (cells). The algorithm commonly preceding the linear time warp will be the LPC algorithm. The SIMD LPC algorithm leaves the data in the PEs in an arrangement that method two can used directly. To use method one, the data must be rearranged, which might require more time than will be saved by using the faster method one.

```
Line  Time in μs
 1    1.75        IF($M = N$) THEN
 2    111                   $T \leftarrow R$
 3    2                     RETURN
 4
 5    23.5        $\text{factor} \leftarrow (M-1)/(N-1)$
 6
 7    2.75        FOR $n \leftarrow 0$ TO $N-1$
 8    11.25                 $\text{tmp} \leftarrow n * \text{factor} + 1$
 9    2.5                   $m \leftarrow \lfloor \text{tmp} \rfloor$
10    1.25                  $s \leftarrow \text{tmp} - m$
11    29                    $T \leftarrow (1-s) * R(m) + s * R(m+1)$
```

Figure 6.16. Algorithm for linear time warping using p PEs. Execution times are for an 8 MHz MC68000. (See Section 7.5).

## 6.4. Dynamic Time Warping

This section presents dynamic time warping algorithms for both the SIMD machine and the VLSI processor array. These algorithms have previously appeared in [YoSi82]. The SIMD algorithms assume that the feature vectors for the entire test word and all template feature vectors needed are stored in every PE memory. The PEs are complete processors, and a general interconnection network handles the needed inter-PE communications. The VLSI array algorithms assume that the cells have less memory, and that fast, fixed inter-PE transfers are a part of the system architecture. In these algorithms, the feature vectors shift from one cell to the next, and the computations are performed in a pipelined fashion.

### 6.4.1 SIMD Algorithms

This section presents two approaches to performing DTW on an SIMD machine. Both assume that the speech recognizer must compare the test template to W reference templates, and each PE contains complete test and reference templates. The serial-parallel approach uses up to W PEs in parallel with each PE doing a serial DTW using a different reference template. The parallel-parallel approach uses many PEs in parallel for each DTW match of the test template with a reference template.

#### *6.4.1.1. Serial-Parallel (SP) SIMD Approach*

A recognizer with a vocabulary of W templates can be implemented on a processor with $N \leq W$ PEs. If $W = N$, then PE w contains template w, $0 \leq w < W$, from the vocabulary, so that every PE contains a different template. Each PE performs a serial DTW between its stored template and the input X. Recursive doubling [Ston80] is used to find the PE containing the smallest distance, in log N time, which represents the template most closely matching the input. See Section 2.6 for an example of recursive doubling.

All DTW algorithms compute the following steps:

1) computing the local distance d(i,j);
2) the two multiplications and four additions in equation (4.7); and
3) two comparisons to find the minimum of three values.

These three steps are defined as one *loop* as discussed in Section 4.5.2. A serial DTW algorithm requires $W(2Ir-I-r^2+r)$ loops to compute W D(A,B)s with the adjustment window r, and $WI^2$ loops without. This does not take into account the possible time saved by pruning. The same algorithm on an SIMD machine with N = W PEs requires $(2Ir-I-r^2+r)$ loops with the adjustment window, and $I^2$ without. This is an ideal speedup (i.e. by a factor of N) over the serial processor. However, if the serial processor uses pruning, the parallel approach will attain a less than ideal speedup. At least one comparison (the minimum distance match) is not pruned, so the time for the SIMD algorithm is not reduced by pruning. Since the time of the serial algorithm may be reduced by pruning, the SP algorithm will no longer attain a factor of N speedup. If $W > N$, (the vocabulary is larger than the number of PEs) then the SP algorithm can be run $\lceil W/N \rceil$ times to match all words. See Table 6.4 for a summary of these results.

#### *6.4.1.2. Parallel-Parallel (PP) SIMD Approach*

Two drawbacks to the SP approach are that pruning will not reduce the computation time unless all PEs can prune at the same time, and that there is no effective way to use $N > W$ PEs. In the parallel-parallel approach each DTW match uses several PEs. Equation (4.7) shows that g(i−2,j−1) and g(i−1,j−2) must be computed before computing g(i,j). The g(i,j)'s that can be computed in parallel are all g(i,j)s for $i+j=2k$ and $i+j=2k+1$, for a fixed value of k between 1 and I inclusive. If g(k) is defined as all g(i,j) with $i+j=2k$ and $i+j=2k+1$, all g(i,j)s in g(k) can be computed in parallel. These g(k) depend only on g(m) for $m < k$. Figure 6.17 shows two diagonal rows that represent a typical group of g(i,j) in a given g(k); the g(m) for $m < k$ are "down" and "to the left" of the diagonal rows. Each g(k) contains at most $2r+1$ points when using an adjustment window of size r. If no adjustment

Table 6.4. Summary of Parallel Dynamic Time Warping Algorithms.

| Algorithm | Adjustment Window | Number of PEs | ΔPEs per Word | Loops for 1 Word | Number of Loops for W Words | Operations per Loop |
|---|---|---|---|---|---|---|
| Serial | no | 1 | 0 | $I^2$ | $WI^2$ | 1ld,2m, |
|  | yes | 1 | 0 | K | WK | 4a,2c |
| SP | no | >1 | 1 | $I^2$ | $\lceil W/N \rceil I^2$ | 1ld,2m |
|  | yes | >1 | 1 | K | $\lceil W/N \rceil K$ | 4a,c2 |
| PP | no | $\geq 2I-1$ | 2I–1 | I | $I\left\lceil \frac{W}{\lfloor (N)/(2I-1) \rfloor} \right\rceil$ | 1ld,2m, |
|  | yes | $\geq 2r+1$ | 2r+1 | I | $I\left\lceil \frac{W}{\lfloor (N)/(2r+1) \rfloor} \right\rceil$ | 4a,2c,4t |
| HSAC | no | $I^2$ | $I^2$ | 2I–1 | $2I-1+\left\lceil \frac{W-1}{\lfloor N/I^2 \rfloor} \right\rceil$ | 1ld,2m,4a, |
| (full) | yes | K | K | 2I–1 | $2I-1+\left\lceil \frac{W-1}{\lfloor N/K \rfloor} \right\rceil$ | 2c,sv,4ss |
| HSAC | no | I | I | 2I–1 | $(2I-1)\left\lceil \frac{W}{\lfloor N/I \rfloor} \right\rceil$ | 1ld,2m,4a, |
| (reduced) | yes | r+1 | r+1 | 2I–1 | $(2I-1)\left\lceil \frac{W}{\lfloor N/(r+1) \rfloor} \right\rceil$ | 2c,sv,4ss |
| BAC | no | 2I–1 | 2I–1 | $I+\left\lceil \frac{I}{2} \right\rceil$ | $I\left\lceil \frac{W}{\lfloor N/(2I-1) \rfloor} \right\rceil + \lceil I/2 \rceil$ | 1ld,2m,4a, |
|  | yes | 2r+1 | 2r+1 | $I+\left\lceil \frac{r}{2} \right\rceil$ | $I\left\lceil \frac{W}{\lfloor N/(2r+1) \rfloor} \right\rceil + \lceil r/2 \rceil$ | 2c,sv,4ss |

| | | | |
|---|---|---|---|
| SP: | Serial Parallel algorithm | ld: | local distance calculation |
| PP: | Parallel Parallel algorithm | m: | multiplication |
| HSAC: | High Speed Array Computer | a: | addition |
| BAC: | Bilinear Array Computer | c: | comparison |
| N: | number of PEs used | sv: | shift vector through pipe to adjacent PE |
| K: | $2Ir-I-r^2+r$ | ss: | shift scalar through pipe to adjacent PE |
| t: | transfer through SIMD interconnection network | | |

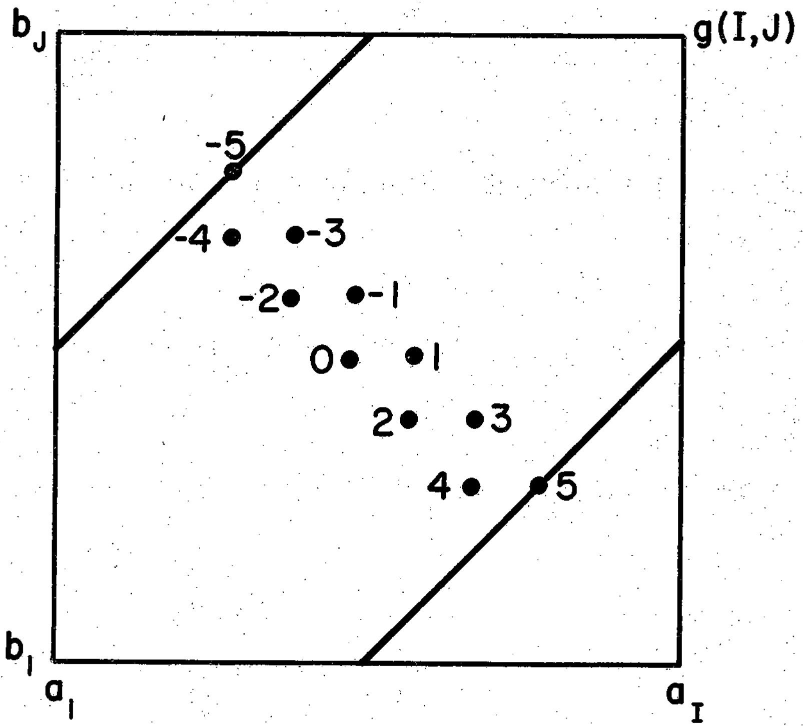


Figure 6.17. A set of g(i,j) that can be computed in parallel, labeled with PE numbers.

window is used, each g(k) has a maximum of 2I−1 points. Figure 6.18 shows the PP algorithm. The PEs are numbered −r,−(r−1),...,−2,−1,0,1,2,...,r−1,r,* and PE n computes g(i,j) for (i=k+n/2, j=k−n/2) for n even, and (i=k+(n+1)/2, j=k−(n−1)/2) for n odd. Figure 6.19 is a data flow diagram for a lines 13-55 of the PP algorithm with each box showing which g(i,j) the given PE is computing and each column of boxs showing the contents of all PEs during a given loop in the algorithm. The arrows between PEs represent the data transfers with the *g* transfers as solid lines and the *d* transfers as dashed lines. The odd (even) numbered PEs correspond to the PEs in the top (bottom) row in Figure 6.17. This assumes that the feature vectors $\underline{a}_i$ and $\underline{b}_j$ are stored in the appropriate PEs before the start of the algorithm. Figure 6.20 is a data flow diagram for the PP algorithm. Each row of boxes indicates which g(i,j) a given PE is computing during each loop of the algorithm. Each column shows which g(i,j)s are computed in parallel for a given k value. A total of 2r+1 PEs per template are needed. If the SIMD machine has N PEs, $\lfloor N/(2r+1) \rfloor$ templates can be matched in I parallel loops, requiring $\left\lceil \frac{W}{\lfloor N/(2r+1) \rfloor} \right\rceil I$ loops for a W template vocabulary. Both the SP and PP methods yield a speedup over the serial algorithm. The following section discusses a parallel DTW algorithm for the VLSI processor array. The section after that compares all the parallel DTW algorithm to each other.

### 6.4.2. VLSI Processor Array Algorithms

Burr, Weste, and Ackland [BAW81,BAW84,WBA83] have presented a high speed array computer (HSAC) in which an I by I grid of cells compares several vocabulary templates to the input template simultaneously. They also presented reduced arrays which can use as few as r+1 cells to "sweep out" the I by I grid. The complexity analysis of the HSAC was presented in Section 5.6. The next section presents a bilinear VLSI array algorithm which incorporates some of the strategy used in the PP SIMD algorithm with the reduced arrays

* If no adjustment window is used, the PEs are numbered −(I−1),−(I−2),...,−1,0,1,...,I−2,I−1).

```
/*
                Algorithm Name:   dtw.s (parallel)
                Section:          6.4.1.2.
                Machine:          SIMD
                Function:         This program performs a dynamic time warp.
                Number of PEs:    2r+1 or 2I−1
                Parameters:       r, the width of the warping path.
                                  p, the number of coefficients per frame.
                                  NetD, the network delay time.
                                  I, the number of frames per utterance.
                Input:            All PEs hold all the input data.
                Output:           PE r holds the distance score.
*/

Time in μs
                PROCEDURE dtw
2               g ← 0
2               gold ← 0
1.5             d ← ∞
1.5             dold ← ∞
28              WHERE ADDR = 0 DO
2                   g ← 0
2               ENDWHERE

4.75            Xindex ← +⌈ADDR/2⌉
4.75            Yindex ←−⌊ADDR/2⌋

1               FOR k ← 1 TO I DO
124                 compute d(Xindex,Yindex)
10.5                WHERE ADDR is even DO
2.5                     dDTR ← dold
2.5                     gDTR ← gold
8                   ELSEWHERE
2.5                     dDTR ← d
2.5                     gDTR ← g
8                   ENDWHERE

3                   USE Shift +1
5+NetD              TRANSFER dDTR TO dup
5+NetD              TRANSFER gDTR TO gup

3                   USE Shift −1
5+NetD              TRANSFER dDTR TO ddown
5+NetD              TRANSFER gDTR TO gdown

7                   WHERE ADDR = r DO
1.5                     gdown ← ∞
2                   ENDWHERE
```

Figure 6.18. Parallel DTW program. Execution times are for an 8 MHz MC68000. (See Section 7.6.)

```

7                WHERE ADDR = -r DO
1.5                  gup ← ∞
2                ENDWHERE

2.5              gold ← g
2.5              dold ← d

4                A ← gdown + 2 * ddown
3                B ← gold  + d
4                C ← gup   + 2 * dup

6.5              WHERE B < A DO
.5                   A ← B
2                ENDWHERE
6.5              WHERE C < A DO
.5                   A ← C
2                ENDWHERE
3                g ← A + d

                 Xindex ← Xindex + 1
                 Yindex ← Yindex + 1

7            WHERE ADDR = 0 DO
2                D(A,B) ← g/(I+J)
2            ENDWHERE
```

Figure 6.18. (Continued)

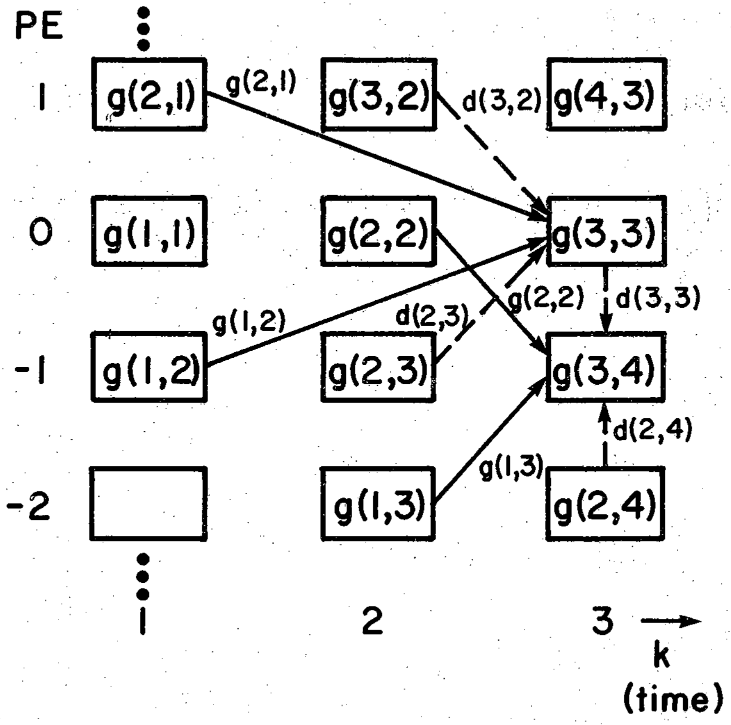
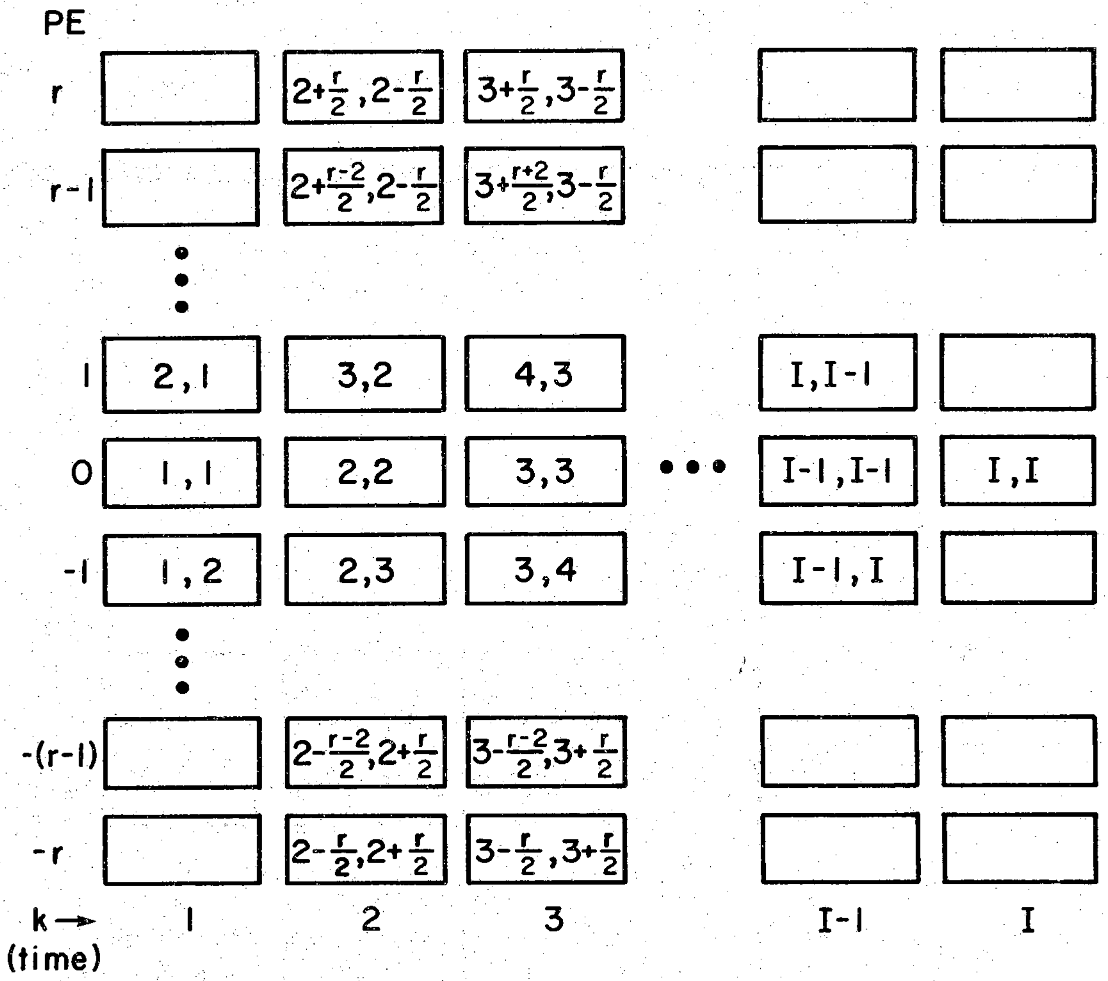


Figure 6.19. Data transfers into even and odd numbered PEs in PP algorithm.



| PE | | | | | | |
|---|---|---|---|---|---|---|
| r | | $2+\frac{r}{2},2-\frac{r}{2}$ | $3+\frac{r}{2},3-\frac{r}{2}$ | | | |
| r-1 | | $2+\frac{r-2}{2},2-\frac{r}{2}$ | $3+\frac{r+2}{2},3-\frac{r}{2}$ | | | |
| | ⋮ | | | | | |
| 1 | 2,1 | 3,2 | 4,3 | | I,I-1 | |
| 0 | 1,1 | 2,2 | 3,3 | ••• | I-1,I-1 | I,I |
| -1 | 1,2 | 2,3 | 3,4 | | I-1,I | |
| | ⋮ | | | | | |
| -(r-1) | | $2-\frac{r-2}{2},2+\frac{r}{2}$ | $3-\frac{r-2}{2},3+\frac{r}{2}$ | | | |
| -r | | $2-\frac{r}{2},2+\frac{r}{2}$ | $3-\frac{r}{2},3+\frac{r}{2}$ | | | |
| k → (time) | 1 | 2 | 3 | | I-1 | I |

Figure 6.20 g(i,j) computations in PP algorithm with r even.

into a VLSI array structure. This work was reported in [YoSi82] and was developed independently of the HSAC reduced array [WBA83,BAW84].

### *6.4.2.1. A Bilinear Array Computer (BAC)*

In general, the single diagonal HSAC uses r+1 cells per DTW comparison. Due to the interdependences discussed in the previous section, it can use no more than r+1 cells per DTW for general path restriction. The bilinear array computer (BAC) presented here restricts the path leading to a given point on the warping graph so that:

$$g(i,j) = d(i,j) + \min \begin{bmatrix} g(i-1,j-2) + 2d(i,j-1) \\ g(i-1,j-1) + d(i,j) \\ g(i-2,j-1) + 2d(i-1,j) \end{bmatrix}$$

$$g(1,1) = 2d(1,1).$$

Because of this restriction, the BAC uses 2r+1 cells per comparison, which results in it requiring half as many loops as the single diagonal HSAC. The single diagonal HSAC uses enough cells to compute one diagonal in Figure 6.17. The BAC uses enough cells to compute two diagonals of points for g(k) shown in Figure 6.17. Figure 6.21a shows the cells are arranged in a bilinear array with the cells in the left column computing the g(i,j)s for the lower diagonal, and the right column for the upper diagonal. Figure 6.21b shows the data paths between adjacent cells. DTtop and DTbot are Data Transfer registers. Storing a value in DTtop in cell i will transfer that value to DTbot in cell i+1. In general, the feature vectors $\underline{a}_i$ and $\underline{b}_i$ are piped in from opposite ends at the rate of one vector every loop. When $\underline{a}_i$ meets $\underline{b}_j$ in cell i−j, it computes d(i,j) and g(i,j) and sends them to cells i−j+1 and i−j−1. On the next loop, $\underline{a}_i$ and $\underline{b}_{j+1}$ meet in cell i−j+1 and it computes d(i,j+1) and g(i,j+1) and sends them to cells i−j+2 and i−j. Figure 6.22 shows the data flow as a function of time. Figure 6.23 shows the instructions executed by each group of cells if I is odd. If I is even, the even cells execute the group B instructions and the odd cells execute the group A. The instruction "a vector down" means to transfer the "a" vector from cell i to cell i−2 for −(I−2) < i < I−2 and transfer in a new "a" vector into both cell I−1 and cell I−2. The instruction "b vector up" is similar to "a vector down" but for the "b" feature vector.

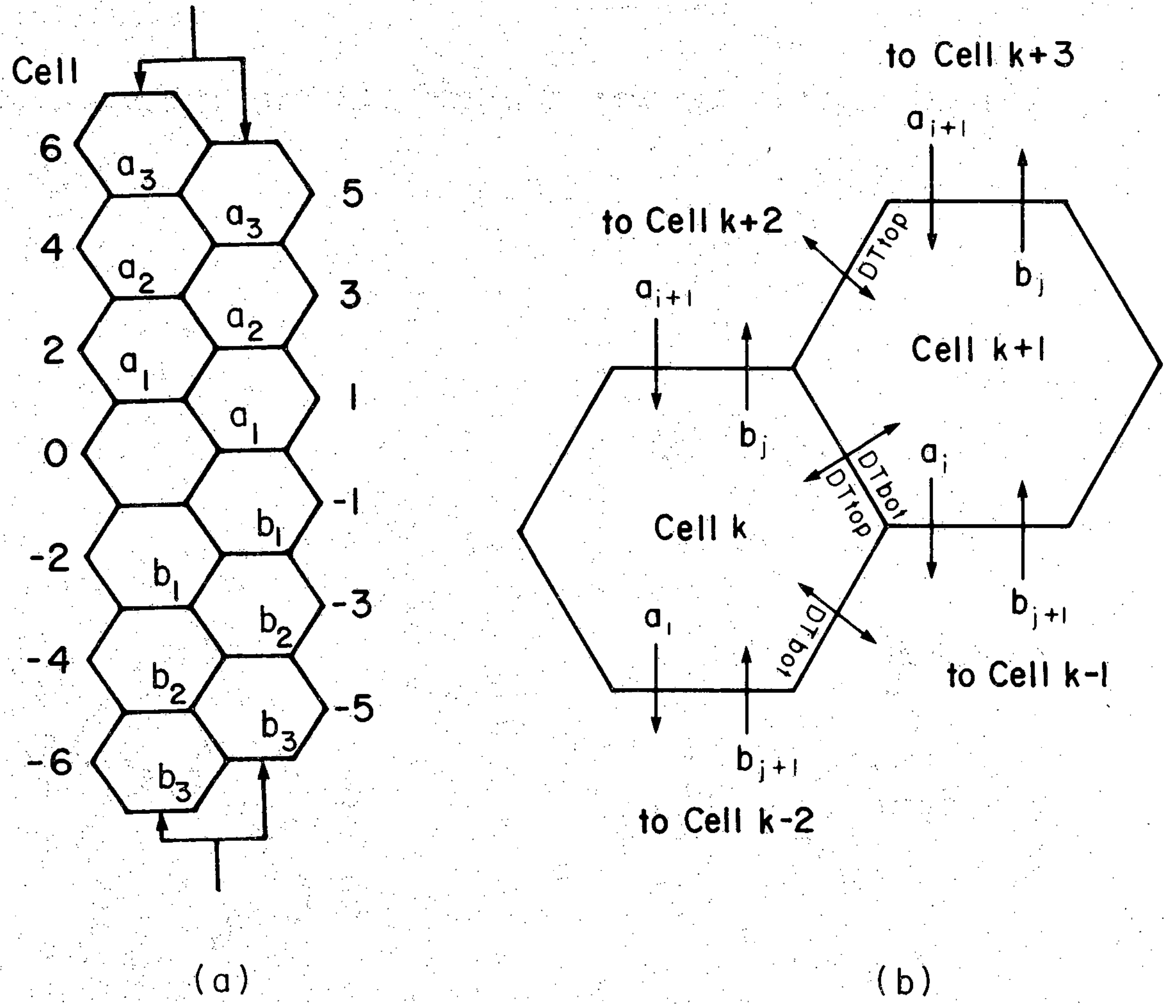


Figure 6.21 a) Bilinear array of cells. b) Data paths between cells in left and right columns.

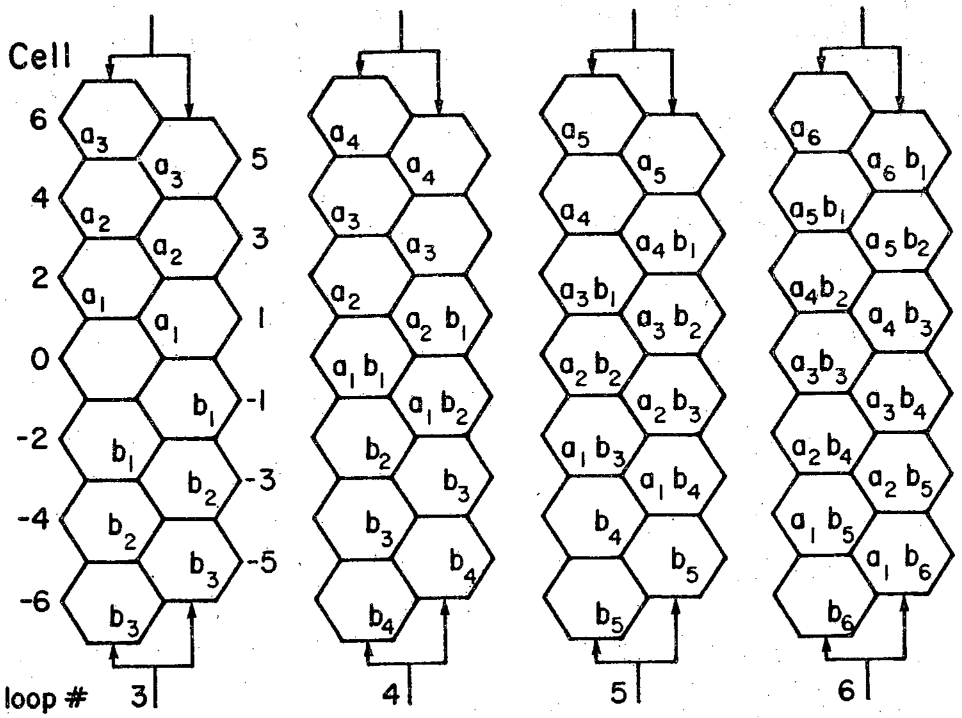


Figure 6.22 Data flow in BAC algorithm.

| Even numbered cells<br>Group A | Odd numbered cells<br>Group B |
|---|---|
| a vector down | a vector down |
| b vector up | b vector up |
| compute d | compute d |
| DTtop ← d | d.bot ← DTbot |
| DTbot ← d | d.top ← DTtop |
| $\mathrm{g} \leftarrow \mathrm{d}+\min\begin{bmatrix} \mathrm{g.bot.old}+2\mathrm{d.bot} \\ \mathrm{g}+\mathrm{d} \\ \mathrm{g.top.old}+2\mathrm{d.top} \end{bmatrix}$ | $\mathrm{g} \leftarrow \mathrm{d}+\min\begin{bmatrix} \mathrm{g.bot}+2\mathrm{d.bot} \\ \mathrm{g}+\mathrm{d} \\ \mathrm{g.top}+2\mathrm{d.top} \end{bmatrix}$ |
| g.top.old←g.top | |
| g.bot.old ← g.bot | |
| g.top ← DTtop | DTbot ← g |
| g.bot ← DTbot | DTtop ← g |
| d.bot ← DTbot | DTtop ← d |
| d.top ← DTtop | DTbot ← d |
| DTtop ← g | g.bot ← DTbot |
| DTbot ← g | g.top ← DTtop |

Figure 6.23. Instructions executed during one loop of the BAC algorithm for I odd. (Exchange columns for I even).

This array computes only one DTW at a time so its throughput is less than the full array HSAC, but it uses twice as many cells as the single diagonal reduced array, so it takes half as long for a comparison. If the BAC requires n cells, and N>n cells are available, $\lfloor N/n \rfloor$ arrays can be built, and $\lfloor N/n \rfloor$ DTWs can be computed simultaneously. The time to compute one DTW is the number of loops from the time $\underline{a}_1$ enters the array until $\underline{a}_I$ enters cell 0. This time is $\lceil I/2 \rceil$ loops to get the first $\underline{a}_1,\underline{b}_1$ pair to cell 0, and I loops until $\underline{a}_I,\underline{b}_I$ arrive at cell 0, giving a total time of $I+\lceil I/2 \rceil$ loops. The $\underline{a}_1,\underline{b}_1$ values for the second template follow the $\underline{a}_I,\underline{b}_I$ values of the first template, so the initial $\lceil I/2 \rceil$ loops used to get $\underline{a}_1,\underline{b}_1$ into cell 0 are not needed for the DTWs that follow. With an adjustment window r, the algorithm needs only 2r+1 cells and r+2I loops.

### 6.4.3. Summary of Results

Table 6.4 summarizes the above results. The column labeled "Number of PEs" lists the minimum number of PEs (cells) needed to use the algorithm. The ΔPE column is the number of PEs (cells) to be added to do another match in parallel. The fifth and sixth columns list the number of loops needed to do one match and W matches. The last column shows the operations done during one loop.

The serial and SP algorithms require the same operations per loop. The PP algorithm requires inter-PE transfers of the *d* and *g* values, which may increase the total loop time. Based on proposed general interconnection networks (e.g., [SiMc81a,SiMc81b]), the transfer time will be negligible compared to the time to compute the local distances. Depending on the implementation, it may be possible to overlap the transfers with the computations, so that little or no extra time is incurred. The loop times for the HSAC and the BAC will be about equal. The operation counts for the SIMD and array algorithms differ significantly; however, time differences will depend on specific implementations. The predominant difference in operation counts arise because the serial and SIMD algorithms assume each PE contains the feature vector before the algorithm starts, whereas the VLSI array algorithms require shifts to bring the test and vocabulary vectors into the cells. The A and B vector shifts occur simultaneously, so the time required is for the transfer of one feature vector. The times to transfer *d* and *g* values may also differ, since the PP algorithm uses a general interconnection network, whereas the VLSI array uses a less general (but most likely faster) fixed pipeline between adjacent cells. If transfer and computation steps can be overlapped, the loop times will be approximately equal, in spite of differences in the operations counts. Figures 6.24 and 6.25 show two plots of the number of loops needed to match W=100 words of length I=40 vs. the number of PEs (cells) with and without an adjustment window. Figures 6.26 and 6.27 are the same as 6.24 and 6.25 except for W=1,000. In Figure 6.24 the BAC and RHSAC lines are plotted almost on top of each other. In Figure 6.25 the BAC, PP, and RHSAC lines are almost one top of each other, with the RHSAC requiring fewer loops in the 1 to 128 PE range. In Figure 6.26 all but the SP are plotted almost on top of each other. The SP requires fewer loops than the other algorithms when using 1 to about 384 PEs, and around 500 PEs, and around 1,000 PEs. In Figure 6.27, the BAC and PP lines are plotted exactly on top of each other, and the RHSAC is plotted slightly below the BAC and PP lines for certain numbers of PEs.

Figure 6.24 shows that the BAC takes a few more loops than the PP algorithm since it requires a few loops to initialize the array which the PP algorithm does not need. The figure also shows that the BAC algorithm requires fewer loops than the HSAC with 544 cells. Since the operations per loop are equivalent, the BAC will therefore be slightly faster. This speed is attained by reducing the number of idle cells. In the BAC, no cells are idle after $\lceil I/2 \rceil$ loops, while the HSAC requires 2I loops before all cells are in use. The PP and BAC algorithms can continue to reduce execution time by adding more PEs (cells), so for these algorithms/architectures, the machine size can be chosen to meet speed requirements.

Figures 6.26 and 6.27 show that when the vocabulary size is increased to 1,000 words, the SP program clearly requires the fewest loops. This is because each cell is executing a serial DTW program which has little overhead of parallelism.

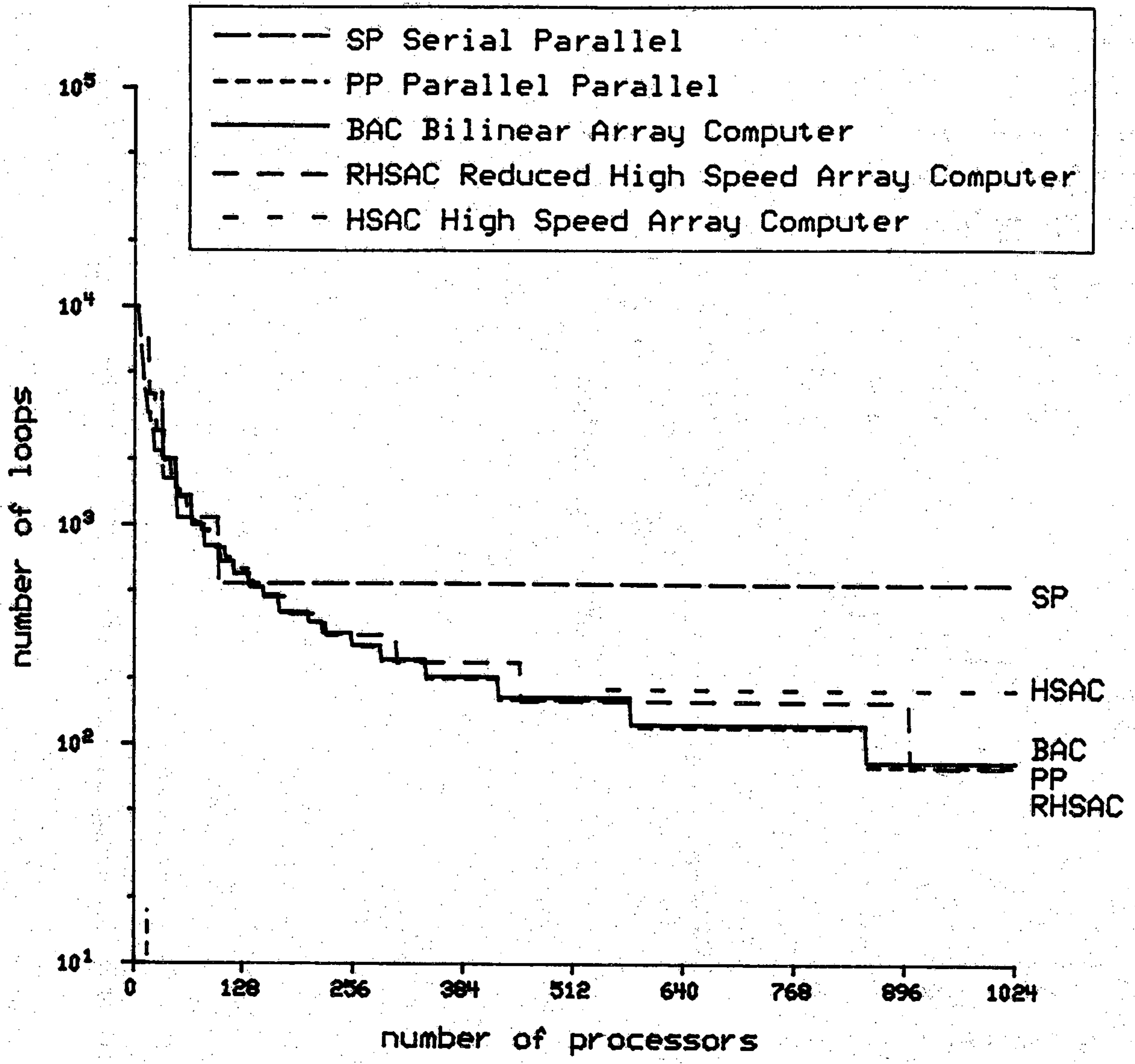


Figure 6.24 Number of loops for W=100, I=40, r=8.

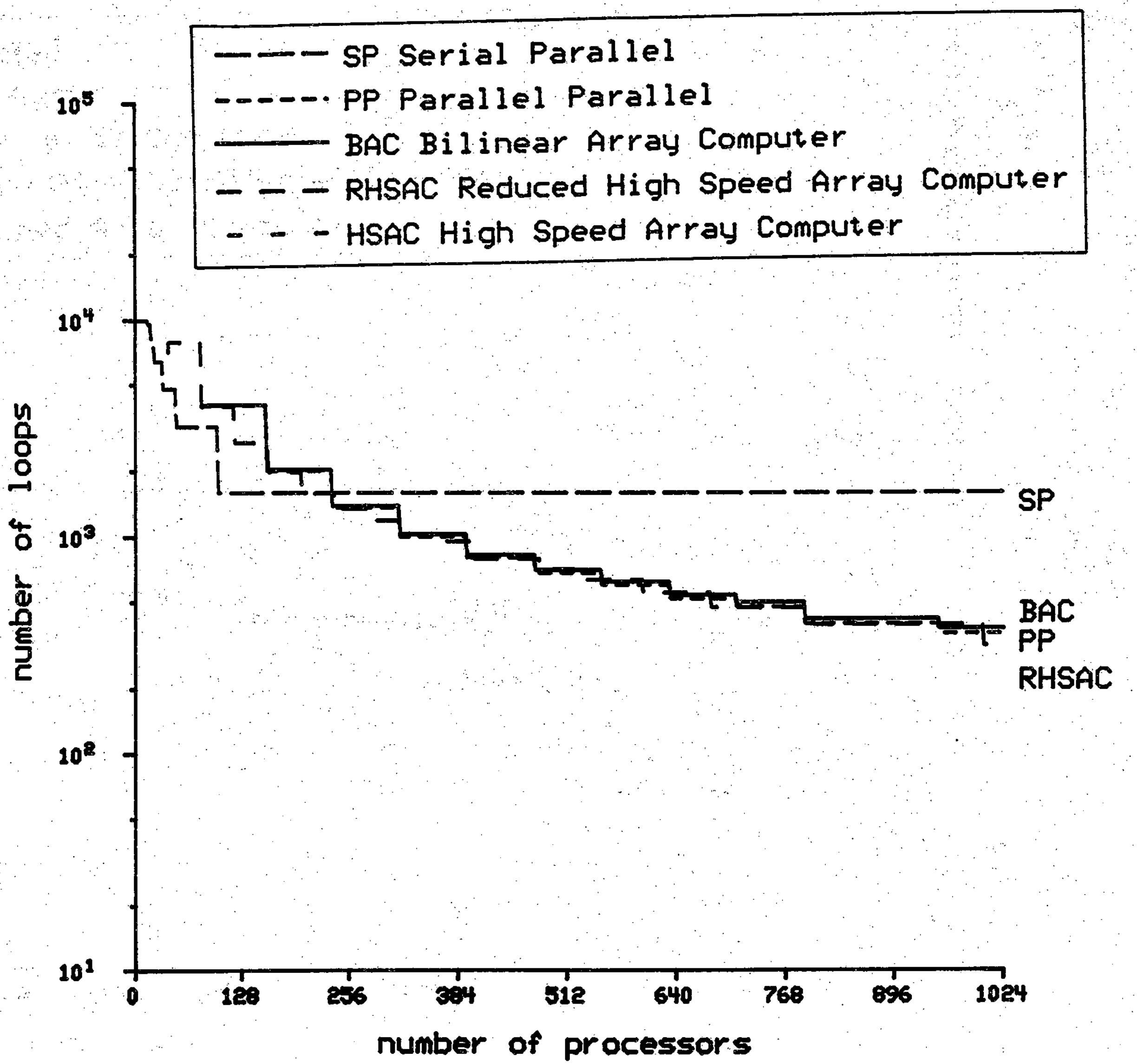


Figure 6.25. Number of loops for W=100, I=40, no window. HSAC not shown, since 1,600 PEs required.

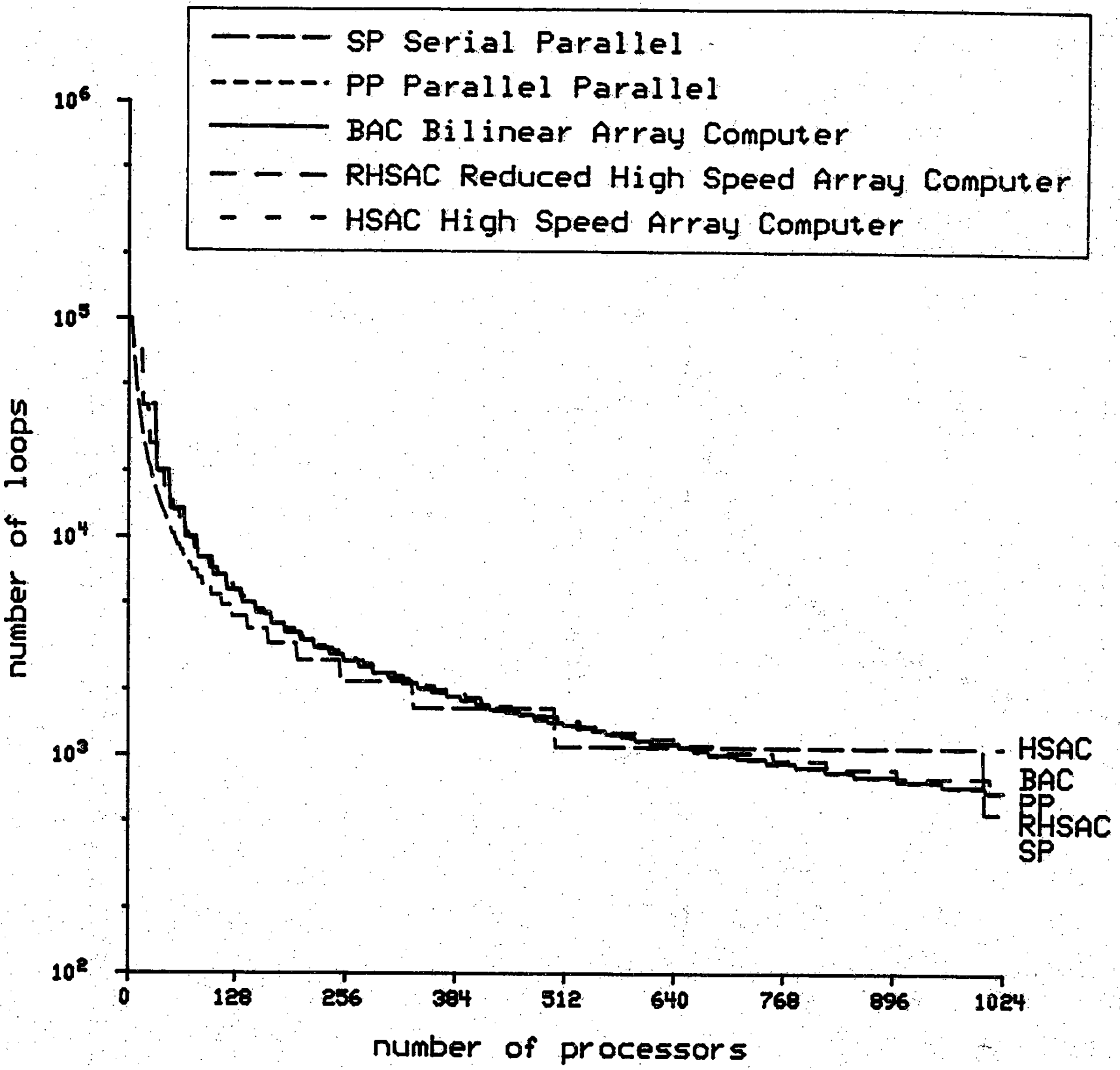


Figure 6.26. Number of loops for W=1,000, I=40, r=8.

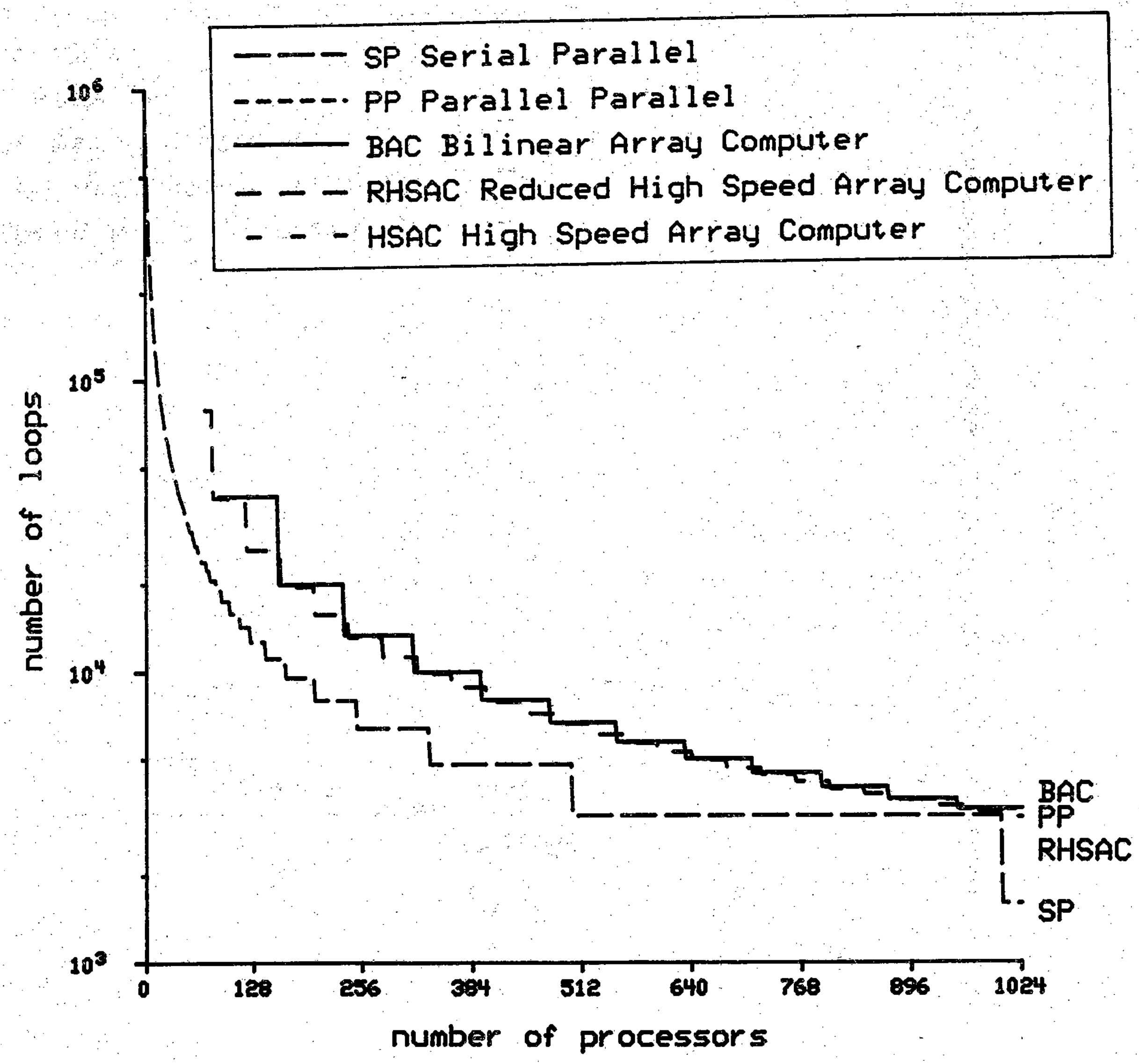


Figure 6.27. Number of loops for W=1,000, I=40, no window. HSAC not shown, since 1,600 PEs required.

## 6.5. Conclusions

Five parallel digital filtering algorithms, an autocorrelation, a linear time warp, and three parallel dynamic time warping algorithms were discussed. To choose the best algorithm, one must consider the need for flexibility, the type of processor used (PEs for SIMD or cells for the VLSI array) available. Also, when using the DTW algorithms the use of pruning and an adjustment window must be considered. The VLSI array algorithms are best suited for a dedicated task since the inter-cell connections are not easily changed. The SIMD interconnection and PEs are more general and could therefore be used to perform other tasks in a recognition system. All the algorithms provide significant speedups for these computationally intensive tasks.

# 7. SIMD MACHINE SIMULATION

This chapter presents the results of simulating many of the SIMD machine algorithms presented in the previous chapters. Section 7.1 describes the *sim68* simulator that is used to run the simulations. These simulations allow the operations of the algorithms to be verified and also give an idea about the execution times of each algorithm assuming the use of current technology processors. The Sections 7.2 through 7.6 present the results of simulating some of the SIMD algorithms from Chapters 5 and 6. Each algorithm is presented as an individual program in these sections and Section 7.7 combines some of the programs into an SIMD machine based isolated word recognition system. This system can process input data sampled at 20 KHz and recognize a 1,000 word vocabulary in real time. Section 7.8 discusses the strengths and weaknesses of using an SIMD architecture for speech processing and suggests improvements to the architecture.

## 7.1. Simulating an SIMD Machine Using Sim68

The *sim68* program performs an assembly language instruction level simulation of an SIMD machine [SiKu82]. All *sim68* programs are written in MC68000 assembly language with the aid of many support programs such as a parallel assembler and loader. The following sections describe the different parts of the SIMD model from Chapter 2 that are simulated.

### 7.1.1. Simulating the PEs and the CU

*Sim68* simulates the PEs and the CU in the SIMD machine as MC68000 microprocessors. The MC68000 is a state-of-the-art 16-bit microprocessor [ToGu81,Mot79], and reasons for its selection are discussed in [SiKu82]. Among these reasons are:

1) It can operate on a variety of data sizes: bit, byte, word (16-bits), and long (32-bits).

2) It has a fast cycle time: from 8 to 12.5 MHz.

3) It has a large address space: 24-bits.

4) It has a regular instruction set. See Figure A.1 in Appendix A.

It has been shown in [SiKu82] that the execution of CU and PEs instructions can overlap by using an instruction queue between them. This overlap can result in a reduction in processing time. *Sim68,* however, assumes that there is no overlap and no delay time for broadcasting instructions to the PEs. Therefore, either the CU is executing an instruction, or the PEs are, but never both at the same time. This assumption means that the execution times given are conservative and might be reduced if an instruction queue were used.

#### *7.1.1.1. The MC68000 Parallel Assembler*

All programming for *sim68* is done in MC68000 assembly language. The parallel assembler used is called *pa68. Pa68* is loosely based on the Digital Equipment Corporation Macro 11 assembler [Dec]. The major differences between *pa68* and a typical serial assembler are:

1) Instructions executed by the CU begin with a "c_", while PE instructions start with a "p_".

2) Instructions opcodes may end with a *.b, .w,* or an *.l* depending on whether the data operated on is *b*yte (8-bits), *w*ord (16-bits), or *l*ong (32-bits).

3) The *.word* directive is used to define data in the CU and the PEs. When defining data in the PEs, argument i of the ***word*** directive is stored in PE i−1. Therefore

```
.word    10,11,12,13
```

would store the value 10 in PE 0, 11 in PE 1, and so on.

4) Instructions starting with a capital letter such as *Where(d0,EQ,d1)* and *Shift(d1)* are macros defined to simulate the functions with the same name in Flock Algol. These are discussed in Section 7.1.2.

5) Unlike some assemblers, the opcode is followed by the source operand, which is followed by the destination operand as defined in [Mot79]. Therefore, *p_mov.w d0,d1* moves the data in register d0 to d1 in all active PEs.

6) The *c_cmp* instruction compares the destination to the source, so if the instructions

```
c_cmp.w   d0,d1
c_blt label
```

are executed, the branch to *label* will occur if d1 is less than d0. This is the reverse of the normal convention. Note that *p_blt* does nothing since the CU must perform all the branching instructions.

Figure 7.1 is a sample listing of a Flock Algol algorithm. It is presented here as an example, and the details of its operation will be discussed in Section 7.2. It shows some of the features of Flock Algol and the conventions that will be used here in presenting algorithms and programs. The left most numbers in Figure 7.1 are the line numbers, while the next number on the line is the execution time, in $\mu$s, of the statement running on an 8 MHz MC68000.

The block of comments before the first numbered line is a standard header that appears before each major program. Each section of the header is described in the following list.

*Program Name* gives the name of the program. This is sometimes referred to if there are several programs that perform the same function.

*Algorithm* will give the figure number of the corresponding Flock Algol code if the program is an assembly language program. The Flock Algol listing will give the figure number of the algorithm it is implementing.

*Machine* will be the SIMD machine.

*Function* will give a brief description of what the program does.

*Precision* lists the number of bits and format for the input, output, and any

```
/*
                    Program name:     filter
                    Section:          6.1.8.
                    Machine:          SIMD
                    Function:         This program preemphasises the
                                      input speech data with a filter
                                      with the transfer function:
                                      H(z) = 1 − coef * z^-1
                    Number of PEs:    N
                    Parameters:       coef, The filter coef. (default = 0.95).
                    Input format:     The input data is stored in
                                      PEs 0 through N-1. PE i contains
                                      sample i for 0 ≤ i ≤ N−1.
                    Output:           The output data is stored in
                                      PEs 0 through N-1. PE i contains
                                      sample i for 0 ≤ i ≤ N.
                    Cycles:           130 + NetD
                    Typical time:     37 μs
                    Variable Usage:  (* means set by calling routine)
                           input:    input data *
                           output:   filter output data
                           tmp,tmp2: temporary values
*/

Line  Time (in μs)
1     3              USE Shift +1
2     8              TRANSFER input TO tmp
3
4     7              WHERE ADDR = 0 DO/* Get value from previous call*/
5     0.5                   tmp2 ← tmp
6     1.5                   tmp  ← oldvalue/* Switch tmp and oldvalue*/
7     1.5                   oldvalue ← tmp2
8     2              ENDWHERE
9
10    12.75          output ← input + tmp * 0.95
```

Figure 7.1 Sample algorithm SIMD machine. The execution time assumes an 8 MHz MC68000.

other important variables used by the program.

*Number of PEs* will list the number of PEs used by the SIMD machine.

*Parameters* lists and describes the parameters that affect the execution times.

*Input* tells how the input data is distributed among the PEs in the SIMD machine.

*Output* is the corresponding information as *Input*.

*Cycles* gives the number of machine cycles needed to process one input sample for the SIMD machine. *Typical Time* gives the execution time in μs for a typical speech recognition system.

Figure 7.2 is a listing of the assembly language program* written for *pa68* to implement the algorithm in Figure 7.1. The numbers on the left are the only part of the listing that would not appear as an input to *pa68*. They show how many cycles each instruction takes. To convert cycles to seconds, divide two by the clock rate and multiply by the number of cycles. Therefore, for an 8 MHz clock, divide the number of cycles by four to get the execution time in μs.

Everything to the right of a semicolon in Figure 7.2 is a comment. The comments written in **boldface** type are the Flock Algol statements which correspond to the assembler statements which follow them. The number to the left of the Flock Algol statement but to the right of the semicolon is the line number of the corresponding Flock Algol listing.

Lines starting with the string *#include* instruct *pa68* to read in another file and process it. The speech processing programs commonly use the *simd.h* and the *defs.h* include files. The include file *simd.h* is listed in Figure A.2. All the data transfer registers, masking unit registers, and other special devices are memory mapped into the CU and PE address spaces. *Simd.h* defines where the various devices appear in the address spaces. It also defines macros for setting up the different interconnection functions and for data conditional masking. These are discussed later.

Figure A.3 is the listing of the include file *defs.h*. *Defs.h* contains definitions for the parameters used by the different speech processing programs.

*The assembly language programs for all the simulations in this chapter are listed in Appendix A.

```
      ;            Program name:       filter
      ;            Algorithm:          Figure filter.1??
      ;            Machine:            SIMD, simulated by a MC68000.
      ;            Function:           This program preemphasises the
      ;                                input speech data with a filter
      ;                                with the transfer function:
      ;                                H(z) = 1 - coef * z^-1
      ;            Number of PEs:      N
      ;            Parameters:         coef, The filter coef. (default = 0.95).
      ;            Input format:       The input data is stored in
      ;                                PEs 0 through N-1. PE i contains
      ;                                sample i for 0 ≤ i ≤ N-1.
      ;            Output:             The output data is stored in
      ;                                PEs 0 through N-1. PE i contains
      ;                                sample i for 0 ≤ i ≤ N.
      ;            Cycles:             130 + NetD
      ;            Typical time:       37 μs
      ;            Register usage:     (* means set by calling routine)
      ;                    d0      pe      used by macros
      ;                    d1      pe      tmp
      ;                    d2      pe      used to swap tmp and oldvalue
      ;                    d7*     pe      WHOAMI  (physical pe address)
      ;                    a0*     pe      points to input signal
      ;                    a1*     pe      points to output signal

      #include  "simd.h"
      #include  "defs.h"

      ;            Data allocation for routine
                   .p_data           ; Data stored in each PE
      coef:        .word   0x8667,0x8667,0x8667,0x8667, \
                           0x8667,0x8667,0x8667,0x8667, \
                           0x8667,0x8667,0x8667,0x8667, \
                           0x8667,0x8667,0x8667,0x8667

                   .p_bss
      oldvalue: .=.+ 2     ; Holds sample N-1 for next time

                   .globl  filter

                   .c_text
      filter:
      ;
      ; 1          USE Shift +1
      ;
12                 Shift(#1)         ; Set up interconnection network addresses
```

Figure 7.2 Sim68 program to perform preemphasis filtering. Numbers to left are execution times in cycles.

```
        ;
        ; 2             TRANSFER input TO tmp
        ;
4                       p_mov.w       (a0),d0
6                       p_mov.w       d0,DTRIN.w    ; transfer inputs from
                                                    ; PE i to PE i-1
                        NetworkDelay(0)
6                       p_mov.w       DTROUT.w,d1

        fixold:
        ;
        ; 4             WHERE ADDR = 0 DO           /* Get value from previous call      */
        ;
28                      Where(d7,EQ,#0)      ; In PE0, get value from last call
        ;
        ; 5                     tmp2 <- tmp
        ; 6                     tmp  <- oldvalue    /* Switch tmp and oldvalue */
        ; 7                     oldvalue <- tmp2
        ;
2                               p_mov.w      d1,d2
6                               p_mov.w      oldvalue.w,d1
6                               p_mov.w      d2,oldvalue.w
        ;
        ; 8             ENDWHERE
        ;
8                       EndWhere

        ;
        ; 10            output <- input + tmp * 0.95
        ;
39                      p_muls coef.w,d1     ; mult. by coef and save in d1.
4                       p_asl.l  #1,d1       ; shift 15 to the right by shifting left one,
2                       p_swap d1            ; and swapping upper and lower words.
2                       p_add.w       d1,d0         ; d0 = d0 + coef * d1
4                       p_mov.w       d0,(a1)       ; save in memory

        filterend:
8                       c_rts
```

Figure 7.2 (Continued)

### 7.1.2. Simulating the Interconnection Network

*Sim68* does not simulate a given interconnection network. Instead, each PE has access to the following three registers:

| | |
|---|---|
| DTRDEST | Physical PE address of destination. |
| DTRIN | Input to the interconnection network. |
| DTROUT | Output from the interconnection network. |

The *DTRDEST* register allows any PE to talk to any other PE. Setting *DTRDEST* to the appropriate values in each PE allows any interconnection function to be simulated. The programs presented here use only the Shift, Cube, and Permutation functions as described in Section 2.4. To assist the programmer, the macros *Shift(x), Cube(x),* and *Perm(x)* define the given functions respectively. See Figure A.2 for the actual macro definitions.

Most interconnection networks take some time for data to travel from the input to the output. The macro *NetworkDelay()* is defined to be a nop *(no operation*, i.e. an operation that does nothing) whose execution time is the same as the typical network transfer time. This value is assumed to be 18 cycles, or 4.5 μs based on the information in [BaLu81,BrSi82]. The interconnection network may have a transfer time as fast as 500 ns for a 16-bit word [Ku84]. If such a network is used, or the transfers are overlapped with the execution time, the effective network could be zero. Therefore the case where the network delay is zero is also presented in many of the tables.

Some algorithms require the CU to make conditional branches based on data stored in the PEs, therefore there is a data path between PE 0 and the CU. Anything PE 0 writes into memory location *TOCU* will appear at the CU in memory location *FROMPE0* after one network delay time.

### 7.1.3. Simulating Broadcasts

*Sim68* simulates broadcasts from the CU to all PEs by using self modifying code. The following two instructions will broadcast the data of size word in register *d0* in the CU to register *d1* in all the active PEs:

```
c_mov.w   d0,.+6
p_mov.w   #0,d1
```

The first instruction writes the data in *d0* into the memory location containing the immediate data for the PE instruction. When the second instruction is broadcast to all active PEs, the new data goes with it. No additional network delays are encountered using this method. The macro *Broadcast(in,out)* is defined to broadcast data from register *in* in the CU to register *out* in the PEs using the above method.

### 7.1.4. Data Conditional Masking

Although the SIMD machine model presented in Chapter 2 includes both PE address masking and data conditional masking, *sim68* simulates only data conditional masking. It uses a mask stack as presented in [ClSi83]. The following example shows how it is performed.

Suppose the following code is to be performed:

```
1    WHERE A>B DO
2        C ← A
3    ELSEWHERE
4        C ← B
5    ENDWHERE
```

Line 1 is executed first in the active PEs by comparing A and B:

```
    p_cmp     B,A
```

Next the flags set by the comparison are moved to the *PE* *c*ondition *c*odes *r*egister (PECCR) of the masking unit:

```
    p_mov.w   sr,PECCR
```

Now the masking unit is given the desired condition:

```
    p_mov.b   #GT,PECCS
```

The *PE* *c*ondition *c*ode *s*elect register (PECCS) tells the masking unit which condition must be met. At this point, all previously active PEs are still active. The CU now tells the masking unit to logically AND the negative of the current condition with the top of the mask stack and push the results on the mask stack. This is done by writing the proper code to the mask control register *(MASKCTL)*:

```
c_mov.w   #Pushs+NDataCond,MASKCTL
```

The negative of the condition enables the PEs for the *ELSEWHERE* condition. Next, the positive condition code is logically ANDed with the value second from the top of the mask stack and pushed on the mask stack:

```
c_mov.w   #Pushss+DataCond,MASKCTL
```

Now the PEs are enabled for the WHERE condition. The statements for line 2 are now executed in those PEs where the condition is true. The ELSEWHERE on line 3 is performed by popping the top of the mask stack:

```
c_mov.w   #Pop+DataCond,MASKCTL
```

Then the statements of line 4 are executed. Finally, line five is executed by again popping the mask stack:

```
c_mov.w   #Pop+DataCond,MASKCTL
```

Now all the PEs that were active before line 1 are again active.

*Sim68* assumes that if all PEs should be disabled during a WHERE or an ELSEWHERE condition, the statements in that block will take no time to execute. This means the hardware must be able to detect that all PEs are disabled and ignore all PE instructions until some PEs are enabled again.

In most cases some PEs will execute the WHERE block, while some will do the ELSEWHERE block, making the execution time the total of both blocks plus the time for enabling and disabling the appropriate sets of PEs.

### 7.1.5. The Typical Speech Recognition System

The programs in the rest of the chapter frequently reference a typical speech recognition system. Table 7.1 lists the parameters for the typical system as used here. These parameters are for a high quality speech recognition system. Most speech recognition systems use 12-bit input samples rather than the 16-bit samples as shown in the table. Also, many high quality systems use an input data rate of 15 KHz, while this system can process data at 20 KHz. This system was chosen to be a conservative system, therefore, it requires more processor throughput than many high quality speech recognition systems.

Table 7.1 Parameters for the typical speech recognition system.

| Parameter | Variable Name | Value |
|---|---|---|
| Sample Rate | | 20 KHz |
| Bits per Sample | | 16, signed |
| Frame Size | M | 100 |
| Autocorrelation Coefs. | autocoef | 9 |
| LPC Coefs. | p | 8 |
| Bits per Coef. | | 16 |
| LTW Output Frames | I | 40 |
| DTW Warping Path Width | r | 6 |
| Range in Vocabulary Size | W | 10-1,000 words |

Execution times that are listed for the typical system are in $\mu s$ and assume the MC68000 uses an 8 MHz clock and data takes 4.5 $\mu s$ to travel through the interconnection network.

### 7.1.6. Execution Times

Execution times for all *sim68* simulations are given in cycles. This paper assumes that the MC68000 runs at an 8 MHz clock rate which gives a register-to-register addition time of 0.5 $\mu s$ for a word (16-bits) data size, or 1 $\mu s$ for a long (32-bit) data size. A 16 by 16 bit signed multiply takes 8.75 $\mu s$.

For each program an expression for the execution time is derived in terms of the parameters of the program. These times are given in terms of:

| | |
|---|---|
| autocoef | The number of autocorrelation coefficients used of LPC. |
| M | The number of samples per LPC frame. |
| I | The number of frames output from the ltw routine. |
| N | The number of PEs the given programs uses. |
| logN | $\lceil \log_2 N \rceil$. |
| NetD | The network delay time in cycles. |
| p | The number of LPC coefficients. |
| r | The width of the dtw warping path. |

In most speech processing systems p=autocoef−1. The times are given in an expanded form, for example:

$$\begin{aligned}\text{cycles} = \quad & 10 + \text{autocoef}[(24+\text{NetD}) + 85 + \\ & (54 + 2\text{NetD})\text{logN} + 2 + 19] - 23 - \text{NetD} + 1\end{aligned}$$

Each term corresponds roughly to the execution time between adjacent labels in the program being considered. In the example above, (54 + 2NetD)logN+2 would correspond to a loop that executed log N times and contained two network transfers. These times do not include the overhead of a main program calling or returning from the given program.

### 7.1.7. Summary

*Sim68* does a good job of simulating an SIMD machine. The important things to know about the simulations are:

1) All Flock Algol times are given in $\mu$s assuming an 8 MHz clock and a 4.5 $\mu$s network delay time.

2) All *pa68* times are given in cycles. Divide cycles by 4 to convert to $\mu$s.

3) If all PEs are disabled, the PE instruction takes no time to execute.

4) The times are conservative because of the assumption that CU and PE instructions are not overlapped.

## 7.2. Digital Preemphasis Filtering

This section presents the SIMD implementation of the Flock Algol algorithm for preemphasis filtering. The filter transfer function is:

$$H(z)=1-az^{-1}$$

where typically $a \simeq .95$. The preemphasis filter is used on the input speech data before autocorrelation analysis is done. To process telephone quality speech in real time, the filtering program must be able to filter 6,670 8-bit samples per second. Filtering high quality speech requires a sampling rate of 15 to 20 KHz using 11 to 12 bits per sample.

Figure A.4 is a parallel MC68000 program to perform the preemphasis filtering on an SIMD machine as discussed in Section 6.1.8. The program uses 16-bit samples and N PEs. It assumes the speech data is stored in the PEs before the program is executed. Sample i is stored in PE i for $0 \leq i < N$, where N is the number of PEs used. The output data uses the same arrangement as the input data. The total execution time is

$$130 + \text{NetD}.$$

Where *NetD* is the network delay time in cycles. This time does not include approximately 26 cycle overhead of calling and returning from the routine. Table 7.2 lists the sampling rates using different network delays and different numbers of PEs. The parameters than are being changed are shown in **boldface** type. Using one PE may be fast enough since Table 7.2 shows that one PE can process data at a sample rate of 27 KHz which is greater than the rate needed for high quality speech processing. This is a lower bound on the maximum sampling rate since if the algorithm uses only one PE, the conditional masking can be replaced by branching instructions and the network transfers are not needed.

Table 7.2 Sampling rates for the SIMD preemphasis program using 16-bit signed data.

| Program | Preemphasis Filter | | | | | |
|---|---|---|---|---|---|---|
| **N** | **1** | **10** | **100** | **1** | **10** | **100** |
| Number of PEs | 1 | 10 | 100 | 1 | 10 | 100 |
| **NetD** | **0** | **0** | **0** | **18** | **18** | **18** |
| Transfers | 1 | 1 | 1 | 1 | 1 | 1 |
| Cycles | 130 | 130 | 130 | 148 | 148 | 148 |
| Time/Sample ($\mu$s) | 32.5 | 32.5 | 32.5 | 37 | 37 | 37 |
| Max Sample Rate (KHz) | 30 | 300 | 3,000 | 27 | 270 | 2,700 |

### 7.2.1. Summary

This section presented a parallel preemphasis filter program. It is able to process speech in real time using as few as 1 PE. By using more PEs, the program can process data at a higher sampling rate. This program assumes that the data was already in the PEs before the program is executed. This is a valid assumption if the program calling the filter program has already loaded the data.

The MC68000 processor is well suited for this type of speech processing since speech data typically uses 12 to 16 bits per sample. The 16 by 16 signed multiplication instruction and the 16-bit signed addition instruction allow the MC68000 to compute the filtered signal quickly.

Filtering usually precedes the computation of autocorrelation coefficients. The next section presents the autocorrelation program and shows how it will work with the preemphasis filtering program to process speech.

## 7.3. Simulation of the Autocorrelation Algorithm

Autocorrelation plays an important role in many isolated word recognition systems. It is used to find the short term autocorrelation coefficients which are then used to find the LPC coefficients. Autocorrelation, as used here, is defined as:

$$R(i) = \sum_{k=0}^{M-i-1} x(k)x(k+i) \qquad 0 \leq i \leq \text{autocoef}$$

where *R(i)* are the autocorrelation coefficients and *x(m)* is the input signal. For speech processing M ranges from 100 to 300 samples, while ***autocoef*** is between 8 and 16 [Myer80]. For the typical system, M=100 and ***autocoef***=9.

In this section, Siegel's autocorrelation algorithm, discussed in Section 5.1.1, is converted to a MC68000 assembly language program and ***sim68*** is used to simulate an SIMD machine executing the program. Figure A.5 is a listing of the program with the execution times, in cycles, on the left, and the corresponding Flock Algol statements as comments in **boldface.** This program assumes 16-bit input data and keeps a 32-bit sum. In general, the total execution time is:

$$\begin{aligned} \text{cycles} &= 10+ \\ &\quad (\text{autocoef})[(30+\text{NetD})+85+(54+2\text{NetD})\log M+2+19] \\ &\quad -23-\text{NetD}+1 \\ &= (\text{autocoef})[136+\text{NetD}+(54+2\text{NetD})\log M]-12-\text{NetD} \end{aligned}$$

Each number in the first line roughly represents the execution time between adjacent labels in Figure A.5.

Table 7.3 gives the execution times for a typical speech application.

Table 7.3 Execution time for autocorrelation program using 16-bit signed inputs and a 32-bit signed sum.

| Program | auto | | auto + filter | |
|---|---|---|---|---|
| **autocoef** | **9** | **9** | **9** | **9** |
| M | 100 | 100 | 100 | 100 |
| logM | 7 | 7 | 7 | 7 |
| Number of PEs | 100 | 100 | 100 | 100 |
| **NetD** | **0** | **18** | **0** | **18** |
| Transfers | 134 | 134 | 135 | 135 |
| Cycles | 4,614 | 7,026 | 4,744 | 7,174 |
| Time | 1,153 μs | 1,757 μs | 1,186 μs | 1794 μs |
| Time/Sample | 11.53 μs | 17.57 μs | 11.86 μs | 17.94 μs |
| Max Sample Rate | 86 KHz | 56 KHz | 84 KHz | 55 KHz |

| Program | auto | | auto + filter | |
|---|---|---|---|---|
| **autocoef** | **17** | **17** | **17** | **17** |
| M | 100 | 100 | 100 | 100 |
| logM | 7 | 7 | 7 | 7 |
| Number of PEs | 100 | 100 | 100 | 100 |
| **NetD** | **0** | **18** | **0** | **18** |
| Transfers | 134 | 134 | 135 | 135 |
| Cycles | 8,726 | 13,316 | 8,856 | 13,464 |
| Time | 2,182 μs | 3,329 μs | 2,214 μs | 3,366μs |
| Time/Sample | 21.82 μs | 33.29 μs | 22.14 μs | 33.66 μs |
| Max Sample Rate | 45 KHz | 30 KHz | 45 KHz | 29 KHz |

### 7.3.1. Effects of NetD on Execution Times

Selecting a value for NetD is difficult. The execution summaries use the values 0 and 18 cycles. 0 is used for a small or negligible delay [Ku84] or when the network transfer is overlapped with the instruction execution. 18 cycles, which is 4.5 $\mu$s, is the value used in [BrSi82]. Another approach is to ask "What is the maximum value NetD can have and still allow the program to run in real time?" Combining the filtering and autocorrelation programs, as they would be in a typical speech system, gives an execution time of 4,744 cycle to process 100 samples for autocoef=9. There are 200 cycles between samples when using a 20 KHz sampling rate, therefore transfers can use 20,000−4,744=15,256 cycles. The programs use 135 transfers, so each transfer can take 113 cycles or 28 $\mu$s per 16-bit word. For example, the Poker system [Snyder82b] requires 12 $\mu$s per byte, or 24 $\mu$s per 16-bit word for transfers which is less than the maximum delay of 28 $\mu$s. An effective sampling rate of 85 KHz with no network delay is reduced to 20 KHz if the network delay is 28 $\mu$s per 16-bit word. This algorithm can tolerate a slow interconnection network and still process high quality speech in real time if autocoef=9. If autocoef=17, then 8,856 cycles are used leaving 11,144 cycles for the 256 transfers which is 43 cycles (10 $\mu$s) per transfer.

### 7.3.2. Using Fewer PEs

The algorithm, as presented, must use as many PEs as there are samples in each frame. In a typical speech recognition system the frame size ranges from 100 to 400 samples which means 100 to 400 PEs must be used. The algorithm (auto/2) in Figure 7.3 can find the autocorrelation coefficients of a M=2N sample frame using N PEs. Before execute, PE i contains samples i and i+N/2 for $0 \leq i \leq N$. As before, the data is shifted between the PEs so that when autocorrelation coefficient j is being computed, PE i contains samples i and i+j, and samples i+N/2 and i+j+N/2. Since each PE contains two samples, two transfers must be used to get this arrangement. The product of samples i and i+j for $0 \leq i \leq 2N$ is found using two multiplication steps per PE and the sum of the products is found using recursive doubling.

```
/*              Algorithm Name:   auto/2
                Section:          7.3.2
                Machine:          SIMD
                Function:         This program finds the autocorrelation
                                  coefficients of input speech data using
                                  half as many PEs as samples in a frame.
                Number of PEs:    N
                Transfers:        Shift(-1), Cube
                Masking:          Data Conditional
                Parameters:       autocoef, The number of coefs. to find.
                                  N, The number of PEs in use.
                                  NetD, The interconnection network
                                          delay time in cycles.
                Input:            The input data is stored in PEs 0 through N-1
                                  with PE i containing sample i and i+N/2
                                  for 0 ≤ i ≤N.
                Output:                   The autocorrelation coefficients, R(i),
                                  for 0 ≤ i ≤autocoef-1 appear in PE i
                                  for 0 ≤ i ≤N (i.e. each PE contains
                                  every coefficient).
                Cycles:           autocoef[136+NetD + (54+2NetD)logN] - 12 - NetD
                Typical Time:     1,757 µs for autocoefs=9, NetD=18, and logN=7.
                Variable Usage: (* means set by calling routine)
                        ADDR:     Address of PE (e.g. ADDR = 0 in PE 0).
                        L:        on completion, PEs 0-L will contains R(i).
                        partsum:  temporary variable holding a partial sum.
                        R():      autocorrelatin coefficients.
                        sig1:*    first half of input signal (sample i)
                        sig2:*    second half of input signal (sample i+N/2)
                        slast1:   after stage i: "slast" in PE m holds sig(m+i).
                        slast2:   after stage i: "slast" in PE m
                                  holds sig(m+N/2+i).
                */
Line  Time in µs
 1    1.75      slast1 ← sig1     /* After stage I, "slast" in
 2                                PE m holds sig(m+i) */
 3    1.57      slast2 ← sig2     /* After stage I, "slast" in
 4                                PE m holds sig(m+i) */
 5
 6    5         FOR i ← 0  TO p DO
 7    1.5           IF i ≠ 0 THEN
 8    3                           USE Shift(-1)
 9    1.5                         DTRin ← slast1
10    4.5                         TRANSFER
11    2.0                         slast1 ← DTRout
12    1.5                         DTRin ← slast2
13    4.5                         TRANSFER
14    2.0                         slast2 ← DTRout
```

Figure 7.3 Algorithm for autocorrelation using N PEs for a frame size of 2N. The execution times assume an 8 MHz MC68000. (See Section 7.3.2.)

```
           
7          WHERE(ADDR,EQ,N−1)
0.5              tmp ← slast1
                 slast1 ← slast2
                 slast2 ← tmp
           ENDWHERE

0.5        partsum ← 0

6.5        WHERE ADDR < M−i DO
10.75            partsum ← slast2 * sig2
2          ENDWHERE

10         partsum ← partsum + slast1 * sig1

2.25       FOR j ← 0 TO max(⌈log(M−i)⌉−1,log(L−1)) DO
3                USE Cube(j)
12.5             TRANSFER partsum TO tmp
0.75             partsum ← tmp + partsum
1.5        R(i) ← partsum
```

Figure 7.3 (Continued)

Figure A.6 is a listing of the corresponding program. The time complexity for auto/2 is:

$$\text{cycles}=18+(\text{autocoef})[(84+2\text{NetD})+87+44+(54+2\text{NetD})\log N+2+19]-$$
$$77-2\text{NetD}+1$$
$$=(\text{autocoef})[236+2\text{NetD}+(54+2\text{NetD})\log N]-58-2\text{NetD}$$

In the proposed speech recognition system using 100 PEs, the autocorrelation program uses 7,174 cycles when autocoef=9, the frame size is 100 samples, and NetD=18. If 50 PEs are used, auto/2 uses 7,214 cycles which is a sampling rate of about 55 KHz. Auto/2, using 50 PEs, requires 188 cycles more than auto, using 100 PEs, which is about 3% more. This is a surprisingly small increase in execute time. Examining the time complexity equations for auto and auto/2 shows that auto requires 136+NetD cycles to perform the Shift transfer and find the product of two samples. Auto/2 requires 236+2NetD cycles to compute the same values, therefore needing almost twice as many cycles. Auto requires (54+2NetD)logM cycles to find the sum of the products using recursive doubling, while auto/2 uses (54+2NetD)logN cycles where N=M/2. Therefore since auto/2 has two samples per PE, it requires one less pass through the interconnection network, so it uses 54+2NetD fewer cycles to compute the sum. The time saved by auto/2 having two samples per PE is slightly less than the extra time it uses to compute the product of two samples per PE, therefore there is only a slight increase in the total computation time.

The same techniques that converted auto to auto/2 can be applied to further reduce the number of PEs used, while increasing the execution time. In general, if there are more samples per each frame than PEs, the algorithm can be modified so each PE will compute $\lceil M/N \rceil$ products where M is the number of samples per frame and N is the number of PEs.

### 7.3.3. Increasing the Throughput Through Serialism

The previous section showed that using half as many PEs resulted in only a 3% increase in the execution time. This result can be used to increase the throughput while using the same number of PEs. Suppose a system uses 100 samples per frame, and has 100 PEs. The execution time will be 7,026 cycles if autocoef=9 and NetD=18. The system can process two frames at a time if PEs 0 through 49 process the first frame, and PE 49 through 99 process the second frame using a modified version of auto/2. The total execution time will be roughly 7,214 cycles (there will be some over head due to processing two frames at a time.) The average execution time per frame is then 7,214/2 = 3,607 cycles which is 52% of the cycles used when processing only one frame at a time.

The above technique could be repeated until 100 frames are being processed in parallel with the 100 PEs doing one frame each. This will certainly increase the throughput, but it will also increase greatly the delay between the time a sample enters the system, and the time the autocorrelation coefficients are computed. This is probably not appropriate for an environment in which real-time processing is desired.

### 7.3.4. Summary

This section presented a program implementing a parallel autocorrelation algorithm. Using M=N PEs it can find the first autocoef=9 autocorrelation coefficients of an M=100 sample frame of speech in 1.7 μs. This gives an effective sample rate of 56 KHz which is more than sufficient for high quality speech processing. Each additional coefficient computed takes 194.5 μs. Combining autocorrelation with the preemphasis filter program from the previous section gives a sampling rate of 55 KHz which is more than enough for high quality speech recognition. Some high quality speech processing uses autocoef=17 coefficients, which gives a sampling rate of 29 KHz which is still more than enough for high quality speech.

The input data is arranged with one 16-bit sample per PE with PE i containing sample i for $0 \leq i < N$. This is the same as the output format of the filter program. The output has PEs 0 through *autocoef* containing all the autocorrelation coefficients.

Fewer PEs than samples in a frame can be used without greatly increasing the execution time. Although the throughput can be increased by computing several frames in parallel using a fewer PEs per frame, the delay time between an input and an output will increase.

The hardware is well suited for this problem since it has a 16 by 16-bit signed multiplications and 32-bit additions. These built-in instructions which perform operations on data the same size as the problem's data size make programming the SIMD machine a straightforward task.

### 7.4. Simulation of the Linear Prediction Algorithm

Linear predictive coding (LPC) is frequently used in both speech synthesis and recognition. The LPC coefficients model the vocal tract as an all pole filter, and the error signal from the coding, models the excitation of the vocal chords. A speech recognition system divides the the speech signal into 10 to 20 ms frames and finds the LPC coefficients for each frame. Therefore, a real-time system that inputs data at 10 KHz to 20 KHz must process one frame of between 100 and 400 samples every 10 to 20 ms. Generally 16-bit coefficients are used, but some applications can use as few as 10 bits [MaGr74].

Figure A.7 is the listing of a program that finds the LPC coefficients given the autocorrelation coefficients. It is based on the algorithm in Figure 5.7. The input data is arranged so each PE contains all the autocorrelation coefficients ($R(i)$ for $0 \leq i <$ autocoef). The output data has LPC coefficient i stored in PE $i-1$ for $1 \leq i \leq p$.

The program uses fixed point arithmetic. The position of the decimal point is shown in the right column. The code *d#=x,y* means that in register *d#*, *x* bits are to the left of the decimal point, and *y* bits are to the right.

The total execution time for the program is:

$$\begin{aligned}\text{cycles} &= 26+p[92+(54+2\text{NetD})\log(p)+2+112+\\&\qquad 125+88+81+\text{NetD}+13]-\text{NetD}-65+1\\&= p[513+\text{NetD}+(54+2\text{NetD})\log(p)]-38-\text{NetD}\end{aligned}$$

where each number in the first line roughly represents the time between labels in Figure A.7. Table 7.4 gives the execution times for a typical speech application. Computing the LPC coefficients alone can be done at a rate of 62 KHz assuming 100 samples per frame, 8 coefficients and NetD=18 using 8 PEs. A typical speech processing system would preemphasize the signal and find the autocorrelation coefficients before finding the LPC coefficients. Using the previous filtering and autocorrelation programs, this can be done with a sampling

Table 7.4 Execution times for LPC program and filter+auto+lpc programs.

| Program | LPC | | filter+auto+LPC | |
|---|---|---|---|---|
| **p** | **8** | **8** | **8** | **8** |
| M | 100 | 100 | 100 | 100 |
| Number of PEs | 8 | 8 | 100 | 100 |
| **NetD** | **0** | **18** | **0** | **18** |
| Transfers | 55 | 55 | 190 | 190 |
| Cycles | 5,362 | 6,352 | 10,106 | 13,526 |
| Time | 1,341 $\mu$s | 1,588 $\mu$s | 2,527 $\mu$s | 3,391 $\mu$s |
| Time/Sample | 13.41 $\mu$s | 15.88 $\mu$s | 25.27 $\mu$s | 33.82 $\mu$s |
| Max Sample Rate | 74 KHz | 62 KHz | 39 KHz | 29 KHz |

| Program | LPC | | filter+auto+LPC | |
|---|---|---|---|---|
| **p** | **16** | **16** | **16** | **16** |
| M | 100 | 100 | 100 | 100 |
| Number of PEs | 8 | 8 | 100 | 100 |
| **NetD** | **0** | **18** | **0** | **18** |
| Transfers | 143 | 143 | 399 | 399 |
| Cycles | 11,626 | 14,200 | 20,482 | 27,664 |
| Time | 2,907 $\mu$s | 3,550 $\mu$s | 5,121 $\mu$s | 6,916 $\mu$s |
| Time/Sample | 29.07 $\mu$s | 35.50 $\mu$s | 51.21 $\mu$s | 69.16 $\mu$s |
| Max Sample Rate | 34 KHz | 28 KHz | 19 KHz | 14 KHz |

rate of 29 KHz, which is sufficient for high quality speech.

A sample rate of 20 KHz and a frame size of 100 samples gives 20,000 cycle between frames. The three programs use 10,052 cycles leaving at most 9,948 cycles for network delays. Since 190 transfers are used, each can take 52 cycles, or 13 $\mu$s per 16-bit word and process speech in real time.

Table 7.4 shows that if p=16 coefficients are used and a 4.5 $\mu$s NetD is assumed, the programs can process data at 14 KHz which is too slow for most high quality speech processing. If the network transfers are fast, or overlapped with the instruction execution so that NetD=0, the speech data can be processed at 19 KHz which is in the range of 15 KHz to 20 KHz used most often for high quality processing.

### 7.4.1. Summary

This section presented a parallel program for computing LPC coefficients from autocorrelation coefficients. It is able to process data at a rate of 62 K samples per second assuming a 100 sample frame, 8 LPC coefficients, and a network delay of 4.5 $\mu$s per 16-bit word. LPC analysis is usually preceded by preemphasis filtering and autocorrelation. The processing rate for these three programs, using the conditions above, is 29 KHz. This is sufficient for real-time processing of high quality speech. A network delay of up to 13 $\mu$s per 16-bit word can be tolerated and still process at the 20 KHz rate needed for high quality speech.

This program uses fixed-point arithmetic and computes coefficients with 16-bit precision. The program uses approximately 7% of the coefficient calculation time to rotate the data so the decimal point is in the correct position. This is a small overhead for implementing fixed point arithmetic.

The LPC program uses both the Cube and Perm interconnection functions and is the only program to use the Perm function. It is possible the interconnection network will not be able to perform the Perm function directly, but instead will use multiple passes through the network. Since the Perm function is used $p$ times and it may take $p$ passes through the interconnection network to implement it, p(p−1) additional network delays may be added to the execution time. For the typical system this is roughly (8)(7)(4.5 $\mu$s) = 252 $\mu$s. This is about a 16% increase over the original time.

## 7.5. Simulation of Linear Time Warping (LTW) Algorithms

In a typical isolated word recognition system, linear time warping occurs after the endpoint detection and before the dynamic time warping. Its purpose is to take an utterance of variable length and linearly stretch or shrink it, in the time domain, until it is a fixed length. Isolated utterances can range from 20 to 80 frames in length in a typical system, where a frame consists of 8 LPC coefficients. Some systems will stretch or shrink the utterance to a 40 frame length. Only after the endpoint routines detect an utterance can the LTW program process the speech data. Since isolated words are about one third to one half second in duration, the LTW must be able to perform its operation in about 300 to 500 ms.

Two LTW algorithms were presented in Section 6.3. Method one places one frame per PE and moves the data between the PEs to do the warping. Method two has one coefficient from each frame in each PE and gets its speed by doing the vector operations in parallel. The following sections present programs implementing each algorithm and gives timing information for each.

### 7.5.1. Method One – One Frame per PE

Figure A.8 is a program for performing method one. The input data is arranged so PE j contains frame j for $0 \leq j < J$, where J is the number for frames in the input utterance and each frame consists of $p$ LPC coefficients. After processing, PE i contains frame i for $0 \leq i < I$, where I is the new utterance length. In a typical system $20 \leq J \leq 80$ and I=40, so the number of PEs is the maximum of J and I.

The time complexity for method one in Figure A.8 is:

$$\text{cycles}=7+210+80+p(29+\text{NetD})+2+10+109p+2+6+6$$
$$+\{42+\text{NetD}+[29+\text{NetD}]p+2+15\}(J-I)+2$$

$$=325+(138+\mathrm{NetD})p+(J-I)[59+\mathrm{NetD}+(29+\mathrm{NetD})p]$$

if J>I. If J=I the linear time warp simplifies to a copy operation taking

$$11+11p$$

cycles. If J<I the time complexity is:

$$\mathrm{cycles}=7+232+(I-J)[42+\mathrm{NetD}+(45+\mathrm{NetD})p+2+13]+2+2+80$$
$$+(29+\mathrm{NetD})p+2+10+109p+2+7$$
$$=344+(138+\mathrm{NetD})p+(I-J)[57+\mathrm{NetD}+(45+\mathrm{NetD})p]$$

Whenever the utterance is being expanded or compressed, the number of operations is based on the amount of change in size. Table 7.5 gives values for J–I = −20, −10, 0, 10, 20, 40 for network delays of 0 and 18 cycles and p=8 coefficients.

### 7.5.2. Method Two – One Coefficient per PE

Figure A.9 is the program for implementing the the second method of linear time warping as discussed in Section 6.3.2. For 8 LPC coefficients, it uses 8 PEs with the input data arranged so that PE k contains coefficient k of frame j for $0 \leq k < p$ and $0 \leq j < J$. The output data uses the same arrangement. Its time complexity is

$$\mathrm{cycles}=7+98+I(45+10+22+106)+2$$
$$=107+183I$$

if J≠I and 450 cycles if J=I. Table 7.6 gives times for a typical speech system.

### 7.5.3. Comparing LTW Methods One and Two

These two methods are an example of the importance of including overhead such as transfers in the time complexities. From Table 6.3. one would expect method one to perform better than method two because method one

Table 7.5 Execution times for linear time warping, method one.

| Program | LTW Method One | | | | | |
|---|---|---|---|---|---|---|
| J–I | −20 | −20 | −10 | −10 | 0 | 0 |
| p | 8 | 8 | 8 | 8 | 8 | 8 |
| Number of PEs | 40 | 40 | 40 | 40 | 40 | 40 |
| **NetD** | **0** | **18** | **0** | **18** | **0** | **18** |
| Transfers | 188 | 188 | 98 | 98 | 0 | 0 |
| Cycles | 9,788 | 13,172 | 5,618 | 7,382 | 99 | 99 |
| Time ($\mu$s) | 2,447 | 3,293 | 1,405 | 1,846 | 34 | 34 |

| Program | LTW Method One | | | | | |
|---|---|---|---|---|---|---|
| J–I | 10 | 10 | 20 | 20 | 40 | 40 |
| p | 8 | 8 | 8 | 8 | 8 | 8 |
| Number of PEs | 50 | 50 | 60 | 60 | 80 | 80 |
| **NetD** | **0** | **18** | **0** | **18** | **0** | **18** |
| Transfers | 98 | 98 | 188 | 188 | 368 | 368 |
| Cycles | 4,339 | 6,103 | 7,249 | 10,633 | 13,069 | 19,693 |
| Time ($\mu$s) | 1,085 | 1,526 | 1,812 | 2,658 | 3,267 | 4,923 |

| Program | LTW Method One | | | | | |
|---|---|---|---|---|---|---|
| J–I | −20 | −20 | −10 | −10 | 0 | 0 |
| p | 8 | 8 | 8 | 8 | 8 | 8 |
| Number of PEs | 40 | 40 | 40 | 40 | 40 | 40 |
| **NetD** | **0** | **18** | **0** | **18** | **0** | **18** |
| Transfers | 356 | 356 | 186 | 186 | 0 | 0 |
| Cycles | 18,092 | 24,500 | 10,322 | 13,670 | 187 | 187 |
| Time ($\mu$s) | 4,523 | 6,125 | 2,580 | 3,418 | 47 | 47 |

| Program | LTW Method One | | | | | |
|---|---|---|---|---|---|---|
| J–I | 10 | 10 | 20 | 20 | 40 | 40 |
| p | 8 | 8 | 8 | 8 | 8 | 8 |
| Number of PEs | 50 | 50 | 60 | 60 | 80 | 80 |
| **NetD** | **0** | **18** | **0** | **18** | **0** | **18** |
| Transfers | 186 | 186 | 356 | 356 | 696 | 696 |
| Cycles | 7,763 | 11,111 | 12,993 | 19,401 | 23,453 | 35,981 |
| Time ($\mu$s) | 1,941 | 2,778 | 3,249 | 4,851 | 5,864 | 8,996 |

Table 7.6 Execution times for linear time warping, method two.

| Program | LTW Method Two | |
|---|---|---|
| I | 40 | 40 |
| **p** | 8 | 16 |
| Number of PEs | 8 | 16 |
| Transfers | 0 | 0 |
| Cycles | 7,427 | 7,427 |
| Time | 1,857 $\mu$s | 1,857 $\mu$s |

uses one scalar and two vector multiplication steps* and method two uses 3I scalar multiplication steps. In a typical system the vectors contain 8 elements and I=40, so method one uses 17 scalar multiplication steps while method two uses 120. Tables 7.5 and 7.6 show that methods one and two both take about 1.8 ms if I−J=10 and NetD=18. This seems inconsistent with Table 6.3 until the transfer times are considered. Method one uses $|J-I|+1$ transfers while method two uses none. The vector and scalar transfers take approximately $(|J-I|+1)(453)$ cycles, and the $|J-I|$ vector multiplications, used in method one, require 872 cycles for p=8 and NetD=18. The vector transfer time is about half the time of a vector multiplication. Therefore when comparing the time complexities of two methods, relative times of all operations should be considered.

### 7.5.4. Summary

A typical speech recognition system has at least 300 to 500 ms between the starting times of two utterances. The LTW program must be performed once for each input utterance, therefore the LTW must executed in less than 300 to 500 ms to run in real time. Both methods presented here can execute in less than 300 to 500 ms assuming that the data is stored in each PE before the LTW program is run. The problem of getting the data in this allocation is discussed in Section 7.7.

The arrangement of the input and output data and the number of PEs used are the main differences between these two methods. Method one uses the maximum of J and I PEs while method two uses p PEs.

Selecting one of these methods may depend on the data arrangement, not the execution time. If a system has each PE processing one frame of speech, method one should be used since it requires one frame per PE as input. If the system has each PE containing one coefficient from each frame, method two should be used since that is how its input data is arranged. If the system uses neither of the above arrangements the data will have to be moved to match

*The time of a multiplication step is the time used by one multiplication operation in several PEs in parallel.

one of the arrangements. The choice of which arrangement to use would be based on the time needed to move the data into one of the arrangements, and the desired output data arrangement.

Neither LTW program can begin execution until after the input utterance has been detected. This causes a delay time since the LTW program and the programs that follow it must wait until the entire utterance is spoken.

## 7.6. Simulation of Dynamic Time Warping Algorithms

Dynamic time warping (DTW) is the process of taking one unknown utterance and comparing it to one known utterance. The DTW algorithm dynamically stretches and shrinks both utterances, in time, to match them to each other as well as possible. This is done, as explained in Section 4.6.2, by computing the local distance d(i,j) between frame i of the known utterance and frame j of the unknown utterance. Dynamic programming theory is used to find the minimum path from d(0,0) to d(I,I) where I is the number of frames in the known and unknown utterances. The local distance scores are accumulated along this minimum path, and the result is a single score telling how closely the two utterances match. A typical isolated word recognition system matches an unknown utterance to every known utterance in the system's vocabulary. A 1,000 utterance vocabulary would therefore require 1,000 DTWs to be performed.

An utterance is a collection of $I$ frames of $p$ coefficients each. $I$ is constant since the LTW program will stretch or shrink the utterance to a fixed length before the DTW program processes it. Typically $I=40$ and $p=8$ and each coefficient has 16 bits.

Section 6.4.1 presented two approachs for implementing a parallel DTW. Both methods are simulated using *sim68*. The first approach is the *s*erial *p*arallel (SP) method. Since a typical speech recognition system needs to perform one DTW match for each word in its vocabulary, the SP method uses one PE for each vocabulary word and broadcasts the unknown utterance to all PEs. Each PE executes a serial DTW to match its known utterance to the unknown utterance.

The second approach is the *p*arallel *p*arallel (PP) method. The PP method uses several PEs to perform one DTW comparison. Two implementations of the PP method are given. The first (PP1) moves the input data to the appropriate PEs and then computes the local distances as they are needed. The second program (PP2) computes the local distances while moving the data to the PEs and then computes the DTW.

The following section presents the *rearrange* routine which is used to rearrange the unknown utterance among the PEs before executing the SP and PP1 programs.

### 7.6.1. Rearrange

Both the SP and PP1 methods need to store the input data in each PE in an unusual manner. The *rearrange* routine moves the data from one arrangement to another so that the DTW programs will have the data in the right places.

The *rearrange* routine expects its input data to be stored with coefficient k of frame i in PE k for $0 \leq k < p$ and $0 \leq i < I$. This arrangement is chosen since it is the arrangement used by the LPC and LTW routines. Rearrange moves the data from this arrangement to the arrangement needed by the DTW program, in which each PE has all the coefficients from all the frames in the unknown utterance. Figure 7.4 is a listing of the rearrange algorithm and Figure A.10 contains a listing of the rearrange program. The rearrange routine sends the data to all PEs by using a series of the *Shift −1* transfer functions. First PE 0 sends its data to the CU by writing it to a memory location called *TOCU*. There is a data path from PE 0 to the CU, so that anything PE 0 stores in memory location *TOCU* appears in memory location *FROMPE0* in the CU after the network delay time. PE 0 sends its data to the CU and the CU broadcasts it to all the PEs. The broadcast if performed by having the CU store the data to be broadcast in the immediate data field of a PE instruction. The PE instruction, with the broadcast data, is broadcast to all PEs as is any other instruction and when the PEs execute it, then the data is stored in each PE's register.

After PE 0 sends its data to the CU, all PEs execute a *Shift −1* transfer function. Now PE i contains the data from PE i+1. PE 0 sends the data it received from PE 1 to the CU and it is broadcast, as before. All PEs execute the *Shift −1* transfer function again, so now PE i has data that was originally in PE i+2 and PE 0 sends its data to the CU. This shift-broadcast loop is

```
/*
                    Algorithm Name:   Rearrange
                    Section:          7.6.1
                    Machine:          SIMD
                    Function:         This program moves data around in preperation
                                      for the DTW program
                    Number of PEs:    2r+1
                    Parameters:       r, the width of the warping path.
                                      p, the number of coefficients per frame.
                                      NetD, the network delay time.
                                      I, the number of frames per utterance.
                    Input:            input[i] contains coefficient k of
                                      input vector i in PE k for 0 ≤ l < k.
                    Output:           output[i][k] contains coefficient k of
                                      vector i of the output in all PEs.
                    Cycles:           26 + I[13 + p(47 + NetD)] + 9 ⌊r/2⌋
                    Typical Time:     5,344 μs for p=8, r=6, I=40, NetD=18

*/
```

```
Line   Time in μs
 1                   PROCEDURE Rearrange
 2     3             USE Shift -1
 3     2.25          FOR i ← 0 TO I-1
 4     1                    tmp ← input[i]; /* tmp contains coefficient i in PE i      */
 5     1.75                 FOR j ← 0 TO p-1
 6     2                           TOCU ← tmp; /* send coefficient to CU                    */
 7     2                           DTRIN ← tmp; /* send coefficient to PE to the left      */
 8     NetD                        TRANSFER;
 9     3                           BROADCAST FROMPE0 TO output[i][j];
                                              /* Send to all PEs */
10     2                           tmp ← DTROUT; /* Get coefficient from PE to right     */
11
12     2.75          FOR i ← 0 TO r/2
13     1                    output[i+I] ← ∞;
```

Figure 7.4 Program to rearrange data from PE k containing coefficient k, $0 \leq k \leq p$ to all PEs containing all coefficients.

repeated until all PEs have shifted their data to PE 0 and PE 0 has sent it to the CU and it is broadcast to all PEs.

The time complexity of the rearrange program is:

$$\begin{aligned}\text{cycles}&=16+I[6+p(47+\text{NetD})+2+5]+2+6+9\lfloor r/2\rfloor+2\\&=26+I[13+p(47+\text{NetD})]+9\lfloor r/2\rfloor\end{aligned}$$

Table 7.7 summarizes the execution times for the rearrange program.

Although some interconnection networks can broadcast data without going through the CU [SiMc81a,SiMc81b], this method of using a data path between PE0 and the CU is used here because it can use a less powerful interconnection network. The method implemented requires one data path going from PE 0 to the CU, and the network must be able to perform a *Shift +1* interconnection function. The execution time for such a broadcast is the time to send the data to the CU plus the 3 μs which are needed for the CU to write the data into a PE instruction and broadcast the instruction.

### 7.6.2. Simulation of the DTW Algorithm – The Serial Parallel Method (SP)

Figure A.11 is the listing of the SP MC68000 program for dynamic time warping. It uses PE 0 and assumes that the rearrange program was run before it so that all the known and unknown frames are stored in PE 0 before executing the program. It differs from a serial program in that the CU executes the branching instructions and performs the loop control as in a parallel program. Some "IF ... THEN ... ELSE" constructs that a serial program would use are replaced by the "WHERE ... ELSEWHERE ... ENDWHERE" constructs in the SP program. Although the serial-parallel program executes on only one PE, it is written to execute on several PEs at the same time. This is the way it would be used on an SIMD system in which each PE compares the unknown utterance to a reference utterance.

The distance score of $\infty$ which is used in the algorithm to represent distances from invalid paths is represented in the MC68000 program as the value $4000_{16}$. This value is used since the local distance scores are stored as 16-bit numbers and they may be multiplied by two and added to each other. For

Table 7.7 Execution times for rearrange routine.

| Program | Rearrange | | | |
|---|---|---|---|---|
| **p** | **8** | **8** | **16** | **16** |
| r | 6 | 6 | 6 | 6 |
| I | 40 | 40 | 40 | 40 |
| Number of PEs | 13 | 13 | 16 | 16 |
| **NetD** | **0** | **18** | **0** | **18** |
| Transfers | 320 | 320 | 640 | 640 |
| Cycles | 15,613 | 21,373 | 30,653 | 42,173 |
| Time/Rearrange | 3,903 $\mu$s | 5,344 $\mu$s | 7,663 $\mu$s | 10,543 $\mu$s |

example if $d1=\infty$ and $d2=\infty$, then $2*d1+d1=C000_{16}$ which can be represented with 16 bits. Using a larger value for $\infty$ could cause the 16-bit value to overflow after the above manipulations are performed.

The time complexity of the SP program is:

$$12+(24+50p+2+7+7+25+13)+ \qquad (1)$$

$$r[24+50p+2+7+23+54+13]+ \qquad (2)$$

$$\sum_{i=1}^{r-1} i[9+13+13]+ \qquad (3)$$

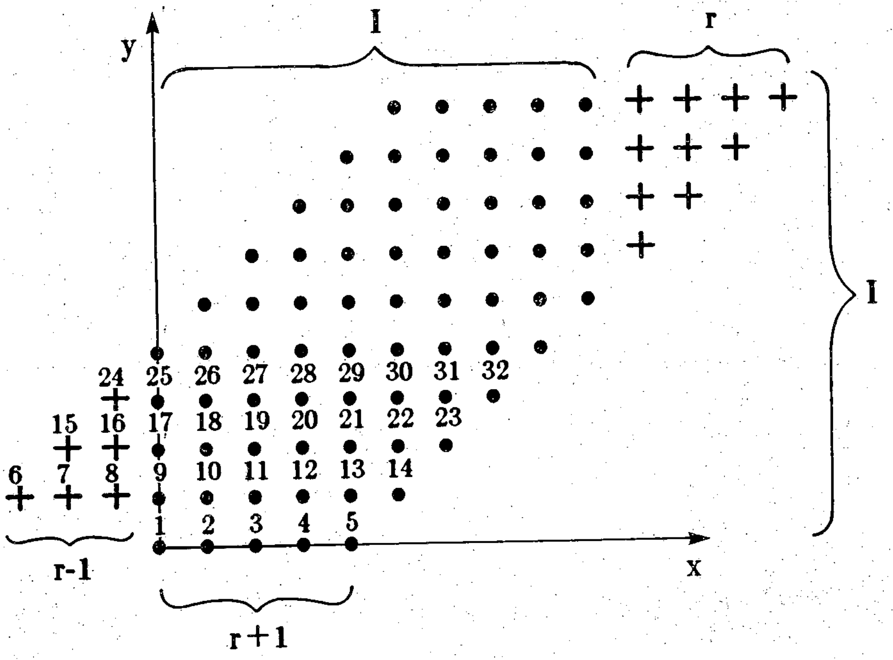
$$r[24+50p+2+15+13+54+13]+ \qquad (4)$$

$$[(I-1)(2r+1)-r-r^2][24+50p+2+16+16+48+44+54+13]+ \qquad (5)$$

$$\sum_{i=1}^{r} i[19+13+13]+ \qquad (6)$$

$$33I+3 \qquad (7)$$

Each number roughly represents the time between two successive labels in the program. Figure 7.5 shows the order in which the distances are computed for I=10 and r=4 and Table 7.8 gives a breakdown of the time spent between adjacent labels. The "."'s in Figure 7.5 are where actual distances are computed and the "+"'s are locations that are "visited" but no distance is computed. A visit to a location means the program sets x and y equal to the coordinates of that location, but the location is not in the warping path. Line (1) in the equation is the time used to initialize the loop counters and compute the special case where $x=0$ and $y=0$ (point 1 in Figure 7.5) Line (2) is the special case where $y=0$ and $x\neq0$ (points 2-5 in Figure 7.5) In general this line is executed $r$ times. Line (3) is the time to skip over the +'s in the lower left triangle. In general there are $r+1$ +'s on the horizontal side of the triangle. Line (4) is the time to compute the special case where $x=0$ and $y\neq0$. Line (5) is the normal case for $x\neq0$ and $y\neq0$. The factor $I-1$ is used because x takes on the values from 0 to $I-1$ with line (2) computing the execution times for $x=0$. The $2r+1$ term in equation (5) is the width of the warping path; the $r+r^2$ term is subtracted to adjust for the time taken into account by lines (3), (4), and (6). Line (6) is the time to skip over the +'s in the upper right triangle. Line (7) is the time used to reset pointers when moving from row y to row y+1.



r=4 I=10

Figure 7.5 Calculation order for accumulated distances of SP DTW program.

Table 7.8 Execution times in cycles between adjacent labels of SP DTW program (x=50p+2+7). The column headings refer to the time complexity equations in Section 7.6.2.

| Line | (1) | (2) | (3) | (4) | (5) | (6) | (7) |
|---|---|---|---|---|---|---|---|
| Times Executed | 1 | r | $\sum_{i=1}^{r-1} i$ | r | $(I-1)(2r+1)$<br>$-r-r^2$ | $\sum_{i=1}^{r} i$ | I |
| dtw: | 12 | | | | | | |
| nextdist: | 24 | 24 | 9 | 24 | 24 | 19 | |
| takediff: | x | x | | x+8 | x+9 | | |
| findA: | | | | | 16 | | |
| findB: | | | | | 48 | | |
| findC: | | | | | 44 | | |
| findG: | | 54 | | 54 | 54 | | |
| nextframe: | 13 | 13 | 13 | 13 | 13 | 13 | |
| newy: | | | | | | | 33 |
| distanceend: | | | | | | | 3 |
| nextpair: | | | 13 | | | 13 | |
| firstrow: | 7 | 23 | | | | | |
| firstcol: | 25 | | | | | | |
| yedge: | | | | 13 | | | |

The simplified time complexity is:

$$12+(78+50p)+r[123+50p]+\sum_{i=1}^{r-1}35i+$$
$$r[121+50p]+[(l-1)(2r+1)-r-r^2][217+50p]+$$
$$\sum_{i=1}^{r}45i+33l+3$$

Table 7.9 gives the execution times for a typical speech recognition system. The SP DTW program is able to execute a match in 74 ms which is 13 matches per second using one PE. A 1,000 word vocabulary can be matched in one second using 77 PEs.

The SP method has little overhead of parallelism because each PE is implementing a serial algorithm. The only parallel construct used is data conditional masking which the program frequently uses for finding the minimum path. The following shows the overhead of using the data conditional mask, and suggests two methods for eliminating the overhead.

The following code performs the same task as the Flock Algol lines 32-35 in Figure A.10, i.e., it stores the minimum of the variables A and B in the variable min.

```
34  WHERE A<B
2       min ← A;
8   ELSEWHERE
2       min ← B;
8   ENDWHERE
```

The numbers on the left are the number of cycles used for each step assuming an 8 MHz MC68000 is used and A, B, and min are stored in registers. The program uses a total of 54 cycles (13.5 $\mu$s). Overlapping the PE and CU instructions by using an instruction queue would not significantly reduce the execution times of these statements since the CU must wait until the PEs have executed the instructions in the queue before enabling the data conditional mask [SiKu82]. The following is the faster method used in Figure A.10.

```
2   min ← A;
26  WHERE B < min
2       min ← B;
8   ENDWHERE
```

Table 7.9 Execution times for serial dynamic time warping (SP).

| Program | DTW | DTW+Rearrange | |
|---|---|---|---|
| **p** | **8** | **8** | **8** |
| r | 6 | 6 | 6 |
| I | 40 | 40 | 40 |
| Number of PEs | 1 | 8 | 8 |
| **NetD** | | **0** | **18** |
| Transfers | 0 | 320 | 320 |
| Cycles | 296,452 | 312,065 | 317,825 |
| Time/Comparison | 74,113 μs | 78,017 μs | 79,456 μs |
| Comparisons/Second | 13 | 13 | 13 |

| Program | DTW | DTW+Rearrange | |
|---|---|---|---|
| **p** | **16** | **16** | **16** |
| r | 6 | 6 | 6 |
| I | 40 | 40 | 40 |
| Number of PEs | 1 | 8 | 8 |
| **NetD** | | **0** | **18** |
| Transfers | 0 | 640 | 640 |
| Cycles | 487,652 | 518,305 | 529,825 |
| Time/Comparison | 121,913 μs | 129,577 μs | 132,457 μs |
| Comparisons/Second | 8 | 7 | 7 |

This requires 38 cycles (9.5 μs), which is 16 cycles less than the first method. The extra cycles are the time needed to push the ELSEWHERE condition on the condition codes stack and to pop it off again. Avoiding the ELSEHERE statement by using the above technique will save 4 μs on the MC68000 when running at 8 MHz.

The following is a serial method to perform the same operation.

```
2    min ← A;
7    IF B < min
2         min ← B;
```

This takes only 11 cycles. A processor using an instruction prefetch may reduce the execution time of the above statements, but its effect will be limited since the second line is a conditional branch which may disrupt the prefetching of instructions. Although this code cannot be used by the parallel DTW program, it does show that the parallel version of finding a minimum takes about 250% longer than the serial version. If the *min* operation, or any other simple operation, is frequently used it should be included in the instruction set of the PEs. Then the PEs could execute the simple function with one instruction rather than using the data conditional masking which requires more time to execute.

A more general approach would be to allow the programmer to define his own instructions, so that he could define simple operations, like the *min* function, as they are needed. On most processors, new instructions are defined by writing microcode, if they can be defined at all. On the MC68000, which is used in the simulations, the microcode cannot be changed. Custom instructions could be implemented by allowing the PEs to execute code out of their own memory while running in SIMD mode. The routines, stored in the local memory of each PE, would be identical in each PE, and would be written so that the execution time of each routine is independent of the data processed. This would take care of the synchronization problems. Then the PEs could perform simple instructions like min without the overhead of data conditional masking.

One other approach, if a custom instruction set were being designed, would be to implement an $M_{CC}$ instruction that works like the $B_{CC}$ instruction on the MC68000. The $B_{CC}$ is a *b*ranch on *c*ondition *c*ode. *cc* can be one of

16 conditions such as, ***less than, greater than,*** etc. The $M_{CC}$ would be a *m*ove on *c*ondition *c*ode. The operation would be to move data from one register to another if the condition is true. Therefore,

```
    2    p_mov    d0,d1; Move data from register d0 to d1.
    2    p_cmp    d1,d2; Compare registers d1 and d2.
    5    p_mlt    d0,d2; Move contents of d0 to d2 if
                       ; d2 is less than d1.
```

would store the minimum of d1 and d2 in d0, without data conditional masking. The minimum, maximum, and absolute value functions are a few of the many functions that could be implemented using the $M_{CC}$ instruction.

### 7.6.3. Simulation of the SIMD DTW Algorithms

Some applications may have more PEs available than there are words in the vocabulary. In cases like this, the SP method may not decrease the execution time of the DTW algorithm as much as wanted since it uses only one PE per DTW match. The parallel parallel (PP) method, discussed in Section 6.4.1.2., uses 2r+1 PEs for each DTW match, therefore decreasing the time needed to do one match. Two alternatives to implementing the PP program are presented. The first, PP1, uses the rearrange routine described earlier to move the data from the output format used by the LTW program to the input format used by the DTW program. Then the PP1 DTW program computes the local distances as they are needed. The second, PP2, uses a variation of the rearrange program which computes the local distances while moving the data. This reduces the amount of data that must be rearranged and stored in each PE. After the data is moved and all the local distances are computed, the PP2 program is executed. The following paragraphs discuss the PP1 program, and the next section covers the PP2 program.

#### *7.6.3.1 PP1*

Figure A.11 is a listing of the PP1 DTW program. The time complexity for the PP1 distance program is:

$$\text{cycles}=58+50p+2+16$$

The time complexity for the PP1 DTW program is:

$$\text{cycles}=4+114+\text{I}[10+\text{dist}+104+2(52+2\text{NetD})+104+ \\ 16+12+16+118+5]+2+44+14\text{r}+6\text{r} \tag{7.1}$$

$$\text{cycles}=164+\text{I}[565+50\text{p}+4\text{NetD}]+20\text{r}$$

where *dist* is the time used by the DTW distance program. The value 118 in equation (7.1) is the time used to run the instruction between labels *findmin* and *incindex* in Figure A.10 Adding up execution times between the labels yields 124 cycles. The six cycles used by the instruction 2 lines before the *incindex* label are not included in the total execution times because it is not normally executed. The *sim68* simulator does not count the execution time if all PEs are disabled. The term 6r is added outside the main loop (the loop starting at the label *nextdist*) to compensate for the few times the statement is executed. Table 7.10 summarizes the execution times for both the PP1 and the PP1+rearrange programs.

In a typical speech recognition system the PP1 program would compare a pair of utterances in less than 16 ms using 13 PEs. The SP requires 80 ms to compute the same comparison using one PE, or it can compare 13 pairs of utterances in 80 ms using 13 PEs. This gives an average of 6 ms per DTW using the SP algorithm with 13 PEs. (All times include the time for the rearrange program.) This means the PP1 program takes about 8/3 times as long as the SP program to execute roughly the same operations. One difference between the SP and PP1 programs is the PP1 uses the interconnection network. If the network delay time is 0, PP1 requires 14 ms per DTW while SP needs 79 ms/13 = 6 ms. Still the PP1 program takes over two times as long to perform a comparison between an unknown and a reference utterance.

The difference is caused by the implementation on the MC68000. The MC68000 has 8 32-bit data registers and 8 32-bit address registers. The SP program stores all of its variables in the data and address registers. The PP1 program uses over 17 variables since it must store the *g* and *d* values for itself and the PEs adjacent to it, plus it must save the old *g* and *d* values for itself and the adjacent PEs. All these variables are stored in memory since there are not enough registers to hold them all. The MC68000 can do a register-to-register move in .5 $\mu$s and a memory-to-memory move in 2.5 $\mu$s, which is 5

Table 7.10 Execution times for parallel dynamic time warping (PP1).

| Program | PP1 DTW | | Rearrange+DTW | |
|---|---|---|---|---|
| **p** | **8** | **8** | **8** | **8** |
| r | 6 | 6 | 6 | 6 |
| I | 40 | 40 | 40 | 40 |
| Number of PEs | 13 | 13 | 13 | 13 |
| **NetD** | **0** | **18** | **0** | **18** |
| **Transfers** | 160 | 160 | 800 | 800 |
| Cycles | 54,884 | 57,764 | 85,537 | 99,937 |
| Time/Match | 13,721 $\mu$s | 14,441 $\mu$s | 21,384 $\mu$s | 24,984 $\mu$s |
| Matches/Second | 72 | 69 | 46 | 40 |

| Program | PP1 DTW | | Rearrange+DTW | |
|---|---|---|---|---|
| **p** | **8** | **8** | **8** | **8** |
| r | 6 | 6 | 6 | 6 |
| I | 40 | 40 | 40 | 40 |
| Number of PEs | 13 | 13 | 13 | 13 |
| **NetD** | **0** | **18** | **0** | **18** |
| **Transfers** | 160 | 160 | 480 | 480 |
| Cycles | 38,884 | 41,764 | 54,497 | 61,137 |
| Time/Comparison | 9,721 $\mu$s | 10,441 $\mu$s | 13,624 $\mu$s | 15,784 $\mu$s |
| Comparisons/Second | 102 | 95 | 73 | 63 |

times as long. In general each memory access takes about 1 $\mu$s more than each register access. Since the memory-to-memory move instruction references memory once to read the value and again to write it to a new location, it takes 2 $\mu$s longer than the register-to-register move. Therefore the PP1 program is slower than the SP program partially because it uses inter-PE transfers, but mainly because the MC68000 does not have enough registers to hold all the PP variables. Some variables must be stored in memory which is slower to access.

This provides another design feature. The processor used in each PE of an SIMD machine for DTW should have more registers than the 8 provided by the MC68000. This would allow more data to be quickly accessed without using main memory.

### *7.6.3.2. Simulation of the DTW Algorithm – PP2*

The time the rearrange program uses to move data between PEs is all parallel overhead since the data movement is not needed on a serial processor. The PP2 program attempts to reduce the rearrange time by computing the local distance as the data is being moved. The rearranging time should be reduced since two frames of $p$ coefficients each are combined into one distance score after the calculation. The next section presents the ***distance*** program which computes the local distances while moving the data. The section after that presents the PP2 program.

#### *7.6.3.2.1. The Distance Program*

Figure 7.6 is the Flock Algol algorithm for computing the local distances. It uses max(p,2r+1) PEs and the input data is arranged so PE k contains coefficient k of frame i for $0 \leq k < p$ and $0 \leq i < I$, where I is the total number of frames.

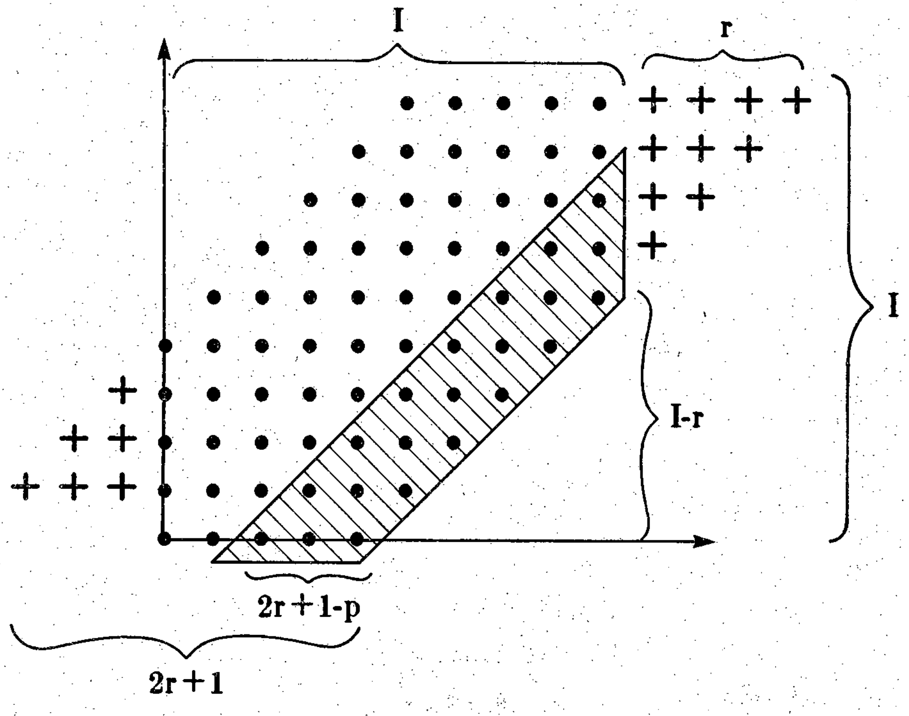
The distance routine computes the local distance between known frame i and unknown frame j in PE 0 through PE p−1 and stores the resulting data in PE i−j. Figure 7.7 represents the local distances with "."s for r=4, p=6, and I=10. The dots outside of the shaded area are are stored in PEs 0 through p−1. The dots in the shaded area are stored in PEs p through 2r+1. Since

```
      /*
                  Algorithm Name:   distance (PP2)
                  Section:          7.6.3.2.1.
                  Machine:          SIMD
                  Function:         This program moves data around and computes
                                    the local distances in preperation
                                    for the DTW program.
                  Number of PEs:    2r+1
                  Parameters:       r, the width of the warping path.
                                    p, the number of coefficients per frame.
                                    NetD, the network delay time.
                                    I, the number of frames per utterance.
                  Input:            known[x] contains coefficient i in PE i of
                                    input vector x.
                                    unknown[y] contains coefficient i in PE i of
                                    input vector y.
                  Output:           d[dptr] contains the local distances.
                                    d[0] contains the first distance needed by the
                                    PE it is stored in for the DTW program.
                                    d[1] contains the next distance, and so on.

*/

Line  Time in μs
 1                PROCEDURE distance
 2    .5          LADDR = ADDR − r  /* Logical address, PE ae numbered −r to r     */
 3    4           FOR i ← 1 TO r/2
 4    13              WHERE |LADDR| > i DO
 5    2                   d[dptr] ← ∞;
 6                        dptr ← dprt + 1;
 7    4               ENDWHERE
 8
 9    8           FOR y ← 0 TO I−1
10    3               FOR x ← −r TO r
11    5                   IF y+x ≤ O AND y+x ≤ 2I−2
12    10.75                   sum ← (known[x] − unknown[y]) ²;
13    3                       FOR k ← 0 TO logN−1
14    3                           USE Cube(k);
15    1.5                         DTRIN ← sum;
16    NetD                        TRANSFER;
17    1.5                         sum ← sum + DTROUT;
18
19                /*
20                            The coefficients are in PE 0 - PE p
21                            and the distance score is needed in PE i
22                            where i > p.  Use the Shift function to
23                            move the data from PE 0 to the desired PE.
24                */
```

Figure 7.6 Algorithm to compute local distances and move data. Execution time are for an 8 MHz MC68000.

```
2                                IF x+r > p
3                                    USE Shift +x+r
4+NetD                               TRANSFER sum

6.5                              WHERE x+r = ADDR          /* Enable PE    */
1                                    d[dptr] ← sum; /* that will use */
                                     dtpr ← dptr + 1;      /* the distance */
2                                ENDWHERE                       /* score.*/

3                     FOR i ← 1 to r/2
1                         d[dptr] ← ∞;
                          dptr ← dptr + 1;
```

Figure 7.6 (Continued)



r=4 I=10

Figure 7.7 Calculation order for accumulated distances of SP DTW program. PEs in shaded area do not start with input data.

the input data is stored in only PEs 0 through p−1, and the distance scores are computed in the same PEs, the distance scores represented by the shaded area in Figure 7.7 must have their scores transferred from a PE outside of the shaded area.

A typical speech recognition system has *p=8* and *r=6,* so 2r+1 is > p and extra transfers are needed to get the data from a PE outside of the shaded area to the proper PE in the shaded area. Lines 25-27 of Figure 7.6 handle this case. If *p=16,* as with some high quality speech recognition systems, p > 2r+1 and lines 25-27 are not ever executed.

The time complexity for the distance routine is:

$$\text{cycles} = 12+85\lfloor r/2 \rfloor+2+12+ \quad (1)$$

$$[I(2r+1)-r-r^{2][20}+43+4+(NetD+31)\log p+2+9+38+13]+ \quad (2)$$

$$(9+7+13)\sum_{i=1}^{r-1} i+ \quad (3)$$

$$(19+7+13)\sum_{i=1}^{r} i+ \quad (4)$$

$$[(2r+1-p)(I-r)+\sum_{i=1}^{2r-p} i](25+NetD+1)+ \quad (5)$$

$$30I+1+ \quad (6)$$

$$6+9\lfloor r/2 \rfloor+2 \quad (7)$$

assuming p < 2r+1. Table 7.11 gives the breakdown on how the time is spent between each label in the assembly language program, given in Figure A.12, for each line of the time complexity. Line (1) is the time used to initialize some variables and store infinity scores in those PEs outside the warping path during the first r/2 loops of the DTW program (see Figure 7.7). Line 2 is the main loop of the program, during which the distances are computed. Line (3) is the time used for visiting the "+"'s in the lower left triangle. Line (4) is the visit time for the upper right triangle. Line (5) is the time used to move data from PE 0 to PE i when i ≥ p. The "."'s in the shaded area of Figure 7.7 represent the time in which this is done. Line (5) can be omitted from the time complexity if p > 2r+1. Line (6) is the time used to prepare to use a new unknown frame. Line (7) is the time needed to pad the d[] array with infinity values for those PEs outside the warping path.

Table 7.11 Execution times in cycles between adjacent labels of PP2 DTW program. The column headings refer to the time complexity equations in Section 7.6.4.1. ($y=\log p(\mathrm{NetD}+31)+2$, $x=85\lfloor r/2\rfloor+2+12$, $z=9\lfloor r/2\rfloor$)

| Line | (1) | (2) | (3) | (4) | (5) | (6) | (7) |
|---|---|---|---|---|---|---|---|
| Times Executed | 1 | $I(2r+1)$<br>$-r-r^2$ | $\sum_{i=1}^{r-1} i$ | $\sum_{i=1}^{r} i$ | $(2r+1-p)(I-r)$<br>$+\sum_{i=1}^{2r-p} i$ | I | 1 |
| distance: | 12 | | | | | | |
| pad: | x | | | | | | |
| nextdist: | | 20 | 9 | 19 | | | |
| takediff: | | 43 | | | | | |
| notinf: | | 4 | | | | | |
| dloop: | | y | | | 25+NetD | | |
| gotd: | | 9 | | | | | |
| easy: | | 38 | | | | | |
| nextframe: | | 13 | 13 | 13 | | | |
| newy: | | | | | | 30 | 6 |
| pad2: | | | | | | | z |
| nextpair: | | | 7 | 7 | | | |

Simplified, the time complexity for the PP2 distance program is:

$$35+94\lfloor r/2\rfloor+30I+29\sum_{i=1}^{r-1} i+39\sum_{i=1}^{r} i+$$

$$[I(2r+1)-r-r^2][129+(NetD+31)\log p]+$$

$$[(2r+1-p)(I-r)+\sum_{i=1} 2r-pi][25+NetD]$$

Table 7.12 gives execution times for a typical speech recognition system.

### *7.6.3.2.2. The PP2 DTW Program*

After the distance program is executed, the DTW program is run. The PP2 DTW program is identical to the PP1 program except the PP2 program does not call a routine to compute the local distances. Instead, it finds the distances in an array, already computed by the distance program. Figure A.12 lists the DTW program along with the main and distance programs. The time complexity for the PP2 DTW program is:

$$4+76+I[106+2(52+2NetD)+104+16+12+16+124+5]+2+44$$

$$126+I[487+4NetD]$$

Table 7.13 summarizes the execution times for a typical speech recognition system. The PP2 program can match 24 pairs of utterances in one second using 13 PEs. The PP1 program is able to match 63 pairs in the same time using the same number of PEs. The execution time has increased because, 1) the number of transfers has increased, and 2) less parallelism is used.

It had been expected that the number of cycles would decrease because two frames of coefficients were being combined into one distance score, which would take less time to pass through the network. This did not happen since in PP2, *p* PEs are used in parallel to compute each local distance. The distance calculation requires log p transfers to sum the square of the differences between coefficients (lines 13-17 in Figure 7.6). This is done once for each distance score, yielding a total of approximately I(2r+1)log p transfers. The rearrange program needs transfers to move the LPC coefficients to the appropriate

Table 7.12 Execution times for distance calculations for PP2.

| Program | PP2 distance | | | |
|---|---|---|---|---|
| **p** | **8** | **8** | **16** | **16** |
| r | 6 | 6 | 6 | 6 |
| I | 40 | 40 | 40 | 40 |
| Number of PEs | 13 | 13 | 16 | 16 |
| **NetD** | **0** | **18** | **0** | **18** |
| Transfers | 1,614 | 1,614 | 1,912 | 1,912 |
| Cycles | 113,387 | 142,439 | 123,705 | 158,121 |
| Time/Compairson | 28,347 $\mu$s | 35,609 $\mu$s | 30,927 $\mu$s | 39,531 $\mu$s |

Table 7.13 Execution times for dynamic time warping program PP2.

| Program | PP2 DTW | | distance + DTW | |
|---|---|---|---|---|
| **p** | | | **8** | **8** |
| r | 6 | 6 | 6 | 6 |
| I | 40 | 40 | 40 | 40 |
| Number of PEs | 13 | 13 | 13 | 13 |
| **NetD** | **0** | **18** | **0** | **18** |
| Transfers | 160 | 160 | 1,774 | 1,774 |
| Cycles | 19,606 | 22,486 | 132,993 | 164,925 |
| Time/Comparison | 4,902 μs | 5,622 μs | 33,249 μs | 41,232 μs |
| Comparisons/Second | 204 | 177 | 30 | 24 |

| Program | distance + DTW | |
|---|---|---|
| p | **16** | **16** |
| r | 6 | 6 |
| I | 40 | 40 |
| Number of PEs | 16 | 16 |
| **NetD** | **0** | **18** |
| Transfers | 2,07 | 2,072 |
| Cycles | 143,311 | 180,607 |
| Time/Comparison | 35,828 μs | 45,152 μs |
| Comparisons/Second | 24 | 22 |

destinations and uses *p* transfers per frame for a total of Ip transfers. If p is greater than (2r+1)log p, the distance program will use fewer transfers.

### 7.6.4. Summary

The previous sections have presented three programs for dynamic time warping. The serial parallel (SP) program broadcasts the unknown input utterance to all PEs and each PE executes a serial DTW program to compare it to a known utterance. The two parallel parallel (PP) programs use 2r+1 PEs to perform each match. The PP1 program moves the data to all PEs, then computes the local distances as they are needed during the DTW program. Each local distance is computed in a single PE, however, all 2r+1 PEs can be computing a different local distance simultaneously. The PP2 program computes the local distances as the data is being moved to the PEs. p PEs are used to compute one distance score. All local distances are computed before the DTW program starts executing.

The SP program is the fastest of the three. It can match 169 pairs of utterances (consisting of 40 frames of 8 coefficients each) in one second using 13 MC68000's running at 8 MHz. The PP1 program is the next fastest matching 63 pairs per second under the same conditions, and PP2 is slowest matching 24 pairs per second. Tables 7.9, 7.7, and 7.12 summarize the execution times for a typical speech recognition system. If faster processing rates are needed, the SP program can use N PEs to compute N comparisons simultaneously. The PP programs can use sets of 2r+1 PEs in parallel so that N PEs can compute $\lfloor N/(2r+1)\rfloor$ DTW comparisons in parallel.

The SP program was fastest since it required fewer data transfers between PEs (none at all after the DTW starts executing except for the recursive doubling needed to find the minimum distance score), and it uses fewer variables than the PP programs. The SP program stores all of its variables in registers, while the PP programs have more variables than registers, so some variables are stored in memory. The MC68000 uses four more cycles to reference memory than a register; therefore the PP programs, while executing about the same number of operations, run slower than the SP program. The PP programs could run faster if the processor in the PE had more registers (at least 18 data registers), or faster memory access.

The PP1 program is the next fastest DTW program since the PP2 distance program uses p PEs to compute one distance score in parallel. The PP1 DTW program uses 2r+1 PEs to compute 2r+1 distance scores serially within each PE. Since $p < 2r+1$ in the typical system, the distance program has $2r+1-p$ PEs idle when computing local distances. Therefore the PP1 DTW program makes better use of the available parallel computing power.

Although the SP program is a serial program running in each PE, if the program is being run under SIMD control, data conditional masking must be used in each PE to find the minimum of two registers. Data conditional masking is a time consuming operation and should be avoided if possible. It would not be needed if the MC68000 could execute a "minimum" instruction directly, but is is unrealistic to expect the processor to have every possible "handy" instruction in its instruction set. A better approach would be to use a processor with programmable microcode or use a custom processor. A library of commonly used microcode operations could be available to the programmer so simple operations like finding the minimum of two register could be executed with one instruction. The would reduce the number of times data conditional masking is used, and should reduce the execution time.

The MC68000 does not have programmable microcode, but this feature could be simulated by letting each PE execute code out of its own memory while running in SIMD mode. Again, a library of commonly used functions could be stored in the local memory of each PE. Each function would be written so the execution time was independent of the data processed so all processors would execute the instruction in the same amount of time.

The DTW programs all used the *Shift* ± *1* transfer functions, the PE 0 to CU link, and the CU broadcast. The PP2 program used the *Cube* transfers and the *Shift* $+n$ transfer function for $p \leq n \leq 2r$.

Overall, the SIMD architecture implemented with MC68000 is well suited for the DTW programs.

## 7.7. SIMD Machine Based Isolated Word Recognition System

Previous sections in this chapter have presented programs for performing various speech recognition tasks. This section shows how these programs are assembled together to perform the function of the speech recognition system shown in Figure 4.1. The parameters listed on Figure 4.1 are for processing telephone quality speech. Table 7.14 lists parameters for telephone quality and high quality speech processing.

The following section presents the main program which calls each of the speech processing programs as they are needed, and contains the endpoint detection program. The main program contains the endpoint detection program since the LPC program is not called until after the begining of an utterance is found, and the LTW and DTW programs are not called until after an entire utterance is found. Section 7.7.2 discusses the data allocation used by each program, and Section 7.7.3 discusses the execution times of the entire system. Section 7.7.4 discusses the size of the input buffers needed to hold the incoming speech samples while the DTW program is executing. Section 7.7.5 summarizes Section 7.7. Figure 7.8 is a Flock Algol algorithm for the main program in the speech recognizer and Figure A.13 is the MC68000 program.

### 7.7.1. Endpoint Detection

The endpoint portion of the main program finds the endpoints based on the energy in each frame as discussed in Section 4.5. The program does not use the zero crossing (ZX) rate discussed in Section 4.5 since Lamel [LRRW81] states it is not always effective.

The endpoint program checks the energy of the current frame by having PE 0 send its autocorrelation coefficient R[0] to the CU after the autocorrelation program is executed. If the energy is greater than *lothresh**, the low

*Unlike the method used in [RaSa75], *lothresh* and *hithresh* are not adaptive. They are constants that are set before the program is executed.

Table 7.14 Parameters for speech recognition systems.

| | Telephone Quality | High Quality | SIMD System |
|---|---|---|---|
| Sample Rate | 6.67 KHz | 20 KHz | 20 KHz |
| Bits per Sample | 8 | 16 | 16 |
| LPC Coefficients | 8 | 16 | 16 |
| Bits per Coefficient | 16 | 16 | 16 |
| Vocabulary Size (words) | 10-1,000 | 10-1,000 | 1,000* |

*The number of words that can be matched in less than one second.

```
/*
            Algorithm Name:   main
            Machine:          SIMD
            Function:         This is the main routine. It calls filter()
                              and auto() to preemphize the signal and find
                              the autocoerrelation coefficients. If R(0)
                              (the energy) is greater than lothresh, it calls
                              lpc(). This main routine also does the
                              endpoint detection. After an utterance is
                              detected, ltw() and dtw() are called.
            Number of PEs:    100
            Parameters:       N, the frame size.
                              autocoef, the number of autocorrelation coefs.
                              r, the width of the warping path.
                              p, the number of LPC coefficients.
                              NetD, the network delay time.
                              I, the number of frames per utterance.
                              VOCABSIZE, the size of the vocabulary.
            Input:            Sample i mod N is is PE i.
            Output:           One distance score per PE.
            Variables Used:
                  i:                Index to current input sample.
                  found:            = TRUE if utterance has been found.
                  lothresh:         Lower threshold.
                  hithresh:         Upper threshold. See section 4.6??
                  M:                Index to current input frame.
                  input[x]:         Input samples, PE i contains sample
                                    i of frame x.
                  filout:           Filtered output for filter program.
                                    PE i contains sample i of frame.
                  R[]:              Autocorrelation coefficients,
                                    all coefficients in all PEs.
                  lpcout[x]:        LPC coefficients PE i contains
                                    coefficient i of frame x.
                  ltwout[x]:        Output utterance from LTW program.
                                    PE i contains coefficient
                                    i from frame x.
                  shuffout[x]:      Output of shuffle program.
                                    All PEs contain all coefficients
                                    from all frames.
                  lib[x]:           Library of known utterances.
                                    PE i contains all coefficients from
                                    all frames for utterance x.
                                    Each PE contains a different
                                    utterance.
                  scores[i]:        Output scores from all DTW matches,
```

Figure 7.8 Flock Algol algorithm for isolated word recognition. Contains endpoint detection algorithm and calls the filter, autocorrelation, LPC, LTW, and DTW algorithms.

```
                      contained in PE r.*/
PROCEDURE main
found ← FALSE;
M ← 0;
i ← 0;
WHILE(TRUE)
/*
        Take one input frame and filter.
*/
        filter(input[i], filout);
/*
        Find autocorrelation coefficients
        for the input frame.
*/
        auto(filout,R[]);
/*
        Take the energy R[0] in PE 0 and
        pass to the CU for endpoit detecton.
*/
        TOCU ← R[0];
        energy ← FROMPE0;

/*
        If the energy is greater than the
        low threshold, compute the LPC
        coefficients and save in lpcout[].
*/
        IF energy > lothresh
            IF energy > hithresh
                found ← TRUE;
            lpc(R[], lpcout[M]);
            M ← M + 1;
/*
        Otherwise, this may be the end of
        an utterances, or between utterances.
*/
        ELSE
            IF found
/*
                It's the end of an utterance.
                Do the LTW.
*/
                ltw(lpcout[],ltwout[],M,40);
/*
                For each word in the vocabulary,
                do a DTW and save the scores.
*/
                shuffle(ltwout[],shuffout[]);
                FOR j ← 0 TO VOCABSIZE−1
                    score[j] ← dtw(shuffout[],lib[j][]);
```

Figure 7.8 (Continued)

```
                        found ← FALSE;
/*
                  It is between words, throw away the saved
                  LPC frames.
*/
                  ELSE
                        M ← 0;
/*
            Use the next frame.
*/
            i ← i + 1;
```

Figure 7.8 (Continued)

threshold, the main program calls the LPC program to compute the LPC coefficients and saves them in an array. If the energy is greater than *hithresh* the *found* flag is set to TRUE. If the energy is less than *lothresh* and the *found* flag is TRUE, the LTW program is called, followed by the rearrange program and the SP DTW program. If the energy is less than *lothresh* and the *found* flag is FALSE, the saved coefficients are discarded.

### 7.7.2. Data Allocation

When combining SIMD machine programs the output data arrangement of one program must match the input data format of the program that follows. The programs presented earlier in this chapter were written so their data formats matched.

The filter program in Section 7.2 expects the input data to be stored with sample i mod N in PE i for $0 \leq i < N$. Where N is the total number of PEs, and mod is the modulus function. The autocorrelation program in Section 7.3 takes the input data in the same format the filter program outputs and stores its output so all autocorrelation coefficients are in all PEs. The LPC program in Section 7.4 uses just 8 PEs, and expects all the autocorrelation coefficients in each PE, just as the autocorrelation program left it. The LPC program leaves LPC coefficient i in PE i for $0 \leq i < p$. The next task in Figure 4.1 is the endpoint detection. The endpoint detection program does not process the data as the other programs do. Instead, it decides whether or not an input utterance has been detected. If it has, the data is sent to the programs which follow. Otherwise, the data is discarded. The LTW routine is called after the endpoint routine has detected an utterance. The LTW routine expects PE i to contain coefficient i of frame j for $0 \leq i < p$ and $0 \leq j < I$. This is the arrangement that the LPC program outputs. The output data arrangement of the LTW program is the same as the input data arrangement.

The SP DTW program needs all frames of the unknown utterance stored in all PEs. This is not the format output by the LTW. The *rearrange* routine moves the data from the arrangement output by the LTW program to the arrangement the DTW program uses as input.

When running the DTW program, each PE contains W/N known utterances where W is the total number of utterances in the vocabulary, and N is the number of PEs. The SP DTW program is executed W/N times and the distance scores are accumulated in the *scores* array in each PE.

### 7.7.3. Execution Times

To process high quality speech in real time the system must meet the specifications in Table 7.14. Table 7.4 shows that if p=8 (not p=16 as shown in Table 7.14) and N=100, the filter, autocorrelation, and LPC programs can process data at 29 KHz. Table 7.9 shows that the SP DTW program can compare 12 utterances per second using one PE which is 1,000 utterances per second using 77 PEs. These two facts show that the SIMD parallel machine can easily process high quality speech in real time. The only problem is the filter, autocorrelation, and LPC programs *and* the DTW program must execute within the allowed amount of time. Figure 7.9 shows the time and number of PEs used for each task in the system. The filter and autocorrelation programs process all input data. If the energy is below the lower threshold, the LPC program is not run. Frames 1 and 2 in Figure 7.9 did not exceed the threshold. Frames 3 through I−1 did, and the LPC coefficients are found for each of them. Frame I was below the low threshold which marks the end of the utterance. The LTW program then is executed. During this time, the input data is being saved in a buffer since the PEs are not running the filter and autocorrelation programs.

After the LTW program, the data is rearranged so all PEs contain the unknown input utterance. Finally, the SP DTW program is executed in all 100 PEs. In the end 100 distance scores are computed and the smallest score comes from the known utterance that best matches the unknown input utterance.

Figure 7.9 shows that most of the system time is spent executing the filtering, autocorrelation, and LPC programs. For a typical utterance with 40 frames, 40(1.8+.16) = 136 ms are spent computing the LPC coefficients from the speech samples. The DTW program uses 79.4 ms for both the rearrange and SP programs. Since the LPC programs uses only 8 PEs, 92 PEs are idle during 64 ms of the LPC computation time. These idle PEs can be used if

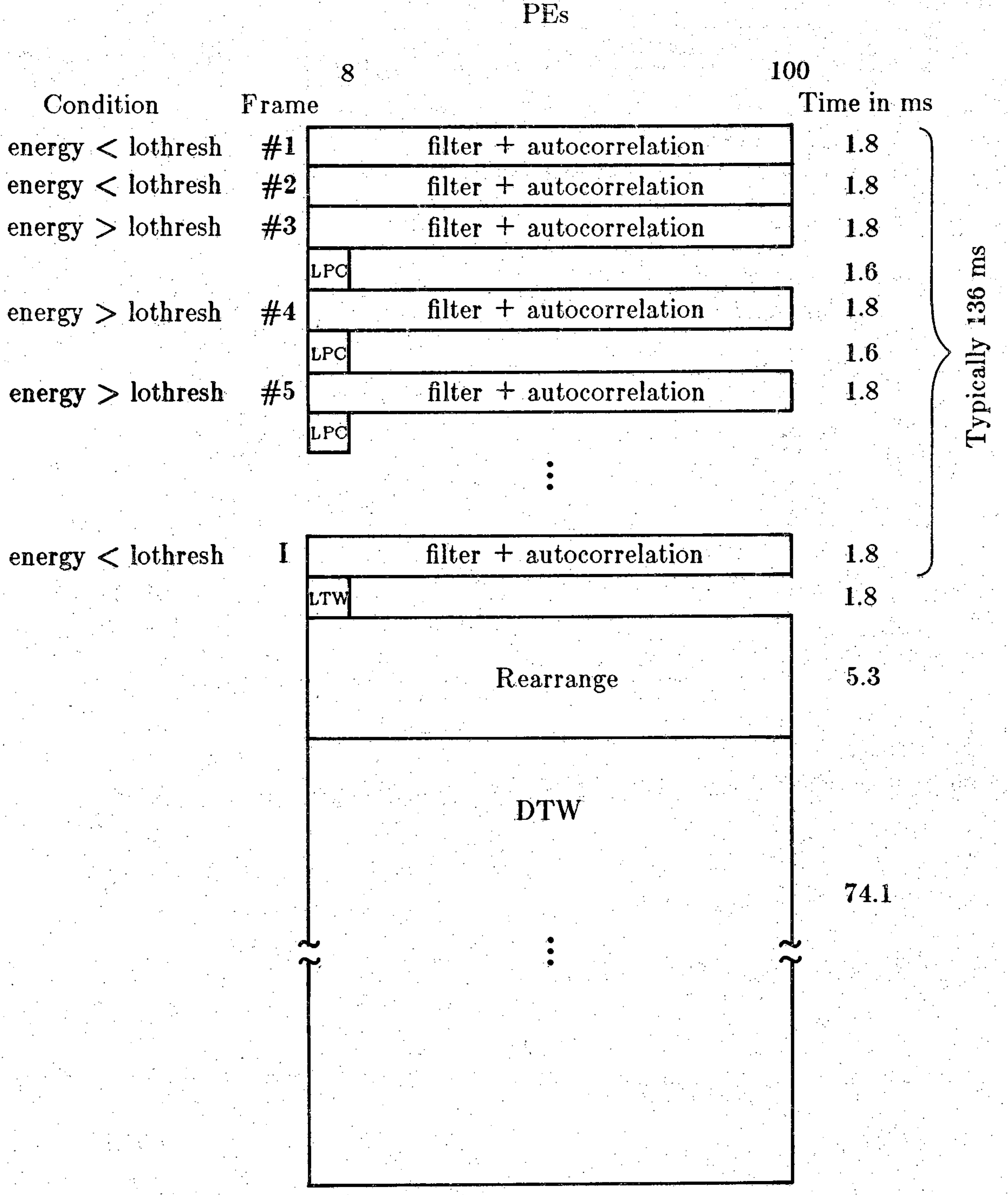


Figure 7.9 Time and PE usage for the parallel isolated word recognition system.

several frames of LPC coefficients are computed in parallel. To do this, the autocorrelation program would leave the first frame of coefficients in PEs 0 through p−1. The LPC program would not be executed as described above; instead the autocorrelation program would be run again. The autocorrelation coefficients from the second run would be stored in PEs p through 2p−1. The would be repeated with the autocorrelation coefficients from frame i stored in PEs ip through (i+1)p−1. Then the LPC program could be run and it would compute $\lfloor N/p \rfloor$ frames of LPC coefficients simultaneously where N is the number of PEs. If this approach were used on the system in Figure 7.9, the filter, autocorrelation, and LPC execution time would be reduced to 78.4 ms not including the time to move data from PEs 0 through p−1 to PEs ip through (i+1)p−1. Although this approach will increase the throughput, it will also increase the delay between the time the speech enters the system and the time LPC coefficients are computed. This is because the computation of the LPC coefficients of frame 0 must wait until the autocorrelation coefficients of frame $\lfloor N/p \rfloor$ are computed. Such a delay is undesirable for real-time processing.

The DTW program could execute in fewer cycles with more PEs if needed. For a 1,000 word vocabulary, the area in Figure 7.9 will be constant, so adding more PEs will decrease the execution time, and removing PEs will increase the execution time. Increasing the execution time will delay the processing of new input samples, which would have to be buffered while the DTW program is running. The next section discusses the effects of the DTW execution time on the input buffer size.

### 7.7.4. Buffering the Input Data

After executing the DTW program, approximately 80 ms have passed since the last input frame was processed. During this time 1,600 new samples will arrive if the sampling rate is 20 KHz. The input data is spread among 100 PEs, so each PE needs a 16 16-bit word buffer to hold the new data while executing the DTW program. Each additional 100 utterances added to the vocabulary require 15 more 16-bit words of buffer space, so the 1,000 word vocabulary needs 151 16-bit words of storage in each PE to hold the new input samples that arrive while the DTW program is running.

The filter, autocorrelation, and LPC programs can process data at 29 KHz when p=8, while high quality speech samples arrive at 20 KHz, therefore the system cam empty the buffer at a rate of 9 KHz. The 100 utterance system takes 178 ms to catch up, while the 1,000 utterance system takes 1,690 ms. Both of these times assume the energy is greater than the low threshold and the LPC coefficients are computed for each frame. If the energy is less than the low threshold, the endpoint routine does not call the LPC program. The sampling rate for the filtering and autocorrelation programs is 55 KHz (Table 7.3), therefore the buffer will empty at 35 KHz. The 100 utterance system will catch up in 45 ms, while the 1,000 utterance system will need 431 ms. Most real-time speech recognition systems can tolerate a delay of 431 ms.

### 7.7.5. Summary

Although the SIMD speech recognition system can process data at 20 KHz and have a 1,000 utterance vocabulary, a buffer is needed to hold the input samples as the DTW program is run, and the utterances must be spaced far enough apart so that a subsequent utterance does not end before the buffers are emptied. Table 7.15 summarizes the buffer requirements for p=8. The buffer requirements were not computed for p=16 since the filer+autocorrelation+LPC programs can process at most 14K samples per second when NetD=18, and 19K samples per second when NetD=0.

This chapter has shown that an SIMD machine using a current technology processor in each of its PEs and CU can process high quality speech in real time. The next section gives concluding remarks and describes the strengths and weaknesses of the SIMD machine for speech processing.

Table 7.15 Buffer requirements for SIMD speech recognition system. p=8, NetD=18, I=40, input sample rate = 20 KHz.

| Vocabulary Size | 100 | 1,000 |
|---|---|---|
| Calls to DTW | 1 | 10 |
| DTW Time | 80 ms | 745 ms |
| Samples Buffered | 1,600 | 15,000 |
| PE Buffer Size | 16 | 150 |
| Catch Up Time with LPC | 178 ms | 1,670 ms |
| Catch Up Time without LPC | 45 ms | 431 ms |

## 7.8. Conclusions

Designing a parallel processor is difficult without knowing the types of program it will run. This chapter has presented a parallel speech recognition system based on an SIMD machine. The experience gained in programming the SIMD machine to recognize isolated words will help in refining the SIMD machine design for speech recognition. The following sections discuss the different parts of the SIMD machine and give details as to which features it should have for real-time speech recognition.

### 7.8.1. The Processor

Each PE and the CU contain a processor. *Sim68* simulated each processor as an MC68000 microprocessor, which proved to be well suited for the typical isolated word recognition system presented in Chapter 4. The following sections discuss what was good about the MC68000, and what improvements could be made if a custom processor were used.

#### *7.8.1.1. Data Size and Type – 16-bit signed fixed point*

Most speech data can be represented as a 16-bit signed integer, therefore the processor should operate on 16-bit data. The autocorrelation LPC and LTW routines used some 32-bit values, so 32-bit addition should also be implemented.

The LPC, LTW, and DTW routines could have used floating point operations, but they were able to be implemented using only fixed point operations. Adding floating point operations would make writing some of the programs easier and might reduce the execution times of the LPC and LTW programs.

Some DTW programs use a distance measure which requires taking the logarithm of a value [Itak75]. The logarithm function can be approximated using fixed-point arithmetic, but this places a burden on the programmer. A system using such a distance measure may benefit from having hardware floating-point operations since it makes the machine easier to program.

### *7.8.1.2. Internal Registers – At Least 18 Data Registers*

The MC68000 has 8 32-bit data registers. Comparing the SP and PP1 DTW programs showed that more registers could be used. The SP program has only a few variables and keeps them all in registers. The PP1 program has 18 variables, which must be stored in memory. Although the two program execute similar code, the SP program takes half the time of the PP1 program because it did not reference variables in memory as often. For the speech recognition system used here, at least 18 data registers are needed since the PP1 program uses 18 variables.

### *7.8.1.3. Memory Size – 2K bytes*

Table 7.16 summarizes the memory requirements for each of the programs in the speech recognition system. Many of the programs can store all their variables in the internal registers, therefore they require no PE memory. The total memory usage for the CU is 1,680 bytes and each PE uses 352 bytes. The main routine passes the data to the other routines by using pointers, therefore most routines use little PE memory, while the main routine (and endpoint) uses the most PE memory.

A CU memory size of 2K bytes and a PE memory size of 512 bytes should be enough for the proposed speech recognition system. Using 512 bytes for the PE memory allows 352 bytes for the variable, and 160 bytes for buffer space.

### *7.8.1.4. Instruction Set – Add* $\mathrm{M}_{\mathrm{CC}}$

The instruction set of the MC68000 is well suited for speech signal processing since it is a 16-bit processor. The most important operations are the 16 and 32-bit signed additions and subtractions, and the 16 by 16-bit signed multiply and the 32 by 16-bit signed divide.

Table 7.16 Memory usage, in bytes, for SIMD based isolated word recognition system.

| | CU Program | PE Data |
|---|---|---|
| filter | 112 | 4 |
| auto | 200 | 0 |
| LPC | 372 | 6 |
| main* | 352 | 342 |
| LTW | 148 | 0 |
| rearrange | 108 | 0 |
| DTW | 388 | 0 |
| Total | 1,680 | 352 |

*Contains the endpoint routine.

The need for data conditional masking could be reduced if a new instruction called $M_{CC}$ were implemented. The $M_{CC}$ instruction is like the $B_{CC}$ instruction which *b*ranches when a *c*ondition *c*ode is true. The $M_{CC}$ instruction would *m*ove data from one register to another when a *c*ondition *c*ode is true. Finding the minimum of two variables takes 9.5 $\mu$s using data conditional masking. The $M_{CC}$ instruction could reduce this to about 3 $\mu$s.

### 7.8.2. Inter-PE Communication – Cube, Shift(± 1), and Broadcasts

Table 7.17 shows the inter-PE communication usages for each of the programs. The *Shift(± 1)* and *Cube* interconnection functions are frequently used by the programs and should be implemented with hardware so they will transfer quickly. The *Perm* function is used only by the LPC routine and does not need a hardware implementation since it is infrequently used.

The broadcasts are all performed by the CU using self modifying code, which requires no special hardware. The TOCU path from PE 0 to the CU is needed by the endpoint routine so the CU can make conditional branches based on the data in the PEs. The rearrange program uses the TOCU path to broadcast data from PE 0 to all PEs.

### 7.8.3. Masking – Data Conditional

Of the two different masking techniques discussed in Section 2.3, the speech recognition system programs used only the data conditional mask. In all but the DTW program, general PE masks could have been used instead of the data conditional masks. The data conditional masks were used since it was clearer which set of PEs were being enabled. In many cases, general PE masks will execute faster than data conditional masks because they can be computed once at compile time. The data conditional masks; however, must be computed at run time, once for every time the mask is used. Table 7.18 summarizes the times the data conditional mask is used and gives the time, in cycles, it takes to set-up the data conditional mask and the time taken by the statements affected by the mask. The LPC program is the only program that used the ELSEWHERE mask, and its times are indicated by 8/91 which mean the

Table 7.17 Inter-PE communication used by SIMD machine.

| | Broadcasts | Transfers | TOCU |
|---|---|---|---|
| filter | | Shift( + 1) | |
| auto | | Shift(± 1), Cube | |
| LPC | | Cube, Perm | |
| endpoint | | | yes |
| LTW | yes | | |
| shuffle | yes | | yes |
| DTW | | Shift(± 1) | |

Table 7.18 Data conditional masking time in cycles.

| Program | Set Up<br>Time | Statment<br>Execution Time |
|---|---|---|
| filter | 36 | 14 |
| auto | 42 | 37 |
| LPC | 34 | 51 |
|  | 50 | 8/91 |
|  | 34 | 65 |
| DTW | 34 | 2 |
|  | 34 | 2 |
|  | 34 | 2 |

WHERE condition takes 8 cycles and the ELSEWHERE takes 91 cycles. The table shows that except for the LPC program, the set up time for the data conditional mask is longer than the time taken by the statements affected by the mask. The $M_{CC}$ instruction (described earlier) could be used in all but the LPC program instead of the data conditional mask. This would reduce the execution times.

### 7.8.4. MC68000 Clock Rate – 8 MHz

All the instruction timings presented have assumed an 8 MHz clock rate. Some versions of the MC68000 can run using a 12.5 MHz clock rate. This clock rate with a no wait state memory will cause the programs to run 50% faster. Although the proposed system can run in real time with the 8 MHz clock, the faster clock rate will allow changes in the system (such as increasing the number of LPC coefficients) and still run in real time.

### 7.8.5. Number of PEs – 100

Table 7.19 summarizes the number of PEs used by each program in the parallel word recognition system. By using 100 PEs, the MC68000 based SIMD machine is able to implement a typical speech recognition system in real time. The value of 100 was chosen because

1) it is the maximum number of PEs that can be used by the autocorrelation program, and

2) the DTW program can compare 1,000 utterances pairs in 0.8 seconds.

The number of PEs used by the autocorrelation, LPC, LTW, and rearrange programs was determined by the problem size. The autocorrelation program uses N=100 PEs, which is more than all the other programs. Its PE usage is equal to the number of samples in a frame of speech. The preemphasis filter program can use any number of PEs, so it uses the same number as the autocorrelation program. The LPC and LTW programs use p=8 PEs. Since $p < N$, N−p=92 PEs are idle during the execution times of the LPC and LTW programs. The DTW program can use any number of PEs too. It uses all 100 since the autocorrelation program uses 100. If there are less than 100

Table 7.19 Number of PEs used by the parallel speech recognition system.

| | Number of PEs | Determined by |
|---|---|---|
| filter | 1 or more | |
| auto | 100 | N (framesize) |
| LPC | 8 | p (Number of LPC coefficients) |
| endpoint | 0 | |
| LTW | 8 | p (Number of LPC coefficients) |
| rearrange | | Number of PEs used by DTW |
| DTW | 1 or more | |

utterances in the vocabulary, some PEs will be idle during the DTW's execution. The rearrange program uses as many PEs as the DTW program since rearrange's job is to rearrange the data for the DTW program.

Using half as many PEs will at increase the execution time of the autocorrelation program by 3%. The following example shows how the proposed system can be implemented using 50 PEs. The filter, autocorrelation, and LPC programs require 148, 7,026, and 6,352 cycles respectively to execute on 100 PEs. If 50 PEs are used, the LPC program will require the same number of cycles since it uses only 8 PEs, and the filter program will use twice as many cycles since it will be executed twice for every input frame. The autocorrelation program will use 7,214 cycles for a total of 2*148 + 7,214 + 6,352 = 13,862, cycles which is a sampling rate of 28 KHz. This is only one 1 KHz slower than when 100 PEs are used. Therefore, 50 PEs can be used and still process speech in real time; however, the DTW program will require twice as much time when using 50 PEs. With 50 PEs the DTW program will use 1.6, seconds on a 1,000 word vocabulary which is considered too long for real time response.

### 7.8.6. Changing the Word Recognition System Parameters

It has been shown that the proposed isolated word recognition system can process high quality speech in real time. The following section discuss the effects of altering the system parameters on the processing throughput.

#### *7.8.6.1. Changing the LPC Frame Size*

If the frame size is increased, the autocorrelation program can use more PEs, and the execution time will increase in proportion to log M (where M is the frame size) based on the time complexity equations. The time between frames will increase if the sample rate remains the same. Suppose the frame size is doubled to 200 samples and the sampling rate remains the same. The autocorrelation program requires 7,836 cycles per frame which is a sampling rate of 102 KHz (assuming NetD=18 and autocoef=9). This is nearly twice the throughput of the program using 100 sample frames (See Table 7.3).

If the frame size of the above example is doubled from 100 to 200 samples, and 100 PEs are still used, the autocorrelation program will use 8,204 cycles, the filter program will use twice as many cycles, and the LPC will used the same number of cycles. The total will be 8,202 + 2*148 + 10,106 = 18,426 cycles which is a sampling rate of 43 KHz. This is faster than using 100 samples per frame, which yields 39 KHz.

Reducing the frame size would reduce the number of PEs used. The duration of a frame is based on the characteristics of the vocal tract and the proposed duration (5 ms) is shorter than what is commonly used (10-20 ms); therefore a frame size reduction would most likely result from a decrease in the sampling rate.

### *7.8.6.2. Changing the Number of LPC Coefficients*

The proposed isolated word recognition system has assumed 8 LPC coefficients are used. Many high quality speech processing systems use as many as 16 LPC coefficients. Table 7.4 shows that the maximum sampling rate for 16 coefficients is 19 KHz; 14 KHz for NetD=18. Although most high quality systems sample at 15 to 20 KHz and these are near that range, there is no time left for executing the DTW program. This shows that the 8 MHz MC68000 SIMD machine based system is able to process in real time, but it does not have much leeway. Increasing the number of LPC coefficients makes it unable to process in real time.

The proposed system assumes a 5 ms frame size. Typically 10 to 20 ms frames are used. If the frame size is increased to 10 ms by using 200 samples per frame and 100 PEs are still used, the time needed will be 8,202 cycle for the autocorrelation program, 2*148 cycles for the filtering program, and 14,200 cycles to the LPC program. This gives a total of 22,698 cycles to process 200 samples for a sampling rate to 35 KHz, which is fast enough of high quality speech.

#### *7.8.6.3. Changing the Number of Frames per Utterance*

The proposed system assumed that I=40 frames per utterance were output from the LTW and processed by the DTW program. The LTW and DTW execution times are proportional to I, so increasing I will increase the LTW and DTW processing times. Thus a larger buffer is needed to store the incoming speech samples while the LTW and DTW programs are executing. Decreasing I, on the other hand, will shorten the LTW and DTW execution times and require a smaller input buffer.

#### *7.8.6.4. Changing the Vocabulary Size*

The DTW program is the only program whose execution time depends on the vocabulary size. The DTW execution time is proportional to $\lceil W/N \rceil$ where W is the number of words in the vocabulary and N is the number of PEs. As with the number of frames per utterance, an increase in W will require a larger input buffer, and a decrease will require a smaller input buffer.

### 7.8.7. Summary

The proposed SIMD machine based isolated word recognition system is able to execute in real time using 100 PEs. Many of the word recognition parameters can be changed and the system will still run in real time. However, increasing the number of LPC coefficients from 8 to 16 without increasing the frame size will cause the system, as it is implemented here, to run slower than real time. The performance of this system is conservative because:

1) a clock rate of 8 MHz was used, although 12.5 MHz MC68000s are available,

2) the PE and CU instruction executions were not overlapped,

3) the LPC frame size was assumed to be 5 ms where 10 to 20 ms are normally used,

4) the network delay was assumed to be 4.5 $\mu$s per 16-bit word and was not overlapped with the instruction execution, and

5) the LPC program uses only 8 PEs and leaves 92 PEs idle.

Increasing the clock rate to 12.5 MHz would increase the throughput by 50% if no wait state memory is used. The table on page 59 of [SiKu82] shows that overlapping the CU and PE instruction execution can result in a 50% speedup. As shown earlier, increasing the frame size and using the same number of PEs reduces the number of computations. Using a faster network and overlapping network transfers can give an effective network delay of 0 which improves the throughput. Finally, computing the LPC coefficients for several frames in parallel will reduce the number of parallel computations needed for the LPC routine.

Considering all of the above, the SIMD based isolated word system has the power needed to execute the proposed system in real time. A system requiring more computations can be implemented in real time if a less conservative model is used.

# 8. SIMULATING VLSI PROCESSOR ARRAYS

Section 5.3 showed how a VLSI processor array could reduce the number of *loops* needed to perform a given task. Of course the question left unanswered was "How much time does a *loop* take?" The following section describes Poker, an emulator for a processor array called Pringle, which has been used to obtain timings. The Poker system was written by members of the Computer Science Department at Purdue University to help in developing the Blue CHiP project [Snyder82a].

## 8.1. Poker Details

The CHiP (*C*onfigurable, *Hi*ghly *P*arallel) computer [Snyder82a] is a family of architectures each constructed from a switch lattice and a collection of microprocessors (called cells*). The switch lattice consists of many switches that can be connected to each other and to adjacent cells. Figure 8.1 shows a possible layout of switches and cells, where the circles represent switches and the squares are cells. Each switch can be dynamically programmed to connect to any of its eight nearest neighbors (i.e., any switch or cell to the north, east, west, south, northeast, northwest, southeast, or southwest). The cells are not connected directly to each other, but communicate through the switch lattice. This connection is a circuit switch rather than a packet switch. The VLSI array structure of two cells being connected can be realized in a CHiP architecture by connecting two cells through a switch. The VLSI array computer can

*Although Poker documentation calls their processors PEs, I will continue to call the processors associated with VLSI arrays *cells,* and reserve the label *PEs* for processors in an SIMD machine.

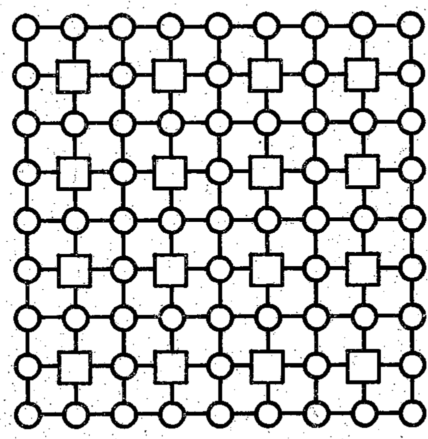


Figure 8.1. Typical switch lattice.

therefore be included as a member of the CHiP computer family by using this type of inter-cell connection.

The Poker System provides a means to emulate Pringle, a CHiP computer [Snyder82a]. The Poker programming environment gives the user the following tools for developing programs for a CHiP computer:

1) A high level language called *xx* that allows one to write code for each cell without having to be concerned with details of the hardware.

2) The ability to set switch settings, thus controlling which port on one cell can communicate to another port on another cell.

3) A simple way to assign which cell will run which *xx* code, and pass different parameters to cells running the same code.

4) A way to map the logical port names given in the *xx* code to the physical ports given in the switch settings.

5) An added feature that allows a user to trace the execution of an *xx* program on a line by line basis.

Details about using 1 through 4 above are given in [Snyder83]. The major difference between the hardware emulated by Poker and a CHiP computer is the switch lattice. Poker does not use a circuit switched interconnection as a CHiP computer does. Instead, each cell has an output latch and an input queue between it and the switch lattice. The latch is polled regularly by the switch hardware. If it contains data, the data is moved to the input queue of the destination cell.

Although Poker does not directly emulate the inter-cell communication of a VLSI array processor it does emulate enough of the VLSI array to obtain meaningful timings. The following sections describe the Poker programming environment and the hardware it emulates.

## 8.1.1. Software for Emulating with Poker

### *8.1.1.1. The xx Programming Language*

The *xx* programming language is a simplified sequential programming language for defining the code for the cells in Poker. Figure B.1 in Appendix B gives a complete description of the language. The example in Figure 8.2 shows some of the features of the language and the conventions that will be used here in presenting Poker programs. The line numbers on the left in the figure are used to refer to portions of the figure.

The block of comments before the first numbered line is a standard header that appears before each major program. Each section of the header is described in the following list.

*Program Name* gives the name of the program as listed in the code names section. The name will be followed by the program name (as used in the text) in () 's if more than one program uses the same name.

*Algorithm* will give the figure number of the corresponding *xx* code if the program is an assembly language program. The *xx* programs will give the figure number of the algorithm it is implementing.

*Machine* will be the VLSI processor array.

*Function* will give a brief description of what the program does.

*Precision* lists the number of bits and format for the input, output, and any other important variables used by the program.

*Number of PEs* will list the number of cells used by the VLSI processor array.

*Parameters* lists and describes the parameters that affect the execution times.

*Input* tells which port the input data comes from in the VLSI processor array.

*Output* is the corresponding information to *Input*.

*Loop Time* tells how many $\mu$s are needed to process one input sample in the VLSI processor array.

*Max Sample Rate* tells how many samples can be processed in one second.

Lines 1-12 of Figure 8.2 show that a comment is enclosed between /* and */, and can span more than one line.

Line 14 declares this code to be named *auto,* and must be stored in a file named *auto.x*. If parameters were passed to this cell, the line would be

```
/*
                    Program Name:       auto (a1)
                    Section:            6.2
                    Machine:            VLSI processor array, simulated by Poker.
                    Function:           Find autocorrelation coefficients R(i)
                                        given input signal x(m), using
                                                 k=M-i-1
                                        R(i)=      ∑     x(k)x(k+i).
                                                  k=0

                    Precision:          Input:  32-bit floating point
                                        Output: 32-bit floating point
                    Number of PEs:      p, the number of coefficients computed.
                    Parameters:         p, the number of coefficients computed.
                    Input:              Arrives at the north port of cell (1,3).
                    Output:             Departs from east port of merge cell.
                    Loop Time:          90 μs to process one input sample.
                    Max Sample Rate:    11 KHz
*/

/*

                    This routine finds the first p autocorrelation coefficients
                    of its input data.  The value of p depends on the number of
                    cells used.  One sample is read from each of the two input
                    ports (in1 and in2).  The sample coming from the in1 port
                    is written to the bottom port (out) so the cell below
                    can use it during the next cycle. The two samples are
                    multiplied together and added the a running sum (sum).  After
                    one frames worth of samples have been read (as determined by
                    the variable samples) the total sum is output to the results
                    port (results).
*/

code                auto;
trace               sum,left,top;
ports               in1,in2,out,results;
begin
                    sint    i,samples;               /* Samples per frame    */
                    real    top,left,sum;            /* These are type int for (a2) */
                    real    in1,in2,out,results;     /* These are type int for (a2) */

                    i  := 0;
                    sum:= 0;
                    samples := 10;
                    out <- sum;             /* Send a zero out to initialize the pipeline */

                    while true do
                            begin
                            i:=i+1;
```

Figure 8.2. An example of an *xx* program.

```
            top <- in1;
            left<- in2;

            if i < samples then       /* Has one frame been processed? */
                    begin             /* No  */
                    out  <- top;      /* Send sample from top to cell below */
                    sum := sum + top * left;/* Find sum    */
                    end

            else begin
                    sum := sum + top * left;         /* Last sample in frame */
                    results <- sum;          /* send out results        */
                    sum := 0;                        /* Reinitialize, sum      */
                    out <- sum;                      /*       and pipeline.    */
                    i := 0;
                    end
            end
end.
```

Figure 8.2 (Continued)

of the form

```
code auto(arg1,arg2);
```

where arg1 and arg2 are given in the code name section which is discussed in Section 8.1.1.3.

Line 15 gives the variables to be traced. All variables listed here (up to four) will appear on the screen during a run, and in the *Trace* file if used. This allows monitoring of the variables during execution, but would not be used in a production setting.

Line 16 tells which I/O ports will be used. These are logical names, and will not be associated with physical names until load time. The data in the port names section tell which logical name to map to which physical direction.

Line 17 starts the beginning of the program.

Line 18 declares *i* and *samples* to be of type *s*hort *int*eger (sint).

Lines 19 and 20 declare several variables to be of type *real.*

Lines 22-24 are assignment statements.

Line 25 writes the values of *sum* to the port *out.* Notice that ":=" is the assignment operator, while "<-" is the read/write port operator.

Line 27 is a *while* statement, and the boolean value *true* is always true, so this loop will go on forever.

Line 28 is the start of a begin/end pair.

The rest of the code is much like any other FORTRAN-like high level language.

### *8.1.1.2. The Switch*

Figure 8.3a is an example of a configuration of cells for a VLSI processor array algorithm, and Figure 8.3b is the switch setting that implements it. The particular algorithm is for autocorrelation, and is used as an example of a typical algorithm. Each box $\left\{\begin{array}{c}+-+\\ \text{x,y}\\ +-+\end{array}\right\}$ is a cell where *x* is the row number of the cell, and *y* is the column number. A "." is a switch and the −,\,/, and | are the data paths.

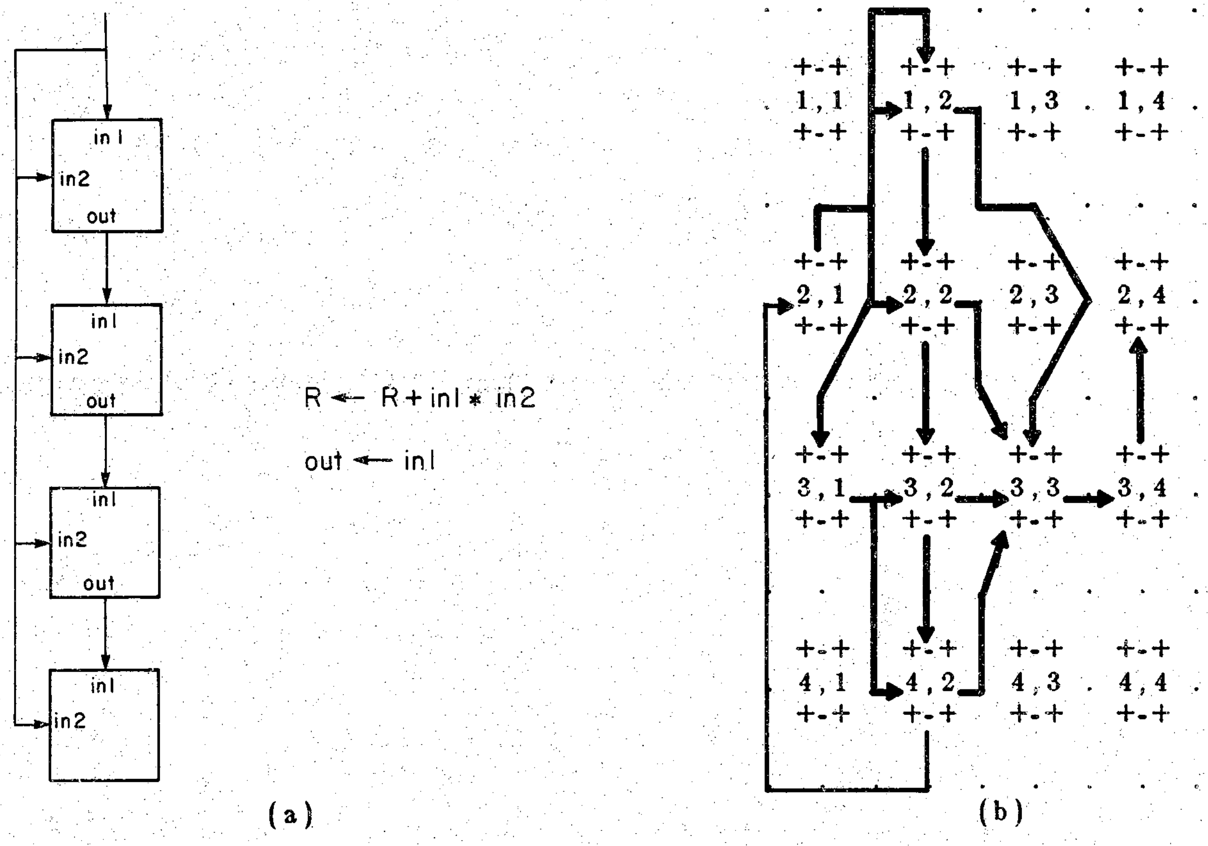


Figure 8.3 (a) Example of a cell configuration for a VLSI algorithm. (b) Example of Poker switch settings for the algorithm.

Each processor has eight logical switch input/output ports. Most programs presented here use a given port for either input or output but not both, so often arrow heads are used to show the direction the data flows. This has no effect on the hardware or software; the arrows are used to make the data flow clearer to the reader. Also, some data paths are used to synchronize two cells. In this case, the arrival of data at cell A marks some event at cell B, and the value of the data passed is ignored. The data paths used in this manner are drawn with light lines, while true data paths are drawn with heavy lines.

#### *8.1.1.3. Code Names*

Each processor can run different code. The code name listing on the right of Figure 8.4 shows which program is run on which processor. There is a correspondence between the left and the right halves of the figure. The upper left cell in the switch runs the code listed in the upper left of the code names. If a cell is unused, no name is listed. In reality, all cells run all the time, but the unused cells run code called *empty* which is a statement jumping to itself.

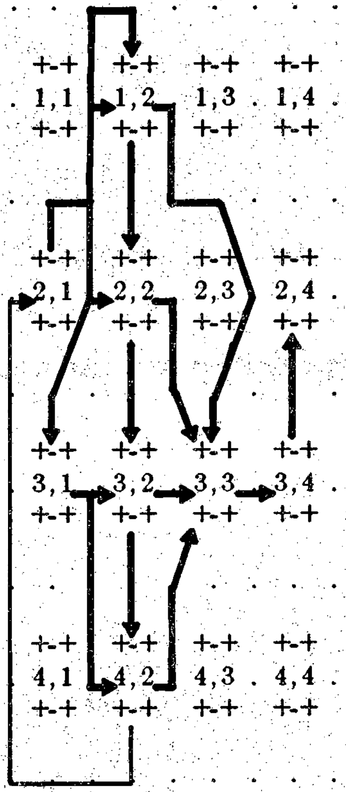
Some programs will have data values listed below the program names. These values are passed to the given program as arguments on the line declaring the name of the code. For example if line 14 of Figure 8.2 were:

```
code auto(arg1,arg2,arg3,arg4);
```

the first value listed below the code name in Figure 8.3 would be passed as *arg1*, the value below it as *arg2*, and so on. Up to four values can be passed. The values need not be the same for different cells running the same code.

#### *8.1.1.4. Port Names*

As mentioned above, each port can be assigned a logical name. This name is mapped to a physical port during load time. The port names given in the example in Figure 8.5 show the mapping from logical names to physical ports. The position of a given name in a cell identifies the port to which it is connected. The positions are:



| cell | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | auto | | |
| 2 | input | auto | | output |
| 3 | pipe | auto | merge4 | lpc |
| 4 | | auto | | |

Figure 8.4. Example of a Poker switch setting and code name assignments.

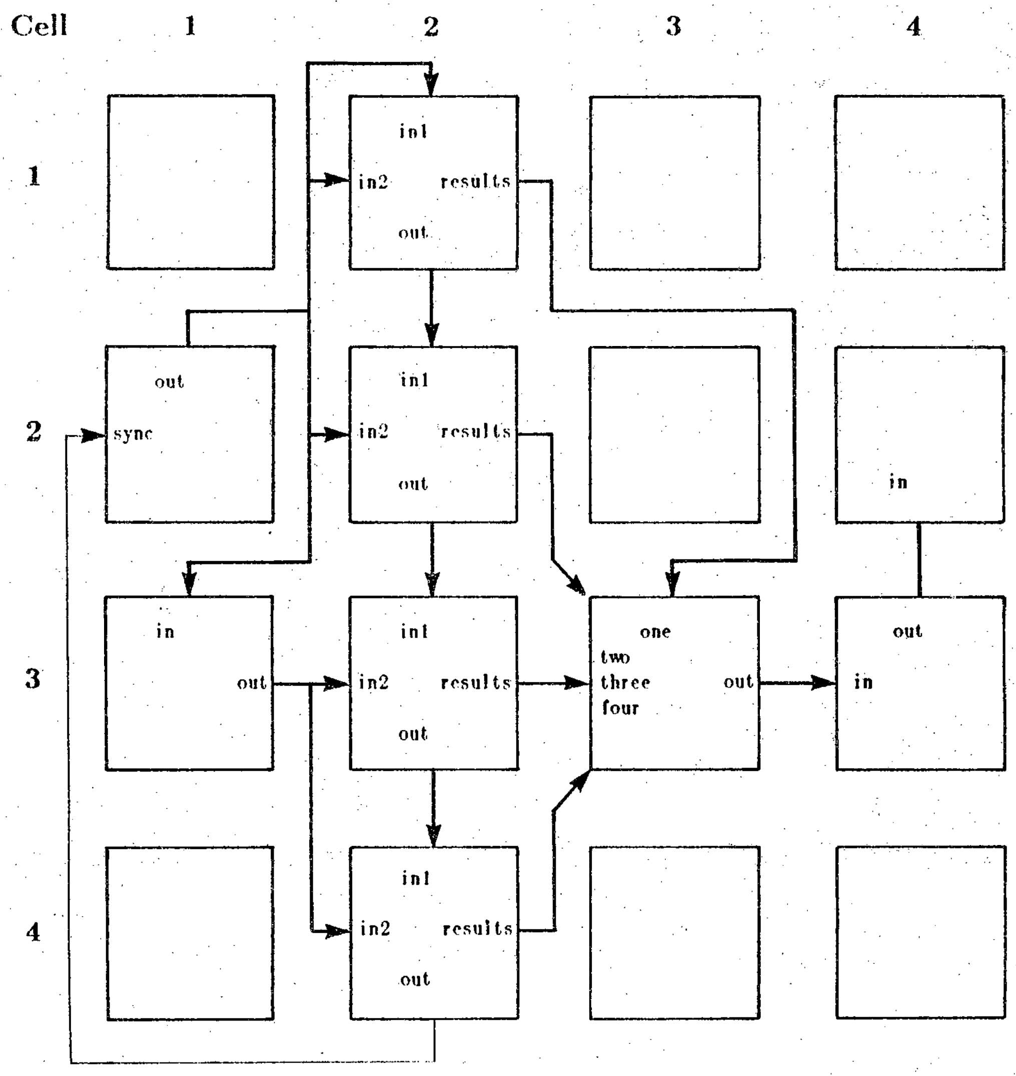


Figure 8.5. Example of Poker port name assignments.

```
        north
nw              ne
west            east
sw              se
        south
```

When running assembly code, data is written to the physical ports, and not logical ports; therefore the port assignment table is not needed.

### 8.1.2. Hardware Emulated by Poker

Figure 8.6 shows the hardware used in one cell of Poker. The main components are an Intel 8051 microprocessor, an Intel 8231 Arithmetic Processor Unit (APU), and the switch interface. The following gives more details about the hardware emulated by Poker.

#### *8.1.2.1. The Intel 8051 Microprocessor*

The heart of the hardware is an Intel 8051 single-component 8-bit microcomputer [Intel]. It is an 8-bit processor designed for single chip operations as a controller or as an arithmetic processor. It runs with a 12 MHz clock and the shortest instruction takes 12 cycles, or 1 $\mu$s. An 8-bit register addition or subtraction takes 1 $\mu$s while an 8-bit unsigned multiplication takes 4 $\mu$s. Figure B.2 is a list of the 8051 instruction set including execution times for each instruction.

The 8051 has two types of RAM, internal and external. There are 256 bytes of internal RAM with the upper 128 bytes being special function registers. These registers allow access to the two built-in 16-bit timers, the four built-in 8-bit I/O ports, and other special features of the 8051. (Figure B.3 gives an example of how to use the built-in timer to control the execution time of a loop.) The lower 128 bytes can be used as regular memory. Most assembly language programs presented here use only the internal RAM.

The external RAM consists of 4K bytes of EPROM and 2K bytes of RAM. The EPROM contains routines used to support the *xx* code. The RAM holds the user's program and data. The external RAM is accessed only through a special register, and thus takes more processor time to use than the internal RAM.

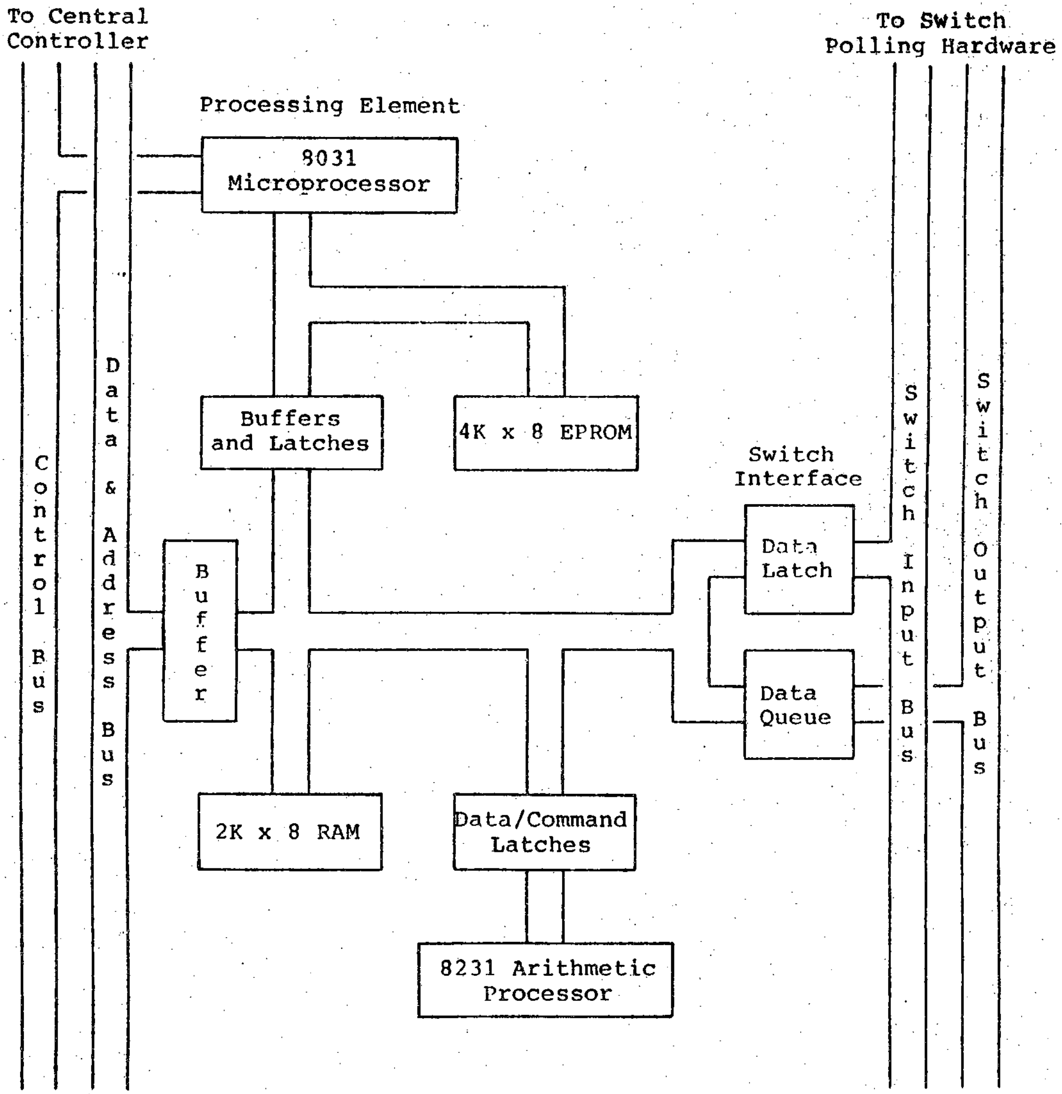


Figure 8.6. Poker cell detail (from [Field]).

### *8.1.2.2. The Arithmetic Processing Unit (APU)*

There is an Intel 8231 APU to assist the 8051 microprocessor with 32-bit floating point arithmetic. The two processors communicate via an 8-bit command latch and an 8-bit data latch. The 8051 pushes data onto the 8231's stack, sends a command, and then pops the result. The APU executes a 32-bit floating-point addition in at most 92 $\mu$s, subtraction in 93 $\mu$s, a multiplication in 42 $\mu$s, and a division in 46 $\mu$s. These maximum execution times are too slow for most speech processing. Also, there is considerable overhead in pushing/popping data to/from the APU, so it is faster for the 8051 to perform some operations than to send them to the APU.

Variables declared to be type *real* or *int* in *xx* are 32 bits long and are processed by the APU. Otherwise, variables of type *sint* are 8 bits each and are processed directly by the 8051.

### *8.1.2.3. The Switch*

An 8051 can communicate with other 8051s through the switch. The switch is a crossbar switch that allows any processor to talk to any other processor. An 8051 talks to the switch through an 11-bit wide output latch, and an 11-bit wide, 16-word deep input queue. Since each processor has 8 logical I/O ports that are implemented by one latch and queue, three of the 11 bits are directional information, i.e., they tell to which port the remaining 8 bits of data are to go. The same is true for the input queue: 8 bits are data, and three bits are the tag telling from which port the data came.

The switch can poll 8 cells every $\mu$s. There are 64 processor cells in an 8 by 8 square, plus 32 more I/O cells along the edges of the square, giving a total of 96 cells, or 12 $\mu$s to do one scan. It is the software's responsibility to wait 12 $\mu$s between writes to the output latch to be sure the previous data was written. If two writes happen between scans, the first data written is lost. Figure B.4 gives an example of how to read/write data from/to the switch.

Once the data is received from the switch, it is the programmer's responsibility to check the tag and buffer the data until all four bytes have arrived from the same direction. In some programs, the data comes from only one direction, or a known direction, so the direction need not be checked. This short cut is used frequently in the assembly routines presented in the following chapters to decrease the execution time of the algorithms.

When using *xx*, the high level language, all port checking and delaying are handled by the compiler and/or loader.

### *8.1.2.4. The 8051 Assembler*

The assembler used for the 8051 supports all the mnemonics for machine instructions specified in [Intel]. The general format of an instruction is:

```
opcode    destination, source
```

so,

```
mov  sum,a
```

would move the data from *a* (the accumulator) to the internal RAM location called *sum*. The output from the assembler, shown in the figures in Appendix B, prints the execution time in $\mu$s for each instruction to the left of the instruction.

The assembler also allows files to be inserted into the current input file. A line of the form:

```
#include "filename.h"
```

will stop the assembler from reading the current file and start reading *filename.h*. Once *filename.h* is read, processing is continued on the previous input file. Two commonly used include file are *ports.h* and *util.h*. *Ports.h* contains the I/O port definition as shown in Figure B.5. *util.h* contains the definition for *writedelay* which waits a fixed amount of time for data to be read from the output latch, and *readwait* which waits for data to appear in the input queue. *util.h* is listed in Figure B.6.

### 8.1.3. Summary

This section has presented the Poker system that is used to simulate VLSI processor arrays. A brief description was given of both the hardware and software, with emphasis on how the hardware affects the software. The important points with respect to the simulations described in the following sections are:

1) Although each cell has an Intel 8231 APU, it is often faster to use the Intel 8051 microprocessor to perform the 8 and 16-bit fixed point arithmetic.
2) Each cell has eight logical I/O ports which are implemented by one output latch and one 16-word deep input queue. 8-bit data is written into the output latch; the latch is polled once every 12 μs, so there must be a 12-μs delay between writes to the latch.
3) The 8051 has two 16-bit timers that can be used to synchronize cells.

Overall, Poker provides an accurate simulation of a VLSI processor array.

## 8.2. Simulation of Filtering Algorithms

This section presents two different digital filtering algorithm simulations. The first is a direct implementation of the VLSI algorithms discussed in Section 6.1.1. This algorithm use no broadcasts and produces one output every two loops. The second algorithm is based on the VLSI algorithms in Section 6.1.2. Here broadcasts are used, and one output is produced during every loop.

The following is a list of requirements a filtering program must meet to process speech data in real time.

*Sampling rate:* The sampling rate for speech data ranges from 6.67 KHz for telephone quality speech to 20 KHz for high quality speech. The filter program must process speech data at these rates to run in real time.

*Precision:* Speech data needs about 8 bits per sample for telephone quality speech and 11 to 12 bits per sample for high quality speech.

*Type of filter:* Selecting values for p and q depends on the type of filter used. The selection of p and q does not affect the execution time of these filtering algorithms; it changes only the number of cells that are used. Therefore during the simulations, p and q are generally set to values that produce convenient sized arrays.

### 8.2.1. Digital Filtering Without Broadcasts

Figures 8.7, 8.8, and 8.9 show the switch settings, port names, and *xx* routines, respectively, used to simulate the first filter algorithm with p=2 and q=2. The values selected for p and q have no effect on the execution time of this program. For convenience, these values were selected so that the array would fit in a four by four cell arrangement. The numbers listed under the name *filter* on the right half of Figure 8.7 is the value of the filter coefficient that is used by the given filter cell. This example is evaluating

$$y_m = b_0 x_m + b_1 x_{m-1} + b_2 x_{m-2} + a_1 y_{m-1} + a_2 + y_{m-2}$$

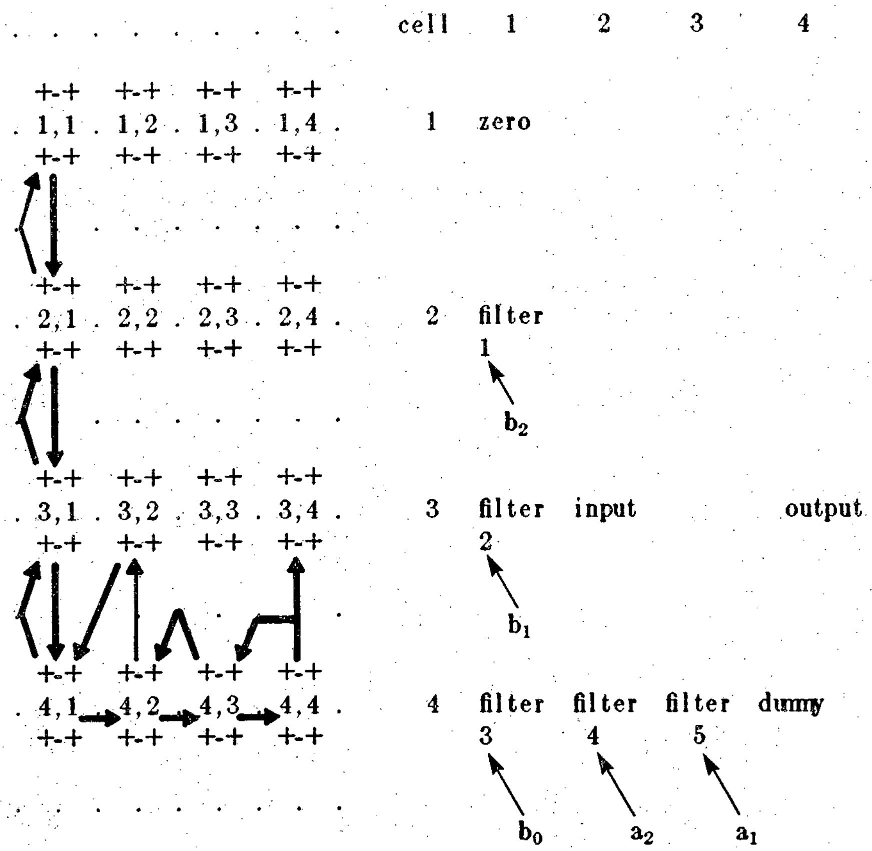


Figure 8.7. Switch settings for no broadcast *xx* filter program, p=2 and q=2.

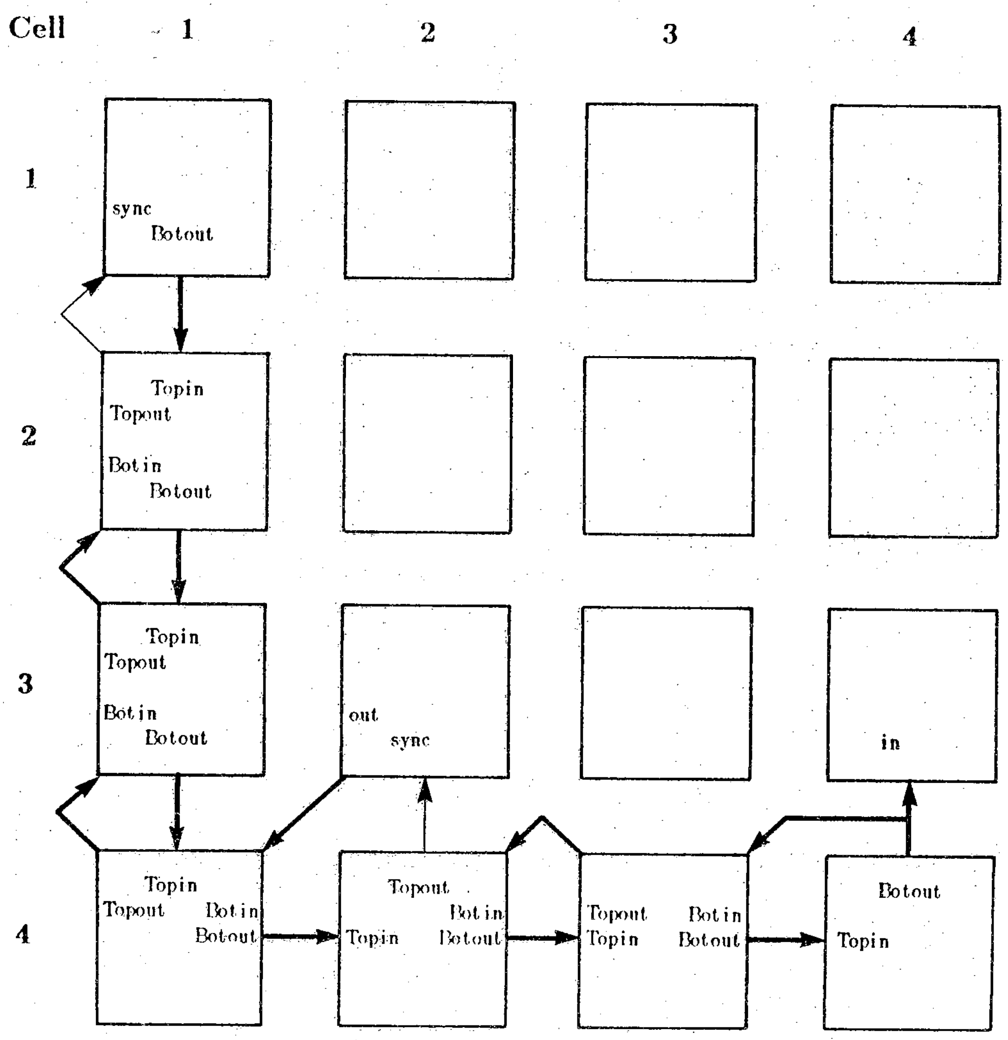
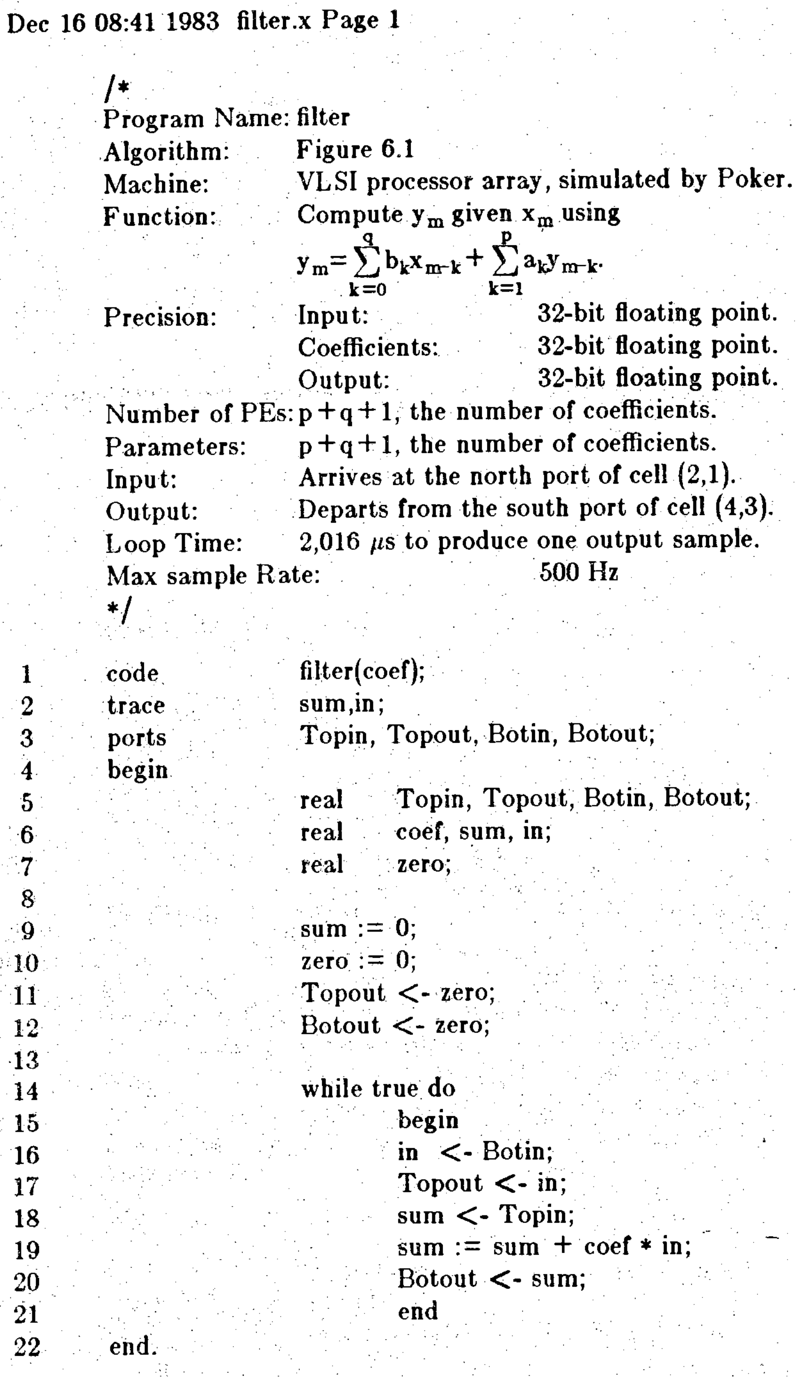


Figure 8.8. Port names for no broadcast $xx$ filter program, p=2 and q=2. Figure 8.12. Port names of fast filter (f1) program.



Dec 16 08:41 1983 filter.x Page 1

```
/*
Program Name: filter
Algorithm:        Figure 6.1
Machine:          VLSI processor array, simulated by Poker.
Function:         Compute $y_m$ given $x_m$ using
                  $y_m=\sum_{k=0}^{q} b_k x_{m-k} + \sum_{k=1}^{p} a_k y_{m-k}$.
Precision:        Input:                32-bit floating point.
                  Coefficients:         32-bit floating point.
                  Output:               32-bit floating point.
Number of PEs:    p+q+1, the number of coefficients.
Parameters:       p+q+1, the number of coefficients.
Input:            Arrives at the north port of cell (2,1).
Output:           Departs from the south port of cell (4,3).
Loop Time:        2,016 μs to produce one output sample.
Max sample Rate:                        500 Hz
*/

code              filter(coef);
trace             sum,in;
ports             Topin, Topout, Botin, Botout;
begin
                  real     Topin, Topout, Botin, Botout;
                  real     coef, sum, in;
                  real     zero;

                  sum := 0;
                  zero := 0;
                  Topout <- zero;
                  Botout <- zero;

                  while true do
                           begin
                           in  <- Botin;
                           Topout <- in;
                           sum <- Topin;
                           sum := sum + coef * in;
                           Botout <- sum;
                           end
end.
```

Figure 8.9. *xx* code for no broadcast filter program.

Dec 16 08:41 1983 dummy.x Page 1

```
code            dummy;
trace           tmp;
ports           Topin, Botout;
begin
                real    tmp;
                real    Topin, Botout;

                tmp := 0.0;

                while true do
                        begin
                        Botout <- tmp;
                        tmp <- Topin;
                        end
end.
```

Dec 16 08:41 1983 input.x Page 1

```
code            input;
trace           i;
ports           out,sync;
begin
                real    tmp,zero;
                real    out,sync;
                real    i;

                zero := 0.0;
                i := 1.0;

                while true do
                        begin
                        tmp <- sync;
                        out <- i;
                        i := i+1;
                        if(i > 10.0) then
                                i := 1.0;

                        tmp <- sync;
                        out <- zero
                        end
end.
```

Figure 8.9 (Continued)

Dec 16 08:41 1983 output.x Page 1

```
code            output;
trace           out;
ports           in;
begin
                real    out;
                real    in;

                while true do
                        begin
                        out <- in;
                        end
end.
```

Dec 16 08:41 1983 zero.x Page 1

```
code zero;
ports Botout, sync;
begin
                real    dumb,z;
                real    Botout, sync;

                z:= 0.0;

                while true do
                        begin
                        dumb <- sync;
                        Botout <- z;
                        end
end.
```

Figure 8.9 (Continued)

for $b_0=3$, $b_1=2$, $b_2=1$, $a_1=5$, and $a_2=4$. Figure 8.10 lists the execution time in $\mu$s for each statement in the filter program. Column one is the number of times the given statement was executed during the simulation. Columns two through four are the minimum, average, and maximum times in $\mu$s for the given statement. The total time for one loop is 1,008 $\mu$s, and two loops are required to process one input. This gives a total time of 2,016 $\mu$s, or a sampling rate of less than 500 Hz. Briefly, the main delays causing the program to be so slow are the time for the inter-cell communications and the time needed to send data to and from the APU. This will be discussed in more detail in Section 8.2.2.2.

500 Hz is not fast enough for speech processing. This problem is overcome by the algorithm discussed in the next section.

### 8.2.2. Digital Filtering Using Broadcasts

The previous filtering algorithm could not process data fast enough to filter speech signals since it required two loops to produce one sample and each loop took 1,008 $\mu$s. An implementation of the VLSI algorithm presented in Section 6.1.2 can produce one sample for every loop. It does this by replacing the upward flowing pipeline with two simultaneous broadcasts. Three programs were written to run this algorithm. They are as follows:

| Name | Language | Data Size | Sum Size |
|---|---|---|---|
| f1 | xx | 32 bit | 32 bit |
| f2 | 8051 | 8 bit | 16 bit |
| f3 | 8051 | 16 bit | 24 bit |

All three programs implement the same algorithm. They differ in the language in which they are written and in the precision of the data they process. Program f1 still cannot process data fast enough for real-time speech filtering. Programs f2 and f3 show that by reducing the precision of the data and coding in assembly language, one can process data fast enough for real-time speech processing. The following sections describe each program.

```
Count  Min   Ave   Max
                         code   filter(coef);
                         trace  sum,in;
                         ports  Topin, Topout, Botin, Botout;
                         begin
                                      real   Topin, Topout, Botin, Botout;
1      10    10    10                 real   coef, sum, in;
1      0     0                        real   zero;

1      178   178   178                sum := 0;
1      178   178   178                zero := 0;
1      91    91    91                 Topout <- zero;
1      91    91    91                 Botout <- zero;

1      0     0                        while true do
                                             begin
29     268   268   268                       in  <- Botin;
29     91    91    91                        Topout_<- in;
29     238   238   238                       sum <- Topin;
29     318   318   318                       sum := sum + coef * in;
29     91    91    91                        Botout <- sum;
29     2     2     2                         end
                         end.
```

Figure 8.10. Execution times for slow *xx* filter program. Execution times are given in $\mu$s.

### *8.2.2.1. Fast xx Filter Program — f1*

Figures 8.11, 8.12, and 8.13 shows the switch settings, port names, and *xx* listings, respectively, for the *xx* program for f1 with p=1 and q=2. For convenience, these values for p and q are chosen so that all the cells used for the filtering operation will fit along one column of a four by four array. As before, the values of p and q have no effect on the execution time of the algorithm unless large values will lengthen the time needed to broadcast a value to all cells.

The heavy lines in Figure 8.11 are the data paths, while the lighter lines are paths used for synchronization. Notice the similarities between the switch setting of Figure 8.11 and Figure 6.2. Figure 8.14 lists the execution time in $\mu$s for each statement in the filter program.

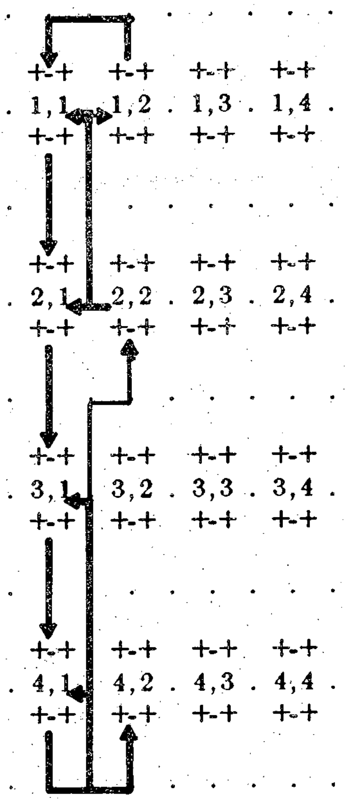
Some general comments about these times are:

1) The variable declarations require some execution time because various flags are set during run time to indicate which variables are traced. In a production system the variables would not need to be traced.

2) All writes to output ports take 91 $\mu$s. They are not buffered and go immediately to the switch lattice.

3) Reads from input ports, on the other hand, can vary greatly in execution time. The data coming from the switch lattice enters a 16-word hardware input buffer. When the cell reads from the buffer, it gets one byte of data along with a tag telling which port the byte came from. If the data did not come from the desired port, the data is stored in a buffer for use when the cell wants to read from the given port.

The total time for one loop is 906 $\mu$s. Since one sample is processed every loop, the sample rate which can be handled is about 1.1 KHz, still too slow for speech processing. The execution time for one loop is spent as shown in Table 8.1. 65% of the time is for I/O, while only 38% is for the actual computation. Figure B.4 shows it takes 49 $\mu$s to write four bytes to an output port while Figure 8.14 shows that writing to an output port takes 91 $\mu$s. The additional 42 $\mu$s are the overhead introduced by the compiler. Part of this overhead is moving the data from external RAM to internal RAM. The *xx* compiler stores all type *reals* in external RAM while the example in Figure B.4 assumed the data is in internal RAM.



```
cell   1      2      3     4

 1    filter  zero
      1

 2    filter  input
      2

 3    filter
      3

 4    filter  output
      4
```

Figure 8.11. Switch settings and code names for fast filter (f1) program for p=1 and q=2. The heavy line are the data paths, while the lighter lines are paths used for synchronization.

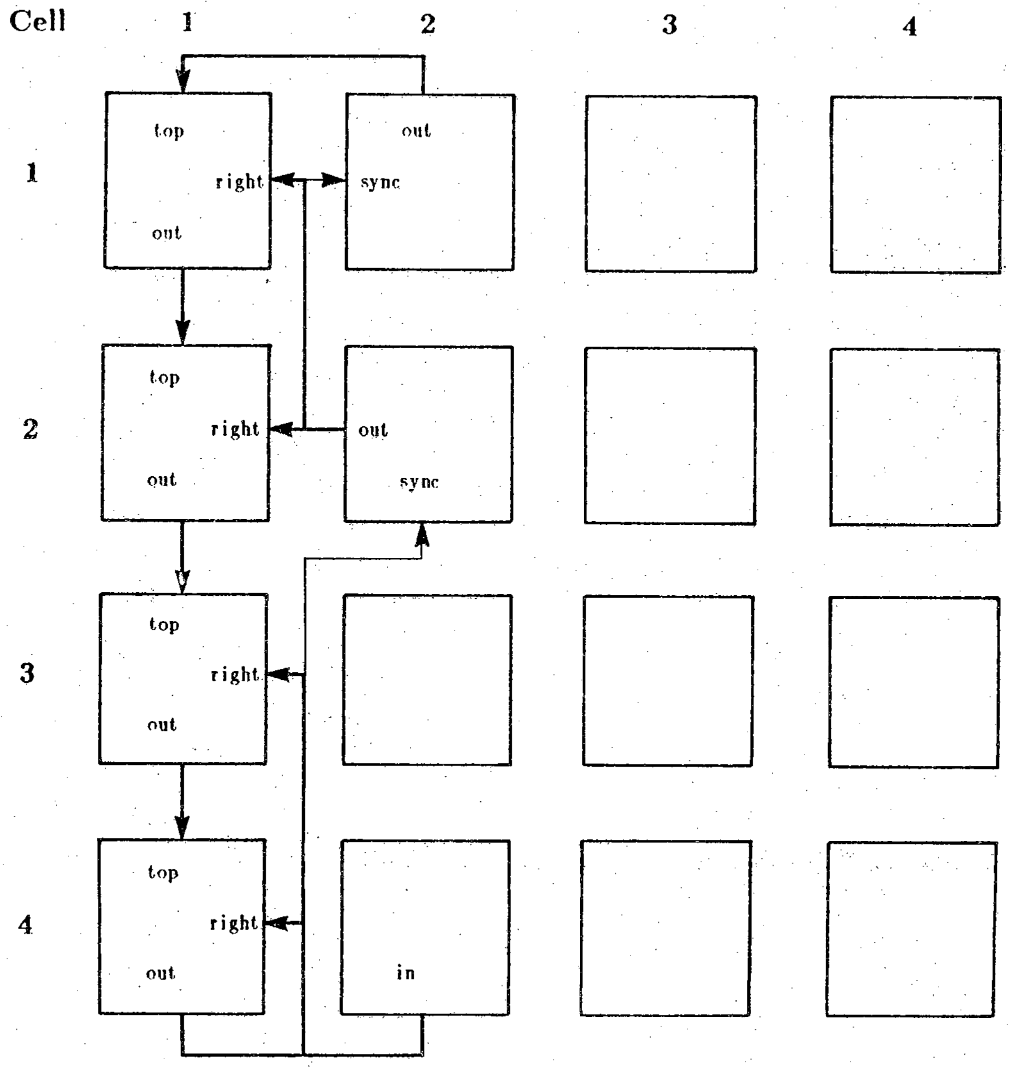
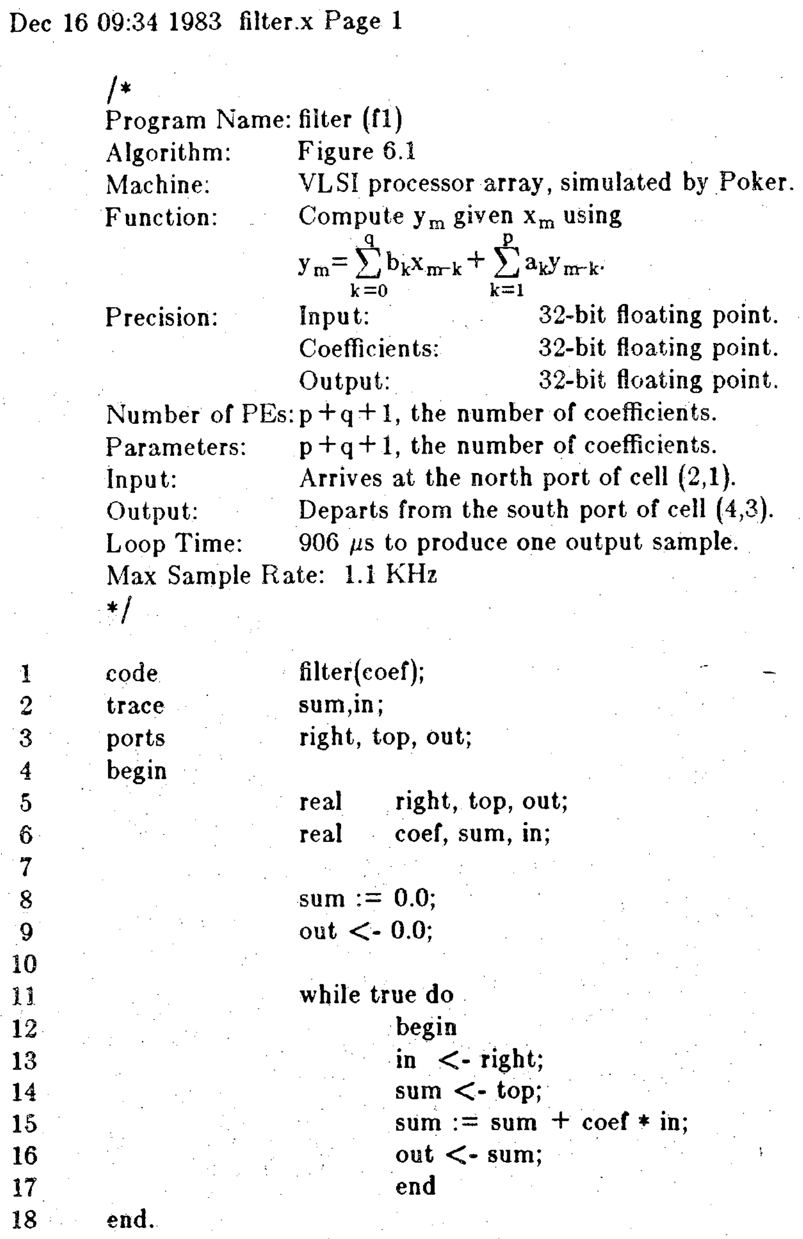


Figure 8.12. Port names of fast filter (f1) program.



Dec 16 09:34 1983 filter.x Page 1

/*<br>
Program Name: filter (f1)<br>
Algorithm: Figure 6.1<br>
Machine: VLSI processor array, simulated by Poker.<br>
Function: Compute $y_m$ given $x_m$ using

$$y_m = \sum_{k=0}^{q} b_k x_{m-k} + \sum_{k=1}^{p} a_k y_{m-k}.$$

Precision: Input: 32-bit floating point.<br>
Coefficients: 32-bit floating point.<br>
Output: 32-bit floating point.<br>
Number of PEs: p+q+1, the number of coefficients.<br>
Parameters: p+q+1, the number of coefficients.<br>
Input: Arrives at the north port of cell (2,1).<br>
Output: Departs from the south port of cell (4,3).<br>
Loop Time: 906 μs to produce one output sample.<br>
Max Sample Rate: 1.1 KHz<br>
*/

```
code            filter(coef);
trace           sum,in;
ports           right, top, out;
begin
                real    right, top, out;
                real    coef, sum, in;

                sum := 0.0;
                out <- 0.0;

                while true do
                        begin
                        in  <- right;
                        sum <- top;
                        sum := sum + coef * in;
                        out <- sum;
                        end
end.
```

Figure 8.13. *xx* code for fast filter (f1) program.

Dec 16 09:34 1983 input.x Page 1

```
code            input;
trace           i,tmp;
ports           out,sync;
begin
                real    out,sync;
                real    i,tmp;

                i := 1.0;

                while true do
                        begin
                        tmp <- sync;
                        out <- i;
                        i := i + 1.0;
                        end
end.
```

Dec 16 09:34 1983 output.x Page 1

```
code            output;
trace           out;
ports           in;
begin
                real    out;
                real    in;

                while true do
                        begin
                        out <- in;
                        end
end.
```

Dec 16 09:34 1983 zero.x Page 1

```
code zero;
ports out,sync;
begin
                real    dumb;
                real    out,sync;

                while true do
                        begin
                        dumb <- sync;
                        out <- 0.0;
                        end
end.
```

Figure 8.13 (Continued)

Table 8.1 Execution times for filtering program f1.

| Function | Time | Percent of Total |
|---|---|---|
| Input | 495 $\mu$s | 55% |
| Output | 91 $\mu$s | 10% |
| Computation | 318 $\mu$s | 35% |
| Loop Control | 2 $\mu$s | <1% |

```
Count  Min    Ave    Max
                            code            filter(coef);
                            trace   sum,in;
                            ports   right, top, out;
                            begin
                                            real     right, top, out;
1      10     10     10                     real     coef, sum, in;

1      52     52     52                     sum := 0.0;
1      143    143    143                    out <- 0.0;

1      0      0                             while true do
                                                     begin
33     250    255    418                             in  <- right;
33     238    240    310                             sum <- top;
32     318    318    318                             sum := sum + coef * in;
32     91     91     91                              out <- sum;
32     2      2      2                               end
                            end.
```

Figure 8.14. Execution times for *xx* fast filter (f1) program.

The computation takes 318 μs to multiply two numbers and add the product to a running sum. Most of this time is spent moving data from external RAM to the APU and back again.

The largest percent of time is spent reading an input port. The data arrives one byte at a time, with a tag telling which port it came from. The software must maintain a separate buffer for each possible tag since a tag represents a logical input port. This buffer management requires a great deal of time, as Figure 8.14 shows.

### *8.2.2.2. Programming Techniques for Reducing Execution Times*

By using assembly language programming, the following techniques can be applied to reduce the execution time of a loop.

1) Reduce the data size. Although most applications do not need 32-bit floating point arithmetic, the current version of *xx* supports only 32-bit floating point and integer arithmetic*. Digital filtering can be done with 8 or 16-bit signed fixed point data. This allows the 8051 to do the computations directly, thus saving the overhead of sending the data to the APU. Also, reducing the data size reduces the amount of data to send through the switch.

2) Use the 12 μs delay time between writing to the switch. Of the 49 μs needed to move four bytes of data from internal RAM to the switch, 33 μs are nops ("no operations") waiting on the switch. These 33 μs could be used to perform a computation.

3) Store variables in internal RAM. In assembly language, all important variables can be stored in internal RAM, thus eliminating the overhead of referencing external RAM.

4) Control the arrival time of data. The arrival of data to the input port can be controlled so that data will arrive in the order needed. This eliminates the need for time consuming buffer management.

* *xx* does have a short integer (sint) which is 8 bits, unsigned. Being unsigned reduces its usefulness for this application.

Given the current implementation of the *xx* programming language, assembly language programs are needed to get the throughput for real-time processing. The following sections describe,f2 and f3. These programs are written in 8051 assembly language and use the above techniques to reduce the time of a loop.

### *8.2.2.3. Fast Assembly Language Filter Program – f2*

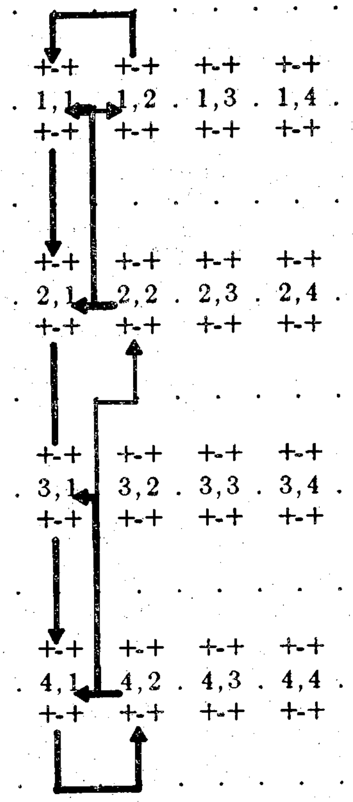
The f2 program uses 8-bit inputs and produces a 16-bit sum. Figure 8.15 shows the switch settings for f2 with p=1 and q=2, and Figure B.7 is a listing of the program. There are no port names given since these assembly language routines reference the physical ports and not the logical ports. Comments have been added to the f2 listing to help explain what it is doing. For example, the line

```
;
; 8    sum <- 0:
;
```

is a comment meaning that the assembly statements that follow perform the same function as line 8 of the corresponding f1 program. The ";" identifies the start of a comment.

Program f2 implements the same algorithm as f1 with one exception. The communication through the switch is carefully controlled so that data arrives in the order it is needed. This eliminates the need to check the source tag and buffer inputs. Unfortunately, the switch settings of Figure 8.11 result in a race condition when cells (3,1) and (4,1) write to their south ports at the same time. The destination of both writes is cell (4,1) and the order of arrival is uncertain. To prevent this, the south port of cell (4,1) goes to the output cell (4,2), which delays the data slightly before writing it to its west port. The arrival times are controlled by using some data paths only for synchronization. Figure 8.16 shows the arrival times and the name of the input port from which the data came the following example:

1) At time one each of the *filter* cells (1,1), (2,1), (3,1), and (4,1) writes data to its south port, and the *zero* cell (1,2) writes to its north port. This data arrives at the north ports of the *filter* cells and the south port of the *output* cell (4,2) at time two.



| cell | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | filter<br>1 | zero | | |
| 2 | filter<br>2 | input | | |
| 3 | filter<br>3 | | | |
| 4 | filter<br>4 | output | | |

Figure 8.15. Switch settings for 8-bit fast filter (f2) for p=1 and q=2.

```
cell        1              2

         |         |    |         |
 1       | 5 north |    |         |
         | 4 east  |    |         |
         | 2 north |    | 4 west  |
         +---------+    +---------+


         |         |    |         |
 2       |         |    |         |
         | 4 east  |    |         |
         | 2 north |    | 3 south |
         +---------+    +---------+


         |         |
 3       |         |
         | 3 east  |
         | 2 north |
         +---------+


         |         |    |         |
 4       |         |    |         |
         | 3 east  |    |         |
         | 2 north |    | 2 south |
         +---------+    +---------+
```

Figure 8.16. Arrival times and port names for f2.

2) The *output* cell (4,2) sends data out its west port after getting data from its south port. This data arrives at the east ports of cells (3,1) and (4,1) and the south port of the *input* cell (2,2) at time three.

3) The arrival of data at the *input* cell (2,2) signals it to write to its west port at time three. This arrives at the east port of *filter* cells (1,1) and (2,1) and the west port of the *zero* cell (1,2) at time four.

4) The arrival of data at the *zero* cell signals it to write a zero value to its north port, which arrives at the north port of cell (1,1) at time five.

Now all cells have data as Figure 8.16 shows. The data is guaranteed to arrive in this order each time through the loop since the transfers are done synchronously.

In f2, cells (1,1), (2,1), (3,1), and (4,1) perform the same code at the same time. At the start of the loop, the data from the north port is in the input queue as shown above. Both bytes are read and saved in internal RAM. Next, the data from the east port is in the queue. It is read and the computation performed. Finally, the sum is written to the south port.

The LSB (least significant byte) of the sum is written before the MSB (most significant byte) is computed. This allows the computation to overlap the 12 $\mu$s switch waiting time.

The input data and the filter coefficients are 8 bits while the running sum is 16. The output cell removes the upper 8 bits when passing the sum back to cells (3,1) and (4,1).

Figure 8.17 gives the equivalent times for each of the *xx* code statements. The total time for one loop is 33 $\mu$s, or a sample rate of about 30 KHz. This is well above the rate needed for speech processing.

There are some practical problems with f2. The data size of 8 bits is adequate for telephone quality speech, but many applications use more than 8 bits. Also the input cell is tightly coupled to the other cells, i.e., it must produce input data at a given time. If it is too soon or too late, the data will enter the queue at the wrong time and be mistaken for other input data. In a real application, the speech sample rate should not have to be tied to the processor clock. The f3 program, as described next, overcomes these problems.

```
f1      f3      f2
xx      8051    8051
32 bit  16 bit  8 bit
                        code            filter(coef);
                        trace   sum,in;
                        ports   right, top, out;
                        begin
                                        real    right, top, out;
10                                      real    coef, sum, in;

52      3       2                       sum := 0.0;
143     31      17                      out <- 0.0;

                                        while true do
                                                begin
255     >13     2                               in  <- right;
240     9       6                               sum <- top;
318     25      12                              sum := sum + coef * in;
91      14      11                              out <- sum;
2       2       2                               end
                          end.

906     59      33      Total loop time
```

Figure 8.17. Execution times in $\mu$s for fast filter programs.

### *8.2.2.4. Fast Assembly Language Filter Program — f3*

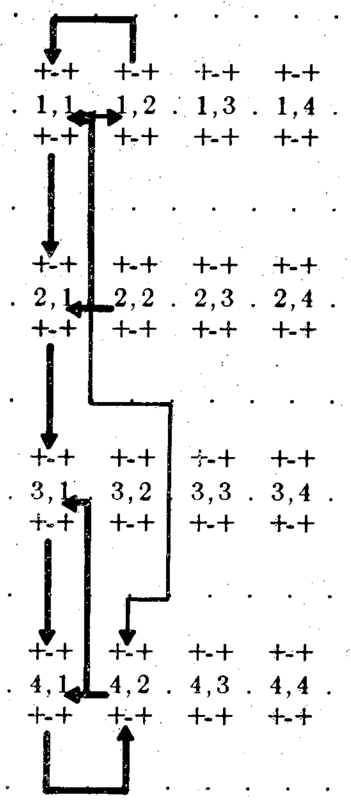
Program f3 overcomes the shortcomings of f2 by using 16-bit input data and keeping a 24-bit sum. Also, the input cell is decoupled from the rest of the cells thus allowing input to arrive at any time following a constant delay after the previous input. Figure 8.18 gives the switch settings for f3, and Figure B.8 lists the program. The switch settings differ from f2 in that the data flow between the output cell (4,2) and the input cell (2,2) is reversed. In program f2 cell (4,2) would signal the input cell (2,2) when a value arrived from cell (4,1). This signal indicated to the input cell that the most recently input data value had produced a result at the end of the pipeline and it was time to start another value. Since program f3 runs asynchronously, the input cell must notify cell (4,2) that new data has arrived, so cell (4,2) can synchronize with the other cells receiving input data.

When an input is produced, the data goes to cells (1,1) and (2,1). Also, cells (1,2) and (4,2) get the data but do not use it. Instead, the input signals them to output data, so that soon after filter cells (1,1) and (2,1) get data in their east ports from cell (2,2), filter cells (3,1) and (4,1) will get data from (4,2).

The filter cells in this program are not synchronized. Instead, each waits for an input and start processing immediately after the input is received. Although the filter cells wait for the first byte of input data, the second byte is assumed to be no later than 24 $\mu$s behind. Therefore, there is no check made on the input queue before reading the second byte.

In f2, the input cell waited for filter cell (4,1) to produce an output before producing another input. In f3, the input cell uses the builtin timer and produces input data at the rate of one sample (2 bytes) every 100 $\mu$s. This is done to show that arrival time of the input data is not tied to the rest of the algorithm.

The 8-bit filter coefficients are treated as if the decimal point is to the left of the most significant bit. This assumes that the filter coefficients are less then one. If they are not all less than one, instructions can be added to shift the data left or right the number of bits needed to produce an output with the decimal in the same position as the input data. This can be done on a cell by cell basis so that a large coefficient in one cell will not affect the precision in



```
cell    1        2        3        4

  1     filter   zero
        128

  2     filter   input
        64

  3     filter
        128

  4     filter   output
        64
```

Figure 8.18. Switch setting for fast filter program (f3) for p=1 and q=2.

another cell. The addition of the shift instructions will add one $\mu$s per bit shifted to the execution times.

When the decimal is to the right of the MSB, the values of the coefficient can range from 1/256 = .0039 to 255/256 = .996. The sum is 24 bits with 16 bits left of the decimal.

Figure 8.17 summarizes the equivalent execution times of f3 for each step of f1. The total time for one loop is 63 $\mu$s, or a sample rate of 15.8 KHz. (If the data arrives at a slower rate the sample rate will be dictated by the arrival rate of the input data.) This is adequate for most speech recognition applications.

### 8.2.3. Summary

Two digital filtering algorithms were simulated using an 8051 8-bit microprocessor running at 12 MHz. Inter-cell communication was through an 8-bit wide pipeline between cells, with the maximum throughput of one byte every 12 $\mu$s.

The first algorithm was based on the pipelined VLSI array algorithm in Section 6.1.1. It used local pipeline communication and no broadcasts. It produced one output sample for every two *loops*. A simulation written in *xx* showed that a loop takes 1,008 $\mu$s giving a sample rate of less than 500 Hz.

The second algorithm was based on the pipelined/broadcast VLSI array algorithm in Section 6.1.2. It used the same local communication as the first algorithm, but it also used broadcasts. It was simulated by three programs, two written in assembly language and one in written *xx*. Table 8.2 presents a summary of the simulations.

Program f3 shows that a VLSI processor array using cells with the power of a current 8-bit microprocessor, can filter 16-bit speech data in real time at a sampling rate of up to 15.8 KHz. The number of coefficients in the filter does not affect the sampling rate. If more coefficients are needed, more cells can be added to the array. The only limitation may be the fan out of a broadcast. The Poker emulator can broadcast from one port to up to four other ports. So if it is necessary to broadcast to more than four ports, extra cells will have to be added as "line drivers" (see the next section). The extra cells will require

Table 8.2 Summary of simulation of digitial filtering algorithms in Poker.

| Name | Language | Input Data Size | Loop Time | Maximum Sampling Rate |
|---|---|---|---|---|
| f1 | xx | 32 bit | 906 $\mu$s | 1.1 KHz |
| f2 | 8051 | 8 bit | 33 $\mu$s | 30 KHz |
| f3 | 8051 | 16 bit | 63 $\mu$s | 15.8 KHz |

more time for the data to move through, therefore the maximum sampling rate will be decreased.

In f2, 7 $\mu$s are wasted (nops) waiting for the switch to poll the output latch, while f3 uses 15 $\mu$s out of 63 $\mu$s for nops. If the output used a queue like the input does, thus eliminating the 12 $\mu$s delay between writes to the output port, the nops could be removed from f2 and f3. Without nops, f2 can process at 33−7 = 26 $\mu$s per loop for a 38 KHz sampling rate, while f3 would run at 63−15 = 48 $\mu$s or 20.8 KHz. A sampling rate of 15.8 KHz (f2 with nops) should be sufficient for most speech applications. If processing of high quality speech requires a 20 KHz rate, this can be achieved with this modification to program f3.

These filtering algorithms map well onto the CHiP architecture. The "pipeline only" algorithm can be implemented on both the Poker emulator and Pringle. The pipeline/broadcast algorithm will run only on the Poker emulator. The Pringle hardware can not broadcast data, while the Poker emulator will allow one port to broadcast to up to four ports. The ability to broadcast is important since it allows the algorithm presented in Section 8.2.2 to be used. This algorithm has a throughput two times faster than the "no broadcast" algorithm in Section 8.2.1.

The filtering algorithms require a fast interconnection network because the data is transferred between cells at the same speed as the sampling rate. The Poker system transfers one byte every 12 $\mu$s, or one 16-bit word every 24 $\mu$s for a throughput of 1/24 $\mu$s = 41 KHz. This rate is sufficient for high quality speech processing if the processor does not have to manage the I/O buffers. The I/O buffer management could be handled by having an output queue (instead of a latch) between the processor and the interconnection network. Also, separate input queues for each port could be used. If such queues are used, the programmer would not have to wait after writing data to the interconnection network to be sure it had been sent. A more general approach would be to have a separate I/O processor which would manage the I/O queue(s) so the main processor could be used mainly for executing programs.

Since most input speech data is 11 to 12 bits, and most computations use 16 bits, each cell should have a 16-bit processor. The internal RAM of the 8051 has the same access time as its registers, making the internal RAM act like 64 16-bit fast general purpose registers. Storage based on a few fast registers is characteristic of the systolic array and is a desirable feature.

Since programming a large project in assembly language is tedious at best, the VLSI processor array must be able to execute programs written in a high level language in real time. The high level language should allow the programmer to select the precision and type (integer, floating point, etc.) of data for each variable. Therefore if only 16 bits are needed, only 16 bits will be used. If the processor which is used is like the 8051 in that it has fast internal RAM, the high level language should allow the programmer to select where the variables are stored.

The Poker system is able to implement the filtering algorithms. The second algorithm can process high quality speech in real time, if the program is carefully written in assembly language.

### 8.3. Simulation of the Autocorrelation Algorithms

Autocorrelation plays an important role in many isolated word recognition systems. It is used to find the short term autocorrelation coefficients which are then used to find the LPC coefficients. Autocorrelation, as used here, is defined as:

$$R(i) = \sum_{k=0}^{M-i-1} x(k)x(k+i) \qquad 0 \leq i \leq p$$

where *R(i)* are the autocorrelation coefficients and *x(m)* is the input signal. For speech processing the frame length, *M*, ranges from 100 to 300 samples, while *p* is between 8 and 16 [Myer80].

For these programs, *M=100* and *p=4*. The value *p=4* is chosen so the arrays of cells will fit conveniently in a four by four grid of cells. Changing *p* to the more common value of 9 will not change the throughput; however, it will change the number of cells needed.

The number of bits used for each input sample ranges from 8 for telephone quality speech to 12 for high quality speech. The number of bits for the sum can range from 16 bits to 32 bits. If all the samples in one frame of speech use 12 significant bits (i.e., the most significant bit is set), the square of each sample (used in finding R(0)) will use 24 bits. The sum of 100 24 bit values will use at most 32 bits, therefore 32 bits is sufficient for computing the sum values. It is possible that long frame sizes can result in a sum that uses more than a 32 bits, but this will happen only when most of the input samples use 12 significant bits. If most samples use all 12 bits, the signal must have a large DC bias which can be subtracted off before processing.

### 8.3.1. Poker Simulation of the Autocorrelation Algorithm

This section discusses the results of simulating the computation of autocorrelation coefficients on a VLSI processor array using five different programs (a1-a5) on the Poker system. The first two programs are written in the *xx* programming language. Program a1 uses 32 bit floating point numbers, while a2 uses 32 bit integers. Both use the APU for processing.

Programs a3-a5 are written in 8051 assembly language. Programs a3 and a5 use 16-bit integer input samples and produce 32-bit integer sums. Program a4 takes 8-bit inputs and produces a 16- bit sum. None of these programs uses the APU.

As with filtering, reducing the precision of the calculations and switching to assembly language results in a greater than tenfold increase in throughput. Although the fastest programs (a3 and a4) can process inputs as fast as 12 KHz and 45KHz*, respectively, like f2 they must be synchronized with the cell producing the input. On the other hand, program a5 processes data at a slower rate (less than 11 KHz), but can run completely asynchronously with respect to the input cell.

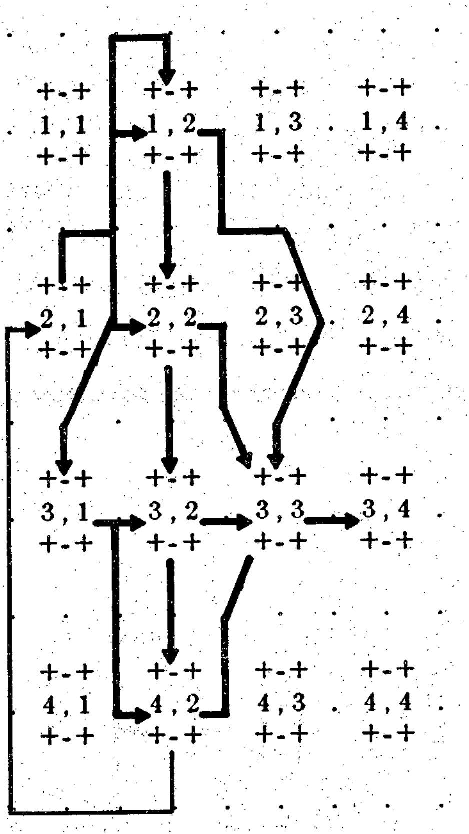
### 8.3.2. High-level Language Programs – a1 and a2

Figures 8.19, 8.20 and 8.21 show the switch settings, code names, port names, and *xx* listings for program a1. Program a2 is not listed since it is identical to a1 except all *real* declarations are changed *int* declarations. Programs a1 and a2 are based on the algorithm in Section 5.1.1. Although the assembly language programs have slightly different settings for the input cells, the autocorrelation cells are connected in the same way. The algorithm works as follows:

1) The input cell (2,1) writes sample x to its output port. This value is broadcast to the west input ports on the autocorrelation cells (1,2), (2,2), (3,2), and (4,2).

2) The Poker switch emulator cannot broadcast to more than four ports. Cell (1,2) uses two input ports, so cell (2,1) must broadcast to five ports.

*These numbers assume $p=4$ and four cells in the machine.



| cell | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | auto | | |
| 2 | input | auto | | |
| 3 | pipe | auto | merge4 | output |
| 4 | | auto | | |

Figure 8.19 Switch setting for autocorrelation programs (a1) and (a2) for VLSI processor array.

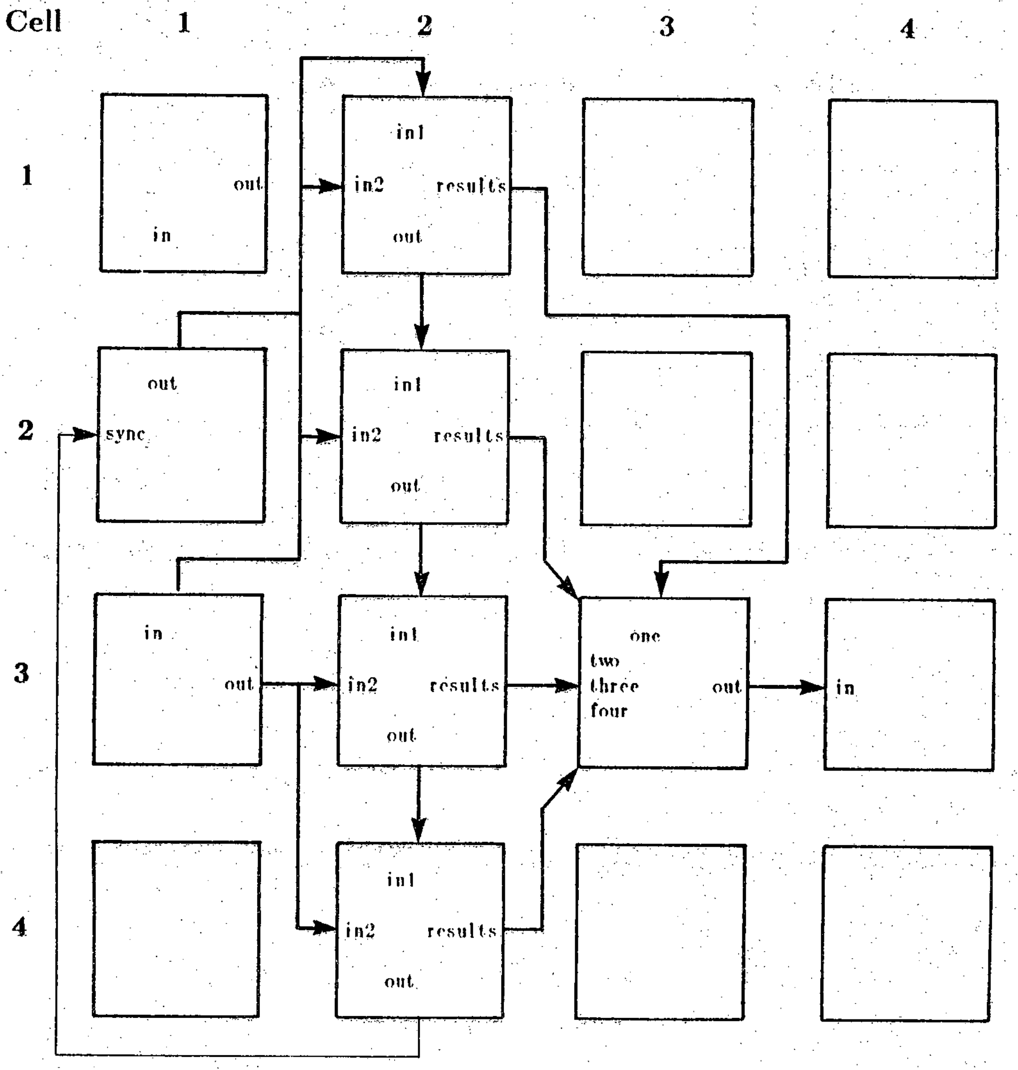
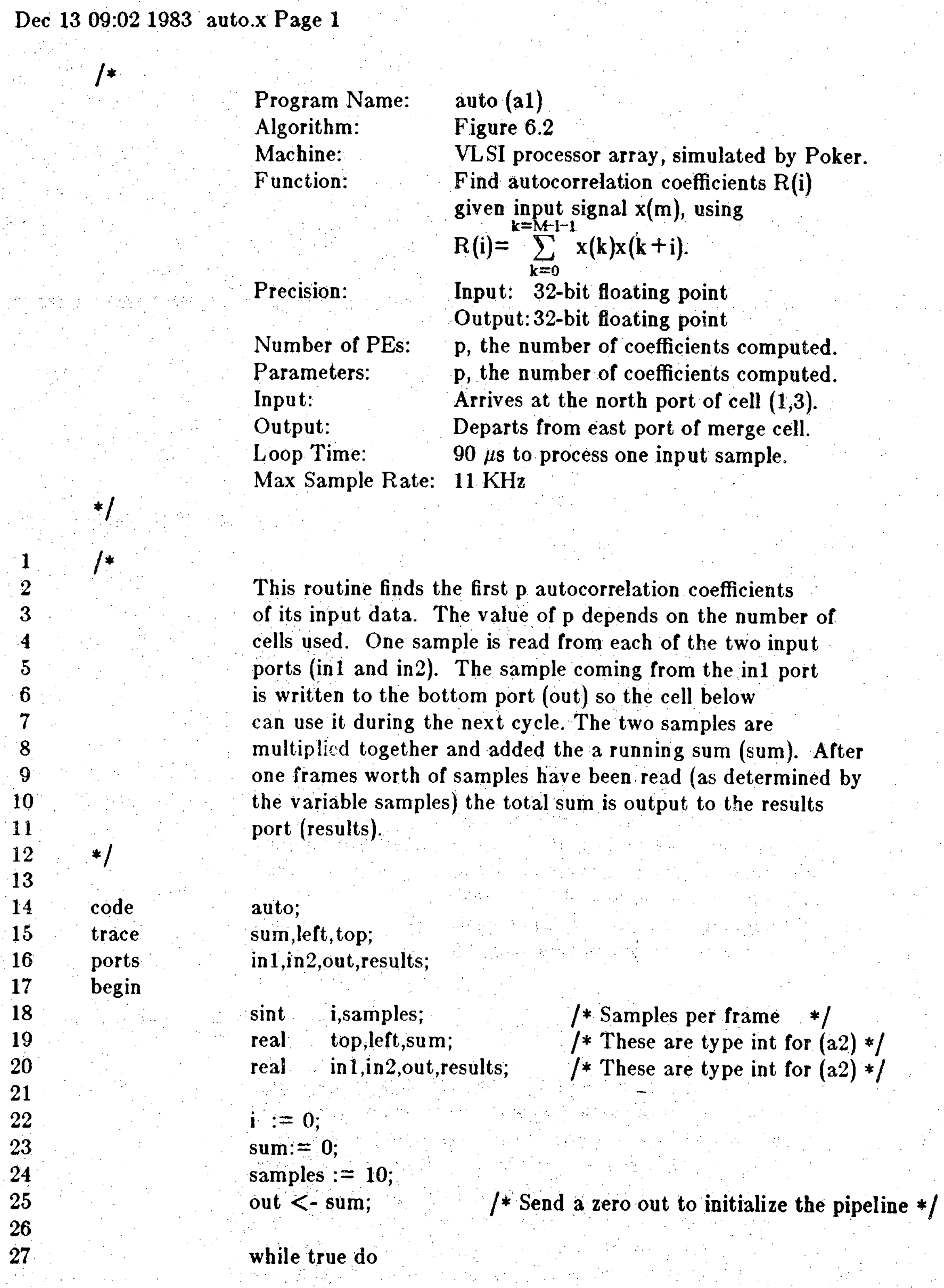


Figure 8.20. Port names for autocorrelation programs (a1) and (a2) for VLSI processor array.



Dec 13 09:02 1983  auto.x Page 1

```
/*
            Program Name:     auto (a1)
            Algorithm:        Figure 6.2
            Machine:          VLSI processor array, simulated by Poker.
            Function:         Find autocorrelation coefficients R(i)
                              given input signal x(m), using
                                    k=M-I-1
                              R(i)=   ∑   x(k)x(k+i).
                                     k=0
            Precision:        Input:  32-bit floating point
                              Output: 32-bit floating point
            Number of PEs:    p, the number of coefficients computed.
            Parameters:       p, the number of coefficients computed.
            Input:            Arrives at the north port of cell (1,3).
            Output:           Departs from east port of merge cell.
            Loop Time:        90 µs to process one input sample.
            Max Sample Rate:  11 KHz
*/

/*
            This routine finds the first p autocorrelation coefficients
            of its input data.  The value of p depends on the number of
            cells used.  One sample is read from each of the two input
            ports (in1 and in2).  The sample coming from the in1 port
            is written to the bottom port (out) so the cell below
            can use it during the next cycle. The two samples are
            multiplied together and added the a running sum (sum).  After
            one frames worth of samples have been read (as determined by
            the variable samples) the total sum is output to the results
            port (results).
*/

code        auto;
trace       sum,left,top;
ports       in1,in2,out,results;
begin
            sint    i,samples;             /* Samples per frame    */
            real    top,left,sum;          /* These are type int for (a2) */
            real    in1,in2,out,results;   /* These are type int for (a2) */

            i  := 0;
            sum:= 0;
            samples := 10;
            out <- sum;             /* Send a zero out to initialize the pipeline */

            while true do
```

Figure 8.21. *xx* listing for autocorrelation programs (a1) and (a2) for VLSI processor array.

```
Dec 13 09:02 1983  auto.x Page 2

                            begin
                            i:=i+1;
                            top <- in1;
                            left<- in2;

                            if i < samples then         /* Has one frame been processed? */
                                    begin               /* No   */
                                    out  <- top;        /* Send sample from top to cell below */
                                    sum := sum + top * left;/* Find sum    */
                                    end

                            else begin
                                    sum := sum + top * left;        /* Last sample in frame */
                                    results <- sum;         /* send out results       */
                                    sum := 0;                       /* Reinitialize, sum      */
                                    out <- sum;                     /*      and pipeline.     */
                                    i := 0;
                                    end
                            end
    end.
```

Figure 8.21 (Continued)

Dec 13 09:02 1983 input.x Page 1

```
/*
                This routine generates input data for the autocorrelation
                array.  The input is the sequence 1,2,3,....
                Each values is written to the output port (out), and the
                next value is written after a dummy value is received at
                the input port (sync).
*/

code            input;
trace           i,tmp;
ports           out,sync;
begin
                real    tmp;            /* These are type int in (a2)     */
                real    i;              /* These are type int in (a2)     */
                real    out,sync;       /* These are type int in (a2)     */
                i := 1;

                while true do
                        begin
                        out <- i;
                        tmp <- sync;    /* Wait for data out of last pe before    */
                        i := i+1;       /* sending any more out.                  */
                        end
end.
```

Dec 13 09:02 1983 merge4.x Page 1

```
/*
                This routine will merge four data streams into one by
                alternating data starting the the top input.
*/
code            merge4;
trace           tmp;
ports           one,two,three,four,out;
begin
                real    tmp;    /* These are type int in (a2)     */
                real    one,two,three,four,out;
                while true do
                        begin
                        tmp <- one;
                        out <- tmp;
                        tmp <- two;
                        out <- tmp;
                        tmp <- three;
                        out <- tmp;
                        tmp <- four;
                        out <- tmp;
                        end;
```

Figure 8.21 (Continued)

Dec 13 09:02 1983  merge4.x Page 2

```
        end.
```

Dec 13 09:02 1983  output.x Page 1

```
        /*
                            This routine is simply a sink.  It reads in real values
                            from its input port (in) and assigns them to a traced
                            variable.  It does no useful processing, but is very
                            handy for seeing what data is coming out of another
                            cell.
        */

        code                output;
        trace               out;
        ports               in;
        begin
                            real      in,out;  /* These are type int in (a2)      */

                            while true do
                                      begin
                                      out <- in;
                                      end
        end.
```

Dec 13 09:02 1983  pipe.x Page 1

```
        /*
                            This routine is like a hardware line driver.  The switch
                            emulator can not broadcast to more than 4 ports at a
                            time, so this pipe is used to increase the number of ports
                            a given cell can send data to at one time.
                            Pipe simply reads data from its input port (in) and
                            writes it (unaltered) to its output port (out).
        */

        code                pipe;
        trace               tmp;
        ports               in, out;
        begin
                            real      in,out,tmp;         /* These are type int in (a2)      */

                            while true do
                                      begin
                                      tmp <- in;
                                      out <- tmp;
                                      end
        end.
```

Figure 8.21 (Continued)

The *pipe* cell at (3,1) is used as a "line driver" so the ports on cells (3,2) and (4,2) will appear as one port to the input cell (2,1). If the problem size is increased to 8 coefficients, the input will have to be broadcast to 9 ports and another line driver will have to be added. Adding more line drivers will increase the execution time of the program since each line driver has a delay between the arrival time of the data and the time the data is broadcast to the output ports.

3) Cell (1,2) receives sample x at both of its input ports *(in1,in2)*. It writes the values from the north port *(in1)* to the south port *(out)*. It then multiplies the two input values together and adds it to the running sum. This cell is computing

$$R(0)=\sum_{k=0}^{M-1} x(k)x(k)$$

4) When cell (2,2) receives the values x from cell (1,2) it also gets the next value (x+1) from the input cell (2,1). It does the same operations (multiplication and addition) as cell (1,2) to compute:

$$R(1)=\sum_{k=0}^{M-2} x(k)x(k+1)$$

Cell (1,2) has provided the one sample delay so that although both cells are performing the same operations, they are computing different autocorrelation coefficients. The same operations are done for the other autocorrelation cells (3,2) and (4,2), with cell (3,2) computing the autocorrelation coefficient with a delay of two and cell (4,2) computing the coefficient with a delay of three.

5) When cell (4,2) writes to its south port, the data is sent to the input cell (2,1). The arrival of the data tells the input cell to write out another value. The value that just arrived has no effect on the value written out. It just synchronizes the input cell to the autocorrelation cells.

6) If fewer than *M* samples have been processed, go back to step 1), otherwise write *sum* to the east port *(results)* set sum to zero, and go to 1).

7) The *merge* cell (3,3) collects the autocorrelation coefficients and combines them in one stream for processing by the *lpc* cell which is discussed in a later section.

### 8.3.3. Execution Times – a1 and a2

Figure 8.22 shows the execution times in μs for each of the statements in the *xx* program. Two things to note about these times are:

1) Short integers *(sint)* are only eight bits long and are handled entirely by the 8051. Variables of type *real* and *int* are 32 bits long and are handled by the APU. This is why *i:=0* takes 5 μs, whereas *sum:=0* take 178 μs.

2) As discussed in Section 8.1.2.3, each cell has one hardware input queue for all the input ports. When data from an input port arrives, the data and a tag indicating the port are written into the queue. When the instruction *top* <- *in2* is executed, the program first checks to see how much data is in the *top* port buffer. If there are less than four bytes, the input queue is read until four bytes from the *top* port are found (this includes the data already in the *top* buffer). Any data read from the queue which is not for the *top* port is stored in the appropriate port buffer. The same process is followed when executing *left* <- *in2*. While *auto* is waiting for data from the north port *(in1)* it may also read data from all the other input ports. Therefore *top* <- *in2* must wait 419 μs for the data to arrive, while *left* <- *in2* requires only 93 μs since most of the data has already been read in and buffered.

A *loop* in this algorithm consists of the operations needed to input, process, and output one sample of speech. For this program, one loop takes 961 μs. After every *M* loops the computation of the autocorrelation coefficients is completed, and the result is written to an output port. If a result is output during the loop, the time increases to 1,223 μs. The execution time of the last loop is longer than the rest of the loops since the result must be written to an output port, and certain variables must be reinitialized. This gives a sampling rate of about 1 KHz which is too slow for speech analysis.

Figure 8.23 is the same algorithm using 32-bit integers for computations instead of real numbers. Here the total time for a loop is 887 μs and 1,010 μs if a result is produced. This is still not fast enough for speech processing.

Table 8.3 shows the the most time-consuming steps in the *xx* routines. Using data of type integer is adequate for speech data processing. Program (a2) can process one sample every 887 μs, which is a sampling rate of 1.1 KHz. This is not fast enough for real-time processing. As with filtering, *xx* in its

```
Count  Min  Ave  Max

                          code    auto;
                          trace   sum,left,top;
                          ports   in1,in2,out,results;
                          begin
1      0    0    0                sint    i,samples;       /* Samples per frame    */
1      15   15   15               real    top,left,sum;
                                  real    in1,in2,out,results;

1      5    5    5                i  := 0;
1      178  178  178              sum:= 0;
1      5    5    5                samples := 10;
                                  /* Send a zero out to initialize the pipeline */
1      91   91   91               out <- sum;

1      0    0    0                while true do
                                  begin
34     14   14   14               i:=i+1;
34     419  419  419              top <- in1;
34     93   93   93               left<- in2;
                                  /* Has one frame been processed? */
34     22   23.8 24                       if i < samples then
                                          begin   /* No   */
31     91   91   91                       out  <- top;
                                          /* Send sample from top to cell below */
31     318  318  318                      sum := sum + top * left;        /* Find sum*/
                                          end

31     2    2    2                else begin         /* Last sample in frame */
3      318  318  318                      sum := sum + top * left;
3      91   91   91                       results <- sum; /* send out results      */
3      178  178  178                      sum := 0;       /* Reinitialize, sum     */
3      91   91   91                       out <- sum;     /* and pipeline.*/
3      5    5    5                        i := 0;
3      0    0    0                        end
34     2    2    2                end
                          end.
```

Figure 8.22. Execution times in $\mu$s for autocorrelation program a1 using real numbers.

```
Count  Min  Ave  Max

                         code   auto;
                         trace  sum,left,top;
                         ports  in1,in2,out,results;
                         begin
1      0    0    0              sint    i,samples;       /* Samples per frame    */
1      15   15   15             int     top,left,sum;
                                int     in1,in2,out,results;

1      5    5    5              i  := 0;
1      29   29   29             sum:= 0;
1      5    5    5              samples := 10;
                                /* Send a zero out to initialize the pipeline */
1      91   91   91             out <- sum;

1      0    0    0              while true do
                                begin
31     14   14   14             i:=i+1;
31     419  419  419            top <- in1;
31     93   93   93             left<- in2;
                                /* Has one frame been processed? */
31     22   23.8 24                     if i < samples then
                                        begin    /* No    */
                                        /* Send sample from top to cell below */
28     91   91   91                     out  <- top;
28     244  244  244                    sum := sum + top * left;        /* Find sum*/
                                        end

28     2    2    2              else begin                /* Last sample in frame */
3      244  244  244                    sum := sum + top * left;
3      91   91   91                     results <- sum; /* send out results        */
3      29   29   29                     sum := 0;       /* Reinitialize, sum       */
3      91   91   91                     out <- sum;     /* and pipeline. */
3      5    5    5                      i := 0;
3      0    0    0                      end
31     2    2    2              end
                         end.
```

Figure 8.23. Execution times in $\mu$s for autocorrelation program a2 using integers.

Table 8.3 Execution times for autocorrelation programs a1 and a2.

| Program | a1 | | a2 | |
|---|---|---|---|---|
| Data Type | int | int | real | real |
| Input | 512 μs | 53% | 512 μs | 58% |
| Finding Sum | 318 μs | 33% | 244 μs | 28% |
| Output | 93 μs | 9% | 93 μs | 10% |
| Total (no result) | 961 μs | 100% | 887 μs | 100% |
| Total (with result) | 1,223 μs | 100% | 1,010 μs | 100% |

current state cannot produce code that executes fast enough for real-time processing. The following section presents assembly language implementations of the algorithm to compute autocorrelation coefficients.

### 8.3.4. Assembly Language Programs – a3 and a4

Autocorrelation of a speech signal does not require 32-bit input data, as used above, for most applications. Instead 8 or 16-bit input data is enough. Using fewer bits reduces the I/O time since less data is sent through the switch lattice, and reduces the execution time since the 8051 can do 16-bit arithmetic without sending data to the APU.

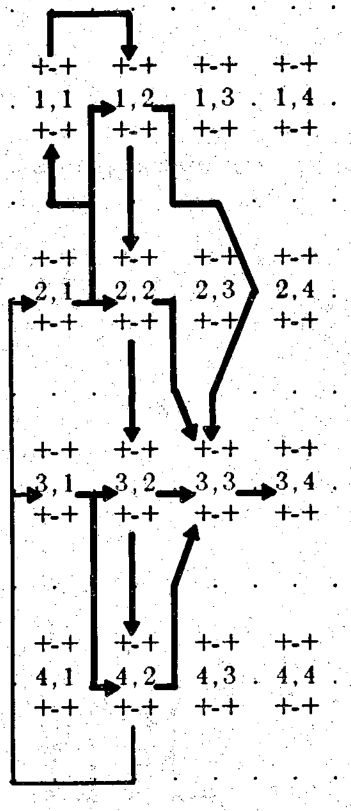
Three 8051 assembly language programs were written to compute autocorrelation coefficients. Two (a3 and a5) use 16-bit input samples and a 32-bit sum, while the other (a4) uses 8-bit inputs and a 16-bit sum. Figures B.9 and B.10 are listings of the first two programs. Figure 8.24 shows the switch setting. They perform the same calculations as the *xx* programs but with less precision. All calculations are done by the 8051; the APU is never used. All inter-cell communication is done blindly, i.e., when receiving data, no check is made to see from which port it came. There is no risk of data arriving at a cell's input queue in the wrong order if

1) all the cells are synchronized (i.e., the main loop requires the same amount of time in each cell) and
2) there is no input from cells which are not synchronized.

Unfortunately debugging synchronized code is tedious because the code in unrelated cells must be carefully timed to take the same amount of time. Data must arrive from the outside world which is not synchronized to Poker, so 2) is an unrealistic constraint. This will be addressed in Section 8.3.6.

Figure 8.25 is a summary of the equivalent execution times for the assembly routine as compared to the integer version of the *xx* routines. Table 8.4 summarizes the total time between input samples for each of the algorithms. Switching to assembly language has produced about a tenfold increase in speed. This increase comes from a combination of:

1) Reducing the input data size from 32 bits to 8 or 16 bits. This allows the 8051 to perform the arithmetic rather than sending it to the APU, which



| cell | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | pipe | auto | | |
| 2 | input | auto | | |
| 3 | input | auto | merge4 | output |
| 4 | | auto | | |

Figure 8.24. Switch settings for assembly language autocorrelation routines.

```
a1     a3     a4     a5
xx     8051   8051   8051
32bit  16 bit 8 bit  16 bit
async  sync   sync   async
                                    code    auto;
                                    trace   sum,left,top;
                                    ports   in1,in2,out,results;
                                    begin
                                            sint    i,samples;
                                            real    top,left,sum;
                                            real    in1,in2,out,results;

5      2      2      2                      i := 0;
29     4      2      4                      sum:= 0;
5                                           samples := 10;
91     17     6      17                     out <- sum;
                                            while true do
                                            begin
11     1      1      1                      i:=i+1;
419    6      3      ≥10                    top <- in1;
91     6      2      ≥10                    left<- in2;
23.8   3/5    3/5    3/5                    if i < samples then
                                                 begin            /* Not done yet */
91     4      2      4                           out  <- top;
244    60     9      60                          sum := sum + top * left;
                                                 end
2                                           else begin
244    60     9      60                          sum := sum + top * left;
91     14     10     13                          results <- sum;
29     4      2      4                           sum := 0;
91     16     7      16                          out <- sum;
5      2      2      2                           i := 0;
                                                 end
2      2      2      2                      end
                                    end.
```

Figure 8.25. Execution times in $\mu s$ for autocorrelation program using 8, 16, and 32-bit inputs.

Table 8.4 Summary of execution times for autocorrelation programs.

| Program | Language | Data Size and Type | Communication | Time for One Loop | Time for Last Loop | Time for 100 Samples | Sample Rate |
|---|---|---|---|---|---|---|---|
| a1 | xx | 32-bit real | async | >961$\mu$s | >1,233$\mu$s | >96,372$\mu$s | <1,037Hz |
| a2 | xx | 32-bit int | async | >887 $\mu$s | >1,010 $\mu$s | >88,823 $\mu$s | <1,125 Hz |
| a3 | 8051 | 16-bit int | sync | 82 $\mu$s | 116 $\mu$s | 8,234 $\mu$s | 12,144 Hz |
| a4 | 8051 | 8-bit int | sync | 26 $\mu$s | 47 $\mu$s | 2,621 $\mu$s | 38,153 Hz |
| a5 | 8051 | 16-bit int | async | >90 $\mu$s | >123 $\mu$s | >9,033 $\mu$s | <11,071 Hz |

is time consuming. Also, the smaller data size requires less time to move through the network.

2) Storing all variables in internal RAM. *xx* stores all variables in external RAM, which requires more time to access.

3) Overlapping data transfers with computation. Therefore, when waiting for the LSB to be read from the output latch, the MSBs are being computed.

Steps 1) and 2) could be implemented by a compiler, thus possible making real-time processing possible without using assembly language.

The seventh column of Table 8.4 shows the time required to process one 100-sample frame of speech. Using 16-bit samples, a sample rate of 10 KHz is easily obtained. This is fast enough for telephone quality speech, but not for high quality speech. Dropping to 8-bit inputs allows a sample rate of about 38 KHz which is fast enough for most speech applications, but is not enough precision for high quality speech. These rates present one problem: it is possible to sample at a high rate (38 KHz), or with high precision (16-bit inputs), but not both. These rates assume there is some buffering of input data during the longer last loop so no data is lost.

### 8.3.5. Potential Problems – a3 and a4

The assembly routines assume the input value will enter at a given time. There is an 8 $\mu$s window in the 8-bit version during which the input data must arrive. The 16-bit version has a 21 $\mu$s window. If the data arrives outside this window, data will be lost.

Because of this narrow window, a *pipe* cell cannot be used to broadcast the input data since it introduces delays in the arrival times. Instead, two identical input cells are used along with broadcasting, as the switch setting in Figure 8.24 shows. The *pipe* (1,1) here is used so the data arriving at cell (1,2) will arrive in the proper order.

This "patch job" of duplicating the input cell is sufficient for demonstrating the system works, but is not practical for processing real data. The Pringle hardware cannot broadcast, therefore one input cell would be needed for each autocorrelation cell. The following section presents a method to overcome this problem.

### 8.3.6. Asynchronous Computing – a5

The last assembly language program (a5) allows the *auto* cells to run asynchronously with respect to the *input* cell. Figure 8.26 is the switch setting and code names for the autocorrelation program listed in Figure B.11. Figure 8.25 and Table 8.4 summarizes the results. Asynchronous execution is achieved as follows: In the synchronous programs, the order of execution is:

1) Read input from external world.
2) Read input from cell above.
3) Compute sum while next external input arrives.
4) Write data to cell below.

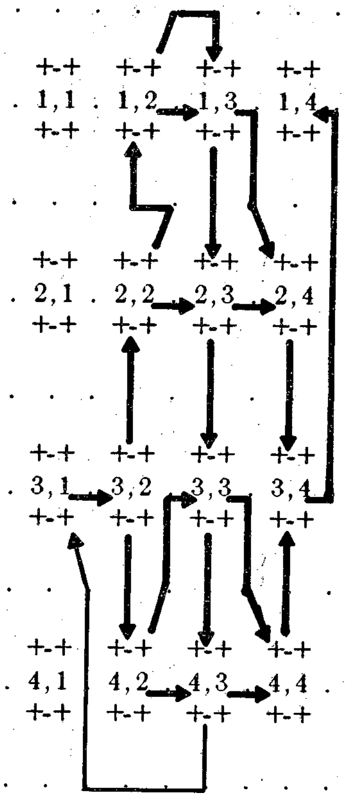
Step 3 overlaps the computations with data input. This program is synchronous since the data must arrive during the computation.

To run asynchronously the order of execution is changed to:

1) Read input from cell above.
2) Wait for input from external world.
3) Write data to cell below.
4) Compute sum while input from cell above arrives.

There is still overlap of computation with input, but the input is from another cell, not the external world. This new program adds only the slight overhead of checking for the arrival of the external input.

The only assumption made is that the external input arrives at all cells at the same time. This is a valid assumption if the hardware can perform a broadcast. Systolic arrays cannot broadcast data, so program a5 uses a tree like configuration of cells to distribute the input data as a broadcast would. This method, however, does not deliver the data to all the cells at the same time. Figure 8.27 shows two columns of cells. The cells in column one form a broadcast tree while the cells in column two are cells receiving the broadcast data. The number in each box is the arrival time of the data assuming it starts in cell (5,1) at time *t=1* and that a write to a port takes one time unit. These arrival times also assume a cell can send data to both output ports with



| cell | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | split<br>2 | auto | output |
| 2 | | split<br>2 | auto | merge<br>1 |
| 3 | input | split<br>1 | auto | merge<br>2 |
| 4 | | split<br>2 | auto | merge<br>1 |

Figure 8.26. Switch setting for autocorrelation program a5.

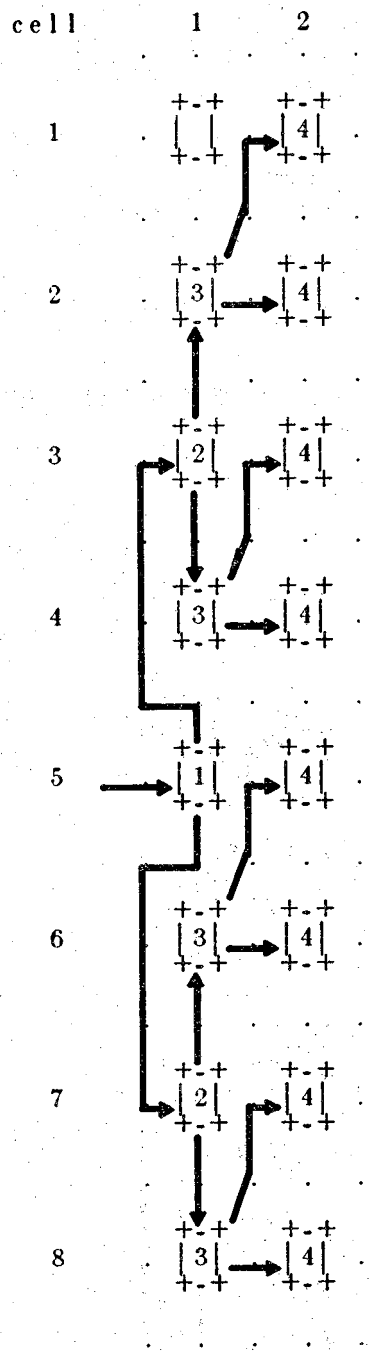


Figure 8.27 Time delays in using tree to broadcast. One port can send data to two ports with one write instruction.

one write instruction. This means that the data will arrive at all the cells in column two at the same time.

Figure 8.28 on the other hand, assumes the program can write to only one port at a time. The data arrives at cell (5,1) at time *t=1*. Cell (5,1) first writes to its southwest port at time *t=2*, then to its northwest port at time *t=3*. Figure 8.28 shows that the southwest port of cell (5,1) goes to cell (7,2), and the northwest port of cell (5,1) goes to cell (3,1). Cell (7,1) gets the data from cell (5,1) at time *t=2* and first sends it to its south port, then its north port. Cell (3,2) receives and sends its data one time unit later. Cells (2,1), (4,1), (6,1), and (8,1) all perform the same operations, only at different times as shown by the Figure 8.28 When using this scheme to broadcast to 8 cells, there is only a one unit delay between adjacent cells. There can be timing problems with such a broadcast tree. In the assembly language algorithm, column two acts as the pipeline in Figure 8.26. Each cell receives two data items; one broadcast item, and one data item being passed through the pipeline. Therefore each cell in column two receives data from two ports. The first port comes from the broadcast tree and is called the broadcast data. The second port comes from the cell above and is called the pipeline data. A cell writes pipeline data to the cell below it after receiving both broadcast data and pipeline data. Although this is how the autocorrelation algorithm functions, this problem can be generalized to any algorithm that fits the above description. Checking the input queue tag and buffering the input data is a time consuming task, so the program is structured so that the data will arrive in the queue in a known order. Since the input queue direction tag is not checked, one of the two assumption must be made:

1) Data comes from the broadcast port first.
2) Data comes from the pipeline port first.

The following shows that either of the above assumptions can result in data arriving in the wrong order.

Assume 1) and consider cell (4,2). Broadcast data arrives at time *t=5;* suppose pipeline data arrives at time *t=5.4** and the pipeline data is written at

*Since the processor can execute instructions faster than the data travels between cells, it is possible for data to be written into the network at a non-integer number of network units from the time the first data was written into the network.

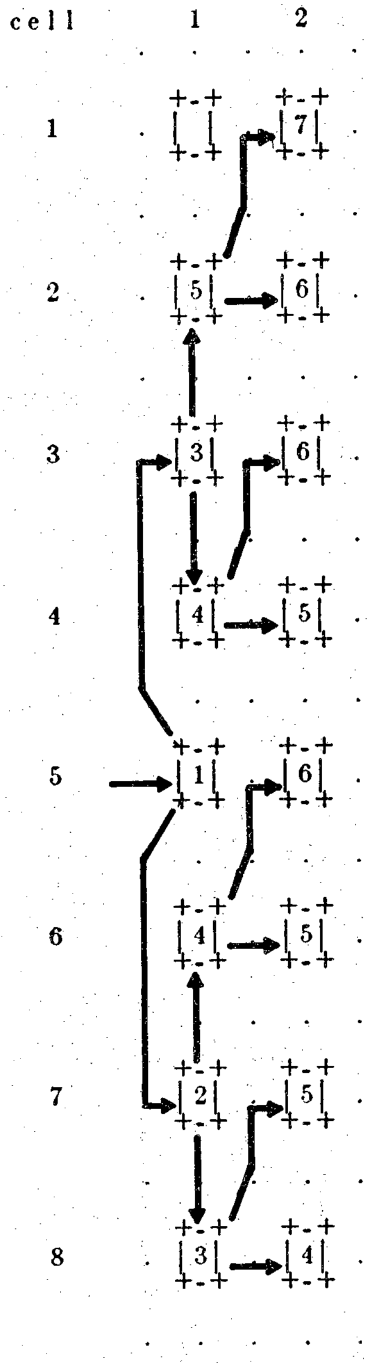


Figure 8.28 Time delays in using tree to broadcast. One port can send data to only one port with one write instruction.

time $t=5.8$. Cell (5,2) assumes the broadcast data will arrive first at $t=6$, but cell (4,2) sent its pipeline data to cell (5,2) before cell (6,1) sent the broadcast data. This is a problem.

Now assume 2), that the pipeline data comes first. Cell (3,2) receives pipeline data from the cell above it and at time $t=6$ receives its broadcast data. Suppose at time $t=6.4$ cell (3,2) sends its pipeline data to cell (4,2). Cell (4,2) expects the pipeline data first, but at time $t=5$ the broadcast data arrives. Again a problem.

There is one receiving cell, call it A, whose data is always written first as it travels through the broadcast tree. In Figure 8.28, this is cell (8,2). There is one cell, call it B, whose data is always written last as it travels through the tree, this is cell (1,2) in Figure 8.28. As the number of cells receiving the data increases, the number of levels in the broadcast tree must be increased. Each additional level of the tree adds on network delay for data arriving at cell A, and two network delays for data arriving at cell B. Therefore, the difference in arrival times increases as the number of cells receiving the broadcast data increases.

Five possible solutions to this problem are:

1) Lower the input rate so all the broadcast data will have propagated through the tree before any pipeline data arrives.
2) Build delays into the broadcast network to be sure the data arrives at all processing cells at about the same time.
3) Use separate input queues for each port.
4) Allow a port to broadcast to two ports.
5) Allow a port to broadcast to any number of ports.

From the programmer's point of view, solution five is the best solution in that not being able to broadcast data is an architectural limitation. The solution to such a limitation is a different architecture. Using a general broadcast frees up the cells in the broadcast tree so they can perform some other task. Solution five is the most expensive solution in that it requires a hardware change.

Solution four is a less expensive solution than five since it may require fewer hardware modifications. The tree broadcast can be used, as shown in Figure 8.27, to broadcast data so that it arrives at the same time at the destination cells, assuming the cells in the tree can broadcast to two other cells with one write instruction. Solution three would require the least expensive hardware modifications. Having separate input queues for each input port would eliminate the arrival order problem.

Solution one is used here since spacing the input samples 150 $\mu$s apart is slow enough for all cells to complete computing before the next sample arrives. This does decrease the throughput, but 150 $\mu$s between samples is fast enough for telephone quality speech. It is not, however, fast enough for high quality speech.

### 8.3.7. Summary

Table 8.4 summarizes the results of the five programs for autocorrelation discussed in this section. As with the filtering algorithms, the programs written in *xx* cannot process data fast enough for real-time speech processing. The three programs written in assembly language show that the 8-bit 8051 microprocessor can process at real-time speeds with throughput ranging from 12 to 38 KHz. Although computing more coefficients may increase the delay time between input and output (because it requires a larger broadcast tree), it does not change the throughput, but only the delay time between the arrival of the last sample and the output of the results. If more coefficients must be computed, more cells can be added to the array.

Program a5 showed that a broadcast can be done with a tree-like structure of cells. The problem with this type of broadcast is the variation in arrival times at the destination cells. This problem could be overcome by allowing a general broadcast to many ports, or simply by allowing a broadcast from one port to two ports. This simple broadcast would allow the tree structure to broadcast data to many cells without the variation in arrival times.

The simplest hardware change that would allow the programs for execute faster would be to have separate input queues for each port. This would allow a5 to process data at 10 KHz instead of 6.67 KHz.

## 8.4. Simulation of Parallel Linear Prediction Algorithms

Both speech synthesis and recognition frequently use linear predictive coding (LPC). The LPC coefficients model the vocal tract as an all pole filter, while the error signal from the analysis represents the excitation of the vocal chords. A speech recognition system divides the the speech signal into 10 to 20 ms frames and finds the LPC coefficients for each frame. Therefore, a real-time system must process one frame of 100 to 400 samples every 10 to 20 ms. Generally, 16-bit signed fixed-point coefficients are used, but some applications can use as few as 10 bits [MaGr74].

The LPC coefficients are found using the autocorrelation method [RaSc78]. The previous section showed that p cells can compute p autocorrelation coefficients. The output from each cell is merged into one cell. This section describes the LPC program that reads the autocorrelation coefficients from one input port and writes the LPC coefficients to the output port.

Although Siegel's method for computing LPC coefficients presented in Section 5.4.1 does achieve some speedup over the serial method, the method simulated here is entirely serial. A single 8051 with an attached APU is able to compute the coefficients in real time. Figure 8.29 lists the *xx* program used. It is a direct implementation of Durbin's recursive solution as discussed in Section 4.4. The execution times, in $\mu$s, for computing 8 LPC coefficients are listed to the left of each statement. Table 8.5 shows the total execution times for various numbers of coefficients. The time to compute 8 coefficients is 42 ms, which is two to four times longer than the desired 10 to 20 ms. Three possible solutions to this problem are: improve the *xx* compiler, use a faster APU, or use multiple cells. The following sections discuss each of these solutions.

```
                            /*
                                    Program Name:   lpc
                                    Algorithm:      Figure 4.2.
                                    Machine:        VLSI processor array, simulated by Poker.
                                    Function:       Find LPC coefficients using
                                                    Durbin's method.
                                    Precision:      Input:  32-bit floating point.
                                                    Output: 32-bit floating point.
                                    Number of PEs:  1
                                    Parameters:     p, the number of coefficients computed.
                                    Input:          Autocorrelation coefficients
                                                    arrive at "in" port.
                                    Output:         Energy (R[0]) is sent out "out" port
                                                    followed by p LPC coefficients.
                                    Typical Time:   38,421 μs for p=8.
                            */

Count  Min  Ave  Max
                            code    lpc;
                            trace   k,E,i,j;
                            ports   in,out;
                            begin
1      10   10   10                 sint    i,j,p;
1      0    0    0                  int     itmp,in;
1      0    0    0                  real    a[10],          /* LPC coefficients        */
1      0    0    0                          aold[10],       /* old LPC coefficients */
1      5    5    5                          E,              /* Prediction error        */
1      5    5    5                          k,
                                            out,            /* output port             */
1      0    0    0                          R[10],          /* autocorrelation coefs  */
1      0    0    0                          tmp;

1      5    5    5                  p := 8;

1      0    0    0                  while true do
                                    begin
1      20   20   20                 for i := 0 to p do      /* Read in autocorrelation coefs*/
                                            begin           /* Starting with R(0)     */
9      262  277  394                        itmp <- in;
9      193  193  193                        k := itmp;
9      79   79   79                         R[i+1] := k;    /* All R[] indexs are +1 since*/
9      4    9    10                 end;                /* xx indexs start at 1          */

1      70   70   70                 E := R[1];
1      91   91   91                 out <- E;               /* Send R[1] to endpoint routine*/

1      20   20   20                 for i := 1 to p do
                                            begin
```

Figure 8.29. Durbin's method for finding LPC coefficients from autocorrelation coefficients.

```
8       52   52   52                      k := 0.0;
8       32   33   38                      for j := 1 to i-1 do
28      385  386  387                            k := k + aold[j] * R[i-j+1];
8       351  351  351                     k := (R[i+1] - k) / E;
8       495  495  495                     tmp := k*k; E := (1.0 - tmp) * E;
8       73   73   73                      a[i] := k;
8       32   32   38                      for j := 1 to i-1 do
28      433  434  435                            a[j] := aold[j] - k * aold[i-j];
8       24   24   24                      for j := 1 to i do
36      102  103  104                            aold[j] := a[j];
8       4    9    10                      end;

1       20   20   20                for i := 1 to p do        /* Send out lpc coefs starting with a1*/
                                          begin
8       73   73   73                      k := aold[i];
8       91   91   91                      out <- k;
8       4    9    10                      end;

1       2    2    2                 end
                              end.
```

Figure 8.29 (Continued)

Table 8.5 Execution times for the LPC program in Figure 8.29.

| Number of Coefficients | Input Time | Computation Time | Output Time | Total Time |
|---|---|---|---|---|
| 4 | 2,810 $\mu$s | 10,246 $\mu$s | 712 $\mu$s | 13,768 $\mu$s |
| 7 | 4,484 $\mu$s | 27,607 $\mu$s | 1,231 $\mu$s | 33,322 $\mu$s |
| 8 | 5,042 $\mu$s | 35,240 $\mu$s | 1,404 $\mu$s | 41,686 $\mu$s |

### 8.4.1. Improve the *xx* Compiler

Since this method for computing LPC coefficients uses real numbers, the *xx* compiler uses the APU. The 8051 accesses the APU by pushing and popping data to and from the APU's stack. The APU is given an operation which it performs on the data on the stack and the result is left on the top of the stack. Pushing and popping data from the APU stack is a time consuming operation because the APU stack is memory mapped as external RAM. The *xx* compiler does not optimize the stack operation, so when:

```
for j := 1 to i−1 do
     k := k + aold[i] * R[i−j];
```

is executed the 8051 uses the following stack operations (assuming i=3):

```
1     push   k
2     push   aold[i]
3     push   R[i−j]
4     *      (multiply top two elements and
             leave the results on top of the stack.)
5     +      (add top two elements and
             leave the results on top of the stack.)
6     pop    k

7     push   k
8     push   aold[i]
9     push   R[i−j]
10    *
11    +
12    pop    k
```

Lines 6 and 7 show an extra push/pop operation which is not needed. A simple improvement to the *xx* compiler would be to allow one variable to be declared as an "APU stack variable" and the compiler would know to leave it on the stack. This could save many unneeded pushes and pops.

### 8.4.2. Use a Faster APU

The Intel 8231 APU requires at most 92 μs for a floating-point addition, 93 μs for a subtraction, 42 μs for a multiplication, and 43 μs for a division. These times are too slow for speech processing. For example, the two most time consuming lines in the LPC program are:

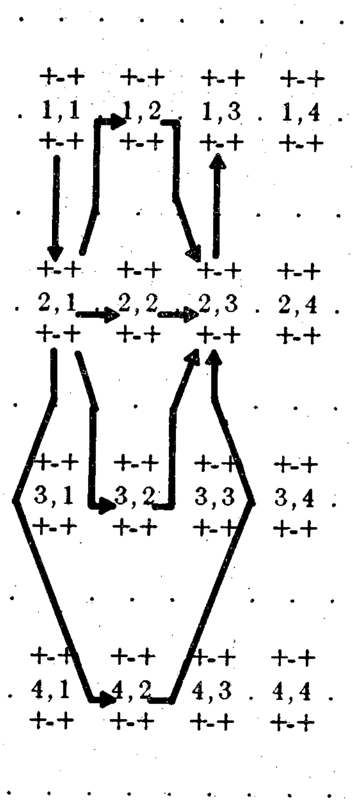
```
A)   k := k + aold[j] * R[i-j+1]; and
B)   a[j] := aold[j] - k * aold[i-j];.
```

Line A uses 387 μs per execution and is run 28 times when p=8. Line B uses 434 μs per and is executed 28 times. If the execution times of lines A and B were reduced to only the time used by the APU, they would require 134 μs and 135 μs respectively. This is 253 μs and 299 μs less time for a total savings of 28*253 + 28*299 = 15,456 μs for the entire program. The total execution time for the *lpc* program is 41,686 μs. Subtracting the time saved from the total time leaves 26,230 μs which is still too slow for real time processing. Therefore, by ignoring the overhead of indexing into arrays and sending data to and from the APU on the two most time consuming statements, the program is still unable to run in real time. A solution to this problem would be to use a faster APU.

### 8.4.3. Use Multiple Cells

Unlike Siegel's method where one LPC computation was divided among many cells, each cell could perform the LPC analysis on a different frame of speech. Figures 8.30, 8.31, and 8.32 show the switch settings and code names, port names, and *xx* program listing, respectively, for the multiple cell LPC program. The program *demux* receives the input coefficients from the autocorrelation program (the autocorrelation program is replaced by the *input* program for testing purposes). *demux* sends the first 9 coefficients (one frame) to the *lpc* cell (1,2). The next 9 coefficients are sent to *lpc* cell (2,2) and so on. After *lpc* cell (2,4) receives its coefficients, the next 9 coefficients are sent to cell (2,1). The *mux* cell collects the outputs from each *lpc* cell to form one data stream similar to the input stream into the *demux* cell.

Each *lpc* cell receives one out of every four frames. If a frame's length is 10 ms, each cell will have 40 ms to compute its LPC coefficients before receiving another input frame. Table 8.5 shows that the *lpc* cell requires ≃42 ms – slightly longer than the 40 ms that is available. The extra 2 ms could be trimmed from the *lpc* program by optimizing the APU stack operations as discussed in the previous section. If a shorter frame length is used, more *lpc* cells can be used to increase the throughput.



| cell | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | input | lpc | output | |
| 2 | demux | lpc | mux | |
| 3 | | lpc | | |
| 4 | | lpc | | |

Figure 8.30. Switch settings and port names for multiple LPC cell program.

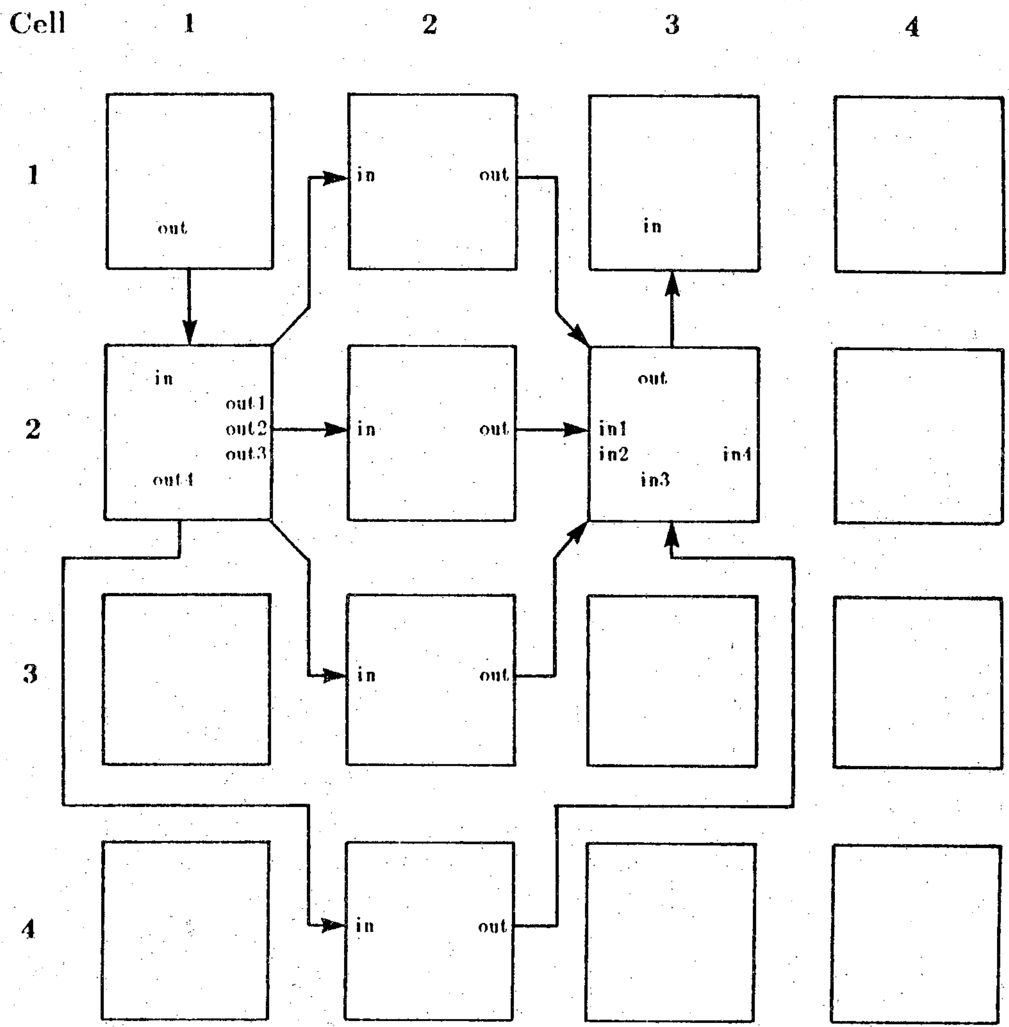
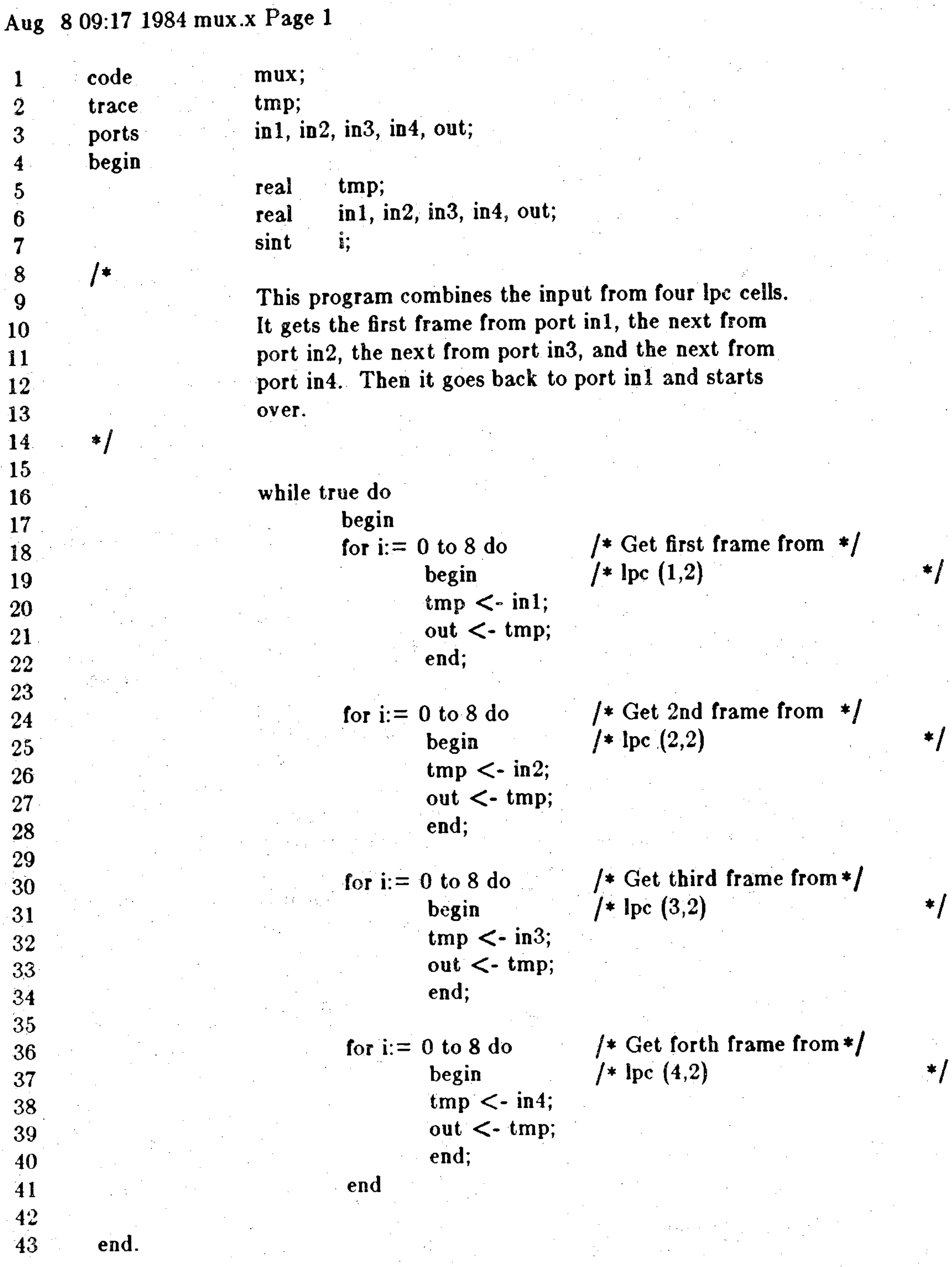


Figure 8.31. Port names for multi-cell LPC program.

Aug  8 09:17 1984 demux.x Page 1

```
code            demux;
trace           tmp;
ports           in, out1, out2, out3, out4;
begin
                int     tmp;
                int     in, out1, out2, out3, out4;
                sint    i;
/*
                This program sends its input to four lpc cells.
                It sends the first frame to port out1, the next to
                port out2, the next to port out3, and the next to
                port out4.  Then it goes back to port out1 and starts
                over.
*/

                while true do
                        begin
                        for i:= 0 to 8 do       /* Send first frame to  */
                                begin           /* lpc cell at (2,1)          */
                                tmp <- in;
                                out1 <- tmp;
                                end;

                        for i:= 0 to 8 do       /* Send 2nd frame to    */
                                begin           /* lpc cell at (2,2)          */
                                tmp <- in;
                                out2 <- tmp;
                                end;

                        for i:= 0 to 8 do       /* Send third frame to  */
                                begin           /* lpc cell at (2,3)          */
                                tmp <- in;
                                out3 <- tmp;
                                end;

                        for i:= 0 to 8 do       /* Send forth frame to  */
                                begin           /* lpc cell at (2,4)          */
                                tmp <- in;
                                out4 <- tmp;
                                end;
                        end
end.
```

Figure 8.32. *xx* program listing for multi-cell LPC program.



Aug  8 09:17 1984 mux.x Page 1

```
code            mux;
trace           tmp;
ports           in1, in2, in3, in4, out;
begin
                real    tmp;
                real    in1, in2, in3, in4, out;
                sint    i;
/*
                This program combines the input from four lpc cells.
                It gets the first frame from port in1, the next from
                port in2, the next from port in3, and the next from
                port in4.  Then it goes back to port in1 and starts
                over.
*/

                while true do
                        begin
                        for i:= 0 to 8 do       /* Get first frame from  */
                                begin           /* lpc (1,2)                  */
                                tmp <- in1;
                                out <- tmp;
                                end;

                        for i:= 0 to 8 do       /* Get 2nd frame from  */
                                begin           /* lpc (2,2)                  */
                                tmp <- in2;
                                out <- tmp;
                                end;

                        for i:= 0 to 8 do       /* Get third frame from*/
                                begin           /* lpc (3,2)                  */
                                tmp <- in3;
                                out <- tmp;
                                end;

                        for i:= 0 to 8 do       /* Get forth frame from*/
                                begin           /* lpc (4,2)                  */
                                tmp <- in4;
                                out <- tmp;
                                end;
                        end

 end.
```

Figure 8.32 (Continued)

Although this form of parallelism has the throughput needed for real-time processing, it introduces a constant delay, i.e., it takes 42 ms to compute one frame of LPC coefficients even though a frame is computed every 10 ms. The result is the input to the cell which follows the *mux* cell will be delayed by about 30 ms (40 ms computation time minus the 10 ms frame length).

### 8.4.4. Summary

This section has presented a serial program to compute LPC coefficients given the autocorrelation coefficients. It showed that 8 coefficients can be computed in 42 ms which is two to four times longer than the time needed for real-time processing. Three solutions where given to improve the execution time. The first was to improve the *xx* compiler to use an "APU stack variable." The compiler would leave this variable on the APU stack thus optimizing the stack operations. This solution will not decrease the execution time enough unless the second solution is used. The second was to use a faster APU since the Intel 8231 is too slow for speech processing. The last solution was to use multiple cells, each running a serial LPC program. A *demux* cell would assign alternate input frames to each LPC cell in a round robin fashion. A *mux* cell would then collect the output from each of the LPC cells. This method has little overhead of parallelism since each cell is running a serial program.

The LPC program is the first speech processing program that uses the APU. Although the APU can perform fixed and floating point arithmetic, until now its use has been avoided. This is due to the overhead in communicating with it and its slow execution times. The APU stack is memory mapped into the 8051's external RAM address space. The 8051 accesses all external RAM through its single *dptr* register. Therefore, if a 32-bit value stored in external RAM is to be pushed on the APU stack, the *dptr* must be set twice for each byte transferred (once to point to a byte in the variable and once to point to the APU stack), giving a total of 8 times. Setting the *dptr* requires 2 $\mu$s, so 16 $\mu$s are used just setting the *dptr*. This extra setting of the *dptr* can be avoided if the APU stack is mapped into one of the 8051's built-in I/O ports. The *dptr* can point to the variable in external RAM, and the 8051 can access the APU stack directly through the its built-in port. This simple modification to the hardware would decrease the time needed to use the APU.

### 8.5. Simulation of Linear Time Warping (LTW) Algorithms

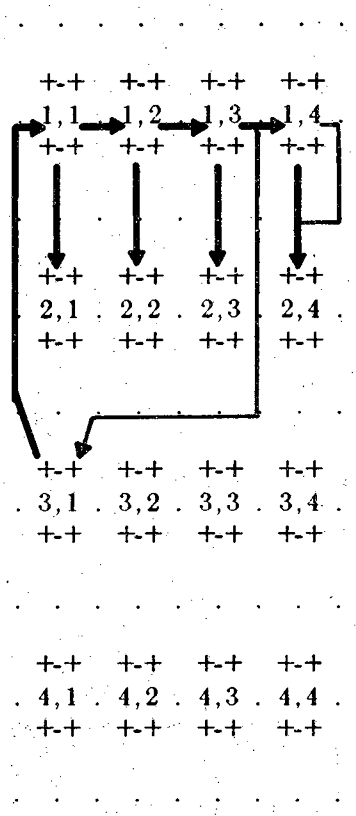
In a typical isolated word recognition system, linear time warping occurs after the endpoint detection and before the dynamic time warping. Its purpose is to take an utterance of variable length and stretch or shrink it, in the time domain, until it is a fixed length. Isolated utterances can range from 20 to 80 frames in length, where a frame consists of 8 LPC coefficients. Some systems will stretch or shrink the utterance to a 40 frame length. Only after detecting the utterance can the LTW program process the speech data. Since isolated words are about one third to one half seconds in duration, the LTW must be able to perform its operation in about 300 to 500 ms.

The LTW algorithm presented in Section 6.3.2 is implemented on the Poker system and the next section presents l1, the resulting program. A later section discusses a second LTW program, l2, which is a single processor algorithm. The data throughput needed by the LTW processor is slow (500 ms between utterances) compared to the other parts of the speech recognition system. A single cell may be able to perform the LTW task in real time.

#### 8.5.1. Parallel LTW – l1

Figures 8.33, 8.34, and 8.35, show the switch settings, port names, and *xx* program listing for l1, the parallel LTW program. The l1 program uses one *ltw* cell per coefficient, therefore in the figure, the switch settings are for four coefficients per frame. The following describes how the program works and discusses the execution times for using l1 in a typical isolated word recognition system.

In the program, cell (3,1) outputs the input frames which go to the west port of *ltw* cell (1,1). All the coefficients enter cell (1,1) and it passes all but one to cells (1,2), (1,3), and (1,4). Each cell keeps one coefficient and passes the other coefficients on to the cells to the right. The algorithm works as follows.



| cell | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | ltw<br>4 | ltw<br>3 | ltw<br>2 | ltw<br>1 |
| 2 | output | output | output | output |
| 3 | input | | | |
| 4 | | | | |

Figure 8.33. Switch setting for multi-cell LTW program.

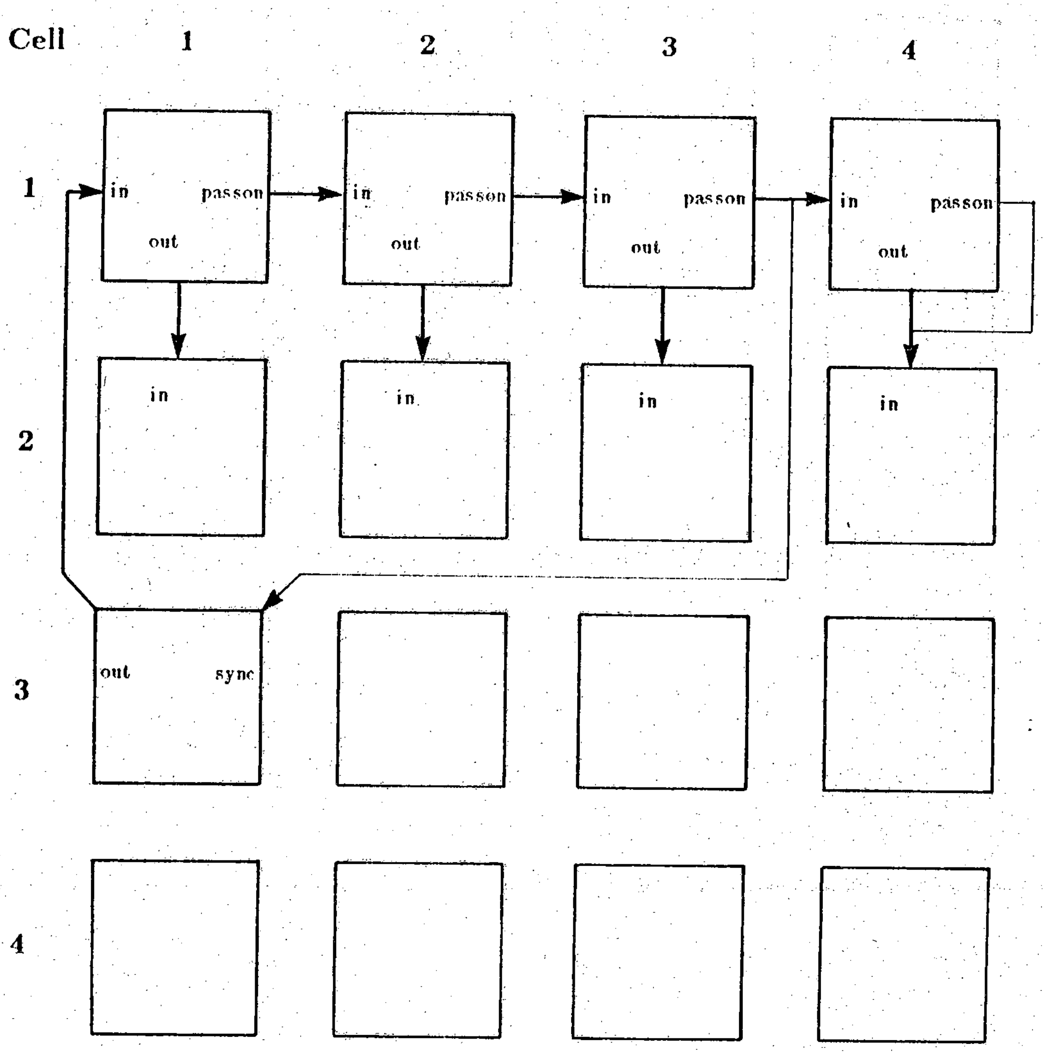


Figure 8.34. Port names for multi-cell LTW program..

```
Dec 20 11:48 1983  ltw.x Page 1

        /*
                        Program Name:     ltw
                        Algorithm:        Section 6.3.2
                        Machine:          VLSI processor array, simulated by Poker
                        Function:         This routine does a linear time warp.
                                          The data enters in from the left as p
                                          coefficients per frame.  The first cell
                                          takes the last coefficient and keeps it
                                          and passes the p-1 preceding
                                          coefficients on the the cells to the
                                          right.  Each of the other cells do the
                                          same thing until the right most cell
                                          only inputs one coefficient.  In the
                                          end, cell 1 will have lpc coefficient k
                                          for all frames, and cell k will have
                                          lpc coef. 1.  -1 is input to show the
                                          end of data and time to start
                                          computing.  Each cells computes the new
                                          warped output using only its lpc coef.
                        Precision:        Input:  32-bit floating point
                                          Output: 8-bit, unsigned
                        Number of PEs:    p, the number of LPC coefficients.
                        Parameters:       p, the number of LPC coefficients.
                        Input:            LPC coefficients enter the leftmost cell.
                        Output:           Warped coefficients exit the south
                                          port of each LPC cell.
                        Loop Time:        Does not apply
                        Typical Time:     92 µs for p=8 and I=40
        */

        code            ltw(number);
        trace           T,j,i,tmp;
        ports           in,out,passon;
        begin
                        bool inputting;

                        sint    k,
                                j,J,      /* Number of frames in input utterance */
                                i,I;      /* Number of frames in output utterance   */
                        int     number;        /* Number of coefs. given cell will get  */
                                          /* leftmost cell should have number=p, */
                                          /* the next will have p-1, until the    */
                                          /* rightmost will have number=1           */

                        real    factor,   /* ratio of J/I  */
```

Figure 8.35.  Code for multi-cell LTW program.

Dec 20 11:48 1983 ltw.x Page 2

```
        R[80],  /* Input utterance      */
        s,      /* scale        */
        onems,  /* 1-s                  */
        tmp,
        T1,T2,  /* Patch Job            */
        T,      /* Ouput utterance      */
        in,passon;
int     out;

I := 2;

while true do
begin
j := 0;
inputting := true;

while inputting do
        begin
        tmp <- in;                /* get first input for yourself     */
        if tmp > 0 then           /* if -1, it's the end
                                          of the input */
                begin
                for k := 2 to number do
                        begin   /* Send the rest to the other cells */
                        passon <- tmp;
                        tmp <- in;
                        end;
                j := j + 1;
                R[j] := tmp;
                end
        else begin
                inputting := false;
                for k := 2 to number do
                        passon <- tmp;
                end;
        end;                      /* of inputting loop       */

        J := j;
        factor := (J-1)/(I-1);

        for i := 1 to I do
                begin
                tmp := (i-1) * factor + 1.0;
                j := tmp;
                s := 64.0 * tmp - j;
                onems := 64.0 - s;
                T1:= onems * R[j];
                T2:=   s  * R[j+1];
```

Figure 8.35 (Continued)

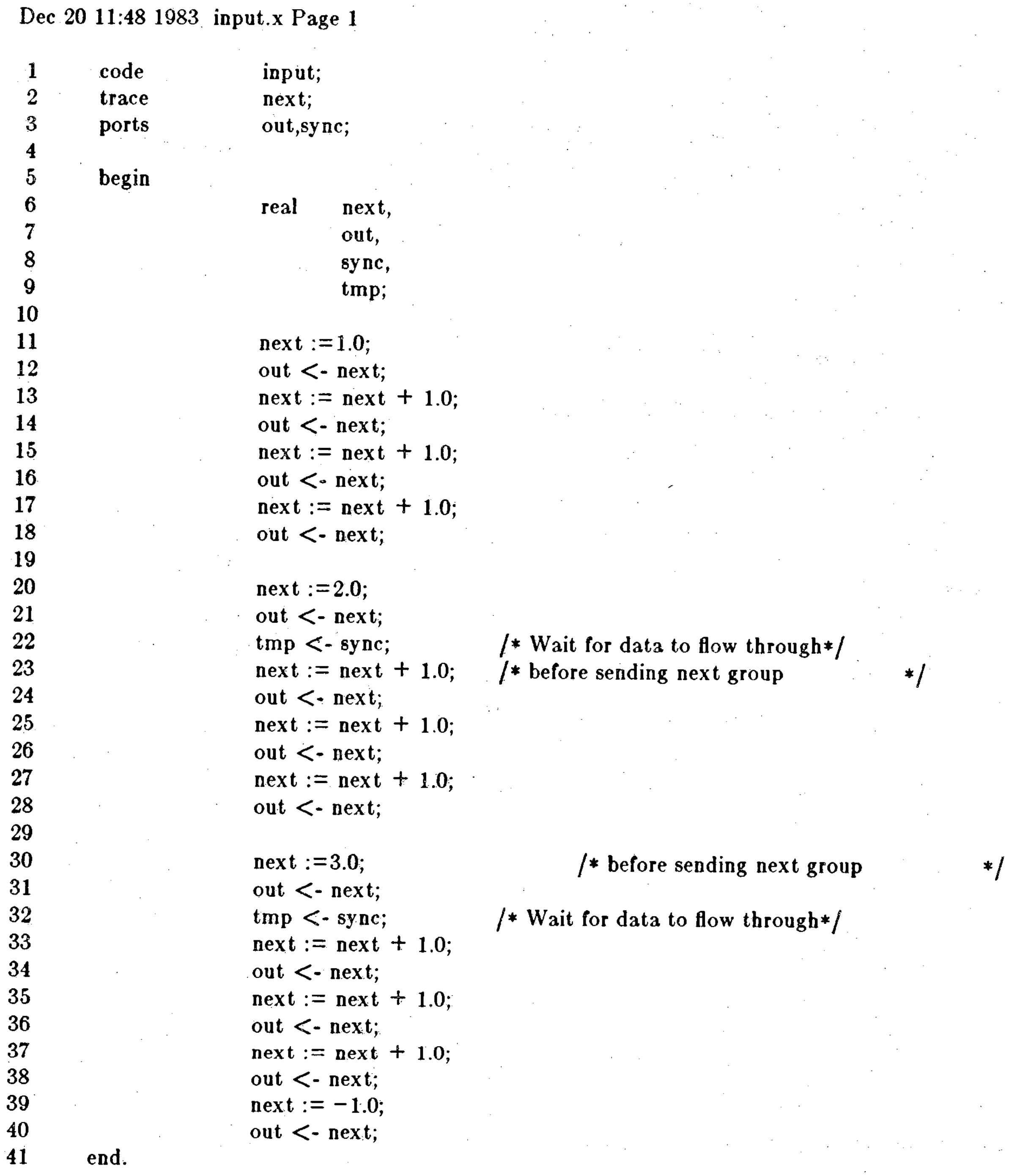
Dec 20 11:48 1983 ltw.x Page 3

```
                                T := T1 + T2 + 128.0;
                                out <- T;
                                end;
                        end;                    /* of while true loop    */
        end.
```

Dec 20 11:48 1983 output.x Page 1

```
        code            output;
        trace           out;
        ports           in;
        begin
                        int     out,
                                in;

                        while true do
                                begin
                                out <- in;
                                end
        end.
```

Figure 8.35 (Continued)



Dec 20 11:48 1983 input.x Page 1

```
code            input;
trace           next;
ports           out,sync;

begin
                real    next,
                        out,
                        sync,
                        tmp;

                next :=1.0;
                out <- next;
                next := next + 1.0;
                out <- next;
                next := next + 1.0;
                out <- next;
                next := next + 1.0;
                out <- next;

                next :=2.0;
                out <- next;
                tmp <- sync;            /* Wait for data to flow through*/
                next := next + 1.0;     /* before sending next group          */
                out <- next;
                next := next + 1.0;
                out <- next;
                next := next + 1.0;
                out <- next;

                next :=3.0;                     /* before sending next group          */
                out <- next;
                tmp <- sync;            /* Wait for data to flow through*/
                next := next + 1.0;
                out <- next;
                next := next + 1.0;
                out <- next;
                next := next + 1.0;
                out <- next;
                next := −1.0;
                out <- next;
end.
```

Figure 8.35 (Continued)

1) Coefficient one of frame one enters cell (1,1). Cell (1,1) passes it to cell (1,2) which passes it to cell (1,3), and finally to cell (1,4).

2) Coefficient two of frame one enters cell (1,1) which passes it to cell (1,2) and then to cell (1,3). Cell (1,3) does not pass it to cell (1,4).

3) Coefficient three enters cell (1,1), which passes it to cell (1,2) where it stops.

4) Coefficient four enters cell (1,1) and stays there.

Now each cell has one coefficient from the first frame. The above process repeats for every frame in the utterance. Once each cell has one coefficient from each frame, the cell starts computing the new frames. After each cell computes a new coefficient it writes to the output cell below it (See Figure 8.33). If p=8, 8 *ltw* cells must be used, and the computation time will not increase. However, the time needed to pipe the 8 coefficients to all the cells will double.

Figure 8.36 shows the execution times for a sample run which uses three frames of four coefficients each for input and produces two frames of output. The total time needed to warp the three frames to two is 11,417 $\mu$s. Of the 11,417 $\mu$s, 6,579 $\mu$s are spent reading in the coefficients and passing them on to other cells. 4,286 $\mu$s are used to compute, and scale each coefficient, while 552 $\mu$s are for outputting the new frames. One way to gauge the performance of the l1 program is to view it in a speech recognition system. In a typical system, the coefficients will arrive one frame at a time about once every 10 to 20 ms. With this slow input rate, most of the time the program uses to input data is spent waiting for the next frame to arrive. The important time is the time after the end of the utterance and before producing new warped frames. This time shows how quickly the program can warp the utterance after all the data has arrived.

Consider a system that produces one frame of 8 coefficients once every 10 ms. The typical word length is 40 frames, so the LTW program must output 40 frames after all the data is input. Program l1 uses 8 *ltw* cells, one cell for each coefficient. The time to compute and output one frame is the time used by lines 56 to 64 of Figure 8.35. This is 2,303 $\mu$s. Since l1 must produce 40 frames, the total time is 92,120 $\mu$s. Therefore the computation time depends on the number of frames outputted. In a typical speech system there is 300 to 500 ms between the beginnings of adjacent utterances. The LTW programs

```
Count   Min Ave Max
                         /*
                                 This routine does a linear time
                                 warp. The data enters in from
                                 the left as p coefficients per
                                 frame. The first cell takes
                                 the last coefficient and keeps
                                 it and passes the p-1 preceding
                                 coefficients on the the cells
                                 to the right. Each of the
                                 other cells do the same thing
                                 until the right most cell only
                                 inputs one coefficient. In the
                                 end, cell 1 will have lpc
                                 coefficient k for all frames,
                                 and cell k will have lpc coef.
                                 1. -1 is input to show the end
                                 of data and time to start
                                 computing. Each cells computes
                                 the new warped output using
                                 only its lpc coef. The new
                                 warped utterance is output one
                                 coefficient at a time at it is
                                 computed. The new coefficients
                                 are scaled and converted to 8
                                 bit unsigned values.
                         */
                       code    ltw(number);
                       trace   T,j,i,tmp;
                       ports   in,out,passon;
                       begin
1        0   0   0             bool inputting;
1        0   0   0             sint    k,
1        5   5   5                     j,J,      /* Number of frames in input utterance  */
1        5   5   5                     i,I;      /* Number of frames in output utterance */
                               int     number;/* Number of coefs. given cell will get */
                                                 /* leftmost cell should have number=p,  */
                                                 /* the next will have p-1, until the    */
                                                 /* rightmost will have number=1         */

1        0   0   0             real    factor, /* ratio of J/I         */
1        0   0   0                     R[80],  /* Input utterance      */
1        0   0   0                     s,      /* scale        */
1        0   0   0                     onems,  /* 1-s                  */
1        5   5   5                     tmp,
1        0   0   0                     T1,T2,  /* Patch Job            */
1        5   5   5                     T,      /* Ouput utterance      */
1        0   0   0                     in,passon;
```

Figure 8.36. Execution times in $\mu$s for multi-cell LTW. Three input frames of four coefficients each, two output frames. 2,303 $\mu$s per output frame.

```
1     0    0    0              int      out;

1     5    5    5              I := 2;

1     0    0    0              while true do
                               begin
2     5    5    5              j := 0;
2     5    5    5              inputting := true;

2     9    9    9              while inputting do
                                       begin
4     274  274  274                    tmp <- in;       /* get first input for yourself */
                                          /* if -1, it's the end of
                                          the of the utterance */
4     314  314  314                    if tmp > 0 then
                                               begin
                           /*  Send the rest to the other cells */
3     66   66   66                             for k := 2 to number do
                                                       begin
9     91   91   91                                     passon <- tmp;
9     274  274  274                                    tmp <- in;
9     4    8    10                                     end;
3     14   14   14                             j := j + 1;
3     73   73   73                             R[j] := tmp;
                                               end
3     2    2    2                  else begin
1     5    5    5                          inputting := false;
1     66   66   66                         for k := 2 to number do
3     95   99   101                                passon <- tmp;
1     0    0    0                          end;
4     11   11   11                 end;    /* of inputting loop      */

1     8    8    8              J := j;
1     210  210  210            factor := (J-1)/(I-1);
1     20   20   20             for i := 1 to I do
                                       begin
2     425  425  425                    tmp := (i-1) * factor + 1.0;
2     157  157  157                    j := tmp;
2     442  442  442                    s := 64.0 * (tmp - j);
2     227  227  227                    onems := 64.0 - s;
2     196  196  196                    T1:= onems * R[j];
2     202  202  202                    T2:=    s  * R[j+1];
2     368  368  368                    T := T1 + T2 + 128.0;
2     276  276  276                    out <- T;
2     4    7    10                     end;
1     2    2    2              end;            /* of while true loop   */
                           end.
```

Figure 8.36 (Continued)

must execute in this amount of time to run in real time. The l1 program requires only 92 ms, therefore it can run in real time.

### 8.5.2. Serial LTW – l2

Since the LTW program needs a low throughput for real-time processing (i.e., 300 to 500 ms per utterance), this section considers a serial approach. Figure 8.37 is the listing for l2, the single-cell LTW program. l2 uses only one cell and executes a serial LTW program. Figure 8.38 shows the timings for each step. The execution times depend on both the number of coefficients and the number of output frames, but not the number of input frames. The total time needed to input three frames of four coefficients each is 18,016 μs. 6,920 μs are used to input the three frames, and 11,096 μs are used to compute and output two frames.

Viewing the l2 program in the same setting as the l1 program shows that no more cells are used when computing 8 coefficients than computing 4. Program l2 must repeat lines 62 to 65 of Figure 8.37 for each coefficient it computes. These lines take 1,042 μs to compute, making a total time of 9,597 μs to compute one frame of 8 coefficients. l2 uses 383,880 μs to compute 40 frames. Table 8.6 summarizes these results. In a typical system, the LTW program has 300 to 500 ms to perform its operations, therefore the l2 program may not be able to process in real time if many short utterances are spoken in a row. In such a case a buffer is needed to store the next frame while the current frame is being processed.

### 8.5.3. Summary

Two programs to perform linear time warping were presented. The first, l1, was based on the algorithm presented in Section 6.3.2 l1 achieves its parallelism by using one cell for each coefficient in a frame. By using $p$ cells (where $p$ is the number of coefficients per frame), l1 is able to warp an input utterance with an arbitrary number of frames to 40 frames in 92 ms. This time does not depend on the number of input frames nor the number of coefficients per frame, but it does depend on the number of output frames. I output frames

```
/*
Program Name:   ltw
Algorithm:      Section 6.3.2.
Machine:        VLSI processor array, simulated by Poker
Function:       This routine does a linear time warp
                using only one cell. 10000 is input to
                show the end of data and time to start
                computing. The new warped utterance is
                output one coefficient at a time at it
                is computed. All outputs are ints,
                multiplied by 64 with 128 added, so the
                fraction part will not be lost.
Precision:      Input:  32-bit floating point
                Output: 8-bit integers multiplies by 64 with 128 added.
Number of PEs:  1
Parameters:     p, the number of LPC coefficients.
                I, the number of frames computed.
Input:          p LPC coefficients are received at the "in" port.
                The value 10001 is received if a word is spotted and
                the ltw program should begin. The value 10000
                is received if the energy previously received
                should be discarded.
Typical Time:   384 ms for p=8 and I=40.
*/

code        ltw;
trace       j,i,T,tmp;
ports       in,out;
begin
            bool inputting;

            sint    j,J,      /* Number of frames in input utterance */
                    i,I;      /* Number of frames in output utterance        */

            real    factor,   /* ratio of J/I  */
                    R1[10],   /* input utterance.      */
                    R2[10],
                    R3[10],
                    R4[10],
                    R5[10],
                    R6[10],
                    R7[10],
                    R8[10],
                    s,                  /* scale          */
                    onems,    /* 1 minus s             */
                    tmp,
                    T1,T2,              /* Patch Job            */
                    T,                  /* Output utterance     */
```

Figure 8.37. Single-cell LTW program.

```
        in;
int     out;

I := 2;

while true do
begin
j := 0;
inputting := true;

while inputting do
        begin
        tmp <- in;

        if tmp = 10000.0 then   /* false alarm, empty buffer       */
                j := 0
        else if tmp = 10001.0 then
                inputting := false        /* end of word, start warping     */
        else
                begin           /* Get next frame                 */
                j := j + 1;
                R1[j] := tmp; tmp <- in;
                R2[j] := tmp; tmp <- in;
                R3[j] := tmp; tmp <- in;
                R4[j] := tmp;
                in <- tmp;                /* Send sync to endpoint          */
                end
        end;                    /* of inputting loop      */

J := j;
factor := (J-1)/(I-1);

for i := 1 to I do
        begin
        tmp := (i-1) * factor + 1.0;
        j := tmp;
        s := 64.0 * (tmp - j);  /* Scale by 128 so it can be       */
        onems := 64.0-s;        /* stored in an 8 bit value        */
        T1:= onems * R1[j];     /* also add 128 bias so it will be */
        T2:=    s  * R1[j+1];   /* always positive                 */
        T := T1 + T2 + 128.0;
        out <- T;
        T1:= onems * R2[j];
        T2:=    s  * R2[j+1];
        T := T1 + T2 + 128.0;
        out <- T;
        T1:= onems * R3[j];
        T2:=    s  * R3[j+1];
        T := T1 + T2 + 128.0;
        out <- T;
```

Figure 8.37 (Continued)

```
                        T1:= onemss * R4[j];
                        T2:=    s  * R4[j+1];
                        T := T1 + T2 + 128.0;
                        out <- T;
                        end;
                end;                    /* of while true loop     */
end.
```

Figure 8.37 (Continued)

```
Count   Min Ave Max
                        /*
                                This routine does a linear time warp
                                using only one cell. 10000 is input to
                                show the end of data and time to start
                                computing. The new warped utterance is
                                output one coefficient at a time at it
                                is computed. All outputs are ints,
                                multiplied by 64 with 128 added, so the
                                fraction part will not be lost.
                        */
                        code    ltw;
                        trace   j,i,T,tmp;
                        ports   in,out;
                        begin
1        0   0   0              bool inputting;
1        5   5   5              sint    j,J,      /* Number of frames in input utterance */
1        5   5   5                      i,I;      /* Number of frames in output utterance */

1        0   0   0              real    factor, /* ratio of J/I             */
1        0   0   0                      R1[10], /* input utterance.         */
1        0   0   0                      R2[10],
1        0   0   0                      R3[10],
1        0   0   0                      R4[10],
1        0   0   0                      R5[10],
1        0   0   0                      R6[10],
1        0   0   0                      R7[10],
1        0   0   0                      R8[10],
1        0   0   0                      s,        /* scale */
1        0   0   0                      onems,  /* 1 minus s    */
1        5   5   5                      tmp,
1        0   0   0                      T1,T2,  /* Patch Job    */
1        5   5   5                      T,        /* Output utterance       */
                                        in;
                                int     out;

1        5   5   5              I := 2;

1        0   0   0              while true do
                                begin
1        5   5   5              j := 0;
1        5   5   5              inputting := true;

1        9   9   9              while inputting do
                                        begin
4      262 363 398                      tmp <- in;
```

Figure 8.38. Execution times is $\mu$s for single cell LTW. Three input frames of four coefficients each, two output frames. 5,429 $\mu$s per output frame.

```
                                         /* false alarm, empty buffer      */
4      432  432  432                     if tmp = 10000.0 then
                                                 j := 0
                                         else if tmp = 10001.0 then
                                                 /* end of word, start warping   */
                                                 inputting := false
1      7    7    7                       else
                                                 begin    /* Get next frame        */
3      14   14   14                              j := j + 1;
3      335  335  335                             R1[j] := tmp; tmp <- in;
3      335  335  335                             R2[j] := tmp; tmp <- in;
3      335  335  335                             R3[j] := tmp; tmp <- in;
3      73   73   73                              R4[j] := tmp;
3      91   91   91                              in <- tmp;       /* Send sync to endpoint*/
3      0    0    0                               end
4      11   11   11                      end;    /* of inputting loop      */

1      8    8    8                  J := j;
1      210  210  210                factor := (J-1)/(I-1);
1      20   20   20                 for i := 1 to I do
                                         begin
2      425  425  425                     tmp := (i-1) * factor + 1.0;
2      157  157  157                     j := tmp;
                                         /* Scale by 128 so it can be stored */
                                         /* in an 8 bit value        */
                                         /* also add 128 bias so it will be  */
                                         /* always positive          */
2      442  442  442                     s := 64.0 * (tmp - j);
2      227  227  227                     onems := 64.0-s;
2      196  196  196                     T1:= onems * R1[j];
2      202  202  202                     T2:=    s  * R1[j+1];
2      368  368  368                     T := T1 + T2 + 128.0;
2      276  276  276                     out <- T;
2      196  196  196                     T1:= onems * R2[j];
2      202  202  202                     T2:=    s  * R2[j+1];
2      368  368  368                     T := T1 + T2 + 128.0;
2      276  276  276                     out <- T;
2      196  196  196                     T1:= onems * R3[j];
2      202  202  202                     T2:=    s  * R3[j+1];
2      368  368  368                     T := T1 + T2 + 128.0;
2      276  276  276                     out <- T;
2      196  196  196                     T1:= onems * R4[j];
2      202  202  202                     T2:=    s  * R4[j+1];
2      368  368  368                     T := T1 + T2 + 128.0;
2      276  276  276                     out <- T;
2      4    7    10                      end;
1      2    2    2                  end;     /* of while true loop   */
                               end.
```

Figure 8.38 (Continued)

Table 8.6 Execution times for LTW programs.

| program | l1 | | l2 | |
|---|---|---|---|---|
| Number of cells | 4 | 8 | 1 | 1 |
| Number of coefficients | 4 | 8 | 4 | 8 |
| Time for one frame | 2,303 μs | 2,303 μs | 5,429 μs | 9,597 μs |
| Time for 40 frames | 92,120 μs | 92,120 μs | 217,160 μs | 383,880 μs |

require I * 2,303 $\mu$s to compute. Data from the LPC program, which precedes the LTW program, arrives at the rate of one frame every 10 to 20 ms in a typical system. 10 to 20 ms to input each of the 40 frames is a long time compared to the 92 ms needed to compute and output 40 frames. Therefore the time used for inputting frames is not included in the total time since it is dependent on the *lpc* cell which is producing the input data.

Program l2 is a serial program using one cell. It requires 384 ms to perform the same task as above using 8 coefficients per frame. Each additional coefficient requires 1,042 $\mu$s to compute. When using 8 coefficients, each additional output frame requires 9,597 $\mu$s to compute.

The Poker system is able to implement both LTW algorithms in real time since it performs the LTW task once every 300 to 500 ms. A buffer may be needed to hold the inputs to the l2 program since it needs 384 ms for its computation. Since the computational requirements are lax, both algorithms are written in *xx* and run in real time.

## 8.6. Poker Simulation of Dynamic Time Warping

Dynamic time warping (DTW) is the process of taking one unknown utterance and comparing it to one known utterance. The result of the DTW operation is a single score telling how closely the two utterances match. A typical isolated word recognition system matches the unknown utterance to every known utterance in the system's vocabulary. A 1,000 word vocabulary would therefore require 1,000 DTWs to be performed.

An utterance is a collection of $I$ frames of $p$ coefficients each. $I$ is constant since the LTW program will stretch or shrink the utterance to a fixed length before the DTW program processes it. Typically $I=40$, $p=8$, and each coefficient is 16 bits.

The Poker system simulates the operations of the BAC using two different programs. The first, d1, is written in *xx*. The second, d2, is written in 8051 assembly language. As with the simulations of the previous algorithms, the *xx* program is too slow for real-time processing. The 8051 program, which, in addition to being written in assembly language, uses less precise data (8-bit coefficients and 16-bit distances), can run in real time. A typical speech recognition system uses 16-bit coefficients and 16-bit distances, so the execution times for such a system are extrapolated from the executions times of the simulated system. The following sections give the highlights of the two programs.

### 8.6.1. BAC written in *xx* – d1

Figures 8.39 and 8.40 show the switch settings, code names, and port names used to simulate a BAC with a warping path of r=2. This value was chosen because it requires a total of 2r+1 *even* and *odd* cells which conveniently fit into a four by four grid of cells. Increasing r to a typical value of 6 will not change the throughput; however, it will increase the time needed to initialize the array. Figure 8.41 gives the *xx* code for the instructions given in

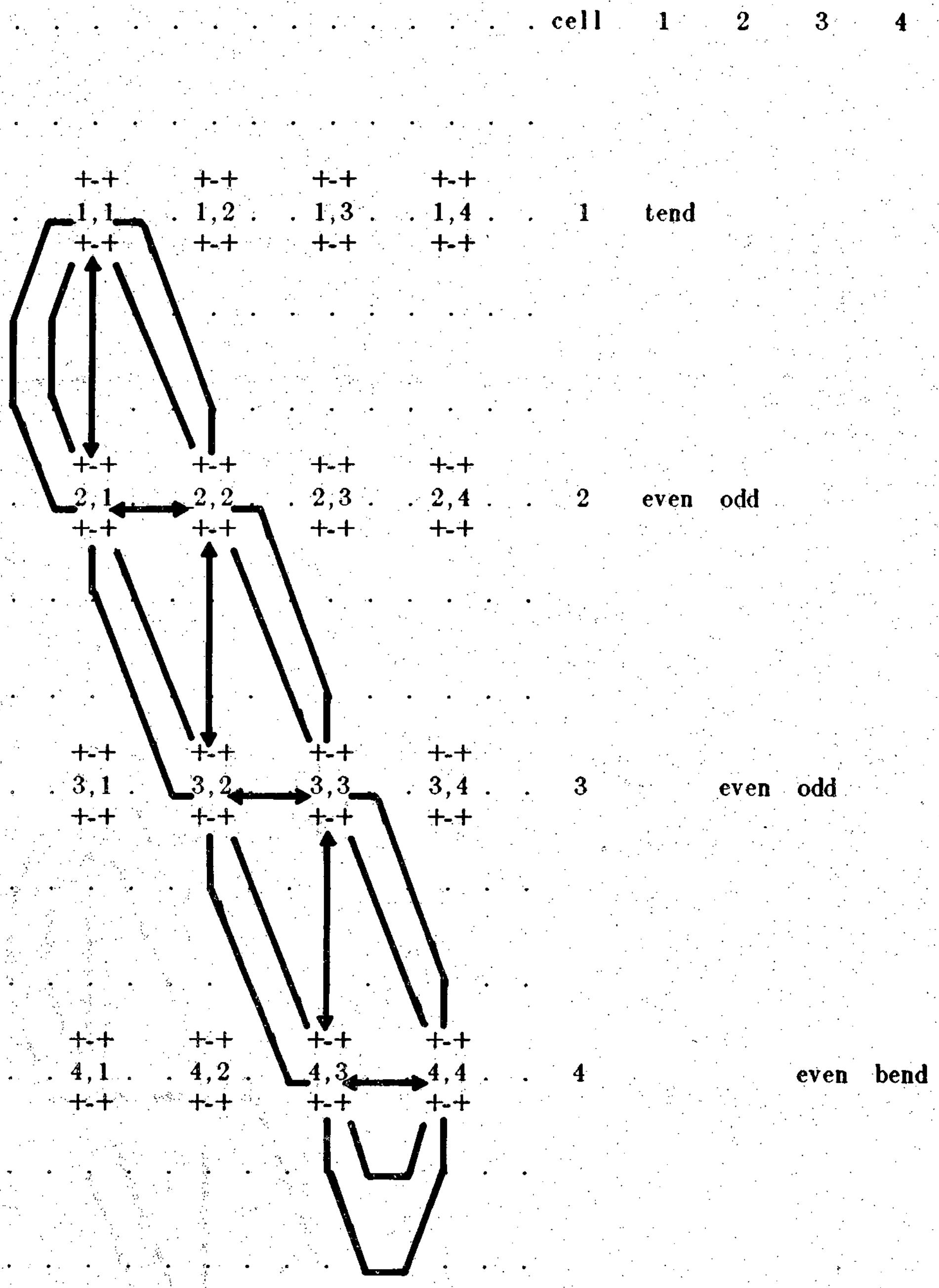


Figure 8.39. Switch settings for DTW program d1.

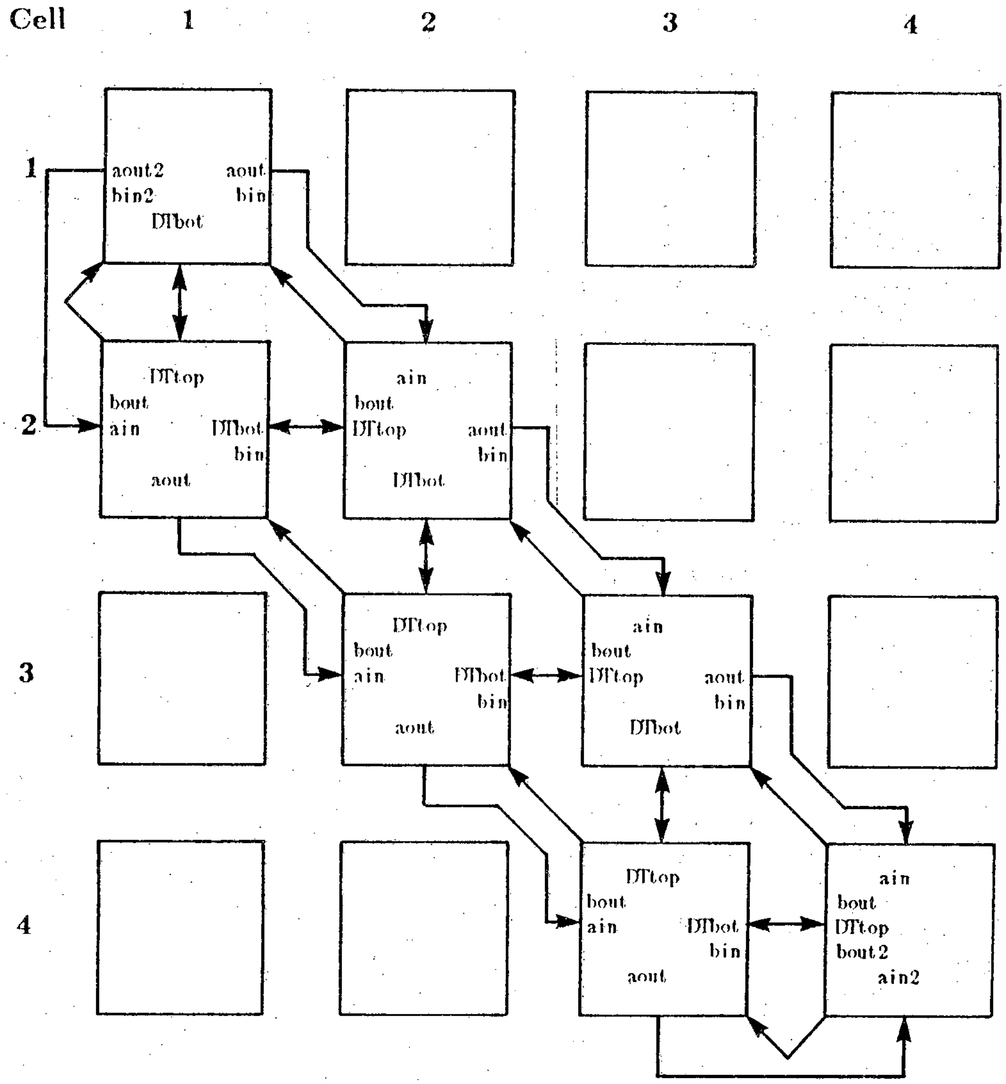
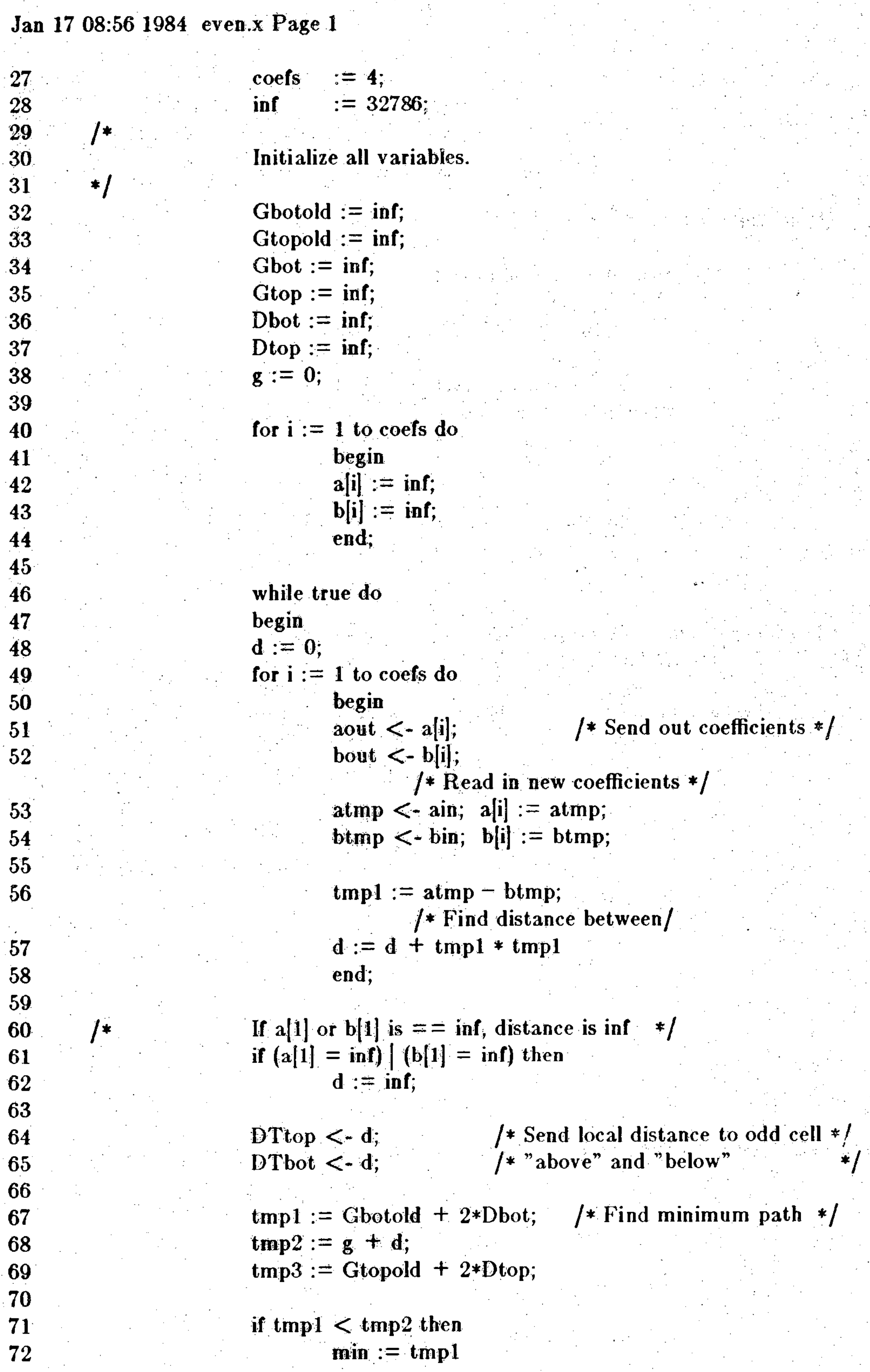


Figure 8.40. Port names for DTW program d1.

```
/*
              Program Name:   dtw (even)
              Algorithm:      Figure 6.23
              Machine:        VLSI processor array, simulated by Poker.
              Function:       This routine does a dynamic time warp.
                              This code is executed by the even cells.
              Precision:      Input:  32-bit floating point
                              d:      32-bit floating point
                              g:      32-bit floating point
              Number of PEs:  2r+1
              Parameters:     r, the width of the warping path.
                              p, the number of coeficients per frame.
                              I, the number of frames per utterance.
              Input:          tend generates the a vectors and sends
                              them up from the bottom of the array.
                              bend generates the b vectors and sends
                              them down from the top of the array.
              Output:         The d and g values are passed to the adjacent
                              odd cells. The a and b vectors are passed
                              to the adjacent even cells.
              Loop Time:      8,960 μs
              Typical Time:   36 ms for I=40, p=4, and r=2
*/
/*
              Patterned after assembly language program d2
*/
code          even;
trace         d,g,atmp,btmp;
ports         bout, bin, aout, ain, DTtop, DTbot;
begin
              sint   i,
                     coefs;   /* Number of coeficients in feature vector */
              int    bout, bin, aout, ain, DTtop, DTbot;
              int    a[10], atmp,
                     b[10], btmp,
                     d,       /* Distance between a and b vectors     */
                     Dbot,
                     Dtop,
                     g,       /* Local optimal distance               */
                     Gbot,
                     Gbotold,
                     Gtop,
                     Gtopold,
                     inf,     /* Infinity                             */
                     min,
                     tmp1,tmp2,tmp3;


```

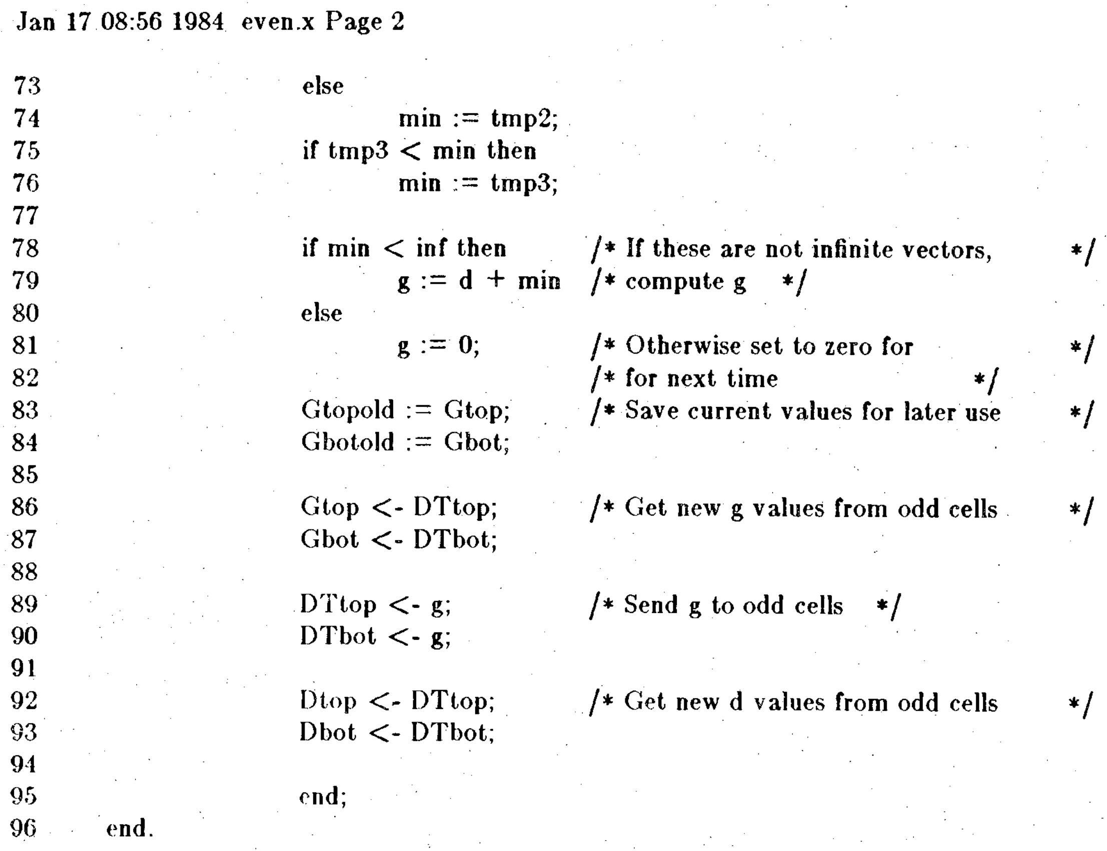
Figure 8.41. *xx* code for DTW program d1.



Jan 17 08:56 1984 even.x Page 1

```
                coefs    := 4;
                inf      := 32786;
    /*
                Initialize all variables.
    */
                Gbotold := inf;
                Gtopold := inf;
                Gbot := inf;
                Gtop := inf;
                Dbot := inf;
                Dtop := inf;
                g := 0;

                for i := 1 to coefs do
                        begin
                        a[i] := inf;
                        b[i] := inf;
                        end;

                while true do
                begin
                d := 0;
                for i := 1 to coefs do
                        begin
                        aout <- a[i];           /* Send out coefficients */
                        bout <- b[i];
                                /* Read in new coefficients */
                        atmp <- ain;  a[i] := atmp;
                        btmp <- bin;  b[i] := btmp;

                        tmp1 := atmp - btmp;
                                /* Find distance between/
                        d := d + tmp1 * tmp1
                        end;

    /*          If a[1] or b[1] is == inf, distance is inf   */
                if (a[1] = inf) | (b[1] = inf) then
                        d := inf;

                DTtop <- d;         /* Send local distance to odd cell */
                DTbot <- d;         /* "above" and "below"             */

                tmp1 := Gbotold + 2*Dbot;   /* Find minimum path  */
                tmp2 := g + d;
                tmp3 := Gtopold + 2*Dtop;

                if tmp1 < tmp2 then
                        min := tmp1
```

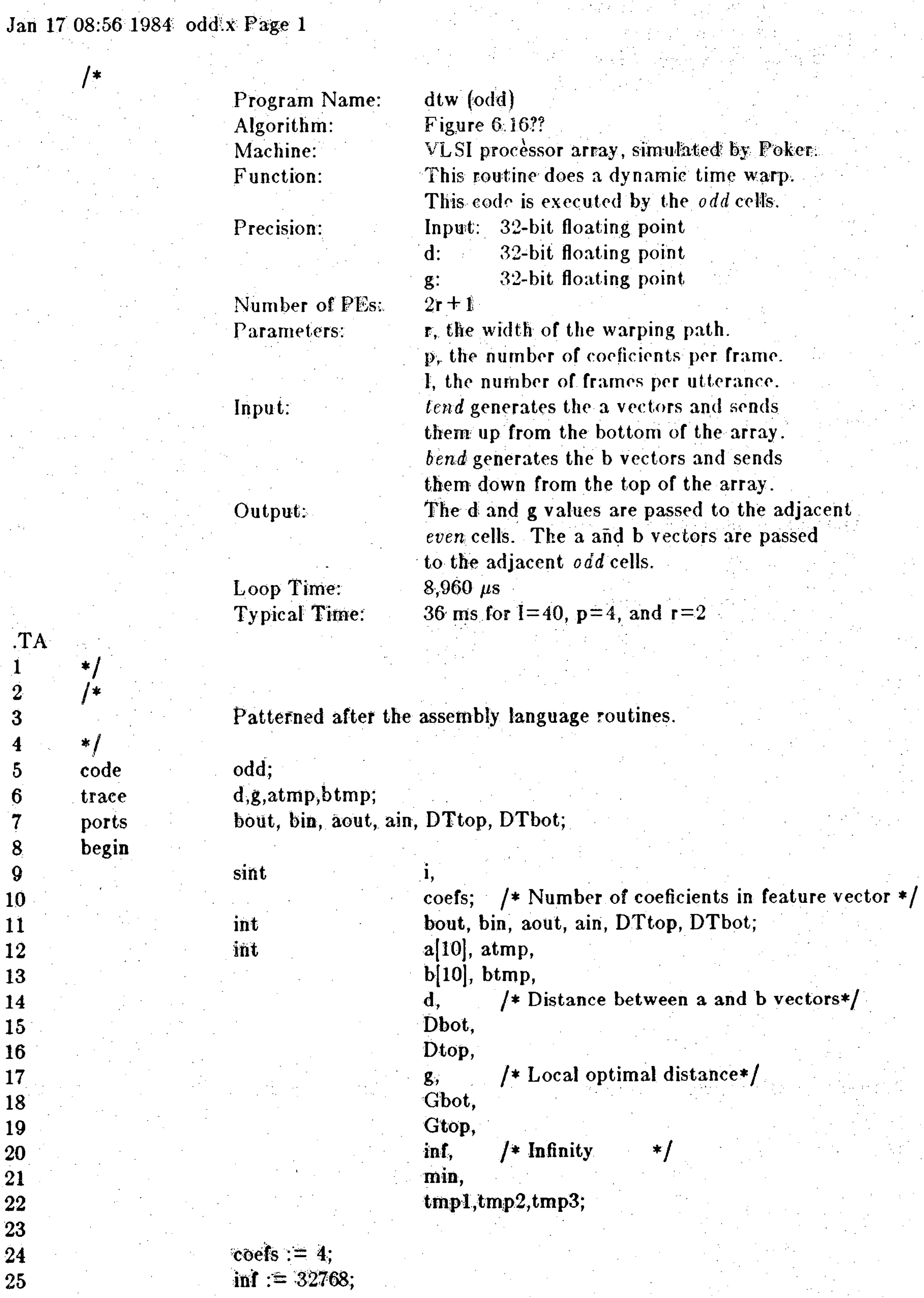
Figure 8.41 (Continued)



Jan 17 08:56 1984 even.x Page 2

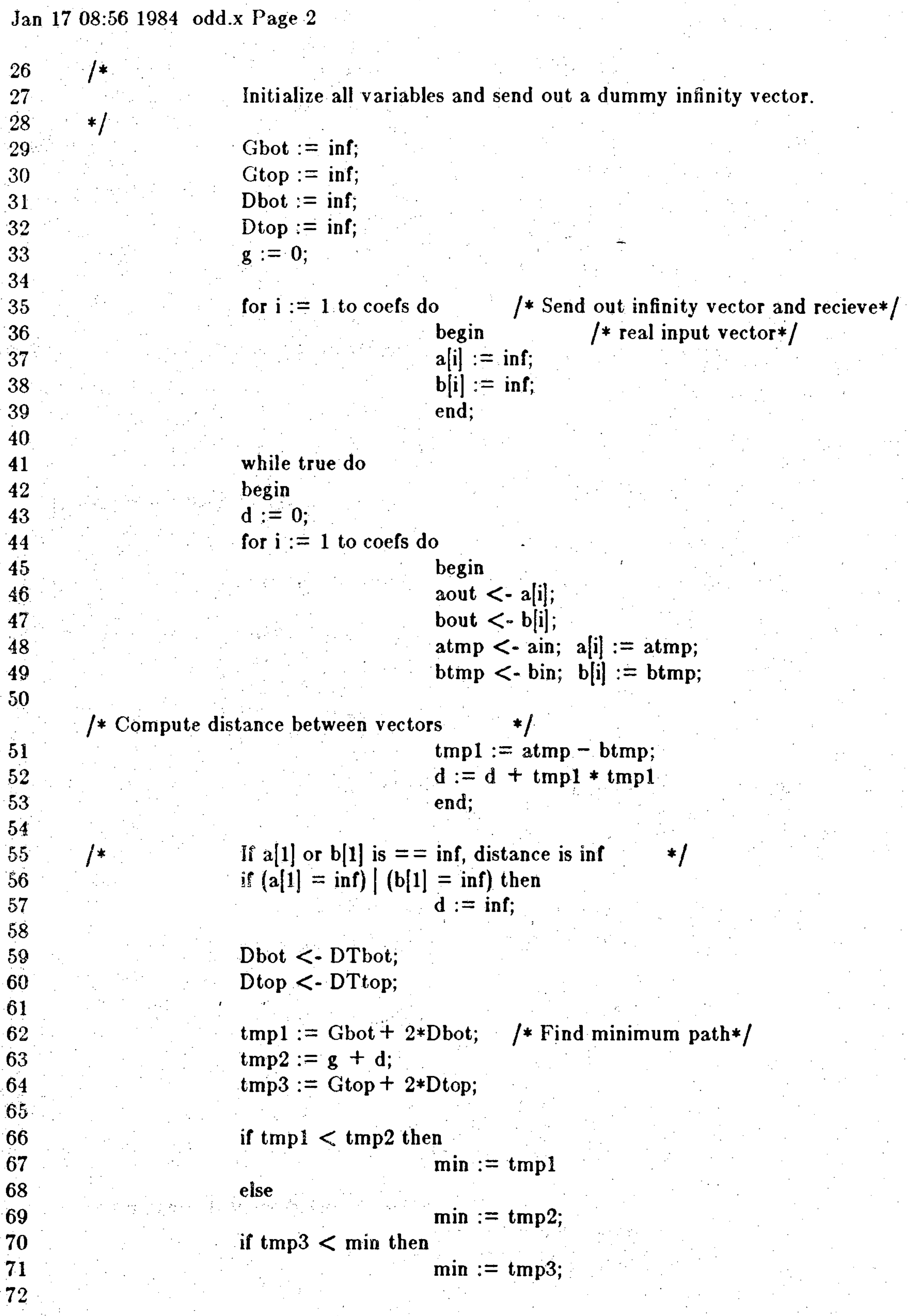
```
                else
                        min := tmp2;
                if tmp3 < min then
                        min := tmp3;

                if min < inf then       /* If these are not infinite vectors,     */
                        g := d + min    /* compute g    */
                else
                        g := 0;         /* Otherwise set to zero for              */
                                        /* for next time                      */
                Gtopold := Gtop;        /* Save current values for later use      */
                Gbotold := Gbot;

                Gtop <- DTtop;          /* Get new g values from odd cells        */
                Gbot <- DTbot;

                DTtop <- g;             /* Send g to odd cells    */
                DTbot <- g;

                Dtop <- DTtop;          /* Get new d values from odd cells        */
                Dbot <- DTbot;

                end;
        end.
```

Figure 8.41 (Continued)



```
Jan 17 08:56 1984  odd.x Page 1

        /*
                        Program Name:   dtw (odd)
                        Algorithm:      Figure 6.16??
                        Machine:        VLSI processor array, simulated by Poker.
                        Function:       This routine does a dynamic time warp.
                                        This code is executed by the odd cells.
                        Precision:      Input:  32-bit floating point
                                        d:      32-bit floating point
                                        g:      32-bit floating point
                        Number of PEs:  2r+1
                        Parameters:     r, the width of the warping path.
                                        p, the number of coeficients per frame.
                                        I, the number of frames per utterance.
                        Input:          tend generates the a vectors and sends
                                        them up from the bottom of the array.
                                        bend generates the b vectors and sends
                                        them down from the top of the array.
                        Output:         The d and g values are passed to the adjacent
                                        even cells.  The a and b vectors are passed
                                        to the adjacent odd cells.
                        Loop Time:      8,960 μs
                        Typical Time:   36 ms for I=40, p=4, and r=2
.TA
        */
        /*
                        Patterned after the assembly language routines.
        */
        code            odd;
        trace           d,g,atmp,btmp;
        ports           bout, bin, aout, ain, DTtop, DTbot;
        begin
                        sint            i,
                                        coefs;  /* Number of coeficients in feature vector */
                        int             bout, bin, aout, ain, DTtop, DTbot;
                        int             a[10], atmp,
                                        b[10], btmp,
                                        d,      /* Distance between a and b vectors*/
                                        Dbot,
                                        Dtop,
                                        g,      /* Local optimal distance*/
                                        Gbot,
                                        Gtop,
                                        inf,    /* Infinity       */
                                        min,
                                        tmp1,tmp2,tmp3;

                        coefs := 4;
                        inf := 32768;
```

Figure 8.41 (Continued)



Jan 17 08:56 1984  odd.x Page 2

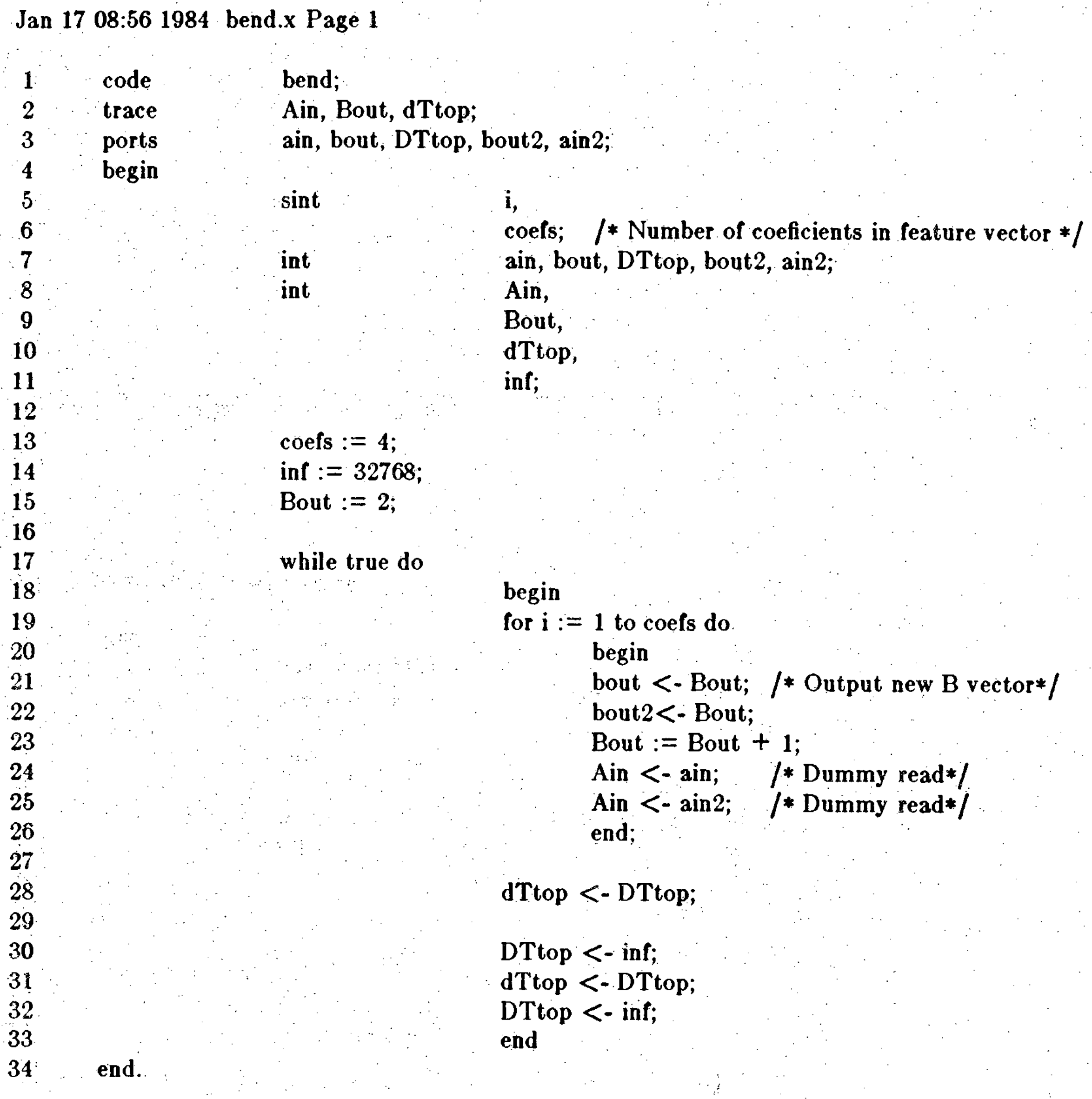
```
/*
                Initialize all variables and send out a dummy infinity vector.
*/
                Gbot := inf;
                Gtop := inf;
                Dbot := inf;
                Dtop := inf;
                g := 0;

                for i := 1 to coefs do          /* Send out infinity vector and recieve*/
                                begin               /* real input vector*/
                                a[i] := inf;
                                b[i] := inf;
                                end;

                while true do
                begin
                d := 0;
                for i := 1 to coefs do
                                begin
                                aout <- a[i];
                                bout <- b[i];
                                atmp <- ain;  a[i] := atmp;
                                btmp <- bin;  b[i] := btmp;

/* Compute distance between vectors         */
                                tmp1 := atmp - btmp;
                                d := d  + tmp1 * tmp1
                                end;

/*              If a[1] or b[1] is == inf, distance is inf          */
                if (a[1] = inf) | (b[1] = inf) then
                                d := inf;

                Dbot <- DTbot;
                Dtop <- DTtop;

                tmp1 := Gbot + 2*Dbot;     /* Find minimum path*/
                tmp2 := g  + d;
                tmp3 := Gtop + 2*Dtop;

                if tmp1 < tmp2 then
                                min := tmp1
                else
                                min := tmp2;
                if tmp3 < min then
                                min := tmp3;

```

Figure 8.41 (Continued)

Jan 17 08:56 1984 odd.x Page 3

```
                if min < inf then
                                g := d + min
                else
                                g := 0;

                DTbot <- g;
                DTtop <- g;

                Gbot <- DTbot;
                Gtop <- DTtop;

                DTbot <- d;
                DTtop <- d;

                end;
        end.
```

Figure 8.41 (Continued)



Jan 17 08:56 1984  bend.x Page 1

```
code        bend;
trace       Ain, Bout, dTtop;
ports       ain, bout, DTtop, bout2, ain2;
begin
            sint            i,
                            coefs;   /* Number of coeficients in feature vector */
            int             ain, bout, DTtop, bout2, ain2;
            int             Ain,
                            Bout,
                            dTtop,
                            inf;

            coefs := 4;
            inf := 32768;
            Bout := 2;

            while true do
                            begin
                            for i := 1 to coefs do
                                  begin
                                  bout <- Bout;  /* Output new B vector*/
                                  bout2<- Bout;
                                  Bout := Bout + 1;
                                  Ain <- ain;     /* Dummy read*/
                                  Ain <- ain2;    /* Dummy read*/
                                  end;

                            dTtop <- DTtop;

                            DTtop <- inf;
                            dTtop <- DTtop;
                            DTtop <- inf;
                            end
end.
```

Figure 8.41 (Continued)

```
Jan 17 08:56 1984  tend.x Page 1

code        tend;
trace       Aout, Bin, dTbot;
ports       bin, aout, DTbot, aout2, bin2;
begin
            sint            i,
                            coefs;   /* Number of coeficients in feature vector */
            int             bin, aout, DTbot, aout2, bin2;
            int             Aout,
                            Bin,
                            dTbot,
                            inf;

            coefs := 4;
            inf := 32768;
            Aout := 1;

            while true do
            begin
            for i := 1 to coefs do
                            begin
                            aout <- Aout;           /* Output new A vector*/
                            aout2<- Aout;
                            Aout := Aout + 2;
                            Bin  <- bin;            /* dummy read*/
                            Bin  <- bin2;           /* dummy read*/
                            end;

            dTbot <- DTbot;

            DTbot <- inf;
            dTbot <- DTbot;
            DTbot <- inf;
            end;
end.
```

Figure 8.41 (Continued)

Figure 6.23, where *even* is Group A, *odd* is Group B, and *tend* and *bend* are the *t*op and *b*ottom *ends*. *tend* and *bend* produce the input data for *even* and *odd*. Therefore a total of 2r+5 cells are used. 2r+1 cells are for the BAC, and the four extra cells are used to produce inputs. Changing the width of the warping path will change the number of cells used, but will not change the throughput of the BAC.

The instructions for the *even* numbered cells in Group A of Figure 6.23 map to the *xx* code as follows: Lines 5-24 of *even* in Figure 8.41 are variable declarations. Lines 27-44 are variable initializations. All the variables and the input frames* are initially set to infinity since it takes time for the frames to fill the bilinear array. During the filling process most cells contain invalid data. Figure 6.22 shows that during loop #4, only cells −1, 0, and 1 have two pairs of valid frames. During loop #4 these three cells compute valid distance scores, while the rest compute values that have no meaning. Initializing these "invalid" cells to infinity allows them to perform their computations and pass their distance scores (which will be infinity) to the cells making valid computations. Since $g$ is picked as the minimum path, the infinite distances from the invalid cells have no effect on the path taken.

Lines 48-57 in *even* in Figure 8.41 move the unknown frames down, the known frames up, and compute *d*. The distance measure used is a sum of squares of differences. It was chosen because its computation time falls between the time needed for a simple "absolute value of differences" and the "Itakura distance measure" [Itak75]. If the Itakura distance measure were used, it would increase the distance computation time because it requires the *log* of a value. Since the APU takes 1,783 $\mu$s for the log computation and the distance measure implemented uses 1,580 $\mu$s for p=8, the Itakura measure would take at least more than double the local distance computation time. Since the local distance measure is computed in a serial fashion, other distance measures can be used without the need for finding parallel algorithms to implement them.

*Section 6.4.2 calls the input data to the BAC *vectors*. I will use the term *frames* instead of vectors since the vectors $a$ and $b$ represent frames of speech data in a speech recognition system.

Lines 61 and 62 check to see if either frame is infinity, if so, $d$ is infinity. Lines 67-81 find the minimum of the three paths. If the minimum is infinity, $g$ is set to zero. This condition occurs after processing one pair of utterances and before the arrival of the next pair. Lines 83-93 send off the $g$ and $d$ values to the adjacent odd numbered cells. The process starts over at line 48.

Because of the internal workings of the *xx* compiler, lines 89,90 and 92,93 are switched from Figure 6.23. If cell A writes two values to the same port of cell B before cell B reads one, the first value is lost. Alternating reads and writes to the same cell insures that no data is lost. A later version of *xx* solves this problem.

The translation from Group B of Figure 6.23 to *odd* of Figure 8.41 follows the same pattern.

Cells (1,1) and (4,4) run *xx* code *tend* and *bend*. *tend* provides the unknown input frame while *bend* produces the known frame. Since all the *even* cells are identical, cell (2,1) will read and write values to the cell "above" it just as cell (3,2) does, but there is no *even* cell above it. The *teven* cell absorbs the $d$ and $g$ values sent to it by cell (2,1) and produces infinity $d$ and $g$ values to send to cell (2,1). These infinity values signal cell (2,1) that there is no valid warping path from the cell above it. The *todd*, *bodd*, and *beven* cells perform the same function for the cells they communicate with as the *teven* cell does for cell (2,1).

Figure 8.42 gives the execution times in $\mu$s for each step of cell (2,1) using four coefficients per frame. The maximum total time needed for one loop of the d1 program, not including variable declarations or initializations, is 8,960 $\mu$s. Table 8.7 shows the percentage of time each part of the DTW program uses when computing four coefficients per frame. A real speech system would use an order of 8 coefficients per frame, which doubles the time to move $a$ and $b$ and find $d$. The total time for an 8 coefficient system is 2*3,891 $\mu$s + 2*1,611 $\mu$s + 1,674 $\mu$s + 1,784 $\mu$s = 14,462 $\mu$s per loop. A typical system will require 40 loops for one comparison, which is 40 * 14,462 = 578,480 $\mu$s. One comparison is performed for every word in the vocabulary, so a vocabulary of only two words can be matched in a little over one second. This is much too slow for real time recognition. As before, coding in assembly language can reduce the time of a loop.

```
Count   Min Ave Max
                       code   even;
                       trace  d,g,atmp,btmp;
                       ports  bout, bin, aout, ain, DTtop, DTbot;
                       begin
1       0   0   0             sint   i,
1       0   0   0                    coefs;  /* Number of coeficients in feature vector */
                              int    bout, bin, aout, ain, DTtop, DTbot;
1       5   5   5             int    a[10], atmp,
1       5   5   5                    b[10], btmp,
1       5   5   5                    d,      /* Distance between a and b vectors*/
1       0   0   0                    Dbot,
1       0   0   0                    Dtop,
1       5   5   5                    g,      /* Local optimal distance        */
1       0   0   0                    Gbot,
1       0   0   0                    Gbotold,
1       0   0   0                    Gtop,
1       0   0   0                    Gtopold,
1       0   0   0                    inf,    /* Infinity      */
1       0   0   0                    min,
1       0   0   0                    tmp1,tmp2,tmp3;


1       5   5   5             coefs  := 4;
1       52  52  52            inf    := 32786;
                       /*
                              Initialize all variables.
                       */
1       52  52  52            Gbotold := inf;
1       52  52  52            Gtopold := inf;
1       52  52  52            Gbot := inf;
1       52  52  52            Gtop := inf;
1       52  52  52            Dbot := inf;
1       52  52  52            Dtop := inf;
1       29  29  29            g := 0;

1       20  20  20            for i := 1 to coefs do
                                     begin
4       73  73  73                   a[i] := inf;
4       73  73  73                   b[i] := inf;
4       4   8.5 10                   end;

1       0   0   0             while true do
                              begin
5       29  29  29            d := 0;
5       20  20  20            for i := 1 to coefs do
                                     begin
                                     /* Send out coefficients */
```

Figure 8.42. Execution times in $\mu$s for d1 using four coefficients per frame.

```
20        164 164 164                   aout <- a[i];
20        164 164 164                   bout <- b[i];
                                        /* Read in new coefficients */
20        166 310 347                   atmp <- ain;  a[i] := atmp;
20        329 329 329                   btmp <- bin;  b[i] := btmp;
20        143 143 143                   tmp1 := atmp - btmp;
                                        /* Find distance between vectors */
                                        d := d + tmp1 * tmp1
20        248 252 254                   end;

                             /*     If a[1] or b[1] is == inf, distance is inf   */
5         304 304 304               if (a[1] = inf) | (b[1] = inf) then
2         52  52  52                    d := inf;

                                    /* Send local distance to odd cell */
5         91  91  91                DTtop <- d;
5         91  91  91                DTbot <- d;    /* "above" and "below" */
                                    /* Find minimum path */
5         229 229 229               tmp1 := Gbotold + 2*Dbot;
5         139 139 139               tmp2 := g + d;
5         229 229 229               tmp3 := Gtopold + 2*Dtop;

5         132 132 132               if tmp1 < tmp2 then
                                        min := tmp1
3         54  54  54                else
2         52  52  52                    min := tmp2;
5         132 132 132               if tmp3 < min then
                                        min := tmp3;
                                    /* If these are not infinite vectors,  */
5         132 132 132               if min < inf then
                                        g := d + min    /* compute g    */
3         141 141 141               else
                                    /* Otherwise set to zero for next time  */
2         29  29  29                    g := 0;

                                    /* Save current values for later use     */
5         52  52  52                Gtopold := Gtop;
5         52  52  52                Gbotold := Gbot;
                                    /* Get new g values from odd cells       */
5         238 238 238               Gtop <- DTtop;
5         342 474 570               Gbot <- DTbot;
                                    /* Send g to odd cells  */
5         91  91  91                DTtop <- g;
5         91  91  91                DTbot <- g;
                                    /* Get new d values from odd cells       */
5         318 321 322               Dtop <- DTtop;
5         459 464 471               Dbot <- DTbot;

5         2   2   2                 end;
                             end.
```

Figure 8.42 (Continued)

Table 8.7 Execution time summary for DTW program d1 using four coefficients per frame.

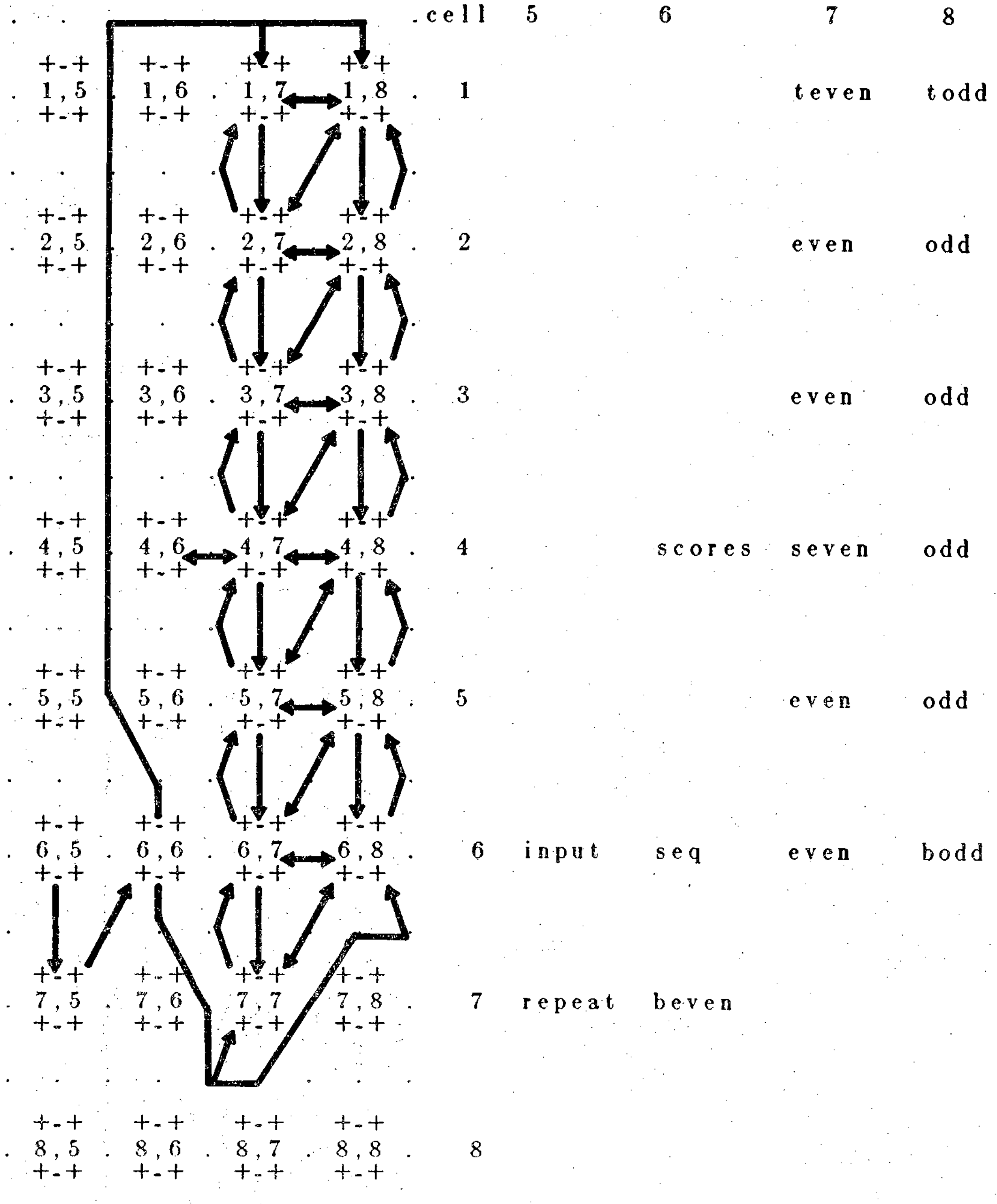
| Operation | Time | Percent of Total |
|---|---|---|
| Moving a and b | 3,891 μs | 43% |
| Finding d | 1,611 μs | 18% |
| Finding g | 1,560 - 1,674 μs | 19% |
| Moving d and g | 1,784 μs | 20% |
| Total Time | 8,960 | 100% |

### 8.6.2. 8051 Assembly Language Version of BAC – d2

Like the *xx* algorithm d1, the assembly language version of the BAC (d2) implements the algorithm in Figure 6.23. Figure 8.43 shows the switch setting for d2. d2 uses 13 cells since a typical speech recognition system uses a warping path width of r=6 and 2r+1 cells must be used. The cells are arranged in two vertical columns. The original "two rows on a diagonal" layout provides a good conceptual map from the task being performed to the program, but it makes poor use of cell space. The two vertical columns, however make better use of the space. d2 does not used the *tend* and *bend* cells, instead it uses the *beven (b*ottom *even), bodd (b*ottom *odd), teven (t*op *even),* and *todd (t*op *odd)* cells. The difference is the *tend* and *bend* cells do not compute distance values, while the top/bottom even/odd cells do. This makes better use of the computing power of each cell. d2 adds a new cell, called *seven,* (*s*cores *even*) to the middle of the array. This is an *even* cell that has an extra output port that outputs the distance scores. The *xx* program has no provisions for outputting scores. If it were to be used for a "real" speech system an output cell would be needed. Since d1 was used only to compute the loop time and not perform a complete DTW comparison, the output cell was not used.

Finally, four new cells are added, *input, repeat, seq,* and *scores. Input* reads unknown frames unknown frames from memory and writes them to its output port, *repeat* takes these frames and sends them out once for every utterance in the vocabulary. The known frames are stored in the *seq* cell. It outputs one known frame for every unknown frame coming from *repeat.* The known frame goes out the south port, while the unknown goes out the north.

Figure B.12 is a listing of all the assembly code for each cell. All the DTW cells execute their instructions in approximately lock-step fashion to minimize the overhead of inter-cell communication. The execution is approximately lock-step in that all the cells start executing their main loops at the same time and the instructions executed are timed so that the writes to the output ports are within a few $\mu$s of each other. Therefore the execution is not strictly lock-step, but it is not asynchronous either. When executing in this manner, all cells write to the switch at about the same time; however, operations between writes to the switch may not be precisely synchronized. After a fixed (and known) amount of time (12 $\mu$s), all cells can read from the switch



```
 .   .   .   .   .   .   .   .  .cell   5        6         7        8
   +-+   +-+   +-+   +-+
 . 1,5 . 1,6 . 1,7 . 1,8 .  1                            teven    todd
   +-+   +-+   +-+   +-+
 .   .   .   .   .   .   .   .
   +-+   +-+   +-+   +-+
 . 2,5 . 2,6 . 2,7 . 2,8 .  2                            even     odd
   +-+   +-+   +-+   +-+
 .   .   .   .   .   .   .   .
   +-+   +-+   +-+   +-+
 . 3,5 . 3,6 . 3,7 . 3,8 .  3                            even     odd
   +-+   +-+   +-+   +-+
 .   .   .   .   .   .   .   .
   +-+   +-+   +-+   +-+
 . 4,5 . 4,6 . 4,7 . 4,8 .  4                  scores    seven    odd
   +-+   +-+   +-+   +-+
 .   .   .   .   .   .   .   .
   +-+   +-+   +-+   +-+
 . 5,5 . 5,6 . 5,7 . 5,8 .  5                            even     odd
   +-+   +-+   +-+   +-+
 .   .   .   .   .   .   .   .
   +-+   +-+   +-+   +-+
 . 6,5 . 6,6 . 6,7 . 6,8 .  6    input         seq       even     bodd
   +-+   +-+   +-+   +-+
 .   .   .   .   .   .   .   .
   +-+   +-+   +-+   +-+
 . 7,5 . 7,6 . 7,7 . 7,8 .  7    repeat        beven
   +-+   +-+   +-+   +-+
 .   .   .   .   .   .   .   .
   +-+   +-+   +-+   +-+
 . 8,5 . 8,6 . 8,7 . 8,8 .  8
   +-+   +-+   +-+   +-+
 .   .   .   .   .   .   .   .
```

Figure 8.43 Switch settings for DTW program d2.

because data is guaranteed to be there. If the cells are not synchronized, it is possible for cell A to write to the switch, wait 12 $\mu$s, then read from the switch but get no data. This occurs when cell B is slightly behind cell A and has not written its data, intended for cell A, to the switch. The cells are run quasi-synchronously by using the built-in timer in each 8051 processor. Because of this, new feature frames must enter the DTW cells at a specific time. The *seq* cell is synchronized with the DTW cells so when *seq* has data to send, it sends it at the proper time. When there is no data to send, the *seq* cell sends infinity frames.

The last new cell is *scores*. DTW cell (4,7) sends each of its $g$ values to the *scores* cell. When the *scores* cell receives a zero value, the DTW cells are starting to compare a new pair of utterances. The value it receives before the zero value is the total score for the previous pair of utterances. The *scores* cell stores this value in an array.

### 8.6.3. Execution Times

Figure 8.44 shows the execution times for both d1 and d2 when using four coefficients per frame. The total time for one loop of d2 is 460 $\mu$s. Table 8.8 shows the percentage of time used by each part of both of the DTW programs.

The execution times for d2 assume there are four 8-bit unsigned coefficients per frame. A typical system would have 8 16-bit signed coefficients per frame. Table 8.9 is a summary of the expected execution times for a version of d2 that uses four and 8 16-bit coefficients per frame. The time used to compute $g$ and move $g$ and $d$ will remain the same when changing either the number of coefficients or the frame size. However, the time used to move the $a$ and $b$ vectors will double when either the number of coefficients or the frame size is doubled since twice as much data is being moved. Also, changing from 8 to 16 bits will increase the time to compute $d$ from 23.5 $\mu$s per coefficient to 74 $\mu$s because the 8-bit multiply-accumulate takes 9 $\mu$s while a 16-bit multiply-accumulate takes about 60 $\mu$s. The computation time for $d$ will be about 296 $\mu$s for four coefficients or 592 $\mu$s for 8 coefficients. The d2 program spends 47% of its time doing the computations (finding $d$ and $g$) while the rest of its time is spent moving data between cells. Of the 703 $\mu$s spent moving data ($a$,

```
d1      d2
xx      8051

                code    even;
                trace   d,g,atmp,btmp;
                ports   bout, bin, aout, ain, DTtop, DTbot;
                begin
0       0                sint    i,
0       0                        coefs;  /* Number of coeficeints in feature vector */
                         int     bout, bin, aout, ain, DTtop, DTbot;
5       0                int     a[10], atmp,
5       0                        b[10], btmp,
5       0                        d,      /* Distance between a and b vectors*/
0       0                        Dbot,
0       0                        Dtop,
5       0                        g,      /* Local optimal distance*/
0       0                        Gbot,
0       0                        Gbotold,
0       0                        Gtop,
0       0                        Gtopold,
0       0                        inf,    /* Infinity         */
0       0                        min,
0       0                        tmp1,tmp2,tmp3;


5       0                coefs   := 4;
52      6                inf             := 32786;
                /*
                         Initialize all variables.
                */
52      4                Gbotold := inf;
52      2                Gtopold := inf;
52      2                Gbot := inf;
52      2                Gtop := inf;
52      2                Dbot := inf;
52      2                Dtop := inf;
29      6                g := 0;

20      2                for i := 1 to coefs do
                                 begin
73      1                        a[i] := inf;
73      1                        b[i] := inf;
8.5     6                        end;

0       5                while true do
                         begin
29      4                d := 0;
20      3                for i := 1 to coefs do
```

Figure 8.44. Execution times in $\mu$s for d1 and d2.

```
d1      d2
xx      8051

                                begin
164     16                      aout <- a[i];     /* Send out coefficients */
164     14                      bout <- b[i];
310     4                       atmp <- ain;  a[i] := atmp;/* Read in new coefficients */
329     4                       btmp <- bin;  b[i] := btmp;
143     7                       tmp1 := atmp - btmp;
        10                      d := d + tmp1 * tmp1  /* Find distance between vectors */
252     2                       end;

                  /*            If a[1] or b[1] is == inf, distance is inf*/
304     9                 if (a[1] = inf) | (b[1] = inf) then
52      4                       d := inf;

91      30                DTtop <- d;    /* Send local distance to odd cell */
91      19                DTbot <- d;    /* "above" and "below"   */

229     12                tmp1 := Gbotold + 2*Dbot;   /* Find minimum path*/
139     6                 tmp2 := g + d;
229     12                tmp3 := Gtopold + 2*Dtop;

132     10                if tmp1 < tmp2 then
54      4                       min := tmp1
                          else
52      4                       min := tmp2;
132     8                 if tmp3 < min then
54      4                       min := tmp3;

132     6                 if min < inf then   /* If these are not infinite vectors,   */
141     6                       g := d + min   /* compute g*/
                          else
29      3                       g := 0;        /* Otherwise set to zero for next time  */

52      4                 Gtopold := Gtop;     /* Save current values for later use*/
52      4                 Gbotold := Gbot;

238     6                 Gtop <- DTtop;              /* Get new g values from odd cells*/
474     6                 Gbot <- DTbot;

91      30                DTtop <- g;          /* Send g to odd cells  */
91      19                DTbot <- g;

321     6                 Dtop <- DTtop;              /* Get new d values from odd cells*/
464     6                 Dbot <- DTbot;

2       11                end;
                  end.
```

Figure 8.44 (Continued)

Table 8.8 Execution time summary for DTW programs d1 and d2.

| Operation | Time | Percent of Total | Time | Percent of Total |
|---|---|---|---|---|
| Program | d1 | | d2 | |
| Coefs/Frame | 4 | | 4 | |
| Bits/Coef | 32 | | 8 | |
| Moving *a* and *b* | 3,891 $\mu$s | 43% | 152 $\mu$s | 33% |
| Finding *d* | 1,611 $\mu$s | 18% | 94 $\mu$s | 20% |
| Finding *g* | 1,560-1,674 $\mu$s | 19% | 76 $\mu$s | 17% |
| Moving *d* and *g* | 1,784 $\mu$s | 20% | 122 $\mu$s | 27% |
| Timer Control | 0 | 0% | $\geq$16 $\mu$s | 3% |
| Waiting for the Switch | | | 146 $\mu$s | 32% |
| Total Time | 8,960 $\mu$s | | 460 $\mu$s | |

Table 8.9 Execution time summary for DTW program d2 using 16 bits per coefficient.

| Operation | Time | Percent of Total | Time | Percent of Total |
|---|---|---|---|---|
| Coefs. per Frame | 4 | | 8 | |
| Moving $a$ and $b$ | 304 $\mu$s | 37% | 608 $\mu$s | 43% |
| Finding $d$ | 296 $\mu$s | 36% | 592 $\mu$s | 42% |
| Finding $g$ | 76 $\mu$s | 9% | 76 $\mu$s | 5% |
| Moving $d$ and $g$ | 122 $\mu$s | 15% | 122 $\mu$s | 9% |
| Timer Control | $\geq$16 $\mu$s | 2% | $\geq$16 $\mu$s | 1% |
| Waiting for the Switch | 226 $\mu$s | 28% | 386 $\mu$s | 27% |
| Total Time | 814 $\mu$s | | 1,414 $\mu$s | |

*b*, d, and g), 386 $\mu$s is spent waiting for the switch. Therefore 27% of the loop time is idle waiting for data to move through the switch.

At least 16 $\mu$s are spent starting and stopping the internal timer. The timer keeps all the DTW cells executing synchronously by doing the following:

1) At the start of the main loop all DTW cells *(even, odd, teven, todd, beven, teven, seven,* and, *seq)* start their timers at the same time.

2) All cells execute the instructions in their loop. Some cells may take longer than others.

3) At the end of the loop all cells wait for the timer to reach a certain predetermined value. Since all cells start at the same time, and all cells wait for the same timer value, all cells will start the next loop at the same time.

An alternative to using the timers is to pad all loops executed by the DTW cells with nops so they are the same length. This makes program development tedious since the programmer must change the code in every cell if the code in one cell is changed.

Although *even* can complete a loop in 1414 $\mu$s while processing 8 16-bit coefficients, *seven* requires an additional 30 $\mu$s to send data to the *scores* cell. Thus, the timer is set so one loop takes 1,445 $\mu$s. A typical speech system uses 40 frames per utterance, giving 40 * 1,445 $\mu$s = 56 ms to match one unknown utterance to one known utterance. Table 8.10 summarizes the execution times for d1 and d2. d2 can match a vocabulary of 17 words in one second using 8 16-bit coefficients per frame and 16-bit coefficients. Multiple BACs can be used in parallel to process a larger vocabulary in real time.

### 8.6.4. Summary

Two parallel programs to implement the BAC algorithm were presented. Program d1, written in *xx*, takes over 578 ms to perform one DTW match between two utterances of 40 frames each with 8 coefficients per frame. Program d2, written in assembly language, takes 57 ms to match the same two utterances. The following techniques were used to obtain this increase in speed.

1) Reducing the precision of the coefficients and distance scores.

2) Synchronizing all the DTW cells.

Table 8.10 Execution times for DTW progams.

| Program<br>Precision | d1<br>32-bit | | d2<br>16-bit | | d2<br>8-bit | |
|---|---|---|---|---|---|---|
| Number of<br>Coefficients | 4 | 8 | 4 | 8 | 4 | 8 |
| Total Time<br>for One Loop | 8,960 μs | 14,462 μs | 845 μs | 1,445 μs | 510 μs | 750 μs |
| Total Time<br>for 40 Loops | 358,400 μs | 578,480 μs | 33,800 μs | 57,760 μs | 20,400 μs | 30,000 μs |
| Word-comparisons<br>per Second | 2 | 1 | 29 | 17 | 49 | 33 |

Changing the precision of the coefficients from 32-bit integers to 8 or 16-bit integers, and the distance scores from 32-bit integers to 16-bit integers reduces the inter-cell communication time because less data is passed between cells. This also reduces the computation time since the 8051 can perform the operations instead of sending them to the APU. In a real speech recognition system, however, 8 bits are not enough for the coefficients [MaGr74]. Instead, a typical system uses 16 bits [WBA83]. Therefore a 16-bit version of d2 would have to be used.

Synchronizing the cells was the second technique used to speed up the program. The order of arrival of data to the input queue for each cell is difficult to determine since each cell normally executes independently of the other cells. The hardware provides a tag for each item in the queue. The tag indicates the port from which the item came. The task of checking this tag and saving the data, if it is not from the desired port, is time consuming. The assembly language BAC program never checks the tag. Instead it controls when the data enters the switch so that the data arrives in the order it is needed. Controlling the arrival of data from several cells running different programs is difficult, so all cells are synchronized by using the 8051's built-in timers. All cells enter their main loops at the same time. Each cell starts its own timer and will not restart the main loop until the given time has elapsed. Therefore when cell A, running *even* code, reads from its port, it knows that cell B, running *odd* code, has sent it some data.

Synchronization can be achieved without the use of timers. The programmer can carefully compute the execution time for the main loop in each cell and pad the cells which have the shortest execution times with nops so all cells have the same time. This makes the tedious task of assembly language programming even more tedious. The programmer must change the code in all cells if he changes the code in one cell. The 8051's build-in timers are a great help to the programmer.

The DTW algorithms have required more inter-cell communication than the previous algorithms. This results in spending 51% of the loop time moving data between cells. Over half (27% of the total time) of this time is spent waiting for the switch. Using an output queue instead of an output latch could eliminate this time and allow the algorithm to run faster. Also, having separate input queues for each port would eliminate the need to synchronize the cells, since there would be no confusion as to the arrival order of the input data.

Since most speech recognition systems use 16-bit coefficients, the processor must be able to perform 16-bit arithmetic. Although the 8051 is an 8-bit machine that can implement 16-bit arithmetic, a better solution would be to use a 16-bit processor. This would allow 16-bit coefficients to be processed without the overhead of implementing 16-bit arithmetic on an 8-bit machine. Likewise, a 16-bit wide data path between cells would reduce the inter-cell communication time.

## 8.7 VLSI Processor Array Isolated Word Recognition System

Previous sections have presented programs for performing various speech recognition tasks. The block diagram in Figure 4.1 shows a typical isolated word recognition system which uses some of these tasks. The parameters listed on it are for processing telephone quality speech. Table 8.11 lists parameters for telephone quality and high quality speech processing. The values listed under the Poker System (implemented) column are the values the system actually simulated. The values under the (possible) column are attainable by using the Poker system with minor changes in the programs.

This section shows how these programs are assembled together to perform the function of the speech recognition system shown in Figure 4.1. When combining VLSI processor array programs, the output data rate and format of one cell must match the input data rate and format of the cell to which it is attached. Figures 8.45, 8.46, and 8.47 show the switch settings, code names, and port names, respectively, for the entire system which uses 51 cells. In the shaded area on the left are all the cells used to compute the autocorrelation coefficients, and the cells in the shaded area on the right perform the DTW. The following sections discuss the new programs and the changes made to the programs from the previous sections so that the system could function.

### 8.7.1. Input Cell

The input cell (1,1) has data from a real speech signal which it sends to the filter cell. Figure 8.48 shows the plot of part of the /a/ sound from the word "all" as spoken by a male speaker. This data is digitized and formatted for input to the assembler and the listing is shown in Figure 8.49. The program in Figure B.13 (called *input*) outputs the first sample as a 16-bit value in two's complement notation. It sends the least significant byte (LSB) first; 16 $\mu$s later it sends the most significant byte (MSB). 160 $\mu$s after sending the LSB

Table 8.11 Parameters for speech recognition systems.

| | Telephone Quality | High Quality | Poker System (implemented) | (possible) |
|---|---|---|---|---|
| Sample Rate | 6.67 KHz | 20 KHz | 6.25 KHz | 6.25 KHz |
| Bits per Sample | 8 | 16 | 16 | 16 |
| LPC Coefficients | 8 | 16 | 4 | 8 |
| Bits per Coefficient | 16 | 16 | 8 | 16 |
| Range of Vocabulary Size (words) | 10-1,000 | 10-1,000 | 49* | 17* |

*The number of words that can be matched in one second using 13 DTW cells.

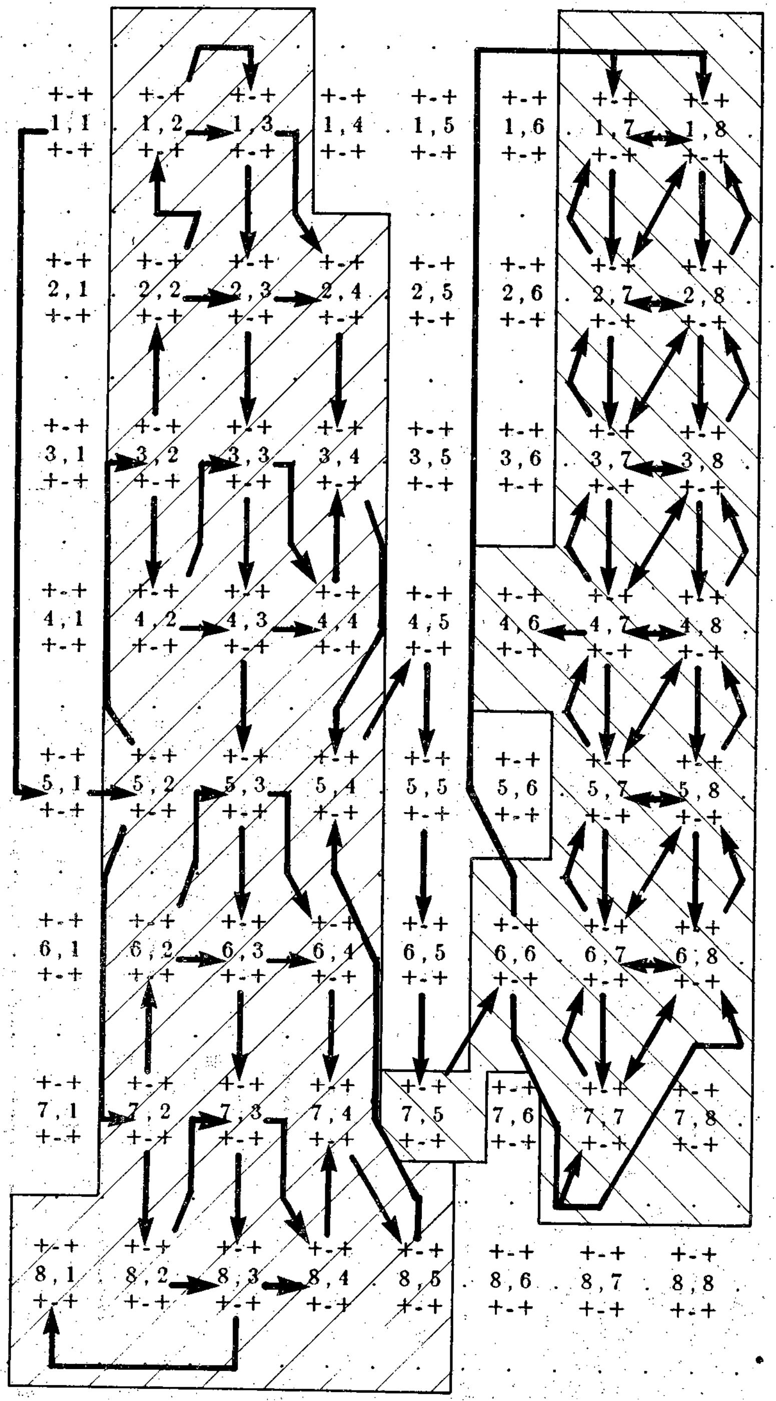
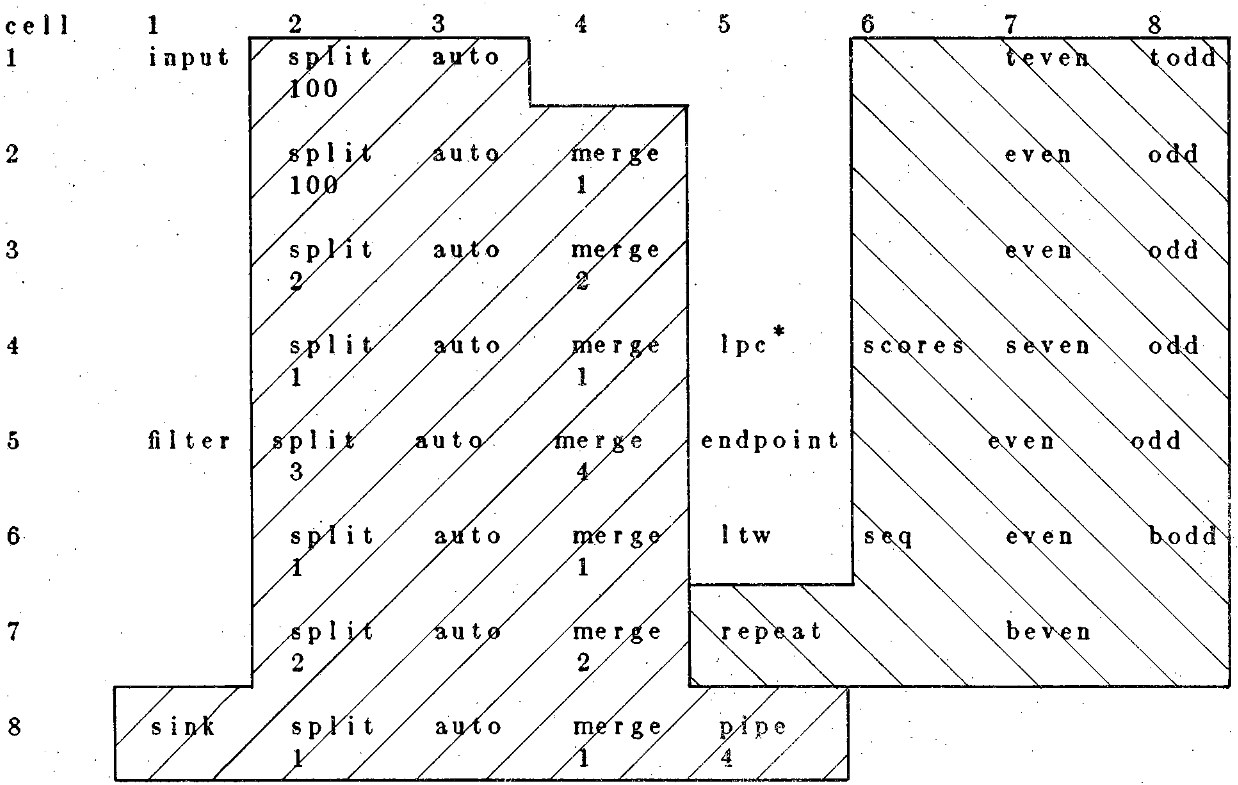


Figure 8.45 Switch settings for word recognition system.



| cell | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | input | split<br>100 | auto | | | | teven | todd |
| 2 | | split<br>100 | auto | merge<br>1 | | | even | odd |
| 3 | | split<br>2 | auto | merge<br>2 | | | even | odd |
| 4 | | split<br>1 | auto | merge<br>1 | lpc* | scores | seven | odd |
| 5 | filter | split<br>3 | auto | merge<br>4 | endpoint | | even | odd |
| 6 | | split<br>1 | auto | merge<br>1 | ltw | seq | even | bodd |
| 7 | | split<br>2 | auto | merge<br>2 | repeat | | beven | |
| 8 | sink | split<br>1 | auto | merge<br>1 | pipe<br>4 | | | |

Figure 8.46. Code names for the word recognition system.

*In a real system, the *lpc* cell will require six cells: one for the *demux* program, one for the *mux* program, and four for the *lpc* program.

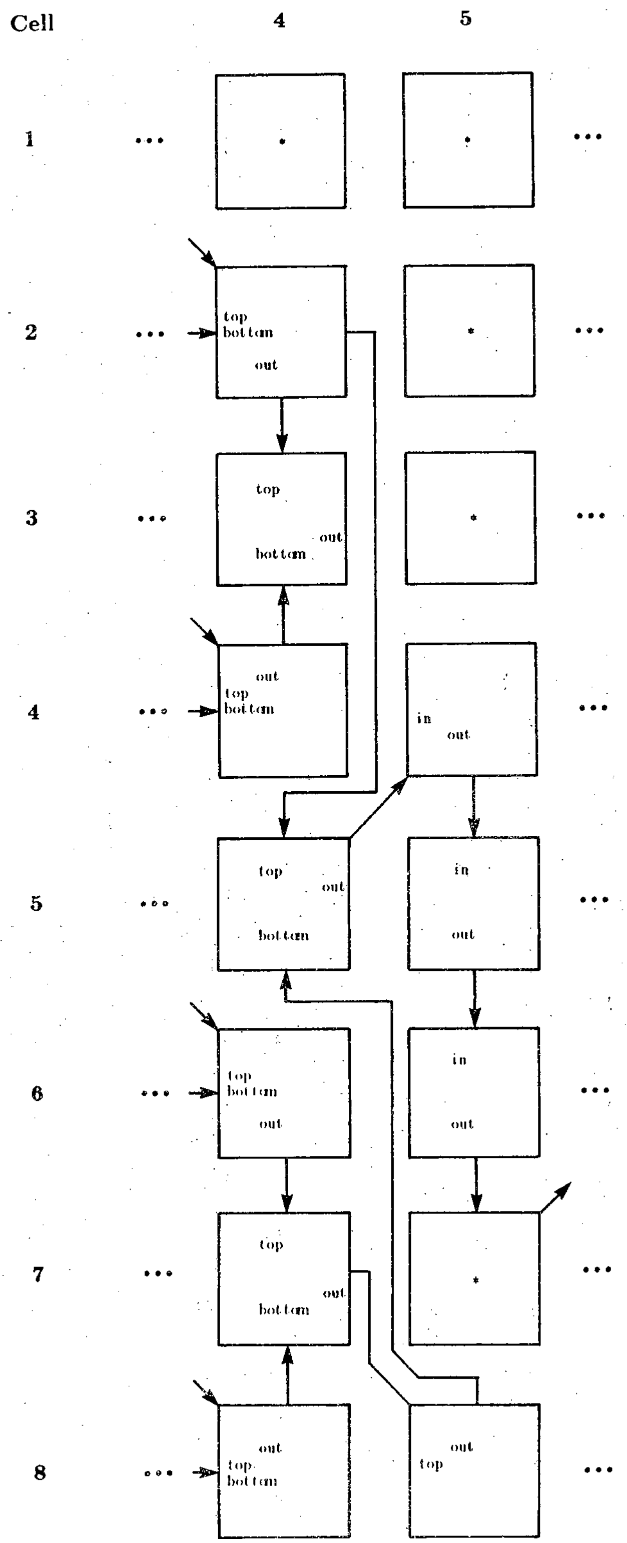


Figure 8.47. Port names for speech recognition system.
*Cells executing assembly language programs reference physical ports names and therefore need on logical port assignments.

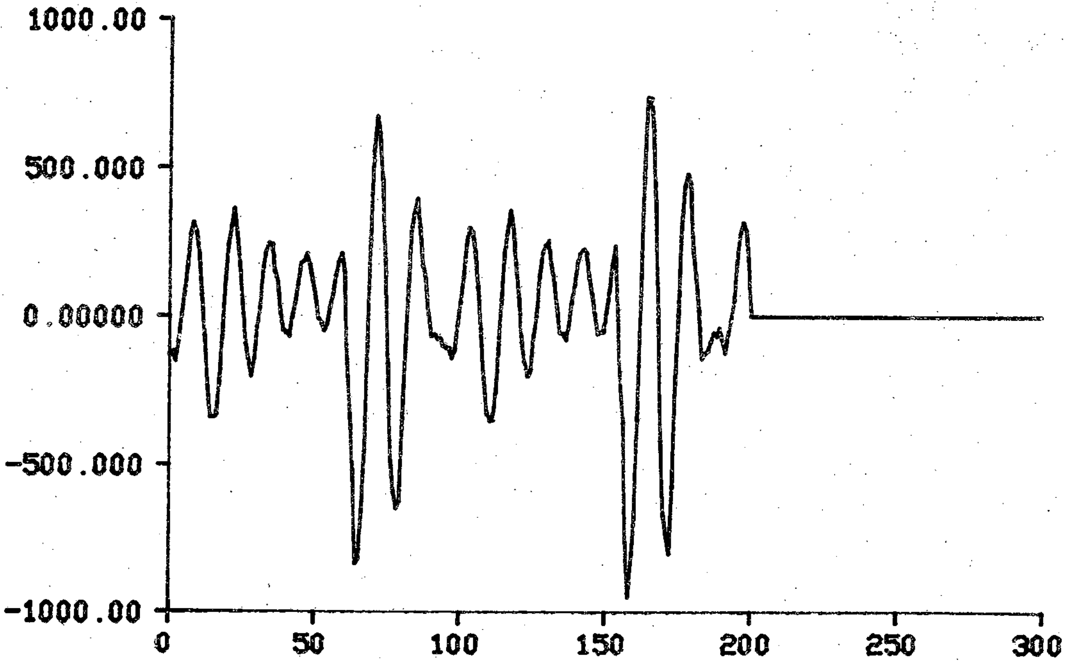


Figure 8.48. Plot of speech data output by the *input* cell.

```
;       This is a portion of the /a/ in the word "all"
;       It is sampled at 10KHz

all1:   dw   -130,  -120,  -151,  -91,   4,     75,    166,   277,   316,   283,
        dw   205,   74,    -66,   -236,  -341,  -340,  -331,  -222,  -73,   108,
        dw   241,   302,   363,   276,   173,   41,    -93,   -147,  -204,  -144,
        dw   -38,   28,    166,   218,   249,   243,   151,   116,   12,    -49,
        dw   -57,   -67,   -5,    43,    106,   180,   186,   209,   172,   121,
        dw   71,    -10,   -22,   -48,   -28,   29,    74,    140,   184,   211,
        dw   148,   -150,  -330,  -519,  -836,  -821,  -659,  -455,  -103,  254,
        dw   554,   670,   614,   458,   187,   -158,  -484,  -591,  -649,  -626,
        dw   -347,  -125,  85,    288,   347,   394,   300,   170,   129,   5,
        dw   -69,   -63,   -76,   -78,   -100,  -107,  -107,  -140,  -115,  -30,
        dw   44,    132,   243,   300,   285,   218,   98,    -42,   -209,  -329,
        dw   -351,  -352,  -260,  -113,  53,    213,   282,   356,   291,   178,
        dw   66,    -87,   -156,  -203,  -181,  -67,   -1,    132,   215,   240,
        dw   253,   174,   115,   30,    -58,   -61,   -77,   -32,   42,    101,
        dw   182,   213,   225,   215,   134,   76,    9,     -60,   -54,   -52,
        dw   10,    82,    170,   238,   21,    -234,  -353,  -761,  -951,  -809,
        dw   -682,  -347,  70,    467,   738,   736,   652,   389,   7,     -417,
        dw   -670,  -738,  -800,  -572,  -222,  5,     306,   438,   480,   448,
        dw   232,   130,   19,    -143,  -125,  -115,  -87,   -53,   -64,   -39,
        dw   -80,   -123,  -60,   -11,   56,    183,   280,   320,   293,   196,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
        dw   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
```

Figure 8.49. Speech input data for word recognition system.

of the first sample, it is sending the LSB of the next sample. The maximum data rate in limited by how fast th broadcast tree can send the data to each autocorrelation cell. The 160 $\mu$s is a sampling rate of 6.25 KHz which is too slow for telephone quality speech, but is the fastest the autocorrelation cells can receive data from the broadcast tree. See Section 8.3.6 for more details on the broadcast tree. The input cell uses the 8051's built-in timer to time the delay between samples.

### 8.7.2. Preemphasis Cell

Although Section 8.2 presented many filtering programs, none of them is used here. The transfer function of the preemphasis filter is $H(z)=1-.95z^{-1}$. It is simple enough for a single cell to perform. Although all the assembly language programs in Section 8.2 used unsigned data, the speech data coming from the input cell is signed data. The filter cell (5,2) uses signed data. The program is shown in Figure B.14. It takes 16-bit two's complement data as input and prcduces filtered 16-bit sign magnitude data as output. The 8051 uses two's complement notation for its addition and subtraction. The 8051 has an unsigned 8-bit by 8-bit multiply, but no signed multiply. There are fewer conversions needed to multiply two sign magnitude numbers than to multiply two two's complement numbers with an unsigned multiply. Therefore since the autocorrelation cells must use a multiplication, the filter cell converts its output to sign magnitude.

### 8.7.3. Autocorrelation Cells

The autocorrelation cells (1,3) - (8,3) run a program based on program *a5* in Figure B.11. The new autocorrelation program, *auto,* differs from *a5* in that *a5* uses unsigned data as input. The program used here takes 16-bit sign magnitude data as input and produces 32-bit two's complement data as output. The program is show in Figure B.15.

A typical speech recognition system uses 9 autocorrelation coefficients. *auto* computes 8 coefficients. This value is chosen since 8 cells fit into the 8 by 8 grid of cells used by Poker. A 9th cell could be added, but it would decrease the clarity of how the program functions because it could not be placed in the same vertical line with the other *auto* cells. Using 8 or 9 cells makes no difference in throughput.

### 8.7.4. The Split, Merge, and Pipe Cells

The *split* and *merge* cells run the same code as shown in Figure B.11. They are used to broadcast data to and collect results from the *auto* cells. The *split* cells form a broadcast tree which sends the input data to all *auto* cells. The *merge* cells collect the autocorrelation coefficients from the *auto* cells into one data stream for input into the *lpc* cell.

The system uses the *pipe* cell (8,5) (see Figure 8.50) so the input buffer on the *merge* cell (5,4) will not overflow when cells (3,4) and (7,4) send their data (16 bytes from each cell) to cell (5,4) at the same time. The *pipe* cell delays the data from cell (7,4) so that cell (5,4) has time to empty its buffer before more data arrives. (This is because of a bug in the *xx* compiler. It is fixed in a later version of the compiler.)

Another function of the *pipe* cell is to discard some of the coefficients the *auto* cell produces. Since this system uses only four LPC coefficients per frame, the LPC cell uses only five autocorrelation coefficients as input. The *pipe* cell discards three out of every four values it receives (it is a leaky pipe) so that the extra coefficients will not reach the LPC cell.

### 8.7.5. The LPC Cell

The *lpc* cell (4,5) runs the code shown in Figure 8.51. This is the same program as Figure 8.29 except that a line (line 27) is added to send the energy of the frame (R(0)) to the *endpoint* program. The *endpoint* program uses this value to detect the endpoints.

This program computes four LPC coefficients while a typical speech recognition system would compute 8. Section 8.4.3 showed that the LPC program can be implemented in real time by using a *demux, mux*, and four *lpc* cells. The single cell LPC program is used here since the system being simulated uses only four LPC coefficients per frame.

```
/*
                Program Name:   pipe
                Machine:        VLSI processor array, simulated by Poker
                Function:       This routine reads four values from the top port
                                and writes the frist value read to the out port.
                                The other three values are discarded.
                                Its main function is to delay the data entering the
                                middle merge cell.
                Precision:      Input:  32-bit integer
                                Output: 32-bit integer
                Number of PEs:  1
                Parameters:     interlace, the number of values to read from
                                top port before write frist value to
                                out port.
*/

code            pipe(interlace);
trace           tmp;
ports           top,out;
begin
                  sint  i;
                  int   tmp,
                          top,out,
                          interlace,
                          tophold[4];

                  while true do
                          begin
                          for i := 1 to interlace do
                                  begin
                                  tmp <- top;
                                  tophold[i] := tmp;
                                  end;
                          out <- tophold[1];
                          end;
end.
```

Figure 8.50. *xx* code for pipe cell.

```
/*
                  Program Name:   lpc
                  Algorithm:      Figure 4.2.
                  Machine:        VLSI processor array, simulated by Poker
                  Function:       Find LPC coefficients using Durbin's algorithm
                  Precision:      Input:  32-bit floating point
                                  Output: 32-bit floating point
                  Number of PEs:  1
                  Parameters:     p, the number of coefficients computed.
                  Input:          Autocorrelation coefficients arrive at
                                  "in" port
                  Output:         Energy (R[0]) is sent out "out"
                                  port followed by p LPC coefficients
                  Loop Time:      Does not apply
                  Typical Time:   42,130 μs for p=8
*/

code              lpc;
trace             k,E,i,j;
ports             in,out;
begin
                  sint    i,j,p;
                  int     itmp,in;
                  real    a[10],        /* LPC coefficients      */
                          aold[10],     /* old LPC coefficients  */
                          E,            /* Prediction error      */
                          k,
                          out,          /* output port           */
                          R[10],        /* autocorrelation coefs */
                          tmp;

                  p := 4;

                  while true do
                          begin
                          for i := 0 to p do    /* Read in autocorrelation coefs */
                                  begin         /* Starting with R(0)            */
                                  itmp <- in;
                                  k := itmp;
                                  R[i+1] := k;  /* All R[] indexs are +1 since   */
                                  end;          /* xx indexs start at 1          */

                          E := R[1];
                          out <- E;             /* Send R[1] to endpoint routine        */

                             for i := 1 to p do
                                     begin
                                     k := 0;
```

Figure 8.51. *xx* program for computing LPC coefficients from autocorrelation coefficients.

```
                for j := 1 to i-1 do
                    k := k + aold[j] * R[i-j+1];
                k := (R[i+1] - k) / E;
                tmp := k*k; E := (1 - tmp) * E;
                a[i] := k;
                for j := 1 to i-1 do
                    a[j] := aold[j] - k * aold[i-j];
                for j := 1 to i do
                    aold[j] := a[j];
                end;

            for i := 1 to p do      /* Send out lpc coefs starting with a1    */
              begin
                k := aold[i];
                out <- k;
              end;

        end
end.
```

Figure 8.51 (Continued)

### 8.7.6 Endpoint Detection

Cell (5,5) executes the endpoint code given in Figure 8.52. The program finds the endpoints based on the energy in each frame as discussed in Section 4.5. The *endpoint* program receives the energy of the current frame followed by $p$ LPC coefficients. If the energy is greater than the low threshold *lothresh* the $p$ LPC coefficients are sent to the LTW cell. If the energy is less than *lothresh* and some previous frame exceeded *hithresh,* the value *10001* is sent to the LTW cell. This signals the LTW program to start processing. If the energy does not exceed *hithresh* the program sends the value *10000* to the LTW cell to tell it to discard all the data received since the last *10001* value.

### 8.7.7. Linear Time Warping

Cell (6,5) executes the linear time warping program given in Figure 8.37. No changes are made to the program.

### 8.7.8. Dynamic Time Warping

The cells executing the DTW programs are identical to those discussed in Section 8.6. No changes are made to the program.

### 8.7.9. Summary

A number of the parallel speech processing programs were combined to form a speech recognition system. Since most speech data is signed, the *filter* and *auto* programs needed major changes so that they could process signed speech data. The other programs needed little or no modification to run on the system. Table 8.11 (Section 8.7.) summarizes the parameters of the system simulated on Poker. The system is unable to process telephone quality speech because its maximum sample rate is 6.25 KHz where 6.67 KHz is needed. Also, it uses only four LPC coefficients of 8 bits each when 8 coefficients of 16 bits each are needed. The conclusion section of this chapter discusses the changes that could be made so the VLSI processor array speech recognition system can process high quality speech in real time.

```
/*
                Program Name:   endpoint
                Algorithm:      Section 4.5 without zero crossing rate
                Machine:        VLSI processor array, simulated by Poker
                Function:       This routine does endpont detection by
                                looking at the value of R(0) out of the
                                autocorrelation routine via the lpc routine.
                                If it is big enough, the following p
                                coefficients are passed on to the ltw routine.
                Precision:      Input:  32-bit floating point
                                        Output:32-bit floating point
                Number of PEs:  1
                Parameters:     p, the number of LPC coefficients computed.
                Input:          Energy (R(0)) is arrives at "in"
                                port followed by p LPC coefficients
                Output:         p LPC coefficients are sent out the "out"
                                port if the energy is greater than "lothresh".
                                The value 10001 is sent if a word is spotted.
                                The value 10000 is sent if the energy drops
                                below "lothresh" before
                                going above hithresh.
*/
code            endpoint;
trace           tmp,energy;
ports           in,out;
begin
                bool    found;          /* == true if a word is spotted */
                sint    i,p;
                real    tmp,
                        energy,
                        lothresh,       /* low threshold        */
                        hithresh,       /* high threshold       */
                        in,out;

                p := 4;                 /* Number of coefficents per frame      */
                lothresh := 1000000.0;
                hithresh := 2000000.0;
                found    := false;

                while true do
                        begin
                        energy <- in;

                        if energy >= lothresh then
                                begin
                                if energy >= hithresh then
                                        found := true;
```

Figure 8.52. *xx* program for finding endpoints.

```
                for i := 1 to p do          /* Send frame to ltw     */
                    begin
                    tmp <- in;
                    out <- tmp;
                    end;
                end
            else
                begin
                if found then
                    begin   /* A word has been spotted       */
                    out <- 10001.0;
                    found := false;
                    end
                else
                    out <- 10000.0; /* No word,
                                    tell ltw to empty buffer */

                for i := 1 to p do          /* Dummy read word    */
                    tmp <- in;                  /* Don't send to ltw      */
                end;
        end;
end.
```

Figure 8.52 (Continued)

## 8.8 Conclusions

This chapter has presented several parallel programs for speech processing. The previous section showed how some of these programs could be combined into a parallel word recognition system. The goal was for this system to process high quality speech, as defined in Table 8.11, in real time. As Table 8.11 shows, the Poker system did not reach this goal for two of the parameters. It can process speech at a rate of 6.25 KHz, not at the rate of 20 KHz as desired and it uses 8-bit coefficients, not the 16-bit coefficients needed for high quality speech processing. The following sections discuss the VLSI processor array and give details as to which features it should have for it to process speech signals in real time.

### 8.8.1. The Processor

Poker emulated each cell as an Intel 8051 8-bit microprocessor. The following sections discuss the desirable properties of a VLSI processor array microprocessor.

#### *8.8.1.1. Data Size and Type – 16-bit signed fixed point*

Most speech data can be represented as a 16-bit signed integer, therefore the processor should operate on 16-bit data. The autocorrelation LPC and LTW routines used some 32-bit values, so 32-bit addition should also be implemented. The LPC and LTW routines used the Intel 8231 APU for floating-point operations, but they could have been implemented using only fixed point arithmetic. Adding floating-point operations would made writing some of the programs easier but it did not make the LPC or LTW programs execute faster. If the APU is to decrease the execution time, the microprocessor must be able to get data to it quickly and it must be able to perform its operations in less time than the 42 $\mu$s the 8231 needs for a floating-point multiply.

### *8.8.1.2. Internal Registers*

The 8051 has 128 bytes of internal RAM. The internal RAM has the same access time as the 8051's data registers for most instructions. This internal RAM can be used as 64 16-bit registers. Having many registers available is good since programs like the BAC can store all of its variables in the register-like memory and not have to use the external memory which is much slower to access.

### *8.8.1.3. Memory Size – 2K bytes*

Table 8.12 summarizes the memory requirements for each of the programs in the speech recognition system. The LTW program used the most memory with 1,280 bytes. The *input* cell does not include the storage needed for the input data. Most likely, the input data would come from an analog to digital converter and not memory. Also, the *seq* cell memory usage does not include the memory needed to store the known templates. A typical system would use 40 frames per utterance, 8 coefficients per frame, and 16-bits per coefficient. This is a total of 640 bytes per utterance. Therefore, if there are more than three words in the vocabulary, the *seq* cell would use more memory than any of the other cells. For a 100 word vocabulary the memory requirements would be 128K bytes if each word used 40 frames of 16 coefficients and 16 bits per coefficient.

Excluding the storage used by the *seq* cell to store known templates, each cell could operate using 2 K bytes of memory. The *seq* cell may have to be a special cell with extra memory to hold all the templates.

Table 8.12 Memory usage, in bytes, for SIMD based isolated word recognition system.

| | Language | Memory Usage |
|---|---|---|
| | | (bytes) |
| input | 8051 | 101 * |
| filter | 8051 | 151 |
| sink | 8051 | 12 |
| split | 8051 | 136 |
| auto | 8051 | 136 |
| merge | *xx* | 450 |
| lpc | *xx* | 848 |
| demux | *xx* | 343 |
| mux | *xx* | 286 |
| endpoint | *xx* | 385 |
| ltw | *xx* | 1280 |
| repeat | 8051 | 211 |
| pipe | 8051 | 142 |
| seq | 8051 | 177 † |
| even | 8051 | 436 |
| teven | 8051 | 445 |
| beven | 8051 | 445 |
| seven | 8051 | 445 |
| odd | 8051 | 429 |
| todd | 8051 | 430 |
| bodd | 8051 | 430 |
| scores | 8051 | 64 |
| Maximum | | 1,280 |

* Does not include storage for input data.

† Does not include storage for known templates.

### 8.8.2. Inter-PE Communications

The following sections discuss features the inter-cell communication should have.

#### *8.8.2.1 The Broadcast*

The VLSI processor array needs to implement a general broadcast that allows one port to broadcast to many ports with one write instruction. Using such a broadcast eliminates the need for the broadcast tree (the ***split*** cells) in Figure 8.45. With a broadcast, the data arrives at the input ports of the ***auto*** cells simultaneously, thus allowing the auto cells to process their data at a sampling rate over 10 KHz – a 60% increase in throughput. Before, the autocorrelation cell would run at 6.25 KHz so the data would have time to travel through the broadcast tree before the next sample arrived.

If a general broadcast is not possible, broadcasting from one port to two ports would be an improvement. This type of broadcast allows the data to propagate through the broadcast tree and arrive at the ***auto*** cells at the same time. Again the ***auto*** cells could process data at over 10 KHz.

The difference between a two port broadcast and the general broadcast is the two port broadcast would have a longer delay between the arrival of the last sample of the frame and the arrival of the autocorrelation coefficients at the *merge* cell. This is because it takes time for the data to travel through the broadcast tree.

#### *8.8.2.2 The I/O Buffer*

Using an output queue to replace the output latch which is between the 8051 and the switch would simplify programming the 8051 in assembly language and decrease the execution time. The *dtw* cells spend 32% of their total execution time waiting for the switch to read the output latch. Most of this wasted time would be eliminated by using a queue.

### 8.8.3. Number of Cells – 51

Table 8.13 summarizes the number of cells used by each program in the parallel word recognition system. By using 51 cells, the 8051 based VLSI processor array is able to process speech in real time, sampling at 6.25 KHz, using 4 8-bit coefficients per frame, and recognizing a 17 word vocabulary in 1 second. The demux/mux approach that was used by the LPC program could be used on the DTW program so that a 100 word vocabulary can be recognized in real-time if 5 copies of the DTW array* were used in parallel. Such a system would use a total of $51+5*15=126$ cells. The *scores* cell could be changed to collect the distance scores from all the DTW arrays.

The demux/mux approach could also be used to improve the throughput of the autocorrelation program, but a better approach would be to use a more powerful cell so the program could run faster.

### 8.8.4. Changing the Word Recognition System Parameters

The following sections discuss the effects of altering the system parameters on the processing throughput.

#### *8.8.4.1 Changing the LPC Frame Size*

Changing the LPC frame size will not change the number of cells used by the autocorrelation program or the throughput. Changing the frame size will only change how often the autocorrelation coefficients are output.

#### *8.8.4.2. Changing the Number of LPC Coefficients*

Increasing the number of LPC coefficients will not change the execution time of the autocorrelation program, however the autocorrelation program would have to use more cells since it uses one cell per coefficient. Increasing the number of LPC coefficients will increase the execution time of the *lpc* cell.

*The DTW array consists of the *repeat*, *seq*, and all the *even* and *odd* cells.

Table 8.13 Number of cells used by the VLSI processor array parallel speech recognition system.

| Function | Type of Cells | Number of Cells | |
|---|---|---|---|
| Input | | | 1 |
| | *input* | 1 | |
| Filter | | | 1 |
| | *filter* | 1 | |
| Autocorrelation | | | 25 |
| | *split* | 8 | |
| | *auto* | 8 | |
| | *sink* | 1 | |
| | *merge* | 7 | |
| LPC | | | 6 |
| | *demux* | 1 | |
| | *lpc* | 4 | |
| | *mux* | 1 | |
| Endpoint | | | 1 |
| | *endpoint* | 1 | |
| LTW | | | 1 |
| | *ltw* | 1 | |
| DTW | | | 16 |
| | *repeat* | 1 | |
| | *seq* | 1 | |
| | *even* | 7 | |
| | *odd* | 6 | |
| | *scores* | 1 | |
| Total | | | 51 |

More *lpc* cells may have to be added to process in real time, and the *demux* and *mux* cells will have to be changed to distribute the autocorrelation coefficients to more *lpc* cells.

#### *8.8.4.3. Changing the Number of Frames per Utterance*

The proposed system assumed that I=40 frames per utterance were output from the LTW and processed by the DTW program. As with the SIMD machine algorithms, the LTW and DTW execution times are proportional to I, so increasing I will increase the LTW and DTW processing times. Decreasing I, on the other hand, will shorten the LTW and DTW execution times.

If the LTW time is increased to greater than 500 ms, the *demux/mux* method used for the LPC program may have to be used to increase the throughput.

#### *8.8.4.4. Changing the Vocabulary Size*

As with the SIMD machine, the DTW program is the only program whose execution time depends on the vocabulary size. Increasing the vocabulary size will require the replication of the cells used for the DTW array and using the *demux/mux* scheme that was used for the LPC program. The *scores* cell could be changed to collect the distance scores from each DTW array and find the minimum score.

### **8.8.5. Summary**

The VLSI processor array, as simulated by Poker, is not able to process telephone quality speech in real time. This inability to process speech in real time is not caused by the VLSI processor array architecture, but by the system that simulated it. The Poker system uses an 8-bit microprocessor in each cell. As Chapter 7 showed, a 16-bit processor is more suited for speech processing since most intermediate speech data is 16 bits.

Poker's inter-cell communications are handled by the switch which can poll a cell only once every 12 $\mu$s. This slow inter-cell communication rate combined with a single input queue and an output latch required the 8051 to use up to 30% of its processing time waiting on the switch. If circuit switched inter-cell communication is used, the time the processor uses to service the I/O queues would be reduced.

If the VLSI processor array uses a 16-bit processor in each cell and has fast inter-cell communications, it should be able to recognize isolated words in real time.

# 9. CONNECTED WORD RECOGNITION

The purpose of this work is to improve the man/machine interface through the use of speech recognition. The idea is that communication between man and machine will improve if the machine can communicate using man's common method of communication (spoken words) rather than have man use the machine's method (terminal). The previous chapters have discussed using an isolated word recognition system which allows the computer to recognize words spoken with short (100 ms) pauses between them. Although isolated word recognition allows man to talk to a machine in a more natural manner, natural speech does not contain pauses between every word. Connected word recognition is an extension of isolated word recognition that allows several (typically less than six) words to be spoken together without pauses between them. An isolated word recognizer can be extended to recognize connected words by changing the DTW algorithm. Section 9.1 describes a level building dynamic time warping algorithm for connected word recognition [MyRi81a]. Section 9.2 presents a parallel DTW algorithm for connected word recognition.

## 9.1. A Level Building Dynamic Time Warping Algorithm

Myers and Rabiner have presented a thorough description of a general DTW algorithm for connected word recognition [MyRi81a,MyRi81b]. The algorithm presented here is from [MyRi81a]. We are given an unknown test pattern T(m) for $1 \leq m \leq M$ where each T(m) is a frame of speech, and M is the total number of frames in the pattern*. T(m) contains L utterances where $1 \leq L \leq L_{MAX}$. The purpose of the DTW is to find which known utterances $R_v$ are contained in pattern T(m), where $1 \leq v \leq V$ and V is the vocabulary size. This is done by making a "super" reference pattern $R^s$ by concatenating L reference patterns, i.e.,

$$R^s = R_{q(1)} R_{q(2)} R_{q(3)} \cdots R_{q(L)}$$

where q(n) for $1 \leq n \leq L_{MAX}$ selects which reference pattern to use in each position. The same DTW algorithm as used for isolated word recognition can then compare the test pattern T(m) to each of the super reference patterns $R^s$ as shown in Figure 9.1.

This is a computationally intense operation since there are many super reference patterns. If V=10, and $L_{MAX} = 5$, there are 11,111 super patterns. Myers' solution is a level building approach. Figure 9.2 shows graphically the computations used for the non-level building approach. The vertical lines show the order in which the distances are computed. The computation starts at the bottom of the leftmost vertical line, and proceeds up the line. After the first line is complete, the next line starts at the bottom and continues up, and so on. The warping path is restricted to the trapezoidal shaped region so that the warping path does not try to compare the end of the super reference pattern to the beginning of the test pattern.

Figure 9.3 shows the level building approach. The computation is as before, moving up from the bottom following the vertical lines. The difference is that the computations are done in levels. The lowest row of heavy dots represents the first level. Figure 9.2 follows the vertical lines up from the bottom comparing a given frame of T(m) to the first utterance in the super reference pattern, then the second utterance, and so on. Figure 9.3 starts at the bottom and compares a frame m=1 of T(m) to the reference pattern $R(n)_v$, but stops at the first level (row of dots). It records the accumulated distance and starts processing back at the bottom with frame T(m+1) and $R(n)_v$. This continues until accumulated distance scores are found for all the dots on level one.

*Previous chapters called the unknown pattern an utterance. Here the unknown pattern may consist of many utterances.

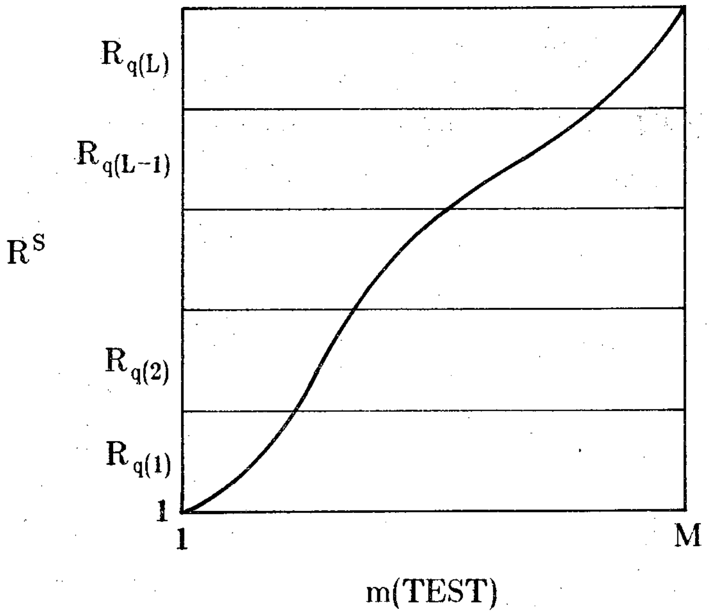


Figure 9.1. Illustration of dynamic warping alignment between text pattern T and super reference pattern $R_S$.

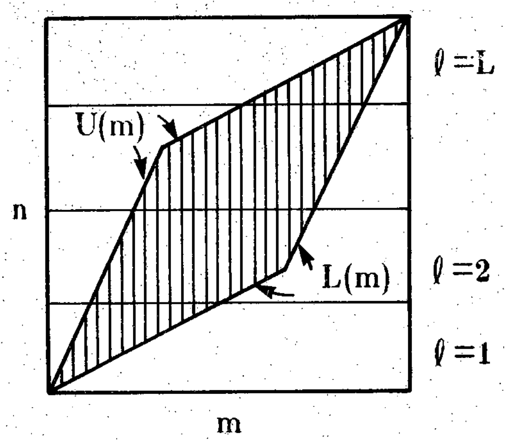


Figure 9.2. Graphical description of the computation order of non-level building algorithm.

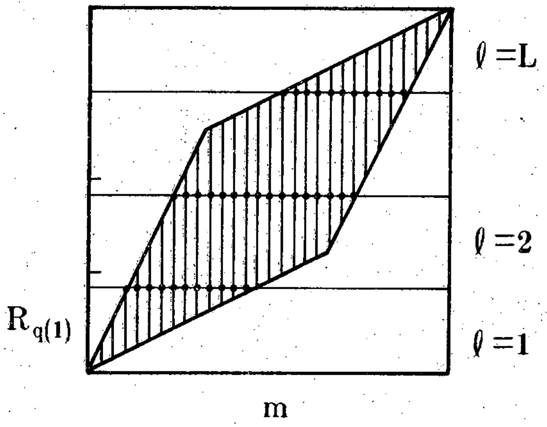


Figure 9.3. Graphical description of the computation order of level building algorithm.

At this point, the next reference pattern, $R(n)_{v+1}$, is compared to T(m) starting at m=1. This continues until all references patterns, $R_v$ for $1 \leq v \leq V$, are compared to T(m) starting at m=1. Each dot on Figure 9.3 has one accumulated distance score for each $R_v$. The minimum distance for each dot is saved and used as initial conditions for the next level of DTWing.

The algorithm in Figure 9.4 outlines the level building process. Table 9.1 shows the translation from the symbols used in [MyRi81a] to those used in Figure 9.4. Line 1 sets the accumulated distance for frame zero of level zero to zero. Lines 2 and 3 set the accumulated distance for all other frames on level zero to infinity. Lines 1-3 constrain the starting endpoint to the start of the super reference and the start of the test reference. Lines 5 and 6 set the accumulated distance of frame zero to infinity on all levels. Line 8 repeats lines 9-27 for each possible number of utterances in the test pattern. Line 9 repeats lines 10-18 for each utterance in the vocabulary. Lines 11 and 12 copy the best accumulated distance scores from the previous level to be used as initial conditions on the current level. For l=1, the previous level was set on lines 1-3 to allow only a path from the beginning of both the test and reference patterns. Lines 14-16 compute the accumulated distance in the same manner the isolated word DTW does. Line 14 selects which vertical line in Figure 9.3 to follow and line 15 selects the position on the line. Line 17 saves the accumulated distance at the locations with the dots in Figure 9.3. Lines 11-17 are repeated for each pattern in the vocabulary and the variable *DT* saves the accumulated distances for each pattern. Lines 20-22 initialize *DTB* and *W* to infinity for the current level. *DTB* is the minimum value of *DT* over all possible super reference patterns and *W* is the index of the minimum reference pattern. Then the shortest distance for each dot in Figure 9.3 is found by Lines 24-27. The best distances and the index of the word giving that distance are saved in *DTB* and *W* respectively. Lines 9-27 are repeated for each level. Lines 29-34 find the level with the smallest distance and set D to the distance.

Myers presents an algorithm with backtracking, so after finding D the reference patterns that composed the pattern can be found. Although backtracking is omitted here because it tends to obscure the function of the algorithm it could have been implemented on the BAC.

```
DTB(0,0) ← 0;                   /* Constrain starting point to   */
FOR m ← 1 TO M                          /* 1st frame of ref. pattern     */
     DTB(0,m) ← ∞;              /* and 1st frame of test patt.   */

FOR l ← 1 TO LMAX               /* Accumulated distance scores = ∞  */
     DTB(l,0) ← ∞;              /* on all levels                    */

FOR l ← 1 TO LMAX               /* For each level                */
     FOR v ← 1 TO V             /* For each vocabulary word         */

          FOR m ← 1 TO M        /* Set initial conditions of        */
               D(m,0) ← DTB(l−1,m);  /* current level to accumulated */
                                /* distances of previous level      */
          FOR m ← 1 TO M        /* Perform DTW as in isolated word system*/
               FOR n ← L(l,m) TO U(l,m)
                    $D(m,n) \leftarrow d(v,m,n) + \min \begin{bmatrix} D(m-1,n-2)+2d(v,m,n-1) \\ D(m-1,n-1)+d(v,m,n) \\ D(m-2,n-1)+2d(v,m-1,n) \end{bmatrix};$

          DT(v,m) ← D(m,Nv);    /* Save accumulated distances for word*/

     FOR m ← 1 TO M             /* Find minimum accumulated distance for*/
          DTB(l,m) ← ∞;         /* each dot.                            */
          W(l,m) ← ∞;

          FOR v ← 1 TO V                  /* For each vocab. word  */
               IF( DT(v,m) < DTB(l,m) )          /* If smaller, save */
                    DTB(l,m) ← DT(v,m);   /* distance, and    */
                    W(l,m) ← v;                  /* index to word too*/

L ← ∞;
D ← ∞;
FOR l ← LMIN TO LMAX            /* For all possible level find   */
     IF( DTB(l,M) < D )         /* shortest path                    */
          L ← l;
          D ← DTB(L,M);
```

Figure 9.4. Algorithm for serial level building DTW.

Table 9.1 Variable name translations for connected word algorithm.

| [MyRi81b] | Algorithm | Description |
|---|---|---|
| $T(m)$ | | Test pattern. |
| $M$ | | Length (in frames) of test pattern. |
| $R_v(n)$ | | Reference pattern $v$. |
| $V$ | | Number of reference words. |
| $R^s$ | | Super reference pattern consisting of a sequence of concatenated reference patterns. |
| $N_v$ | Nv | Length (in frames) of $v$th reference pattern. |
| $L$ | | Number of reference patterns in a string. |
| $D$ | D | Global distance between test pattern and super reverence pattern. |
| $D_l(m,n)$ | D(m,n) | Accumulated distance to frame $m$ of the test pattern, and frame $n$ of the $l$th reference of the super reference pattern. |
| $\tilde{D}_l^v(m)$ | DT(v,m) | Accumulated distance to frame $m$ of the text pattern, and the last frame of the $l$th reference of the super reference pattern for reference pattern $v$. |
| $d_l(m,n)$ | d(v,m,n) | Local distance between the $m$th frame of the test pattern, and the $n$th frame of the $l$th reference of the super reference pattern. |
| $L_l(m)$ | L(l,m) | Modified lower boundary function for the $l$th level. |
| $U_l(M)$ | U(l,m) | Modified upper boundary function for the $l$th level. |
| $L_{MAX}$ | LMAX | Maximum number of references in a super reference pattern. |
| $L_{MIN}$ | LMIN | Minumum number of references in a super reference pattern. |
| $\tilde{D}_l^B(m)$ | DTB(l,m) | Minimum value of $\tilde{D}_l(m)$ over all possible super reference patterns of length $l$. |
| $W_l(M)$ | W(l,m) | The index $v$, of the reference pattern $R_v$, that gives $\tilde{D}_l^B(m)$. |

## 9.2. An SIMD Level Building DTW Algorithm

The previous section presented a serial level building DTW algorithm. This section shows how it can be implemented on an SIMD machine.

The parallel level building DTW algorithm starts with the *s*erial *p*arallel (SP) algorithm discussed in section 6.4.1.1. The SP algorithm uses one PE for every utterance in the vocabulary. The unknown utterance is broadcast to all PEs and each PE executes a serial DTW program. Figure 9.5 shows the parallel version of Figure 9.4. Only a few changes are needed. The unknown pattern is broadcast to all PEs, and each PE does the level building warp similar to the serial program. After the accumulated distances are computed for each level, recursive doubling is used to find the minimum distance for each dot over all the vocabulary words. The minimum distance is stored in all PEs.

Line 9 of the serial algorithm is missing since all the vocabulary words are done in parallel. Lines 24-27 of Figure 9.4 are changed since the accumulated distances are spread across the PEs. Lines 24-36 of Figure 9.5 use recursive doubling to find the utterance with the smallest distance. The arrays *DTB* and *W* contain the same values in all PEs, therefore lines 39-44 for Figure 9.5 are the same as lines 32-37 of Figure 9.4.

Table 9.2 gives some computational comparisons between the serial and parallel level building DTW algorithms. The serial column is from [MyRi81a]*. $\bar{N}$ is the average reference pattern length in frames and M is the frame length of the test pattern. $\bar{N}M/3$ is the average number of distances at each level. This is shown by the the shaded area in Figure 9.2. Table 9.3 gives typical computational requirements for $L_{MAX}=5$, $V=10$, $M=120$, and $\bar{N}=35$.

*Table I-A on page 295.

```
DTB(0,0) ← 0                        /* Constrain starting point to first frame*/
FOR m ← 1 TO M                      /* of test pattern and first frame of*/
    DTB(0,m) ← ∞;                   /* reference pattern.*/

FOR l ← 1 TO LMAX                   /* Accumulated distance scores = ∞*/
    DTB(l,0) ← ∞; /* on all levels          */

FOR l ← 1 TO LMAX                           /* For each level*/
                                                    /* In all PEs*/

    FOR m ← 1 TO M                          /* Set initial conditions for current*/
        D(m,0) ← DTB(l−1,m);                /* level to acc. dists. on previous*/
                                                    /* level.*/
    FOR m ← 1 TO M                          /* Perform DTW as in isolated word*/
        FOR n ← L(l,m) TO U(l,m)                /* system.*/
            $D(m,n) \leftarrow d(m,n) + \min \begin{bmatrix} D(m-1,n-2)+2d(m,n-1) \\ D(m-1,n-1)+d(m,n) \\ D(m-2,n-1)+2d(m-1,n) \end{bmatrix};$

    DT(m) ← D(m,Nv);                        /* Save accumulated dist. for word v*/

FOR m ← 1 TO M                              /* Use recursive doubling to find*/
    DTB(l,m) ← ∞;                   /* minimum dist. for each dot*/
    W(l,m) ← ∞;

    DT ← DT(m);                             /* Store current frame's dist. in DT*/
    v ← ADDR+1;                             /* and index in v*/
    FOR i ← 0 TO $\lceil \log_2 V \rceil - 1$
        USE Cube(i);                        /* Send to another PE*/
        TRANSFER DT to DT′;
        TRANSFER v to v′;
        WHERE DT′ < DT
            DT ← DT′;                       /* Put smallest value in DT*/
            v ← v′;
        ENDWHERE

    DTM(l,m) ← DT;                          /* DT is the same in all PEs*/
    W (l,m) ← v;


L ← ∞;
D ← ∞;
FOR l ← LMIN TO LMAX                        /* Find smallest dist. for each*/
    IF( DTB(l,M) < D )                      /* level. Done serially in all*/
        L ← l;                              /* PEs with the same data.*/
        D ← DTB(L,M);
```

Figure 9.5. Algorithm for parallel level building DTW.

Table 9.2 Comparison of serial and parallel leveling building DTW algorithms.

|  | Serial | Parallel |
|---|---|---|
| Number of Basic Time Warps | $L_{MAX}V$ | $L_{MAX}$ |
| Size of Time Warps | $\bar{N}M/3$ | $\bar{N}M/3$ |
| Total Computations for Distances | $L_{MAX}V\bar{N}M/3$ | $L_{MAX}\bar{N}M/3$ |

Table 9.3 Comparison of serial and parallel leveling building DTW algorithms. Counts are for $L_{MAX}=5$, $\bar{N}=35$, $V=10$, and $M=120$.

| | Serial | Parallel |
|---|---|---|
| Number of Basic Time Warps | 50 | 5 |
| Size of Time Warps | 1,400 | 1,400 |
| Total Computations for Distances | 70,000 | 7,000 |

## 9.3. A VLSI Processor Array DTW Algorithm

The level building DTW algorithm can also be implemented on the BAC. Figure 9.6 gives the instructions that are executed by the array in Figure 6.21. The level building algorithm differs from the isolated word DTW in that the infinity vectors that separated utterances are used differently. Previously, all elements of the infinity vectors were infinity values. Now the second and third elements of the infinity vectors instruct the cell how to initialize its variables. The algorithm works as follows. An infinity vector enters the top and bottom of the array before any data is entered. The second element of the *a* vector (a[1]) is set to NEWUNKNOWN. This instructs the cells to set *ginit*, *g*, and *gmin* to infinity. *ginit* is the initial value of *g* on the current level, *gmin* is the minimum accumulated value of *g* for the current level, and *g* is as before, the current accumulated distance. The third element of *a* (a[2]) is an index telling which known utterance is being entered.

The known and unknown utterances enter as before, with the unknown utterances entering the bottom of the array one frame at a time and the known utterances entering the top of the array. The infinity vector that follows the known utterance has the second element set to NEWWORD, which instructs the cell to compare the current *g* value to the minimum *g* value of all the utterances which have been processed since the last NEWUNKNOWN value. If *g* is smaller than the previous *g*'s, its value is assigned to *gmin*, the index of the current utterance is saved in *minindex,* and *g* is assigned the initial value for the current level, *ginit*. After the infinity vector has propagated from the top to the bottom cell, a new known/unknown pair is started through the cells. Between levels, the second element of the infinity vector is set to NEWLEVEL which instructs the cell to set the initial value for the new level to the minimum value of the previous level, and set *gmin* to infinity.

Some observations about this approach are:

1) all the even (odd) cells are not executing the same code since those cells processing infinity values must execute extra instructions, and

2) one pair of known/unknown utterances must pass completely through the BAC before the next pair can enter. The original BAC allowed pairs of utterances to be separated by a single infinity vector, thus overlapping the computations and eliminating the initialization time.

Even numbered cells
Group A

```
a vector down
b vector up
if a[1] = NEWUNKNOWN
   ginit ← $\infty$
   g ← $\infty$
   gmin ← $\infty$
   index = a[2]
if a[1] = NEWWORD
   if(g < gmin)
      gmin ← g
      g ← ginit
      minindex ← index
if a[1] = NEWLEVEL
   ginit ← gmin
   gmin ← $\infty$
   index ← a[2]
   g ← ginit

compute d
DTtop ← d
DTbot ← d
$g \leftarrow d + \min \begin{bmatrix} g.bot.old + 2d.bot \\ g + d \\ g.top.old + 2d.top \end{bmatrix}$
g.top.old←g.top
g.bot.old ← g.bot
g.top ← DTtop
g.bot ← DTbot
d.bot ← DTbot
d.top ← DTtop
DTtop ← g
DTbot ← g
```

Odd numbered cells
Group B

```
a vector down
b vector up
if a[1] = NEWUNKNOWN
   ginit ← $\infty$
   g ← $\infty$
   gmin ← $\infty$
   index = a[2]
if a[1] = NEWWORD
   if(g < gmin)
      gmin ← g
      g ← ginit
      minindex ← index
if a[1] = NEWLEVEL
   ginit ← gmin
   gmin ← $\infty$
   index ← a[2]
   g ← ginit

compute d
d.bot ← DTbot
d.top ← DTtop
$g \leftarrow d + \min \begin{bmatrix} g.bot + 2d.bot \\ g + d \\ g.top + 2d.top \end{bmatrix}$

DTbot ← g
DTtop ← g
DTtop ← d
DTbot ← d
g.bot ← DTbot
g.top ← DTtop
```

Figure 9.6. Instructions executed during one loop of the BAC algorithm for I odd. (Exchange columns for I even).

Although the level building DTW can be implemented on the BAC, it is not a "clean" implementation in that cells executing the same code are not executing synchronously as before, and the data flow must be disrupted between each utterance to initialize various variables.

## 9.4. Summary

This chapter has presented two parallel algorithms for a level building dynamic time warp: one for the SIMD machine and the other for the VLSI processor array. The SIMD machine used one PE per vocabulary word and required only a few simple changes to the serial level building DTW algorithm. The VLSI processor array algorithm required only simple changes to the code executed in each cell, however the changes contained conditional branches which caused the cell taking the branch to be unsynchronized with the other cells. Also, the pipeline between cells must be reinitialized between utterances, therefore disrupting the data flow.

The HSAC [BAW81,BAW84] cannot implement the level building DTW since the HSAC has all cells in a diagonal executing the same instructions and it is not possible to instruct individual cells to save their accumulated distances.

The SIMD machine is well suited for performing the level building DTW since it requires few changes from the isolated word DTW, and the changes that are made do not alter the time needed to perform a basic time warp.

# 10. CONCLUSIONS

In this thesis, parallel algorithms for isolated word recognition were written for an VLSI processor array and an SIMD machine. These algorithms were simulated to determine if real-time execution was achievable and to obtain detailed measurements on the ways in which the architecture features were used. The simulations were run using parameters that a typical speech recognition system would use. The SIMD simulations showed that an SIMD machine with a MC68000 microprocessor in the CU and each PE could run in real time using 100 PEs. The VLSI processor array simulations showed that a processor array using Intel 8051s, in each cell, could not process speech data in real time. This inability to process fast enough was attributed to the 8-bit 8051 and the slow inter-cell communication, and not the parallel architecture model.

It is not meaningful to compare the execution times of an algorithm implemented on both parallel architectures since the SIMD machine uses a 16-bit microprocessor with an 8 MHz clock rate, and the VLSI processor array used an 8-bit microprocessor with a 12 MHz clock rate. Any such comparison will show that the 16-bit processor is better for speech processing than an 8-bit processor.

Desirable features for the SIMD machine architecture for real-time speech recognition are:

1) The processor should operate on 16-bit signed fixed-point data, have at least 18 data registers, and at least 2K bytes of general purpose memory in the CU and 512 bytes in each PE.

2) The interconnection network should implement the *Cube* and *Shift(± 1)* interconnection functions and have a data path from PE0 to the CU.

3) All PE masking operations can be performed using data conditional masks; however, many of the masks can be computed at compile time and executed as general PE masks.

4) If 100 PEs are used, the time to compare the input utterance to 1,000 known utterances will be less than 500 ms. Also, many of the speech system parameters can be changed and 100 PEs will still be able to process speech in real time.

Desirable features for a VLSI processor array for real-time speech processing are:

1) The processor should operate on 16-bit signed fixed point data and have at least 2K bytes of general purpose memory. The internal RAM of the 8051 is also a desirable feature since it can be used as if it were many fast general purpose registers.

2) The inter-cell communication must include a broadcast capability, and each cell should have both an input and output buffer between it and the other cells.

3) The speech recognition system needs at least 51 cells. Fewer cells could be used if the architecture supports broadcasting.

One comparison that can be made, however, is to compare the SIMD DTW program to the HSAC [BAW84]. The HSAC can produce a DTW comparison once every 40 $\mu$s for a throughput of 25,000 matches per second using a full array of 400 processors. An 8 by 16 reduced array of PEs can compute 5,000 matches per second. The SP DTW program, using 128 PEs, can compute 1,664 matches per second which is about 1/3 the rate of the reduced array HSAC. These figures show that the dedicated processors of the HSAC are able to compare utterances faster than the SIMD machine. However, the SIMD machine is more flexible in that it can perform a level building DTW for connected speech recognition and the HSAC cannot.

The work in this thesis could be extended by considering the following problems.

1) Simulate the cells of the VLSI processor array as if they were a digital signal processing chip instead of the 8051 microprocessor. The TMS32010 [TI83] digital signal processor can perform a 16 by 16-bit multiply in 200 ns and a 32-bit addition in 200 ns. Such performance is more than sufficient for speech processing and should improve the VLSI processor array throughput.

2) Simulate different inter-cell communications on the VLSI processor array.

Simulating circuit switched and packet switched communications with 1, 8, and 16 bit wide data paths would indicate which method is best suited for speech processing.

3) Simulate an instruction queue between the CU and PEs in the SIMD machine. Previous work has shown a 50% improvement in execution times for image processing [SiKu82]. Such simulations will show if speech processing can yield the same improvements.

4) Write algorithms for continuous speech recognition and simulate them. Continuous speech recognition is more time consuming than isolated word recognition and a powerful parallel processor might be able to recognize continuous speech in real time.

In summary, this thesis has shown that parallel processing is useful for real-time speech recognition. Through simulations, it demonstrated that both the SIMD machine and the VLSI processor array could implement an isolated word recognition system.

# LIST OF REFERENCES

# LIST OF REFERENCES

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffery D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass, 1974.

[ASV79] A. V. Ashajayanthi, S. Rajaram, and N. Viswanadham, "A Parallel Processor for Real-Time Speech Signal Processing," *1979 IEEE International Conference on Acoustics, Speech, and Signal Processing,* April 1979, pp. 868-871.

[AtHa71] Bishnu S. Atal and Susan L. Hanauer, "Speech Analysis and Synthesis by Linear Prediction of the Speech Wave," *Journal of the Acoustical Society of America,* Vol. 50, August 1971, pp. 637-655.

[Ba79] Kenneth Batcher, "MPP -- a Massively Parallel Processor," *1979 International Conference on Parallel Processing,* August 1979, pp. 249.

[BaLu81] George H. Barnes and Stephen F. Lundstrom, "Design and Validation of a Connected Network for Many-Processor Multiprocessor Systems," *Computer,* Vol. 14, No. 12, December 1981, pp. 31-41.

[Barn68] George H. Barnes, "The Illiac IV Computer," *IEEE Transactions on Computers,* Vol. C-17, August 1968, pp. 746-757.

[BAW81] David J. Burr, Bryan D. Ackland, and Neil Weste, "A High Speed Array Computer for Dynamic Time Warping," *Proceedings of 1981 the IEEE Acoustics, Speech, and Signal Processing,* April 1981, pp. 471-474.

[BAW84] David J. Burr, Bryan D. Ackland, and Neil Weste, "Array Configurations for Dynamic Time Warping," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* Vol. ASSP-23, No. 1, February 1984, pp. 119-128.

[BBGI80] Jeffrey A. Barnett, Morton I. Bernstein, Richard A. Gillmann, and Iris M. Kameny, "The SDC Speech Understanding System," in *Trends in Speech Recognition,* Prentice-Hall Inc, Englewood Cliffs, NJ 07632, 1980, pp. 272-293.

[BBN76] William Woods et. al., "Speech Understanding Systems, Final Technical Progress Report," No. 3438, Bolt Beranek and Newman Inc., October 1976.

[Bouk72] W. J. Bouknight, Stewart A. Denenberg, David E. McIntyre, J. M. Randall, Amed H. Sameh, and Daniel L. Slotnick, "The Illiac System," *Proceedings of the IEEE,* Vol. 60, No. 4, April 1972, pp. 369-388.

[BrSi82] Edward C. Bronson and Leah Jameison Siegel, "A Parallel Architecture for Acoustic Processing in Speech Understanding," *Proceedings of the 1982 International Conference on Parallel Processing,* Bellaire, Michigan, August, 1982, pp. 307-311.

[ClSi83] Carolyn Cline and Howard Jay Siegel, "Extension of ADA for SIMD Parallel Processing," *The IEEE Computer Society's Seventh International Computer Software and Applications Conference,* November 1983, pp. 366-372.

[Cran72] B. A. Crane, "PEPE Computer Architecture," *IEEE Computer Society Conference,* September 1972, pp. 57-60,

[Dec] Digital Equipment Corporation, *Macro-11 Assembler,* Publication number DEC-11-DMACA-A-D.

[Dodd81] George R. Doddington and Thomas B. Schalk, "Speech Recognition: Turning Theory to Practice," *IEEE Spectrum,* Vol. 18, No. 9, September 1981.

[Field] J. Timothy Field, Alejandro A. Kapauan, and Lawrence Snyder, "Pringle: A Parallel Processor to Emulate CHiP Computers," Purdue University Department of Computer Science, CSD-TR-443.

[Flyn66] Michael J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE,* Vol. 54, No. 12, December 1966, pp. 1901-1909.

[Hodg80] C. J. M. Hodges, Thomas P. Barnwell, and Daniel McWhorter, "The Implementation of an All Digital Speech Synthesizer Using a Multimicroprocessor Architecture," *IEEE International Conference on Acoustics, Speech, and Signal Processing,* April 9-11, 1980, pp. 855-858.

[Intel] Intel Corporation, "Intel MCS-51(tm) Family of Single Chip Microcomputers Users's Manual," Part Number 121517-001, July 1981.

[Itak75] Fumitada Itakura, "Minimum Prediction Residual Principle Applied to Speech Recognition," *IEEE Transactions Acoustics,* *Speech, and Signal Processing,* Vol. ASSP-23, No. 1, February 1975.

[JeWi74] Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report, second edition,* Springer-Verlag, New York, NY, 1974.

[KeRi78] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1978.

[KoSt73] Peter M. Kogge and Harold S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers,* Vol. C-22, No. 8, August 1973, pp. 786-793.

[Ku84] James T. Kuehn, internal correspondence.

[Kuck77] David J. Kuck, "A Survey of Parallel Machine Organization and Programming," *Computing Surveys,* Vol. 9, No. 1, March 1977, pp. 29-59.

[KuLe] H. T. Kung and Charles E. Leiserson, "Algorithms for VLSI Processing Arrays," in *Introduction to VLSI Systems,* edited by Carver Mead and Lynn Conway, Addison-Wesley, Reading, MA, 1980, pp. 271-294.

[Kung80] H. T. Kung, "The Structure of Parallel Algorithms," in *Advances in Computers,* Vol. 19, edited by Marshall C. Yovits, Academic Press, New York, NY, 1980.

[LeLi81] Stephen E. Levinson and Mark Y. Liberman, "Speech Recognition by Computer," *Scientific American,* April 1981, pp. 64-76.

[LMMB83] Menahem Lowy, Hy Murveit, David M. Mintz, Robert W. Broderson, "An Architecture for a Speech Recognition System," *IEEE 1983 International Solid State Circuits Conference,* Vol. 26, February 1983, pp. 118-119.

[LRRW81] Lori F. Lamel, Lawrence R. Rabiner, Arron E. Rosenberg, and Jay G. Wilpon, "An Improved Endpoint Detector for Isolated Word Recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* Vol. ASSP-29, No. 4, August 1981, pp. 777-785.

[MaGr74] John D. Markel and Augustine H. Gray, "Fixed-Point Truncation Arithmetic Implementation of a Linear Prediction Autocorrelation Vocoder," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* Vol. ASSP-22, No. 4, August 1974, pp 273-282.

[MaGy76] John D. Markel, Augustine H. Gray, Jr., "Linear Prediction of Speech", *Springer-Verlag,* New York, NY, 1976.

[Makh75] John Makhoul, "Linear Prediction: A Tutorial Review," *Proceedings of the IEEE,* Vol. 63, No. 4, April 1975, pp. 561-580.

[Mot79] Motorola Semiconductor, *MC68000 16-bit Microprocessor User's Manual,* Motorola IC Division, Austin, TX, 1979.

[MRR80] Cory Myers, Lawrence R. Rabiner, and Aaron E. Rosenberg, "Performance Tradeoffs in Dynamic Time Warping Algorithms for Isolated Word Recognition," *IEEE Transactions Acoustics, Speech, and Signal Processing,* Vol. ASSP-28, No. 6, December 1980, pp. 623-635.

[MSS80] Philip T. Mueller, Jr., Leah J. Siegel, and Howard Jay Siegel, "Parallel Algorithms for the Two-Dimensional FFT," *5th International Conference on Pattern Recognition,* December 1980, pp. 497-502.

[Myer80] Cory S. Myers, "A Comparative Study of Several Dynamic Time Warping Algorithms for Speech Recognition," Masters Thesis, Massachusetts Institute of Technology, February 1980.

[MyRa81a] Cory S. Myers and Lawrence R. Rabiner, "A Level Building Dynamic Time Warping Algorithm for Connected Word Recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* Vol. ASSP-29, No. 2, April 1981, pp. 284-297.

[MyRa81b] Cory S. Myers and Lawrence R. Rabiner, "Connected Digit Recognition Using a Level-Building DTW Algorithm," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* Vol. ASSP-29, No. 3, June 1981, pp. 351-363.

[Noll67] A. N. Noll, "Cepstrum Pitch Determination," *Journal of the Acoustic Society of America,* Vol. 41, February 1967, pp. 293-309.

[OpSc75] Alan V. Oppenheim and Ronald W. Schafer, *Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, NJ, 1975.

[Pe77] Marshall C. Pease, "The Indirect Binary N-cube Microprocessor Array," *IEEE Transactions on Computers,* Vol. C-26, No. 5, May 1977, pp. 458-573.

[RaGo75] Lawrence R. Rabiner and Ben Gold, *Theory and Application of Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, NJ, 1975.

[RaSa75] Lawrence R. Rabiner and Marvin R. Sambur, "An Algorithm for Determining the Endpoints of Isolated Utterances," *Bell System Technical Journal,* Vol. 54, No. 2, February 1975.

[RaSc78] Lawrence R. Rabiner and Ronald W. Schafer, *Digital Processing of Speech Signals,* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[RLRW79] Lawrence R. Rabiner, Stephen E. Levinson, Aaron E. Rosenberg, and Jay G. Wilpon, "Speaker-Independent Recognition of Isolated Words Using Clustering Techniques," *IEEE Transactions Acoustics, Speech, and Signal Processing,* Vol. ASSP-27, No. 4, August 1979, pp. 336-349.

[SaCh71] Hiroaki Sakoe and Seibi Chiba, "Dynamic Programming Algorithm Optimization for Spoken Word Recognition," *IEEE Transactions Acoustics, Speech, and Signal Processing,* Vol. ASSP-26, No. 1, February 1978, pp. 43-49.

[Sakoe79] Hiroaki Sakoe, "Two-Level DP-Matching - A Dynamic Programming-Based Pattern Matching Algorithm for Connected Word Recognition," *IEEE Transactions Acoustics, Speech, and Signal Processing,* Vol. ASSP-27, No. 6, December 1979, pp. 588-595.

[Saf82] Robert J. Safranek, "Speech Processing on SIMD Computers" Master of Science Thesis, Purdue University, School of Electrical Engineering, August 1982.

[SBK77] Herbert Sullivan, T. R. Bashkow, and David Klappholz, "A Large Scale Homogeneous, Fully Distributed Parallel Machine," *4th Symposium on Computer Architecture,* March 1977, pp. 105-124.

[Si77] Howard Jay Siegel, "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks," *IEEE Transactions on Computers,* Vol. C-26, No. 2, February 1977, pp. 153-161.

[Si79] Howard Jay Siegel, "Interconnection Networks for SIMD Machines," *Computer,* Vol. 12, June 1979, pp. 57-65.

[Si80a] Leah J. Siegel, "Parallel Processing Algorithms for Linear Predictive Coding," *IEEE International Conference on Acoustics, Speech, and Signal Processing,* April 1980, pp. 960-963.

[Si80b] Leah J. Siegel, Howard Jay Siegel, Robert J. Safranek, and Mark A. Yoder, "SIMD Algorithms to Perform Linear Predictive Coding for Speech Processing Applications," *1980 International Conference on Parallel Processing,* August, 1980, pp. 193-196.

[Si81] Leah J. Siegel, "Using SIMD Machines for Speech Analysis," *14th Annual Hawaii International Conference on System Sciences,* January, 1981, Vol. 1, pp. 309-318.

[Sieg81a] Howard Jay Siegel, Leah J. Siegel, Frederick Kemmerer, Philip T. Mueller, Jr., Harold E. Smalley, Jr., and S. Diane Smith, "PASM: a Partitionable Multimicrocomputer SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Transactions on Computers,* Vol. C-30, No. 12, December 1981, pp. 934-947.

[Sieg81b] Leah J. Siegel et al., "Parallel Image Processing/Feature Extraction Algorithms and Architecture Emulation: Interim Report for Fiscal 1981," Technical Report, TR-EE-81-35, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907.

[SiKu82] Howard Jay Siegel and James T. Kuehn, "Design and Simulation of a Multimicroprocessor System for Mapping Applications," Technical Report, TR-EE-83-18, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907, December 1984.

[SiMc81a] Howard Jay Siegel and Robert J. McMillen, "Using the Augmented Data Manipulator Network in PASM," *Computer,* Vol. 14, No. 2, February 1981, pp. 25-33.

[SiMc81b] Howard Jay Siegel and Robert J. McMillen, "The Multistage Cube: A Versatile Interconnection Network," *Computer,* Vol. 14, No. 12, December 1981, pp. 65-76.

[SiMu78] Howard Jay Siegel and Philip T. Mueller, Jr., "The Organization and Language Design of Microprocessors for an SIMD/MIMD System," *2nd Rocky Mountain Symposium on Microcomputers,* August 1978, pp. 311-340.

[SMS79] Leah J. Siegel, Philip T. Mueller, and Howard Jay Siegel, "FFT Algorithms for SIMD Machines," *Seventeenth Annual Allerton Conference on Communication, Control, and Computing,* October 1979, pp. 1006-1015.

[Snyder82a] Lawrence Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer* Vol. 15, No. 1, January 1982, pp 47-56.

[Snyder82b] Lawrence Snyder, "The Poker (1.0) Programmers Guide," Technical Report CSD-TR-434, Computer Science Department, Purdue University, West Lafayette, IN 47907, December 1982.

[Snyder83] Lawrence Snyder, "Introduction to the Poker Parallel Programming Environment," *1983 International Conference on Parallel Processing,* August 1983, pp 289-292.

[Sond68] Man Mohan Sondhi, "New Methods of Pitch Extraction," *IEEE Transactions on Audio Electroacoustics,* Vol. AU-16, No. 3, June 1968, pp. 262-266.

[Ston80] Harold S. Stone, "Parallel Computers," in *Introduction to Computer Architecture, 2nd edition,* edited by Harold S. Stone, Science Research Associates, Inc., Chicago, IL, 1980, pp. 362-425.

[Thre] Threshold Technology, Inc., Delran, NJ.

[TI83] "TMS32010 Digital Signal Processor," Texas Instruments, Dallas, Texas 75265, May 1983.

[ToGu81] H-m. D. Toong and A. Gupta, "An Architectural Comparison of Contemporary 16-bit Microprocessors," *IEEE Micro,* Vol. 1, May 1981, pp. 26-37.

[Verb] Verbex Corp., Bedford, Mass.

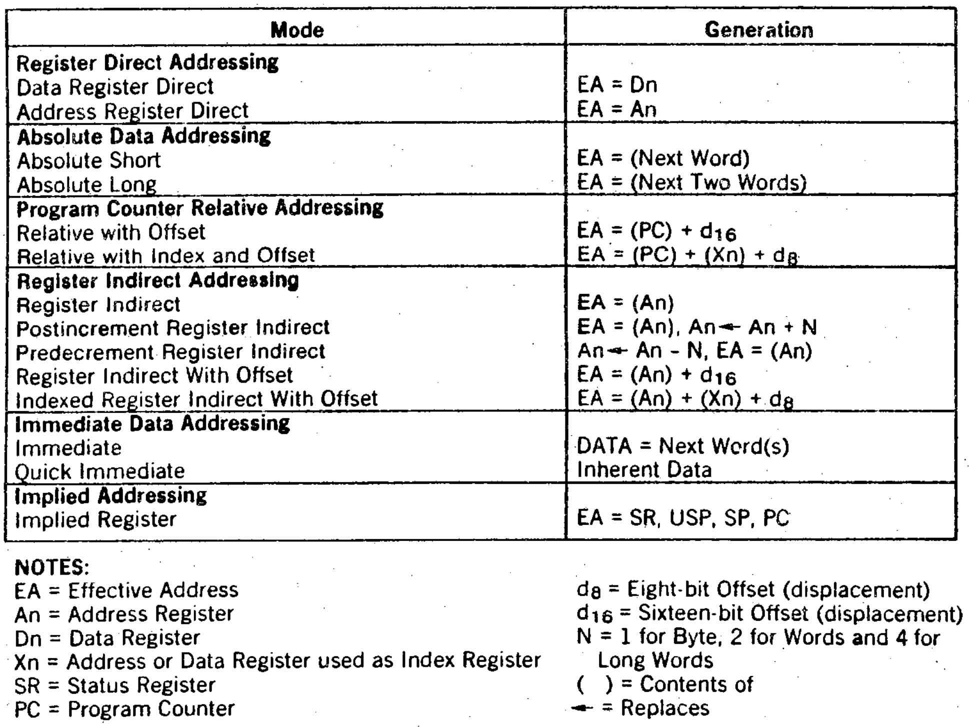
[WBA83] Neil Weste, David J. Burr, and Bryan D. Ackland, "Dynamic Time Warp Pattern Matching Using an Integrated Multiprocessing Array," *IEEE Transactions on Computers,* Vol. C-32, No. 8, August 1983, pp 731-744.

[YoSi81] Mark A. Yoder and Leah J. Siegel, "Systolic and SIMD Algorithms for Digital Filtering," *Proceeding of the Nineteenth Annual Allerton Conference on Communication, Control, and Computing,* October 1981, pp. 880-889.

[YoSi82] Mark A. Yoder and Leah J. Siegel, "Dynamic Time Warping Algorithms for SIMD Machines and VLSI Processor Arrays," *IEEE International Conference on Acoustics, Speech, and Signal Processing,* May 1982, pp. 1274-1277.

# APPENDICES

# APPENDIX A: SIMD Machine Assembly Language Programs

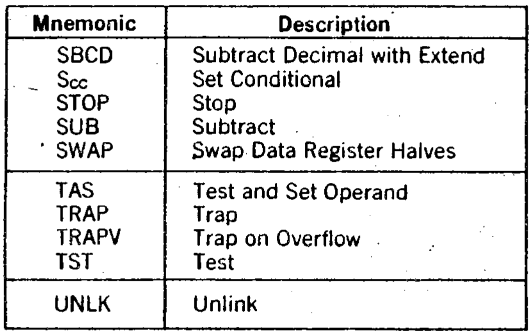


| Mode | Generation |
|---|---|
| **Register Direct Addressing** | |
| Data Register Direct | EA = Dn |
| Address Register Direct | EA = An |
| **Absolute Data Addressing** | |
| Absolute Short | EA = (Next Word) |
| Absolute Long | EA = (Next Two Words) |
| **Program Counter Relative Addressing** | |
| Relative with Offset | EA = (PC) + $d_{16}$ |
| Relative with Index and Offset | EA = (PC) + (Xn) + $d_8$ |
| **Register Indirect Addressing** | |
| Register Indirect | EA = (An) |
| Postincrement Register Indirect | EA = (An), An ← An + N |
| Predecrement Register Indirect | An ← An - N, EA = (An) |
| Register Indirect With Offset | EA = (An) + $d_{16}$ |
| Indexed Register Indirect With Offset | EA = (An) + (Xn) + $d_8$ |
| **Immediate Data Addressing** | |
| Immediate | DATA = Next Word(s) |
| Quick Immediate | Inherent Data |
| **Implied Addressing** | |
| Implied Register | EA = SR, USP, SP, PC |

**NOTES:**

EA = Effective Address
An = Address Register
Dn = Data Register
Xn = Address or Data Register used as Index Register
SR = Status Register
PC = Program Counter

$d_8$ = Eight-bit Offset (displacement)
$d_{16}$ = Sixteen-bit Offset (displacement)
N = 1 for Byte, 2 for Words and 4 for Long Words
( ) = Contents of
← = Replaces



| Mnemonic | Description |
|---|---|
| SBCD | Subtract Decimal with Extend |
| Scc | Set Conditional |
| STOP | Stop |
| SUB | Subtract |
| SWAP | Swap Data Register Halves |
| TAS | Test and Set Operand |
| TRAP | Trap |
| TRAPV | Trap on Overflow |
| TST | Test |
| UNLK | Unlink |

Figure A.1 MC68000 instruction set. (From [Mot79])

| Mnemonic | Description |
|---|---|
| ABCD | Add Decimal with Extend |
| ADD | Add |
| AND | Logical And |
| ASL | Arithmetic Shift Left |
| ASR | Arithmetic Shift Right |
| Bcc | Branch Conditionally |
| BCHG | Bit Test and Change |
| BCLR | Bit Test and Clear |
| BRA | Branch Always |
| BSET | Bit Test and Set |
| BSR | Branch to Subroutine |
| BTST | Bit Test |
| CHK | Check Register Against Bounds |
| CLR | Clear Operand |
| CMP | Compare |
| DBcc | Test Cond., Decrement and Branch |
| DIVS | Signed Divide |
| DIVU | Unsigned Divide |
| EOR | Exclusive Or |
| EXG | Exchange Registers |
| EXT | Sign Extend |
| JMP | Jump |
| JSR | Jump to Subroutine |
| LEA | Load Effective Address |
| LINK | Link Stack |
| LSL | Logical Shift Left |
| LSR | Logical Shift Right |
| MOVE | Move |
| MOVEM | Move Multiple Registers |
| MOVEP | Move Peripheral Data |
| MULS | Signed Multiply |
| MULU | Unsigned Multiply |
| NBCD | Negate Decimal with Extend |
| NEG | Negate |
| NOP | No Operation |
| NOT | One's Complement |
| OR | Logical Or |
| PEA | Push Effective Address |
| RESET | Reset External Devices |
| ROL | Rotate Left without Extend |
| ROR | Rotate Right without Extend |
| ROXL | Rotate Left with Extend |
| ROXR | Rotate Right with Extend |
| RTE | Return from Exception |
| RTR | Return and Restore |
| RTS | Return from Subroutine |

Figure A.1 (Continued)

```
;       Fixed addresses in CU space

MASKCTL              =       0x404   ; masking operations unit control port
FROMPE0              =       0x408   ; Data path from PE0

;       The following are standard definitions for the
;       PE transfer registers.

DTRDEST              =       0x400   ; PE address where  data is transfered to
DTRIN                =       0x402   ; Data transfer in from interconnection network
DTROUT               =       0x404   ; Data transfer out of network
TOCU                 =       0x40c   ; Data path to CU from PE0
#define NetworkDelay(x)      p_mov.10,0
                                      ; Network Delay (4.5 microseconds at 8 MHz)

;       The following are standard definitions for the
;       PE condition code registers.

PECCR =         0x408   ; Condition codes register (SR), size W, write only
PECCS =         0x40a   ; Condition codes select register, size, B, write only

;       The following are control words for the masking operations unit
;       See [SiKu82] for more details.

;OP
Pushs                =       0x0000  ; Push*
Pushss               =       0x1000  ; Push**
Pop                  =       0x2000  ; Pop
Initialize           =       0x3000

;S
DataCond             =       0x0600  ; Positive Data conditional mask
NDataCond            =       0x0700  ; Negative Data conditional mask

;       The following are control word for the condition codes select register
;       From page A-3 of the 68000 Assembly Language Programming Manual

T                    =       0x0     ; True
F                    =       0x1     ; False
HI                   =       0x2     ; High
LS                   =       0x3     ; Low or same
CC                   =       0x4     ; Carry clear
CS                   =       0x5     ; Carry set
NE                   =       0x6     ; Not equal
EQ                   =       0x7     ; Equal
VC                   =       0x8     ; No overflow
VS                   =       0x9     ; Overflow
PL                   =       0xa     ; Plus
```

Figure A.2 Contents of simd.h, the file describing the device locations in the address space.

```
MI                      =       0xb     ; Minus
GE                      =       0xc     ; Greater than or equal
LT                      =       0xd     ; Less than
GT                      =       0xe     ; Greater than
LE                      =       0xf     ; Less than or equal

;       Macro definitions for inter PE communications
;       These deinitions assume d0 is available for use,
;       and d7 contains WHOAMI.
;       In Transfer_l(in,out), in and out must be different D registers.
;       In Broadcast, in must be a register

#define Broadcast(in,out)                                       % \
                        c_mov.w         in,.+6.w                % \
                        p_mov.w         #0,out

#define Transfer_w(in,out)                                      % \
                        p_mov.w         in,DTRIN.w              % \
                        NetworkDelay(0)                         % \
                        p_mov.w         DTROUT.w,out

#define Transfer_l(in,out)                                      % \
                        p_mov.w         in,DTRIN.w              % \
                        NetworkDelay(0)                         % \
                        p_mov.w         DTROUT.w,out            % \
                        p_swap          in                      % \
                        p_swap          out                     % \
                        p_mov.w         in,DTRIN.w              % \
                        NetworkDelay(0)                         % \
                        p_mov.w         DTROUT.w,out            % \
                        p_swap          in                      % \
                        p_swap          out

#define Transfer_l1(in)                 % \
                        p_mov.w         in,DTRIN.w              % \
                        NetworkDelay(0)                         % \
                        p_mov.w         DTROUT.w,in             % \
                        p_swap          in                      % \
                        p_mov.w         in,DTRIN.w              % \
                        NetworkDelay(0)                         % \
                        p_mov.w         DTROUT.w,in             % \
                        p_swap          in

#define Shift(x)                                                % \
                        p_mov.w         d7,d0                   % \
                        p_add.w         x,d0                    % \
                        p_mov.w         d0,DTRDEST.w
```

Figure A.2 (Continued)

```
#define Cube(x)                                           % \
                    p_mov.w       d7,d0                   % \
                    p_bchg        x,d0                    % \
                    p_mov.w       d0,DTRDEST.w

#define Perm(x,y)                                         % \
                    p_mov.w       y,d0                    % \
                    p_sub.w       x,d0                    % \
                    p_subq.w #1,d0                        % \
                    p_mov.w       d0,DTRDEST.w

;                   Macro definitions for Data Conditional masking
#define Where(x,cond,y)                                   % \
                    p_cmp.w       y,x                     % \
                    p_mov.w       sr,PECCR.               % \
                    p_mov.b       #cond,PECCS.w           % \
                    .lock                                 % \
                    c_mov.w       #Pushs+DataCond,MASKCTL.w% \
                    .unlock

#define WhereElse(x,cond,y)                               % \
                    p_cmp.w       y,x                     % \
                    p_mov.w       sr,PECCR.w              % \
                    p_mov.b       #cond,PECCS.w           % \
                    .lock                                 % \
                    c_mov.w       #Pushs+NDataCond,MASKCTL.w% \
                    c_mov.w       #Pushss+DataCond,MASKCTL.w% \
                    .unlock

#define ElseWhere   c_mov.w       #Pop+DataCond,MASKCTL.w
#define EndWhere    c_mov.w       #Pop+DataCond,MASKCTL.w
```

Figure A.2 (Continued)

```
;               Definitions for main routine
STACK           =       0x1000          ; Put the stack at the top of memory

;               Definitions for autocorrelation
autocoef        =       9

;               Definitions for endpoint
lothresh        =       0x100
hithresh        =       0x200

;               Definitions for lpc
p               =       8

;               Definitions for ltw
MAXFRAMES       =       80

;               Definitions for dtw
inf             =       0x4000          ; Infinity
I               =       40              ; Number of frames in utterance
r               =       6               ; Width of warping path N = 2r+1
VOCABSIZE       =       10              ; Number of utterances in the vocabulary
```

Figure A.3 Contents of defs.h, the definition file.

```
;               Program Name:      filter
;               Algorithm:         Figure 6.10
;               Machine:           SIMD, simulated by a MC68000.
;               Function:          This program preemphasises the input speech
;                                  data with a filter with the transfer function:
;                                  H(z) = 1 - coef * z^-1
;               Precision:         Input:  16-bit signed
;                                  coef:   16-bit signed    (15 bits to the right
;                                                             of the decimal point.)
;                                  Output: 16-bit signed
;               Number of PEs:     N
;               Transfers:         Shift(+1)
;               Masking:           Data Conditional
;               Parameters:        coef, The filter coefficient (default = 0.95).
;                                  NetD, The interconnection network delay
;                                          time in cycles.
;               Input:             The input data is stored in PEs 0 through N-1.
;                                  PE i contains sample i for 0 ≤ i ≤ N-1.
;               Output:            The output data is stored in PEs 0 through N-1.
;                                  PE i contains sample i for 0 ≤ i ≤ N.
;               Cycles:            130 + NetD
;               Typical Time:      37 μs for one N sample frame
;               Register Usage:    (* means set by calling routine)
;                       d0      pe      used by macros
;                       d1      pe      tmp
;                       d2      pe      used to swap tmp and oldvalue
;                       d7*     pe      WHOAMI  (physical pe address)
;                       a0*     pe      points to input signal
;                       a1*     pe      points to output signal

#include "simd.h"
#include "defs.h"

;               Data allocation for routine
                .p_data         ; Data stored in each PE
coef:           .word   0x8667,0x8667,0x8667,0x8667,  ; 0x8667 = 0.95 * 2^15
                        0x8667,0x8667,0x8667,0x8667, \
                        0x8667,0x8667,0x8667,0x8667, \
                        0x8667,0x8667,0x8667,0x8667

                .p_bss
oldvalue: .=.+  2               ; Holds sample N-1 for next time

                .globl  filter

                .c_text
filter:
;
```

Figure A.4 Sim68 program to perform preemphasis filtering. Numbers on left are execution times in cycles.

```
          ; 1             USE Shift +1
          ;
12                        Shift(#1)        ; Set up interconnection network addresses

          ;
          ; 2             TRANSFER input TO tmp
          ;
4                         p_mov.w          (a0),d0
6                         p_mov.w          d0,DTRIN.w      ; transfer inputs from PE i to PE i-1
                          NetworkDelay(0)
6                         p_mov.w          DTROUT.w,d1

          fixold:
          ;
          ; 4             WHERE ADDR = 0 DO            /* Get value from previous call      */
          ;
28                        Where(d7,EQ,#0)              ; In PE0, get value from last call
          ;
          ; 5                     tmp2 <- tmp
          ; 6                     tmp  <- oldvalue     /* Switch tmp and oldvalue */
          ; 7                     oldvalue <- tmp2
          ;
2                                 p_mov.w       d1,d2
6                                 p_mov.w       oldvalue.w,d1
6                                 p_mov.w       d2,oldvalue.w
          ;
          ; 8             ENDWHERE
          ;
8                         EndWhere

          ;
          ; 10            output <- input +tmp * 0.95
          ;
39                        p_muls coef.w,d1        ; mult. by coef and save in d1.
4                         p_asl.l        #1,d1          ; shift 15 to the right by shifting left one,
2                         p_swap d1               ; and swapping upper and lower words.
2                         p_add.w        d1,d0          ; d0 = d0 + coef * d1
4                         p_mov.w        d0,(a1)        ; save in memory

          filterend:
8                         c_rts
```

Figure A.4 (Continued)

```
;               Program Name:   auto
;               Algorithm:      Figure 5.1
;               Machine:        SIMD, simulated by a MC68000.
;               Function:       This program finds the autocorrelation
;                               coefficients of input speech data.
;               Precision:      Input:  16-bit signed
;                               Output: 32-bit signed
;               Number of PEs:  N
;               Transfers:      Shift(−1), Cube
;               Masking:        Data Conditional
;               Parameters:     autocoef, The number of coefs. to find.
;                               N, The number of PEs in use.
;                               NetD, The interconnection network delay
;                                               time in cycles.
;               Input:          The input data is stored in PEs 0 through N−1
;                               with PE i containing sample i for 0 ≤ i ≤N.
;               Output:         The autocorrelation coefficients, R(i),
;                               for 0 ≤ i ≤autocoef−1 appear in PE i
;                               for 0 ≤ i ≤N (i.e. each PE contains
;                               every coefficient).
;               Cycles:         autocoef[136+NetD + (54+2NetD)logN]−12−NetD
;               Typical Time:   1,757 µs for autocoefs=9, NetD=18, and logN=7.
;               Register Usage: (* means set by calling routine)
;                       d0      pe      used by macros
;                       d1      pe      j,tmp
;                       d1      cu      j
;                       d2      pe      N−i, tmp
;                       d3      pe      partsum
;                       d4      pe      sig             input data
;                       d5      pe      slast
;                       d6      pe      i
;                       d6      cu      i
;                       d7*     pe      WHOAMI  (physical address)
;                       a0*     pe      pointer to input data
;                       a1*     pe      pointer to output coefficients
N               =       16
logN            =       4

#include "simd.h"
#include "defs.h"

;               Data allocation for routine

                .globl  auto

                .c_text
auto:
;
```

Figure A.5 Program performing autocorrelation. Numbers on left are execution times in cycles.

```
            ; 1          slast ← sig              /* After stage I, "slast" in PE m
            ;                                     holds sig(m+i) */
            ;
  4                      p_mov.w      (a0),d4         ; store sig in register
  2                      p_mov.w      d4,d5           ; slast <- sig

            ;
            ; 3          FOR i ← 0  TO p DO
            ;
  2                      p_clr.w  d6              ; i <- 0 in PE
  2                      c_clr.w  d6              ; i <- 0 in CU
            loop1:
            ;
            ; 4          IF i ≠ 0 THEN
            ;
  2                      c_tst.w  d6              ; if i == 0 jump to lab1
  5/4                    c_beq.s  lab1

            ;
            ; 5          USE Shift(-1)
            ;
  12                     Shift(#-1)

            ;
            ; 6          DTRin ← slast
            ; 7          TRANSFER
            ; 8          slast ← DTRout
            ;
  6                      p_mov.w        d5,DTRIN.w
  NetD                   NetworkDelay(0)
  6                      p_mov.w        DTROUT.w,d5

            lab1:
            ;
            ; 9          partsum ← 0
            ;
  2                      p_clr.w  d3
            ;
            ; 10         WHERE ADDR < N-i DO
            ;
            ;            where(N-i,GE,WHOAMI)
  4                      p_mov.w        #N,d2          ; d2 = N
  2                      p_mov.w        d6,d1          ; d1 = i
  2                      p_sub.wd1,d2               ; d2 = N-i

  26                     Where(d2,GT,d7)               ; d7 = WHOAMI
            ;
            ; 11         partsum ← slast * sig
            ;
  2                              p_mov.w        d4,d3   ; partsum <- sig
```

Figure A.5 (Continued)

```
35                         p_mul.s d5,d3    ; partsum <- slast * sig
         ;
         ; 12          ENDWHERE
         ;
8                      EndWhere

         ;
         ; 13          FOR j ← 0 logN-1 DO
         ;
2                      p_clr.w d1               ; j <- 0 in PEs
2                      c_movq #logN-1,d1        ; j <- log2(N-1) in CU
         loop2:
         ;
         ; 14          USE Cube(j)
         ;
12                     Cube(d1)
         ;
         ; 15          TRANSFER partsum TO tmp
         ;
32+2NetD               Transfer_l(d3,d2)
         ;
         ; 16          partsum ← tmp + partsum
         ;
3                      p_add.l d2,d3

         ;             Loop back for "FOR j ← 0 TO logN-1 DO"
2                      p_addq.w #1,d1           ; j++ in PEs
5/7                    c_dbf    d1,loop2                 ; j-- in CU

         ;
         ; 17          R(i) ← partsum
         ;
6                      p_mov.ld3,(a1)+

         ;             Loop back for "FOR i ← 0  TO p DO"
2                      p_addq.w        #1,d6             ; i++ in PEs
2                      c_addq.w        #1,d6             ; i++ in CU
4                      c_cmp.w         #autocoef,d6      ; if i < number autocorrelation coefs.
5/6                    c_blt    loop1           ;        loop

         autoend:
8                      c_rts
```

Figure A.5 (Continued)

```
;               Program Name:     auto/2
;               Algorithm:        Figure 7.3
;               Machine:          SIMD, simulated by a MC68000.
;               Function:         This program finds the autocorrelation
;                                 coefficients of input speech data using
;                                 half as many PEs as samples in a frame.
;               Precision:        Input:  16-bit signed
;                                 Output: 32-bit signed
;               Number of PEs:    N
;               Transfers:        Shift(-1), Cube
;               Masking:          Data Conditional
;               Parameters:       autocoef, The number of coefs. to find.
;                                 N, The number of PEs in use.
;                                 NetD, The interconnection network delay
;                                               time in cycles.
;               Input:            The input data is stored in PEs 0 through N-1
;                                 with PE i containing sample i for 0 ≤ i ≤N.
;               Output:           The autocorrelation coefficients, R(i),
;                                 for 0 ≤ i ≤autocoef-1 appear in PE i
;                                 for 0 ≤ i ≤N (i.e. each PE contains
;                                 every coefficient).
;               Cycles:           autocoef[136+NetD + (54+2NetD)logN]-12-NetD
;               Typical Time:     1,757 μs for autocoefs=9, NetD=18, and logN=7.
;               Register Usage: (* means set by calling routine)
;                         d0      pe      used by macros
;                         d1      pe      j,tmp
;                         d1      cu      j
;                         d2      pe      N-i, tmp
;                         d3      pe      partsum
;                         d4      pe      sig          input data
;                         d5      pe      slast
;                         d6      pe      i
;                         d6      cu      i
;                         d7*     pe      WHOAMI  (physical address)
;                         a0*     pe      pointer to input data
;                         a1*     pe      pointer to output coefficients

N               =       4
logN            =       2

#include "simd.h"
#include "defs.h"

;               Data allocation for routine

                .globl   auto2
```

Figure A.6 Program performing autocorrelation using half as many PEs as frames. Numbers on left are execution times in cycles.

```
                .c_text
        auto2:
        ;
        ; 1     slast1 ← sig1  /* After stage I, "slast" in
        ; 2                        PE m holds sig(m+1) */
        ; 3     1.57   slast2 ← sig2  /* After stage I, "slast" in
        ; 4                        PE m holds sig(m+1) */
        ;
4               p_mov.w      (a0)+,d4       ; store first half of frame in a reg.
2               p_swap d4            ; move to upper 16 bits
4               p_mov.w (a0),d4             ; store second half in other half of reg.
2               p_swap d4            ; Keep first half in lower 16 bits
2               p_mov.ld4,d5         ; slast <- sig

        ;
        ; 6     FOR i ← 0  TO p DO
        ;
2               p_clr.w  d6          ; i <- 0 in PE
2               c_clr.w  d6          ; i <- 0 in CU
        loop1:
        ;
        ; 7     IF i ≠ 0 THEN
        ;
2               c_tst.w  d6          ; if i == 0 jump to lab1
5/4             c_beq.s  lab1

        ;
        ; 8     USE Shift(−1)
        ;
12              Shift(#−1)                  ; TRANSFER USING shift(−1)

        ;
        ; 9     DTRin ← slast1
        ; 10    TRANSFER
        ; 11    slast1 ← DTRout
        ;
                                     ; Send first half through network
6               p_mov.w      d5,DTRIN.w    ; DTRIN <- slast
NetD            NetworkDelay(0)
6               p_mov.w      DTROUT.w,d5 ; slast <- DTROUT

        ;
        ; 12    DTRin ← slast2
        ; 13    TRANSFER
        ; 14    slast2 ← DTRout
        ;
2               p_swap d5            ; Send second half through network
6               p_mov.w      d5,DTRIN.w    ; DTRIN <- slast
NetD            NetworkDelay(0)
6               p_mov.w      DTROUT.w,d5 ; slast <- DTROUT
```

Figure A.6 (Continued)

```
2                  p_swap d5

        ;
        ; 16       WHERE(ADDR,EQ,N-1)
        ; 17               tmp ← slast1
        ; 18               slast1 ← slast2
        ; 19               slast2 ← tmp
        ; 20       ENDWHERE
        ;
28                 Where(d7,EQ,#N-1)
2                     p_swap      d5
8                  EndWhere

        lab1:
        ;
        ; 22       partsum ← 0
        ;
2                  p_clr.w d3               ; partsum <- 0
        ;
        ; 24       WHERE ADDR < M-i DO
        ; 25               partsum ← slast2 * sig2
        ; 26       ENDWHERE
        ;

        ;          where(N-i,GE,WHOAMI)
4                  p_mov.w        #N,d2          ; d2 = N
2                  p_mov.w        d6,d1          ; d1 = i
2                  p_sub.wd1,d2             ; d2 = N-i
26                 Where(d2,GT,d7)          ; d7 = WHOAMI
2                     p_mov.l     d4,d3          ; partsum <- sig (first half)
2                     p_swap      d3
2                     p_swap      d5
35                    p_mul.s     d5,d3          ; partsum <- slast * sig
2                     p_swap      d5
8                  EndWhere

        lab2:
        ;
        ; 28       partsum ← partsum + slast1 * sig1
        ;
2                  p_mov.w        d4,d2          ; now compute second half
35                 p_mul.s d5,d2
3                  p_add.l d2,d3            ; Add to halves together

        ;
        ; 30       FOR j ← 0 TO logN-1 DO
        ;
2                  p_clr.w d1               ; j <- 0
2                  c_movq #logN-1,d1        ; j <- log2(N-1)
        loop2:
```

Figure A.6 (Continued)

```
          ;
          ; 31          USE Cube(j)
          ; 32          TRANSFER partsum TO tmp
          ;
12                      Cube(d1)              ; Transfer Using CUBE
32+2NetD                Transfer_l(d3,d2)
          ;
          ; 33          partsum ← tmp + partsum
          ;
3                       p_add.l d2,d3

2                       p_addq.w #1,d1        ; j++ in pe
5/7                     c_dbf    d1,loop2     ; j-- in cu

          ;
          ; 34          R(i) ← partsum
          ;
6                       p_mov.ld3,(a1)+       ; R(i) <- partsum

2                       p_addq.w #1,d6        ; i++ in PE
2                       c_addq.w #1,d6        ; i++ in cu
4                       c_cmp.w      #autocoef,d6    ; if i < number autocorrelation coefs.
5/6                     c_blt    loop1        ;         loop

          autoend:
8                       c_rts
```

Figure A.6 (Continued)

```
;               Program Name:    lpc
;               Algorithm:       Figure 5.7
;               Machine:         SIMD, simulated by a MC68000.
;               Function:        This routine finds the LPC coefficients using
;                                       Durbin's method.
;               Precision:       Input:         16-bits, signed
;                                Output: 16-bits, signed with 12 bits to the
;                                               right of the decimal point
;               Number of PEs:   p, the number of LPC coefficients.
;               Transfers:       Cube, Perm
;               Masking:         Data Conditional
;               Parameters:      p, the number of LPC coefficients.
;                                NetD, the interconnection network delay
;                                       time in cycles.
;               Input:           Each PE contains all the autocorrelation
;                                coefficients R(i) for 0 ≤ i ≤ p.
;               Output:          The results are stored in a(i), with
;                                PE(i-1) containing a(i) for 0 ≤ i < p.
;               Cycles:          p[513+NetD+(54+2NetD)log(p)]-38-NetD
;               Typical Time:    1,588 μs for p=8 and NetD=18
;               Register usage:  (* means set by calling routine)
;                       d1      pe      a               LPC coefficients
;                       d2      pe      k
;                       d3      pe      indexing
;                       d4      pe      j
;                       d4      cu      j
;                       d5      pe      i
;                       d5      cu      i
;                       d6      pe      LADDR           logical PE address
;                       d7*     pe      WHOAMI          physical PE address
;                       a1*     pe      R               points to the array of
;                                                       autocorrelation coef.
;                       a2      pe      R               points to current R value

p               =       8
logp            =       3

#include "simd.h"
#include "defs.h"

        .globl          lpc
        .p_data
LADDR:  .word           1,2,3,4,5,6,7,8

        .p_bss
E       .=.+            3

        .c_text
```

Figure A.7 Program to finding LPC coefficients.

```
        lpc:
6                     p_mov.w     LADDR.w,d6
2                     p_mov.l     a1,a2              ; a2 points to current R value

        ;
        ; 1           E ← R(0)
        ;
8                     p_mov.w     (a2)+,E.w          ; E = R[0]                        E=16.0
        ;
        ; 2           a ← 0
        ;
2                     p_clr.w     d1                 ; a = 0

        ;
        ; 3           FOR i ← 1 TO p DO              /* Compute k(i)                       */
        ;
4                     p_mov.w     #1,d5              ; i = 1
4                     c_mov.w     #1,d5              ; i = 1

        mainloop:
        ;
        ; 4           k ← 0
        ;
3                     p_clr.l     d2                 ; k = 0                          d2=4.12
        ;
        ; 5           WHERE LADDR < i DO
        ;
26                    Where(d6,LT,d5)
        ;
        ; 6           k ← a * R(i-LADDR)
        ;
2                     p_mov.w     d5,d3              ; d3 = i
2                     p_sub.w     d6,d3              ; d3 = i-LADDR
5                     p_asl.l     #1,d3              ; *2 for word addressing
7                     p_mov.w     0(a1,d3.w),d2      ; d2=k=R[i-LADDR]                d2=16.0
35                    p_muls      d1,d2              ; d2=k=a*R[i-LADDR]
        ;                                                                   d1=4.12 d2=4.12
        ;
        ; 7           ENDWHERE
        ;
8                     EndWhere

        ;
        ; 11          FOR j ← 0 TO logp - 1 DO
        ;
2                     p_clr.w     d4                 ; j = 0
2                     c_movq      #logp-1,d4         ; j = log2(p) - 1

        ;
        ; 12          USE Cube(j)
```

Figure A.7 (Continued)

```
          ;
          again:
12                      Cube(d4)    ; cube(j)
          ;
          ; 13          DTRIn ← k
          ; 14          TRANSFER
          ;
32+2NetD                Transfer_l(d2,d0)
          ;
          ; 15          k ← k + DTRout
          ;
3                       p_add.l     d0,d2              ; k = k + R[i−LADDR]

          ;             Loop back for "FOR j ← 0 TO logp − 1 DO"
2                       p_addq.w #1,d4                 ; j++
5/7                     c_dbf       d4,again

          ;
          ; 17          k ← [R(i) − k] / E
          ;
          cubedone:
4                       p_mov.w     (a2)+,d3           ; d3 = R[i]                       d3=16.0
2                       p_ext.l     d3                 ; sign extend                     d3=16.0
2                       p_movq      #12,d0             ; shift by 12 to move decimal place
16                      p_asl.l     d0,d3              ;                                 d3=16.12
3                       p_sub.l     d2,d3              ; d3 = R[i]−k                     d3=16.12
2                       p_mov.l     d3,d2              ;                                 d3=16.12
83                      p_divs      E.w,d2             ; k = (R[i]−k)/E            E=16.0 d2=4.12

          ;
          ; 18          E ← 1 − $k^2$ * E
          ;
          findE:
2                       p_mov.w     d2,d3              ;                                 d3=4.12
35                      p_muls      d3,d3              ;                                 d3=8.24
16                      p_asr.l     d0,d3              ;                                 d3=8.12
3                       p_neg.l     d3                 ;                                 d3=8.12
8                       p_add.l     #0x1000,d3         ; k = 1 − k^2                     d3=8.12
39                      p_muls      E.w,d3             ; E = (1−k^2)*E            E=16.0 d3=20.12
16                      p_asr.l     d0,d3              ;                                 d3=16.0
6                       p_mov.w     d3,E.w             ;                                  E=16.0

          findk:
          ;
          ; 21          USE $\mathrm{PERM}_{LADDR/1-LADDR}$
          ;
12                      Perm(d6,d5)
          ;
          ; 22          WHERE LADDR = 1 DO
          ; 23          a ← k       /* $a_i^{(i)}$←k(i)          */
```

Figure A.7 (Continued)

```
        ;
34              WhereElse(d6,EQ,d5)
2               p_mov.w    d2,d1          ; a = k               d1=4.12 d2=4.12
6               p_mov.w    d1,DTRIN.w     ; DTRIN = a                   d1=4.12
        ;
        ; 24    ELSEWHERE
        ;
8               ElseWhere
        ;
        ; 25    WHERE LADDR < 1 DO
        ;
26              Where(d6,LT,d5)
        tonet:
        ;
        ; 26    DTRin ← a
        ; 27    TRANSFER
        ;
6               p_mov.w    d1,DTRIN.w     ; DTRIN = a                   d1=4.12
NetD            NetworkDelay(0)
        ;
        ; 28    a ← a − k * DTRout
        ;
39              p_muls     DTROUT.w,d2    ; k = k * a                   d2=8.24
2               p_movq     #12,d0
16              p_asr.l    d0,d2          ;                             d2=8.12
2               p_sub      d2,d1          ; a=a−k*DTROUT    d1=4.12 d2=4.12
        ;
        ; 29    ENDWHERE
        ;
8               EndWhere
        ;
        ; 30    ENDWHERE
        ;
8               EndWhere

        ;       Loop back for "i ← 1 TO p DO"
        inccounters:
2               p_addq.w   #1,d5          ; i++
2               c_addq.w   #1,d5          ; i++
4               c_cmp.w    #p+1,d5
5/6             c_bne      mainloop
        endlpc:
8               c_rts
```

Figure A.7 (Continued)

```
;               Program Name:   ltw
;               Algorithm:      Figure 6.13
;               Machine:        SIMD, simulated by a MC68000
;               Function:       This program does a linear time warp
;                               on the input data
;               Precision:      Input:  16-bit signed
;                               Output: 16-bit signed
;               Number of PEs:  Max number of frames.
;                               (J or I whichever is greater.)
;               Transfers:      Shift(-1), Broadcast
;               Masking:        Data Conditional
;               Parameters:     J-I, the changed in the number of frames
;                               p, the number of coefficients per frame
;                               NetD, the network delay time.
;               Input:          PE j holds frame j for
;                               0 ≤ j < number of input frames (J)
;               Output:         PE i holds frame i for
;                               0 ≤ i < number of output frames (I)
;               Cycles:         if J>I  325+(138+NetD)p
;                                          +(J-I)[59+NetD+(29+NetD)p]
;                               if J=I  47 + 11p
;                               if J<I  344+(138+NetD)p
;                                          +(I-J)[57+NetD+(45+NetD)p]
;               Typical Time:   7,382 μs for I-J=10
;               Register Usage: (* means register is set by the calling routine)
;                       d0      pe      used by macros
;                       d0*     cu      J               Starting number of frames
;                       d1*     cu      I               Finishing number of frames
;                       d3      pe      1-s
;                       d3      cu      i
;                       d4      pe      i
;                       d4      cu      i
;                       d5      pe      factor,s
;                       d5      cu      factor
;                       d6      pe      i_tmp
;                       d6      cu      J
;                       d7*     pe      WHOAMI  (physical pe address)
;                       d7      cu      I
;                       a4      pe      R1
;                       a5*     pe      R               Points to current input frame
;                       a6*     pe      Tout            Points to current output frame

#include "simd.h"
#include "defs.h"

            .globl      ltw
            .c_text
ltw:
```

Figure A.8 Program for linear time warping using one frame per PE.

```
        ;
        ; 1         IF(I = J) THEN
        ;
2                   c_cmp.w     d0,d1               ; IF(J==I)
4/5                 c_bne.s     lab1
        ;
        ; 2         T ← R
        ;
2                   c_movq      #p-1,d0             ; Number of coefs to transfer
        loop1:
6                   p_mov.w     (a5)+,(a6)+         ; T <- R
5/7                 c_dbf       d0,loop1
        ltwend2:
        ;
        ; 3         RETURN
        ;
8                   c_rts

        lab1:
2                   c_mov.w     d0,d6               ; J
2                   c_mov.w     d1,d7               ; I

        ;
        ; 5         factor ← (J-1) / (I-1)
        ;

2                   c_subq.w    #1,d0               ; J-1
2                   c_subq.w    #1,d1               ; I-1

        ;           Since "factor", "tmp", and "s" are fractions,
        ;           they are represented as fixed decimal by shifting them left
        ;           by X places.  The notation X.Y means there are X bits to
        ;           the left of the decimal and Y bits to the right.

        ;           shift lower 16 bits to upper 14 bits
        ;           since quot. is between .5 and 2  d0=2.14
6                   c_ror.l     #2,d0               ; Faster to rotate right and
2                   c_swap      d0                  ; swap words
70                  c_divu      d1,d0               ; d0 <- d0/d1           d1=16.0 d0=2.14
2                   c_mov.w     d0,d5               ; factor <- (J-1)/(I-1)         d5=2.14

        ;           BROADCAST d5 From CU to PEs
10                  Broadcast(d5,d5)

        ;
        ; 6         i ← ⌈ADDR/factor⌉
        ;

2                   p_mov.w     d7,d1
2                   p_swap      d1
```

Figure A.8 (Continued)

```
11              p_asl.l     #23-16,d1       ; asl.l #23,d2 the fast way
70              p_divu      d5,d1           ; d1 <- d1/d5              d5=2.14 d1=7.9
4               p_add.w     #0x1ff,d1       ; find ceiling of d1
2               p_movq      #9,d2
12              p_asr.w     d2,d1           ; i <- ceil(ADDR/factor)           d1=16.0
2               p_mov.w     d1,d4           ;                                  d4=16.0

        ;
        ; 12    IF(I > J) THEN
        ;
2               c_cmp.w     d7,d6           ; IF(I>J)
5/6             c_bgt       lab2

        ;
        ; 14    FOR i ← 1 TO I − J
        ;
        ;       FOR i <- I-J-1 TO 0 STEP -1
2               c_mov.l     d7,d3           ; i <- I
3               c_sub.l     d6,d3           ; i <- I-J
4               c_subq.l    #1,d3           ; i <- I-J-1

        ;
        ; 13    USE Shift +1
        ;
12              Shift(#1)   ; Set up network for TRANSFER USING shift(1)

        ;       At this point, all PEs transfer their "i" values to the
        ;       network, but only the enabled PEs will read the values

        for1:
2               p_movq      #p+p,d0
2               c_movq      #p-1,d0
6               p_mov.w     d4,DTRIN.w      ; TRANSFER i USING shift(+1)

        ;
        ; 15    WHERE(ADDR < i) DO
        ;
26              Where(d4,GT,d7)

NetD            NetworkDelay(0)
6               p_mov.w     DTROUT.w,d4     ; TRANSFER i USING shift(+1)

        ;
        ; 16    TRANSFER i
        ; 17    TRANSFER R
        ;
        loop2:

        ;       The section of code turns on all PEs so they can write thier
        ;       data to the network, it then turns off the PEs that were
```

Figure A.8 (Continued)

```
        ;       disabled before.

8               c_mov.w    #Pop+DataCond,MASKCTL.w                  ; enable all PEs

2               p_subq.w   #2,d0
11              p_mov.w    0(a5,d0.w),DTRIN.w              ; TRANSFER each R coef.

8               c_mov.w    #Pushs+DataCond,MASKCTL.w               ; disable some PEs

NetD            NetworkDelay(0)
11              p_mov.w    DTROUT.w,0(a5,d0.w)                      ; USING shift(+1)
5/7             c_dbf      d0,loop2

        ;
        ; 18    ENDWHERE
        ;
8               EndWhere

5/7             c_dbf      d3,for1


        ;
        ; 19    i ← ADDR
        ;
2               p_mov.w    d7,d4

        lab2:
        ;
        ; 21    tmp ← i * factor +1
        ;

2               p_mov.w    d4,d0            ;                             d0=16.0
35              p_mulu     d5,d0            ;                      d5=2.14 d0=2.14
8               p_add.l    #0x4000,d0       ; add 1 (factor is still shifted)

        ;
        ; 22    j ← ⌊tmp⌋
        ;

2               p_mov.l    d0,d1            ; d1 <- tmp                   d1=2.14
8               p_and.l    #0xffffc000,d1   ; d1 == j
        ;
        ; 23    s ← tmp − j
        ;
3               p_sub.l    d1,d0
2               p_mov.l    d0,d5            ; s == d5                     d5=2.14

        ;
        ; 24    USE Shift −1
        ;
```

Figure A.8 (Continued)

```
12                  Shift(#-1)
        ;
        ; 25        TRANSFER R to R1
        ;
4                   p_mov.w    #R1,a4

2                   c_movq     #p-1,d0
2                   p_movq     #p+p,d0
        loop3:
2                   p_subq.w   #2,d0

11                  p_mov.w    0(a5,d0.w),DTRIN.w                      ; DTRin <- R
NetD                NetworkDelay(0)
11                  p_mov.w    DTROUT.w,0(a4,d0.w)      -             ; R1 <- DTRout

5/7                 c_dbf      d0,loop3

        ;
        ; 26        T ← (1-s) * R +s * R1
        ;
4                   p_mov.w    #0x4000,d3         ; move a shifted 1 to d3        d3=2.14
2                   p_sub.w    d5,d3              ; (1-s) == d3

2                   c_movq     #p-1,d0
2                   p_movq     #p+p,d0
        loop4:
2                   p_subq.w   #2,d0

2                   p_mov.w    d3,d1              ; d1 <- (1-s)
40                  p_muls     0(a5,d0.w),d1      ; d1 <- (1-s) * R              d1=2.14
2                   p_mov.w    d5,d2              ; d2 <- s
40                  p_muls     0(a4,d0.w),d2      ; d2 <- s * R1                 d2=2.14
3                   p_add.l    d1,d2
6                   p_rol.l    #2,d2              ; shift right by 14 by rotating
2                   p_swap     d2                 ; left 2 and swapping
7                   p_mov.w    d2,0(a6,d0.w)      ; T <- (1-s)*R + s*R1

5/7                 c_dbf      d0,loop4

        ;
        ; 31        IF(I < J) THEN
        ;
2                   c_cmp.w    d7,d6              ; IF(I<J)
5/4                 c_blt.s    ltwend

        ;
        ; 32        FOR i ← 1 TO J - I
        ;
        ;           FOR i <- J-I-1 TO 0 STEP -1
2                   c_mov.w    d6,d3              ; i <- J
```

Figure A.8 (Continued)

```
2               c_sub.w     d7,d3               ; i <- J-I
2               c_subq.w    #1,d3               ; i <- J-I-1

        for2:
        ;
        ; 33    TRANSFER i TO i_tmp
        ;
6               p_mov.w     d4,DTRIN.w          ; DTRin <- i
NetD            NetworkDelay(0)
6               p_mov.w     DTROUT.w,d6         ; i_tmp <- DTRout

        ;
        ; 34    WHERE(i_tmp ≤ ADDR) DO
        ;
26              Where(d6,LE,d7)

2               c_movq      #p-1,d0
2               p_movq      #p+p,d0

        ;
        ; 35    TRANSFER T
        ;
        loop5:
2               p_subq.w    #2,d0

11              p_mov.w     0(a6,d0.w),DTRIN.w                          ; DTRin <- T
NetD            NetworkDelay(0)
11              p_mov.w     DTROUT.w,0(a6,d0.w)                         ; T <- DTRout

5/7             c_dbf       d0,loop5
        ;
        ; 36    i ← i_tmp
        ;
2               p_mov.w     d6,d4               ; i <- i_tmp
        ;
        ; 37    ENDWHERE
        ;
8               EndWhere

5/7             c_dbf       d3,for2             ; FOR i <- J-I-1 TO 0 STEP -1

        ltwend:
8               c_rts
```

Figure A.8 (Continued)

```
        ;               Program Name:   ltw2
        ;               Algorithm:      Figure 6.16
        ;               Machine:        SIMD, simulated by a MC68000
        ;               Function:       This program does a linear time warp on
        ;                               the input data
        ;               Precision:      Input:  16-bits, signed
        ;                               Output: 16-bits, signed
        ;               Number of PEs:  p, the number of coefficients.
        ;               Transfers:      Broadcast
        ;               Masking:        Data Conditional
        ;               Parameters:     p, the number of coefficients.
        ;                               I, the number of output frames.
        ;               Input:          PE k contains coefficient k of frame j
        ;                               for 0 ≤ k < p and 0 ≤ j < Total number of frames
        ;               Output:         PE k contains coefficient k of frame j
        ;                               for 0 ≤ k < p and 0 ≤ j < I.
        ;               Cycles:         if J ≠ I, 107 + 183I
        ;                               if J = I, 450
        ;               Typical Time:   1,857 µS for I=40.
        ;               Register usage: (* means register is set by the calling routine)
        ;                       d0      pe      used by macros
        ;                       d0*     cu      J-1
        ;                       d1*     cu      I-1
        ;                       d3      pe      1-s
        ;                       d4      pe      i
        ;                       d4      cu      i
        ;                       d5      pe      factor
        ;                       d5      cu      factor
        ;                       d6      pe      s
        ;                       d7*     pe      WHOAMI  (physical pe address)
        ;                       a4      pe      R1
        ;                       a5*     pe      R       Pointer to input frames
        ;                       a6*     pe      Tout    Pointer to output frames

        #include "simd.h"
        #include "defs.h"

                        .globl          ltw
                        .c_text
        ltw:
        ;
        ; 1             IF(J = I) THEN
        ;
2                       c_cmp.w         d0,d1                   ; IF(J==I)
5/4                     c_bne.s         lab1
        ;
        ; 2             T ← R
        ;
```

Figure A.9 Program for linear time warping using p PEs.

```
2                 c_subq.w   #1,d0              ; Number of coefs to transfer
        loop1:
6                 p_mov.w    (a5)+,(a6)+        ; T <- R
5/7               c_dbf      d0,loop1
        ;
        ; 3       RETURN
        ;
8                 c_rts

        ;
        ; 5       factor ← (J-1)/(I-1)
        ;
        lab1:
2                 c_subq.w   #1,d0              ; J-1
2                 c_subq.w   #1,d1              ; I-1

        ;         Since "factor", "tmp", and "s" are fractions,
        ;         they are represented as fixed decimal by shifting them left
        ;         by X places.  The notation X.Y means there are X bits to
        ;         the left of the decimal and Y bits to the right.

        ;         shift lower 16 bits to upper 14 bits
        ;         since quot. is between .5 and 2  d0=2.14
6                 c_ror.l    #2,d0              ; shift to left and then
2                 c_swap     d0                 ; swap to upper byte
70                c_divu     d1,d0              ; d0 <- d0/d1          d1=16.0 d0=2.14
2                 c_mov.w    d0,d5              ; factor <- (J-1)/(I-1)        d5=2.14

        ;         BROADCAST d5 From CU to PEs
10                Broadcast(d5,d5)
        ;
        ; 7       FOR i ← 0 TO I-1
        ;
        ;         FOR i <- 0 to I-1 in pes and FOR i <- I-1 to 0 STEP -1 in cu

2                 p_clr.w    d4
2                 c_mov.w    d1,d4              ; i = I-1

        ;
        ; 8       tmp ← i * factor + 1
        ;
        for1:
2                 p_mov.w    d4,d0              ;                              d0=16.0
35                p_mulu     d5,d0              ;                      d5=2.14 d0=2.14
8                 p_add.l    #0x4000,d0         ; add 1 (factor is still shifted)

        ;
        ; 9       j ← ⌊tmp⌋
        ;
        findm:
```

Figure A.9 (Continued)

```
2               p_mov.l     d0,d1             ; d1 <- tmp                       d1=2.14
8               p_and.l     #0xfffc000,d1     ; d1 == j
        ;
        ; 10    s ← tmp − j
        ;
        finds:
3               p_sub.l     d1,d0
2               p_mov.l     d0,d6             ; s == d6                         d5=2.14

        ;       T <- (1−s) * R + s * R1

4               p_mov.w     #0x4000,d3        ; move a shifted 1 to d3          d3=2.14
2               p_sub.w     d6,d3             ; (1−s) == d3

        ;       Adjust j (d1) so it can index R()
        ;       shift by 13 since indexing words d1=15.0
7               p_rol.l     #3,d1             ; It's faster to shift right 3
2               p_swap      d1                ; and swap
2               p_subq.w    #2,d1             ; of 2 bytes each

        ;
        ; 11    T ← (1−s) * R(j) + s * R(j+1)
        ;
        findT:
2               p_mov.w     d3,d0             ; d0 <- (1−s)
40              p_muls      0(a5,d1.w),d0     ; d0 <- (1−s) * R                 d1=2.14
2               p_mov.w     d6,d2             ; d2 <- s
40              p_muls      2(a5,d1.w),d2     ; d2 <- s * R1                    d2=2.14
3               p_add.l     d0,d2
6               p_asl.l     #2,d2             ; Unshift to normal range         d2=16.0
2               p_swap      d2                ; Save time by shifting left 2 and
                                              ; swaping words
4               p_mov.w     d2,(a6)+          ; T <- (1−s)*R + s*R1

2               p_addq.w    #1,d4             ; i = i + 1

5/7             c_dbf       d4,forl
        ltwend:
8               c_rts
```

Figure A.9 (Continued)

Jul 17 08:51 1984 main.s Page 1

```
;               Program Name:   main (PP1)
;               Algorithm:      Figure 6.18
;               Machine:        SIMD, simulated by a MC68000
;               Function:       This program calls the dynamic time
;                               warp routine.
;               Precision:      Input:  16-bit signed
;                               Output: 16-bit signed
;               Number of PEs:  2r+1
;               Transfers:      Shift(± 1)
;               Masking:        Data Conditional
;               Parameters:     r, the width of the warping path.
;                               p, the number of coefficients per frame.
;                               NetD, the network delay time.
;                               I, the number of frames per utterance.
;               Input:          All PEs hold all the input data.
;               Output:         PE r holds the distance score.
;               Cycles:         See text
;               Register Usage:
;                       d5      pe      LADDR    (logical address)
;                       d7      pe      WHOAMI   (physical address)
;                       a0      pe      Pointer to start of Input data
;                       a1      pe      Pointer to start of Unknown data
;                       a2      pe0     Pointer to where to store results.
;                                       a2++ after stored

#include "simd.h"
#include "defs.h"

                .p_text
instr:          .=.+ 10         ; Space for PE instructions

                .p_data
                .globl  WHOAMI
                .globl  input
                .globl  unknown

WHOAMI:         .word   0,1,2,3,4

unknown:                        ; Unknown input utterance
                .word   2,3,4,5,0
                .word   6,7,8,1,0
                .word   8,7,6,5,0
                .word   4,3,2,1,0

                .word   0,0,0,0,0
```

Figure A.10 Parallel program for parallel-parallel DTW algorithm.

Jul 17 08:51 1984 main.s Page 2

```
                .word   0,0,0,0,0
                .word   0,0,0,0,0
                .word   0,0,0,0,0

                .word   inf,inf,inf,inf,inf, inf,inf,inf,inf,inf, \
                        inf,inf,inf,inf,inf, inf,inf,inf,inf,inf
                .word   inf,inf,inf,inf,inf, inf,inf,inf,inf,inf, \
                        inf,inf,inf,inf,inf, inf,inf,inf,inf,inf

input:                          ; Known input utterance
                .word   1,1,1,1,1, 2,2,2,2,2, 3,3,3,3,3, 4,4,4,4,4
                .word   5,5,5,5,5, 6,6,6,6,6, 7,7,7,7,7, 8,8,8,8,8
                .word   8,8,8,8,8, 7,7,7,7,7, 6,6,6,6,6, 5,5,5,5,5
                .word   4,4,4,4,4, 3,3,3,3,3, 2,2,2,2,2, 1,1,1,1,1

                .word   0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0
                .word   0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0
                .word   0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0
                .word   0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0

                .word   inf,inf,inf,inf,inf, inf,inf,inf,inf,inf, \
                        inf,inf,inf,inf,inf, inf,inf,inf,inf,inf
                .word   inf,inf,inf,inf,inf, inf,inf,inf,inf,inf, \
                        inf,inf,inf,inf,inf, inf,inf,inf,inf,inf

                .p_bss
scores:         .=.+    10      ; Distance scores

                .globl  d
                .globl  dold
                .globl  dDTR
                .globl  dup
                .globl  ddown
                .globl  g
                .globl  gold
                .globl  gDTR
                .globl  gup
                .globl  gdown

d:              .=.+    2       ; Local distance
g:              .=.+    2       ; Optimal subpath distance
dold:           .=.+    2       ; Old local distance
gold:           .=.+    2       ; Old optimal subpath distance
dDTR:           .=.+    2       ; d values to transfer
gDTR:           .=.+    2       ; g values to transfer
dup:            .=.+    2       ; d values to transfer to PE with next higher addr.
gup:            .=.+    2       ; g values to transfer to PE with next higher addr.
ddown:          .=.+    2       ; d values to transfer to PE with next lower addr.
```

Figure A.10 (Continued)

```
Jul 17 08:51 1984 main.s Page 3

        gdown:          .=.+    2       ; g values to transfer to PE with next lower addr.

                        .word   inf,inf,inf,inf,inf, inf,inf,inf,inf,inf, \
                                inf,inf,inf,inf,inf, inf,inf,inf,inf,inf
                        .word   inf,inf,inf,inf,inf, inf,inf,inf,inf,inf, \
                                inf,inf,inf,inf,inf, inf,inf,inf,inf,inf

        fixed:          .=.+    1280    ; Input data rearranged for dtw

                        .c_text
        main:
4                       c_mov.w         #STACK,a7                       ; set up stack pointer
8                       c_mov.w         #Initialize,MASKCTL.w           ; Initialize CMS
8                       c_mov.w         #0x8000,MASKCTL.w

6                       p_mov.w         WHOAMI.w,d7 ; d7 always has WHOAMI in it
        ;                                           (Physical address)
4                       p_mov.w         #inf,d6 ; Store infinity in d6
2                       p_mov.w         d7,d5       ; Store logical address in d5
2                       p_subq.w        #r,d5

4                       p_mov.w         #unknown,a0 ; Take data in the order put out by lpc, and
4                       p_mov.w         #fixed,a1   ; and rearrange it for the dtw routine.
10                      c_jsr           shuffle

4                       p_mov.w         #input,a0
4                       p_mov.w         #fixed,a1
4                       p_mov.w         #scores,a2
10                      c_jsr           dtw         ; Find distance between input and unknown

        mainend:
2                       c_stop          #0000
```

Figure A.10 (Continued)

```
Jul 17 09:09 1984 dtw.s Page 1

;               Program Name:   dtw (PP1)
;               Algorithm:      Figure 6.18??
;               Machine:        SIMD, simulated by a MC68000
;               Function:       This program does a dynamic time warp
;                               on the input data
;               Precision:      Input:  16-bit signed
;                               Output: 16-bit signed
;               Number of PEs:  2r+1
;               Parameters:     r, the width of the warping path.
;                               p, the number of coefficients per frame.
;                               NetD, the network delay time.
;                               I, the number of frames per utterance.
;               Input:          All PEs hold all the input data.
;               Output:         PE r holds the distance score.
;               Cycles:         See text
;               Typical Time:   10 ms for I=40, r=6, NetD=18, and p=8.
;               Register Usage: (* means register is set by the calling routine)
;                       d2      pe      tmp storage
;                       d5*     pe      LADDR   (logical address)
;                       d6      pe      inf (infinity)
;                       d7*     pe      WHOAMI  (physical address)
;                       d7      cu      k
;                       a0*     pe      Pointer to start of Input data
;                       a1*     pe      Pointer to start of Unknown data
;                       a2*     pe0     Pointer to where to store results.
;                                       a2++ after stored

;               The subroutine  distance(x,y) returns the distance between
;               frame x of utterance 1 and frame y of utterance 2.
;               x and y are passed in the d0 and d1.
;               The result is returned in d0.

#include "defs.h"
#include "simd.h"

;               Data allocation for routine
                .p_bss
                .globl  d
                .globl  dold
                .globl  dDTR
                .globl  dup
                .globl  ddown
                .globl  g
                .globl  gold
                .globl  gDTR
                .globl  gup
                .globl  gdown
```

Figure A.10 (Continued)

```
Jul 17 09:09 1984 dtw.s Page 2

                        .c_text
                        .globl   dtw
        dtw:
4                       p_mov.w        #inf,d6 ; Store infinity in d6

        ;
        ; 10            Xindex ←+⌈ADDR/2⌉
        ;
        findindex:
2                       p_mov.w        d5,d0    ; Xindex <- ceil(LADDR/2)
2                       p_addq.w       #1,d0
4                       p_asr.w #1,d0  ; LADDR/2
7                       p_asl.w #4,d0  ; Multiply index by 2^4 for p=8.
                                                ; 2^4 = autocoef * word size
4                       p_add.w        d0,a0

        ;
        ; 11            Yindex ←−⌊ADDR/2⌋
        ;
2                       p_mov.w        d5,d0    ; Yindex <- floor(LADDR/2)
4                       p_asr.w #1,d0
2                       p_neg.wd0
7                       p_asl.w #4,d0  ; Multiply index by 2^4 for p=8.
                                                ; 2^4 = autocoef * word size
4                       p_add.w        d0,a1

        ;
        ; 2             g ← 0
        ; 3             gold ← 0
        ; 4             d ← ∞
        ; 5             dold ← ∞
        ;
8                       p_clr.w g.w             ; g <- 0
8                       p_clr.w gold.w          ; gold <- 0
6                       p_mov.w        d6,d.w         ; d <- inf
6                       p_mov.w        d6,dold.w      ; dold <- inf

        ;
        ; 6             WHERE ADDR = 0 DO
        ; 7                     g ← 0
        ; 8             ENDWHERE
        ;
28                      Where(d5,EQ,#0)
8                               p_clr.w g.w     ; g <- 0
8                       EndWhere

        ;
        ; 13            FOR k ← 1 TO I DO
```

Figure A.10 (Continued)

Jul 17 09:09 1984 dtw.s Page 3

```
        ;
4                       c_mov.w       #I-1,d7 ; FOR k <- I-1 TO 0 STEP -1 DO

        ;
        ; 14            compute d(Xindex,Yindex)
        ;
        for1:
10                      c_jsr         distance
6                       p_mov.w       d1,d.w

        ;
        ; 15            WHERE ADDR is even DO
        ; 16                 dDTR ← dold
        ; 17                 gDTR ← gold
        ;
2                       p_mov.w       d5,d0             ; WHERE LADDR is even DO
4                       p_and.w       #1,d0
36                      WhereElse(d0,EQ,#0)
10                           p_mov.w       d.w,dDTR.w
10                           p_mov.w       g.w,gDTR.w
        ;
        ; 18            ELSEWHERE
        ; 19                 dDTR ← d
        ; 20                 gDTR ← g
        ; 21            ENDWHERE
        ;
8                       ElseWhere
10                           p_mov.w       dold.w,dDTR.w
10                           p_mov.w       gold.w,gDTR.w
8                       EndWhere

        ;
        ; 23            USE Shift +1
        ; 24            TRANSFER dDTR TO dup
        ; 25            TRANSFER gDTR TO gup
        ;
        movedataup:
12                      Shift(#1)                          ; TRANSFER dDTR TO dup
10                      p_mov.w       dDTR.w,DTRIN.w
NetD                    NetworkDelay(0)
10                      p_mov.w       DTROUT.w,dup.w

10                      p_mov.w       gDTR.w,DTRIN.w       ; TRANSFER gDTR TO gup
NetD                    NetworkDelay(0)
10                      p_mov.w       DTROUT.w,gup.w

        ;
        ; 27            USE Shift -1
```

Figure A.10 (Continued)

Jul 17 09:09 1984 dtw.s Page 4

```
        ; 28          TRANSFER dDTR TO ddown
        ; 29          TRANSFER gDTR TO gdown
        ;
        movedatadown:
12                    Shift(#-1)                          ; TRANSFER dDTR TO ddown
10                    p_mov.w       dDTR.w,DTRIN.w
NetD                  NetworkDelay(0)
10                    p_mov.w       DTROUT.w,ddown.w

10                    p_mov.w       gDTR.w,DTRIN.w        ; TRANSFER gDTR TO gdown
NetD                  NetworkDelay(0)
10                    p_mov.w       DTROUT.w,gdown.w

        fixends:
        ;
        ; 31          WHERE ADDR = r DO
        ; 32                gdown ← ∞
        ; 33          ENDWHERE
        ;
28                    Where(d5,EQ,#r)                     ; WHERE LADDR = +r DO
6                           p_mov.w       d6,gdown.w      ; gdown <- inf
8                     EndWhere

        ;
        ; 35          WHERE ADDR = −r DO
        ; 36                gup ← ∞
        ; 37          ENDWHERE
        ;
28                    Where(d5,EQ,#−r)                    ; WHERE LADDR = −r DO
6                           p_mov.w       d6,gup.w        ; gup <- inf
8                     EndWhere

        ;
        ; 39          gold ← g
        ; 40          dold ← d
        ;
10                    p_mov.w       g.w,gold.w      ; gold <- g
10                    p_mov.w       d.w,dold.w      ; dold <- d

        ;
        ; 42          A ← gdown + 2 * ddown
        ;
        findA:
6                     p_mov.w       ddown.w,d0      ; A <- gdown + 2 * ddown
4                     p_asl.w #1,d0           ; 2*ddown
6                     p_add.w       gdown.w,d0

        ;
```

Figure A.10 (Continued)

Jul 17 09:09 1984 dtw.s Page 5

```
         ; 43           B ← gold  +d
         ;
         findB:
6                       p_mov.w      d.w,d1  ; B <- gold + d
6                       p_add.w      gold.w,d1

         ;
         ; 44           C ← gup   +2 * dup
         ;
         findC:
6                       p_mov.w      dup.w,d2       ; C <- gup + 2 * dup
4                       p_asl.w #1,d2
6                       p_add.w      gup.w,d2

         ;
         ; 46           WHERE B < A DO
         ; 47                A ← B
         ; 48           ENDWHERE
         ;
         findmin:
26                      Where(d1,LS,d0)             ; g <- min(A,B,C) + d
2                            p_mov.w      d1,d0
8                       EndWhere

         ;
         ; 49           WHERE C < A DO
         ; 50                A ← C
         ; 51           ENDWHERE
         ;
26                      Where(d2,LS,d0)
2                            p_mov.w      d2,d0
8                       EndWhere

         ;
         ; 52           g ← A +d
         ;
6                       p_add.w      d.w,d0         ; d0 now holds min(d0,d1,d2)
6                       p_mov.w      d0,g.w         ; g <- min(A,B,C)
26                      Where(d6,LS,d0)             ; where(d0 < inf)
6                            p_mov.w      d6,g.w  ; g <- inf
8                       EndWhere

         incindex:
5/7                     c_dbf        d7,for1  ; FOR d7 <- I/2-2 TO -1 STEP -1

         ;
         ; 57           WHERE ADDR = 0 DO
         ; 58                D(A,B) ← g/(I+J)
```

Figure A.10 (Continued)

```
Jul 17 09:09 1984 dtw.s Page 6

        ; 59            ENDWHERE
        ;
28                      Where(d5,EQ,#0)
8                               p_mov.w         g.w,(a2)+
8                       EndWhere

        dtwend:
8                       c_rts
```

Figure A.10 (Continued)

```
Jul 17 09:10 1984 distance.s Page 1

        ;               Program Name:     distance (PP1)
        ;               Algorithm:        Figure 6.18??
        ;               Machine:          SIMD, simulated by a MC68000
        ;               Function:         distance finds the distance
        ;                                 between to sets of coefficients
        ;               Precision:        Input:  16-bit signed
        ;                                 Output: 16-bit signed
        ;               Number of PEs:    2r+1
        ;               Transfers:        None
        ;               Masking:          Data Conditional
        ;               Parameters:       p, the number of coefficients per frame.
        ;               Input:            All PEs hold all the input data.
        ;               Output:           PE r holds the distance score.
        ;               Cycles:           50p + 76
        ;               Typical Time:     474 µs for p=8

        ;               Register usage:   (* means passed as argument)
        ;                       d0    pe     Used by macros
        ;                       d1    pe     running sum
        ;                       d1    pe     returns total distance
        ;                       d2    pe     Current lpc coefficient
        ;                       d4    cu     loop counter
        ;                       d5    pe     LADDR     (logical pe address)
        ;                       d6    pe     inf
        ;                       d7    pe     WHOAMI  (physical pe address)
        ;                       a0*   pe     points to Input frames
        ;                       a1*   pe     points to Unknown frames

        #include "simd.h"
        #include "defs.h"

                        .globl   distance
                        .c_text
        distance:
        ;               where( (a0) == inf || (a1) == inf )
4                       p_cmp  (a0),d6          ; is (a0) == inf ?
3                       p_mov.w       sr,d0
4                       p_cmp.w       (a1),d6  ; is (a1) == inf ?
3                       p_mov.w       sr,d1
2                       p_or.w  d1,d0                      ; is (a0) == inf OR (a1) == inf ?
6                       p_mov.w       d0,PECCR.w
8                       p_mov.b       #EQ,PECCS.w
                        .lock
8                       c_mov.w       #Pushs+NDataCond,MASKCTL.w
8                       c_mov.w       #Pushss+DataCond,MASKCTL.w
                        .unlock

2                               p_mov.w       d6,d1   ; return(inf)
```

Figure A.10 (Continued)

```
Jul 17 09:10 1984 distance.s Page 2

6                            p_add.w       #p+p,a0
6                            p_add.w       #p+p,a1

8                    ElseWhere

2                    p_movq #0,d1          ; sum = 0;

2                    c_movq #p-1,d4
       loop1:
4                    p_mov.w       (a0)+,d2
4                    p_sub.w(a1)+,d2
35                   p_muls  d2,d2
2                    p_add.w       d2,d1          ; sum += [input - unknown]^2

5/7                  c_dbf         d4,loop1

       distanceend:
8                    EndWhere

8                    c_rts
```

Figure A.10 (Continued)

```
Jul 17 10:23  1984 shuffle.s Page 1

        ;               Program Name:     shuffle (PP1)
        ;               Algorithm:        Figure dtw.4??
        ;               Machine:          SIMD, simulated by a MC68000
        ;               Function:         This program rearranges the output data
        ;                                 from lpc for input to dtw.
        ;               Precision:        Input:   16-bit signed
        ;                                 Output: 16-bit signed
        ;               Number of PEs:    2r+1
        ;               Transfers:        FROMPE0, Shift(-1), Broadcast
        ;               Masking:          None
        ;               Parameters:       p, the number of coefficients per frame.
        ;                                 NetD, the network delay time.
        ;                                 I, the number of frames per utterance.
        ;               Input:            PE i contains coefficient i of frame j
        ;                                 for 0 ≤ i <p and 0 ≤j < I.
        ;               Output:           All PEs contain all coefficients for all PEs
        ;               Cycles:           16+I[6+p(47+Netd)+2+5]+2+6+9⌊r/2⌋
        ;               Typical Time:     5,344 μs
        ;               Register usage:   (* means passed as argument)
        ;                       d0      pe     Used by macros
        ;                       d1      pe     Current lpc coefficient
        ;                       d3      cu     loop counter
        ;                       d4      cu     loop counter
        ;                       d6      pe     Infinity
        ;                       a0*     pe     points to Input frames
        ;                       a1*     pe     points to Unknown frames

        #include "simd.h"
        #include "defs.h"


                        .globl   shuffle
                        .c_text
        shuffle:
4                       c_mov.w         #I-1,d3          ; Total of I frames will be shuffled.
12                      Shift(#-1)

        sloop2:
2                       c_movq #p-1,d2          ; shift over p lpc coefficients

4                       p_mov.w         (a0)+,d1

        sloop:
8                       p_mov.w         d1,TOCU          ; Data in PE0 goes to CU
8                       p_mov.w         d1,DTRIN
NetD                    NetworkDelay(0)
6                       c_mov.w         FROMPE0.w,d1
12                      Broadcast(d1,(a1)+)     ; Send data from PE0 to all PEs
```

Figure A.10 (Continued)

Jul 17 10:23  1984 shuffle.s Page 2

```
8                           p_mov.w       DTROUT,d1

5/7                         c_dbf         d2,sloop
5/7                         c_dbf         d3,sloop2

2                           c_movq #r-2,d2
4                           c_asr.w #1,d2             ; FOR i = r/2 - 1 TO 0 STEP -1
          paddit:
4                           p_mov.w       d6,(a1)+        ; Pad with infinites
5/7                         c_dbf         d2,paddit

          shuffleend:
8                           c_rts
```

Figure A.10 (Continued)

```
        ;                Program Name:     dtw
        ;                Algorithm:        Figure 4.8
        ;                Machine:          SIMD, simulated by a MC68000
        ;                Function:         This program does a dynamic time warp
        ;                                  on the input data
        ;                Precision:        Input:  16-bit signed
        ;                                  Output: 16-bit signed
        ;                Number of PEs:    1
        ;                Transfers:        None
        ;                Masking:          Data Conditional
        ;                Parameters:       r, the width of the warping path.
        ;                                  p, the number of coefficients per frame.
        ;                                  NetD, the network delay time.
        ;                                  I, the number of fame per utterance.
        ;                Input:            PE 0 holds all the input data.
        ;                Output:           PE 0 holds the distance score.
        ;                Cycles:           See text
        ;                Typical Time:     74 ms for p=8, r=6, and I=40
        ;                Register Usage: (* means register is set by the calling routine)
        ;                        d0     pe                used by macros
        ;                        d1     cu     x          Index into known template
        ;                        d2     cu     y          Index into unknown template
        ;                        d3     pe                local distance
        ;                        d3     cu     j
        ;                        d6     pe     inf        Infinity
        ;                        d7     pe     WHOAMI     (physical pe address)
        ;                        a1     pe     points to known template(x)
        ;                        a2*    pe     points to unknown template(y)
        ;                        a3*    pe     points to local distances
        ;                        a4*    pe     points to accumalted distances

        N                =       1
        logN             =       0

        #include  "simd.h"
        #include  "defs.h"

        ;                Data allocation for routine
                         .globl  lib1
                         .globl  dtw

                         .c_text
        dtw:
4                        p_mov.w        #inf,d6 ; d6 = infinity

        ;
        ; 2              FOR y := 0 TO I-1
        ;
```

Figure A.11 Serial program for implementing the serial-parallel DTW algorithm.

```
2                          c_clr.w  d2              ; For y := 0 to I-1
          ;
          ; 3              FOR x := -r TO r
          ;
2                          c_clr.w  d1              ; For x := -r to r (x = 0 for first pass)

4                          p_mov.w        #lib1,a1        ; Known template (r must be even)

          ;
          ; 4              IF (y+x ≥ 0) AND (y+x ≥ 2I-2)
          ;
          nextdist:
2                          c_mov.w        d2,d4           ; if(y+x < 0) continue
2                          c_add.wd1,d4             ; d4 = y + x
5/6                        c_blt          nextpair
4                          c_cmp.w        #I,d4           ; if(y+x > 2I-2) continue
5/6                        c_bge   nextpair

          ;
          ; 9              FOR i := 0 TO p-1
          ;
2                          c_movq #p-1,d3           ; Sum d1 over all PEs

          ;
          ; 8              sum := 0;
          ;
2                          p_clr    d3

          ;
          ; 10             sum := sum + (known[x][i] - unknown[y][i]) ²;
          ;
          takediff:
4                          p_mov.w        (a2)+,d1        ; d1 = unknown frame
4                          p_sub.w(a1)+,d1          ; d1 = unknown - known
35                         p_muls  d1,d1            ; d1 = (unknown - known) ^ 2
2                          p_add.w        d1,d3           ; d3 = sum

5/7                        c_dbf    d3,takediff

2                          c_tst.w  d2              ; if y = 0 jump to firstrow
5/6                        c_beq          firstrow
2                          c_tst.w  d4              ; if y+x = 0 jump to yedge
5/6                        c_beq          yedge
          ;
          ; 26             A := g[x-1][y-2] + 2d[x][y-1];
          ; 32             min := A
          ;
          findA:
6                          p_mov.w        r+r+r+r(a3),d4        ;  d(i,j-1)
4                          p_asl.w #1,d4                        ; 2d(i,j-1)
6                          p_add.w        r+r+r+r(a4),d4        ; g(i-1,j-2) + 2d(i,j-1)
```

Figure A.11 (Continued)

```
            ;
            ; 27            B := g[x-2][y-1] + 2d[x-1][y];
            ;
            findB:
4                           p_mov.w       (a3),d5  ;  d(i-1,j)
4                           p_asl.w #1,d5          ; 2d(i-1,j)
4                           p_add.w       (a4),d5  ; g(i-2,j-1) + 2d(i,j-1)


            ;
            ; 33            WHERE B < A
            ; 34            min := B;
            ; 35            ENDHWERE
            ;
26                          Where(d5,LS,d4)
2                                   p_mov.w        d5,d4
8                           EndWhere
            ;
            ; 28            C := g[x-1][y-1] + 2d[x][y];
            ;

            findC:
6                           p_mov.w       r+r+r+r+2(a4),d5     ; g(i-1,j-1)
2                           p_add.w       d3,d5                ; g(i-1,j-1) + d(i,j)


            ;
            ; 36            WHERE C < min
            ; 37                    min := C;
            ; 38            ENDWHERE
            ;
26                          Where(d5,LS,d4)
2                                   p_mov.w        d5,d4
8                           EndWhere


            ;
            ; 40            g[x][y] := d[x][y] + min;
            ;
            findG:
2                           p_add.w       d3,d4                ; g <- d(i,j) + min(A,B,C)
            ;
            ; 46            WHERE g[x][y] ≥ ∞
            ; 47                    g[x][y] := ∞;
            ; 48            ENDWHERE
            ;
26                          Where(d6,LS,d4)
2                                   p_mov.w        d6,d4        ; g <- inf
8                           EndWhere
            ;
            ; 11            d[x][y] := sum;
            ;
6                           p_mov.w       d3,r+r+r+r+2(a3)     ; store in d array (2r)
```

Figure A.11 (Continued)

```
6                   p_mov.w       d4,r+r+r+r+r+r+r+r+6(a4)
                                                     ; store in g array (2(2r+1)-1)
2                   p_addq.w      #2,a3   ; move d pointer
2                   p_addq.w      #2,a4   ; move g pointer

        ;
        ; 2         FOR y := 0 TO I-1   (cont.)
        ;
        nextframe:
2                   c_addq.w      #1,d1   ; x = x + 1
2                   p_subq.w      #p+p,a2        ; Reset y pointer
4                   c_cmp.w       #r,d1
5/6                 c_ble   nextdist

        ;
        ; 3         FOR x := -r TO r    (cont.)
        ;
        newy:
6                   p_sub.w#rp+rp+rp+rp,a1       ; x = x - 2rp
        ;                                   (times 2 for word addressing)
2                   p_addq.w      #p+p,a2               ; y = y + 1
4                   c_mov.w       #-r,d1          ; x = -r
2                   c_addq.w      #1,d2           ; y = y + 1
2                   p_addq.w      #2,a3           ; move d pointer (Skip over inf value)
2                   p_addq.w      #2,a4           ; move g pointer
6                   p_mov.w       d6,r+r+r+r(a4)        ; g(i-1,j-2) <- inf
4                   c_cmp.w       #I,d2
5/6                 c_bne         nextdist

        ;
        ; 50        RETURN g[I][I];
        ;
2                   p_mov.w       d5,d0

        dtwend:
8                   c_rts

        nextpair:
2                   p_addq.w      #p+p,a1         ; move input data pointer
2                   p_addq.w      #p+p,a2         ; move unknown data pointer
2                   p_addq.w      #2,a3   ; move d pointer
2                   p_addq.w      #2,a4   ; move g pointer
5                   c_bra.s       nextframe

        ;
        ; 20        ELSE IF X = 0          /* Check bottom edge          */
        ; 21        min := 2 * d[x-1][y];
        ;
        firstrow:
2                   c_tst.w d1                   ; if x = 0 jump to first column
```

Figure A.11 (Continued)

```
5/6                     c_beq           firstcol
6                       p_mov.w         r+r+r+r(a3),d4          ;  d(i,j-1)
4                       p_asl.w #1,d4                      ; 2d(i,j-1)
5                       c_bra           findG

        ;
        ; 15            IF Y = 0 AND X=0
        ; 16            g[x][y] := 2 * d[x][y];
        ;
        firstcol:
6                       p_mov.w         d3,r+r+r+r+2(a3)      ; store in d array (2r)
4                       p_asl.w #1,d3                     ; store in g array (2(2r+1)-1)
6                       p_mov.w         d3,r+r+r+r+r+r+r+r+6(a4)
2                       p_addq.w        #2,a3                 ; move d pointer
2                       p_addq.w        #2,a4                 ; move g pointer
5                       c_bra           nextframe

        ;
        ; 18            IF Y = 0        /* Check left edge      */
        ; 19            min := 2 * d[x][y-1];
        ;
        yedge:
4                       p_mov.w         (a3),d4
4                       p_asl.w #1,d4             ; 2d(i-1,j)
5                       c_bra   findG
```

Figure A.11 (Continued) -

```
Jul 17 08:51 1984 main.s Page 1

        ;               Program Name:   main (PP2)
        ;               Algorithm:      Figure 6.18
        ;               Machine:        SIMD, simulated by a MC68000
        ;               Function:       This program calls the dynamic time
        ;                               warp routine.
        ;               Precision:      Input:  16-bit signed
        ;                               Output: 16-bit signed
        ;               Number of PEs:  2r+1
        ;               Transfers:      Shift(±1)
        ;               Masking:        Data Conditional
        ;               Parameters:     r, the width of the warping path.
        ;                               p, the number of coefficients per frame.
        ;                               NetD, the network delay time.
        ;                               I, the number of frames per utterance.
        ;               Input:          All PEs hold all the input data.
        ;               Output:         PE r holds the distance score.
        ;               Cycles:         See text
        ;               Register usage:
        ;                       d5      pe      LADDR     (logical address)
        ;                       d7      pe      WHOAMI    (physical address)
        ;                       a0      pe      Pointer to start of Input data
        ;                       a1      pe      Pointer to start of Unknown data
        ;                       a2      pe0     Pointer to where to store results.
        ;                                       a2++ after stored

                        .p_text
        instr:          .=.+ 10         ; Space for PE instructions

                        .globl  lib1
                        .p_data

        WHOAMI:         .word   0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

                        .word   inf,inf,inf,inf ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                        .word   inf,inf,inf,inf ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                        .word   inf,inf,inf,inf ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
        lib1:
                        .word   2,3,4,5 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                        .word   6,7,8,1 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                        .word   8,7,6,5 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                        .word   4,3,2,1 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                        .word   0,0,0,0 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                        .word   0,0,0,0 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                        .word   0,0,0,0 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                        .word   0,0,0,0 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
```

Figure A.12 Parallel program for DTWing (PP2).

Jul 17 08:51 1984 main.s Page 2

```
lib1end:
                .word   inf,inf,inf,inf ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   inf,inf,inf,inf ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   inf,inf,inf,inf ,0,0,0,0 ,0,0,0,0 ,0,0,0,0

unknown:
                .word   1,2,3,4 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   5,6,7,8 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   8,7,6,5 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   4,3,2,1 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   0,0,0,0 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   0,0,0,0 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   0,0,0,0 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   0,0,0,0 ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
unknownend:
                .word   inf,inf,inf,inf ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   inf,inf,inf,inf ,0,0,0,0 ,0,0,0,0 ,0,0,0,0
                .word   inf,inf,inf,inf ,0,0,0,0 ,0,0,0,0 ,0,0,0,0

                .p_bss
dist:           .=.+    10      ; Local distance scores
scores:         .=.+    10      ; Global Distance scores

                .globl  d
                .globl  dold
                .globl  dDTR
                .globl  dup
                .globl  ddown
                .globl  g
                .globl  gold
                .globl  gDTR
                .globl  gup
                .globl  gdown

d:              .=.+    2       ; Local distance
g:              .=.+    2       ; Optimal subpath distance
dold:           .=.+    2       ; Old local distance
gold:           .=.+    2       ; Old optimal subpath distance
dDTR:           .=.+    2       ; d values to transfer
gDTR:           .=.+    2       ; g values to transfer
dup:            .=.+    2       ; d values to transfer to PE with next higher addr.
gup:            .=.+    2       ; g values to transfer to PE with next higher addr.
ddown:          .=.+    2       ; d values to transfer to PE with next lower addr.
gdown:          .=.+    2       ; g values to transfer to PE with next lower addr.


                .c_text
main:
```

Figure A.12 (Continued)

Jul 17 08:51 1984 main.s Page 3

```
4                       c_mov.w       #STACK,a7                       ; set up stack pointer
8                       c_mov.w       #Initialize,MASKCTL.w           ; Initialize CMS
8                       c_mov.w       #0x8000,MASKCTL.w

10                      p_mov.w       #15,DTRDEST           ; Set all transfers to PE15
6                       p_mov.w       WHOAMI.w,d7           ; d7 always has WHOAMI in it
                                                    (Physical address)
2                       p_mov.w       d7,d5         ; Store logical address in d5
2                       p_subq.w      #r,d5

4                       p_mov.w       #unknown,a2
4                       p_mov.w       #dist,a3
10                      c_jsr         distance; Find distance between input and unknown

        dtwstart:
4                       p_mov.w       #dist,a0
4                       p_mov.w       #scores,a1
10                      c_jsr         dtw

        mainend:
2                       c_stop        #0000
```

Figure A.12 (Continued)

Jul 17 09:09 1984 dtw.s Page 1

```
;           Program Name:     dtw (PP2)
;           Algorithm:        Figure 6.18??
;           Machine:          SIMD, simulated by a MC68000
;           Function:         This program does a dynamic time warp
;                             on the input data.  The local distances
;                             have already been computed before this
;                             routine is called.
;           Precision:        Input:  16-bit signed
;                             Output: 16-bit signed
;           Number of PEs:    2r+1
;           Transfers:        Cube, Shift
;           Masking:          Data Conditional
;           Parameters:       r, the width of the warping path.
;                             p, the number of coefficients per frame.
;                             NetD, the network delay time.
;                             I, the number of frames per utterance.
;           Input:            All PEs hold all the input data.
;           Output:           PE r holds the distance score.
;           Cycles:           See text
;           Typical Time:     5.6 ms for r=6, p=8, NetD=18, and I=40.
;           Register usage:   (* means passed as parameter)
;                  d2     pe     tmp storage
;                  d5*    pe     LADDR     (logical address)
;                  d6     pe     inf (infinity)
;                  d7*    pe     WHOAMI  (physical address)
;                  d7     cu     k
;                  a0*    pe     Pointer to local distances
;                  a1*    pe0    Pointer to where to store results.
;                                a1++ after stored

#include "defs.h"
#include "simd.h"

;           Data allocation for routine
            .p_bss
            .globl  d
            .globl  dold
            .globl  dDTR
            .globl  dup
            .globl  ddown
            .globl  g
            .globl  gold
            .globl  gDTR
            .globl  gup
            .globl  gdown

            .c_text
            .globl  dtw
```

Figure A.12 (Continued)

Jul 17 09:09 1984 dtw.s Page 2

```
        dtw:
4                       p_mov.w         #inf,d6 ; Store infinity in d6

        ;
        ; 2             g ← 0
        ; 3             gold ← 0
        ; 4             d ← ∞
        ; 5             dold ← ∞
        ;
        findstart:
8                       p_clr.w g.w             ; g <- 0
8                       p_clr.w gold.w          ; gold <- 0
6                       p_mov.w         d6,d.w          ; d <- inf
6                       p_mov.w         d6,dold.w       ; dold <- inf

        ;
        ; 6             WHERE ADDR = 0 DO
        ; 7                     g ← 0
        ; 8             ENDWHERE
        ;
28                      Where(d5,EQ,#0)
8                               p_clr.w g.w     ; g <- 0
8                       EndWhere

        ;
        ; 13            FOR k ← 1 TO I DO
        ;
4                       c_mov.w         #I-1,d7         ; FOR k <- I-1 TO 0 STEP -1 DO

        forl:
        ;
        ; 14            compute d(Xindex,Yindex)
        ;
8                       p_mov.w         (a0)+,d.w

        ;
        ; 15            WHERE ADDR is even DO
        ; 16                    dDTR ← dold
        ; 17                    gDTR ← gold
        ;
2                       p_mov.w         d5,d0           ; WHERE       LADDR is even DO
4                       p_and.w         #1,d0
36                      WhereElse(d0,EQ,#0)
10                              p_mov.w         d.w,dDTR.w
10                              p_mov.w         g.w,gDTR.w
        ;
        ; 18            ELSEWHERE
        ; 19                    dDTR ← d
```

Figure A.12 (Continued)

Jul 17 09:09 1984 dtw.s Page 3

```
        ; 20                gDTR ← g
        ; 21            ENDWHERE
        ;
8                       ElseWhere       ; Elsewhere
10                              p_mov.w         dold.w,dDTR.w
10                              p_mov.w         gold.w,gDTR.w
8                       EndWhere

        ;
        ; 23            USE Shift +1
        ; 24            TRANSFER dDTR TO dup
        ; 25            TRANSFER gDTR TO gup
        ;
        movedataup:
12                      Shift(#1)                               ; TRANSFER dDTR TO dup
10                      p_mov.w         dDTR.w,DTRIN.w
NetD                    NetworkDelay(0)
10                      p_mov.w         DTROUT.w,dup.w

10                      p_mov.w         gDTR.w,DTRIN.w          ; TRANSFER gDTR TO gup
NetD                    NetworkDelay(0)
10                      p_mov.w         DTROUT.w,gup.w

        ;
        ; 27            USE Shift -1
        ; 28            TRANSFER dDTR TO ddown
        ; 29            TRANSFER gDTR TO gdown
        ;
        movedatadown:
12                      Shift(#-1)                              ; TRANSFER dDTR TO ddown
10                      p_mov.w         dDTR.w,DTRIN.w
NetD                    NetworkDelay(0)
10                      p_mov.w         DTROUT.w,ddown.w

10                      p_mov.w         gDTR.w,DTRIN.w          ; TRANSFER gDTR TO gdown
NetD                    NetworkDelay(0)
10                      p_mov.w         DTROUT.w,gdown.w

        ;
        ; 31            WHERE ADDR = r DO
        ; 32                    gdown ← ∞
        ; 33            ENDWHERE
        ;
        fixends:
28                      Where(d5,EQ,#r)         ; WHERE LADDR = +r DO
6                               p_mov.w         d6,gdown.w      ; gdown <- inf
8                       EndWhere
```

Figure A.12 (Continued)

```
Jul 17 09:09 1984 dtw.s Page 4

        ;
        ; 35          WHERE ADDR = −r DO
        ; 36                  gup ← ∞
        ; 37          ENDWHERE
        ;
28                    Where(d5,EQ,#−r)                    ; WHERE LADDR = −r DO
6                             p_mov.w      d6,gup.w       ; gup <- inf
8                     EndWhere


        ;
        ; 39          gold ← g
        ; 40          dold ← d
        ;
10                    p_mov.w      g.w,gold.w      ; gold <- g
10                    p_mov.w      d.w,dold.w      ; dold <- d


        ;
        ; 42          A ← gdown  + 2 * ddown
        ;
        findA:
6                     p_mov.w      ddown.w,d0      ; A <- gdown + 2 * ddown
4                     p_asl.w #1,d0  ; 2*ddown
6                     p_add.w      gdown.w,d0


        ;
        ; 43          B ← gold  + d
        ;
        findB:
6                     p_mov.w      d.w,d1  ; B <- gold + d
6                     p_add.w      gold.w,d1


        ;
        ; 44          C ← gup   + 2 * dup
        ;
        findC:
6                     p_mov.w      dup.w,d2        ; C <- gup + 2 * dup
4                     p_asl.w #1,d2
6                     p_add.w      gup.w,d2


        ;
        ; 46          WHERE B < A DO
        ; 47                  A ← B
        ; 48          ENDWHERE
        ;
        findmin:
26                    Where(d1,LS,d0)              ; g <- min(A,B,C) + d
2                             p_mov.w      d1,d0
8                     EndWhere
```

Figure A.12 (Continued)

Jul 17 09:09 1984 dtw.s Page 5

```
        ;
        ; 49            WHERE C < A DO
        ; 50                A ← C
        ; 51            ENDWHERE
        ;
26                      Where(d2,LS,d0)
2                           p_mov.w       d2,d0
8                       EndWhere

        ;
        ; 52            g ← A +d
        ;
6                       p_add.w       d.w,d0        ; d0 now holds min(d0,d1,d2)
6                       p_mov.w       d0,g.w        ; g <- min(A,B,C)
26                      Where(d6,HI,d0)             ; where(d0 < inf)
6                           p_mov.w       d6,g.w    ; g <- inf
8                       EndWhere

        incindex:
5/7                     c_dbf         d7,for1 ; FOR d7 <- I/2-2 TO -1 STEP -1

        ;
        ; 57            WHERE ADDR = 0 DO
        ; 58                D(A,B) ← g/(I+J)
        ; 59            ENDWHERE
        ;
28                      Where(d5,EQ,#0)
8                           p_mov.w       g.w,(a1)+
8                       EndWhere

        dtwend:
8                       c_rts
```

Figure A.12 (Continued)

```
Jul 17 09:10 1984 distance.s Page 1

            ;         Program Name:   distance (PP2)
            ;         Algorithm:      Figure 6.18??
            ;         Machine:        SIMD, simulated by a MC68000
            ;         Function:       This program computes the local distances
            ;                         for the DTW program.
            ;         Precision:      Input:  16-bit signed
            ;                         Output: 16-bit signed
            ;         Number of PEs:  2r+1
            ;         Parameters:     r, the width of the warping path.
            ;                         p, the number of coefficients per frame.
            ;                         NetD, the network delay time.
            ;                         I, the number of frames per utterance.
            ;         Input:          PE i contains coefficient i of frame j.
            ;         Output:         PE i-j+r contains the local distance
            ;                         between known frame i and unknown frame j.
            ;         Cycles:         See text.
            ;         Typical Time:   35 ms for r=6, p=8, NetD=18, and I=40.
            ;         Register usage  (* means set by calling routine)
            ;                d0    pe               used by macros
            ;                d1    cu    x          Index into known template
            ;                d2    pe    x          Index into known template + r
            ;                d2    cu    y          Index into unknown template
            ;                d3    cu    j
            ;                d6    pe    inf        Infinity
            ;                d7    pe    WHOAMI     (physical pe address)
            ;                a1    pe    points to known template(x)
            ;                a2*   pe    points to unknown template(y)
            ;                a3*   pe    points to local distances

            ;         Data allocation for routine
                      .globl   lib1
                      .globl   distance

                      .c_text
            distance:
4                     p_mov.w       #inf,d6 ; d6 = infinity

            ;
            ; 3       FOR i ← 1 TO r/2
            ;
2                     p_movq #1,d3          ; FOR i := 1 to r-1 step 2
2                     c_movq #r-2,d3
4                     c_asr.w #1,d3         ; d3 = r/2 - 1

            ;
            ; 4       WHERE |LADDR| > i DO
            ; 5             d[dptr] ← ∞;
            ; 6             dptr ← dprt + 1;
```

Figure A.12 (Continued)

Jul 17 09:10 1984 distance.s Page 2

```
        ; 7            ENDWHERE
        ;
        pad:
26                     Where(d5,GT,d3)                      ; where(LADDR > i)
4                             p_mov.w        d6,(a3)+       ; (a3) <- inf
8                      EndWhere
2                      p_neg.wd3
26                     Where(d5,LT,d3)                      ; where(LADDR < -i)
4                             p_mov.w        d6,(a3)+       ; (a3) <- inf
8                      EndWhere
2                      p_addq.w      #1,d3                  ; i:= i + 1
5/7                    c_dbf         d3,pad
        ;
        ; 9            FOR y ← 0 TO I-1
        ;
2                      c_clr.w  d2              ; For y := 0 to I-1
        ;
        ; 10           FOR x ← -r TO r
        ;
2                      c_clr.w  d1              ; For x := -r to r (x = 0 for first pass)
4                      p_mov.w       #r,d2            ; d2 in PE = d1+r in CU

4                      p_mov.w       #lib1,a1         ; Known template (r must be even)

        ;
        ; 11           IF y+x ≤ O AND y+x ≤ 2I-2
        ;
        nextdist:
2                      c_mov.w       d2,d4            ; if(y+x < 0) continue
2                      c_add.wd1,d4             ; d4 = y + x
5/6                    c_blt         nextpair
4                      c_cmp.w       #I,d4            ; if(y+x > 2I-2) continue
5/6                    c_bge         nextpair

        ;
        ; 12           sum ← (known[x] - unknown[y]) /u2/d;
        ;
        takediff:
4                      p_mov.w       (a2),d1  ; d1 = unknown frame
4                      p_sub.w(a1)+,d1        ; d1 = unknown - known
35                     p_muls d1,d1           ; d1 = (unknown - known) ^ 2

        ;
        ; 13           FOR k ← 0 TO logN-1
        ;
        notinf:
2                      c_movq #logN-1,d3      ; Sum d1 over all PEs
2                      p_clr         d3
```

Figure A.12 (Continued)

```
Jul 17 09:10 1984 distance.s Page 3

        ;
        ; 14            USE Cube(k);
        ;
        dloop:
12                      Cube(d3)

        ;
        ; 15            DTRIN ← sum;
        ; 16            TRANSFER;
        ; 17            sum ← sum +DTROUT;
        ;
6                       p_mov.w         d1,DTRIN.w
NetD                    NetworkDelay(0)
6                       p_add.w         DTROUT.w,d1

2                       p_addq.w        #1,d3
5/7                     c_dbf           d3,dloop


        ;
        ; 25            IF x+r > p
        ;
        gotd:
4                       c_cmp.w         #N-r,d1
5/4                     c_blt.s         easy      ; If destination PE is >= N,


        ;
        ; 26            USE Shift +x+r
        ;
12                      Shift(d2)         ; data isn't in desired PE, so shift
        ;
        ; 27            TRANSFER sum
        ;
8                       p_mov.w         d1,DTRIN        ; it there from PE0.
NetD                    NetworkDelay(0)
8                       p_mov.w         DTROUT,d1


        ;
        ; 29            WHERE x+r = ADDR                  /* Enable PE that will use the*/
        ; 30                    d[dptr] ← sum;          /* distance score       */
        ; 31                    dtpr ← dptr + 1;
        ; 32            ENDWHERE
        ;
        easy:
26                      Where(d2,EQ,d7)                 ; where(x == ADDR)
4                               p_mov.w         d1,(a3)+
8                       EndWhere

        nextframe:
2                       c_addq.w        #1,d1           ; x = x + 1
```

Figure A.12 (Continued)

```
Jul 17 09:10 1984 distance.s Page 4

2                   p_addq.w     #1,d2        ; x = x + 1
4                   c_cmp.w      #r,d1
5/6                 c_ble        nextdist

     newy:
6                   p_sub.w#r+r+r+r,a1 ; x = x - 2r
     ;                                        (times 2 for word addressing)
6                   p_add.w      #2,a2        ; y = y + 1
4                   c_mov.w      #-r,d1       ; x = -r
2                   p_clr.w d2         ; x = 0 in pe
2                   c_addq.w     #1,d2        ; y = y + 1
4                   c_cmp.w      #1,d2
5/6                 c_bne        nextdist

     ;              Stick infinities on the end of each list

     ;
     ; 34           FOR i ← 1 to r/2
     ;
2                   c_movq #r-2,d3
4                   c_asr.w #1,d3  ; d3 = r/2 - 1
     pad2:
     ;
     ; 35           d[dptr] ← ∞;
     ; 36           dptr ← dptr + 1;
     ;
4                   p_mov.w      d6,(a3)+     ; (a3) <- inf
5/7                 c_dbf   d3,pad2

     distanceend:
8                   c_rts

     nextpair:
2                   p_addq.w     #2,a1
5                   c_bra.s nextframe
```

Figure A.12 (Continued)

```
;               Program Name:   main
;               Algorithm:      Figure 7.8
;               Machine:        SIMD, simulated by a MC68000
;               Function:       This is the main routine.  It calls filter()
;                               and auto() to preemphize the signal and find
;                               the autocoerrelation coefficients.  If R(0)
;                               (the energy) is greater than lothresh, it calls
;                               lpc().  This main routine also does the
;                               endpoint detection.  After an utterance is
;                               detected, ltw() and dtw() are called.
;               Precision:      Input:  16-bit signed
;                               Output: 16-bit signed
;               Number of PEs:  100
;               Parameters:     N, the frame size.
;                               autocoef, the number of autocorrelation coefs.
;                               r, the width of the warping path.
;                               p, the number of LPC coefficients.
;                               NetD, the network delay time.
;                               I, the number of frames per utterance.
;                               VOCABSIZE, the size of the vocabulary.
;               Input:          Sample i mod N is is PE i.
;               Output:         One distance score per PE.
;               Cycles:         See text.
;               Typical Time:   See text.
;               Register usage:
;                       d0      cu      M (number of frames in word)
;                       d7      pe      WHOAMI (physical address)
;                       d7      cu      i
;                       a0      pe      pointer to input data
;                       a1      pe      pointer to output data
;
;               The data is stored as follows:
;               Routine Number  Data Storage
;                       of PEs  Input                           Output
;               filter  100     1 sample/PE                     1 sample/PE
;               auto    100     1 sample/PE                     Each PE has all coefs.
;               lpc     8       Each PE has all coefs           PE i has lpc coef i
;               ltw     8       PE i has coef i from each frame. SAME
;               shuffle 100     "                               Each PE has all coefs.
;               dtw     100     Each PE has all coefs.

#include "simd.h"
#include "defs.h"

                .p_text
inst:           .=.+ 10         ; Space where pe instructions are broadcast to

                .globl  WHOAMI
```

Figure A.13 SIMD program for isolated speech recognition system. Contains endpoint routine.

```
                      .p_data
        ;             Physical addresses (stored in d7)
        WHOAMI:       .word   0,1,2,3,4,5,6,7, 8,9,10,11,12,13,14,15
        ;             Input signal
        input:        .word   1,2,3,4,5,6,7,8, 9,10,11,12,13,14,15,16
                      .word   11,12,13,14,15,16,17,8, \
                              9,10,11,12,13,14,15,16
                      .word   1,-2,-3,-4,-5,-6,-7,-8, \
                              -9,-10,-11,-12,-13,-14,-15,-16
                      .word   0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0
                      .word   1,1,1,1,1,1,1,1 ,1,1,1,1,1,1,1,1

        lib:          .word   1,2,3,4,5,6,7,8, 9,10,11,12,13,14,15,16
                      .word   11,12,13,14,15,16,17,8, \
                              9,10,11,12,13,14,15,16
                      .word   1,\-2,-3,-4,-5,-6,-7,-8, \
                              -9,-10,-11,-12,-13,-14,-15,-16
                      .word   0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0
                      .word   1,1,1,1,1,1,1,1, 1,1,1,1,1,1,1,1

                      .p_bss
        sig:          .=.+    2                     ; Filtered signal
        R:            .=.+    autocoef+autocoef ; autocorrelation coefficients
        lpcout:       .=.+    MAXFRAMES             ; autocorrelation coefficients
        Tout:         .=.+    MAXFRAMES             ; output of ltw
        fixed:        .=.+    MAXFRAMES             ; output of ltw arranged for dtw

                      .c_bss
        found:        .=.+    2        ; ==1 if R(0) > hithresh
        libcount:     .=.+    2        ; number of utterances compared

                      .c_text
        main:
8                     c_mov.w      #Initialize,MASKCTL.w       ; Initialize CMS
8                     c_mov.w      #0x8000,MASKCTL.w           ; stop mask unit

4                     p_mov.w      #STACK,a7     ; set up PE stack pointer
4                     c_mov.w      #STACK,a7     ; set up CU stack pointer

6                     p_mov.w      WHOAMI.w,d7 ; d7 aways has WHOAMI in it

        ;
        ; 5           WHILE(TRUE)
        ;
        newword:
4                     p_mov.w      #input,a0     ; Pointer to input data
4                     p_mov.w      #lpcout,a1    ; Pointer to output lpc coefs.
        ;
        ; 2           found ← FALSE;
        ; 3           M ← 0;
```

Figure A.13 (Continued)

```
            ;
8                          c_clr          found.w; found := false;
2                          c_clr          d0              ; M = 0

            nextframe:
4                          p_mov.w a0,-(a7)       ; push a0 on stack
4                          p_mov.w a1,-(a7)       ; push a1 on stack
4                          c_mov.w d0,-(a7)       ; push d0 on stack

            ;
            ; 9            filter(input[i], filout);
            ;
            filterdo:
4                          p_mov.w        #sig,a1 ; a0 points to input data
10                         c_jsr          filter

            ;
            ; 13           auto(filout,R[]);
            ;
            autodo:
4                          p_mov.w        #sig,a0 ; a0 points to input data
4                          p_mov.w        #R,a1
10                         c_jsr          auto

            autodone:
4                          p_mov.w        (a7)+,a1        ; pull a1 off stack
4                          p_mov.w        (a7)+,a0        ; pull a0 off stack
4                          c_mov.w        (a7)+,d0        ; pull d0 off stack
6                          p_add.w        #2,a0           ; Point to next input data sample

            ;
            ; 18           TOCU ← R[0];
            ; 19           energy ← FROMPE0;
            ;
14                         p_mov.w        R,TOCU          ; Get R(0) from PE0 and compare it
NetD                       NetworkDelay(0)                ; to lothresh and hithresh to see if
8                          c_mov.w FROMPE0,d1 ; a word is present.

            ;
            ; 25           IF energy > lothresh
            ;
            gotit:
4                          c_cmp.w        #lothresh,d1    ; if energy >= lothresh then
5/4                        c_blt.s  isend

            ;
            ; 26           IF energy > hithresh
            ; 27                   found ← TRUE;
            ;
4                          c_cmp          #hithresh,d1              ; if energy >= hithresh then
```

Figure A.13 (Continued)

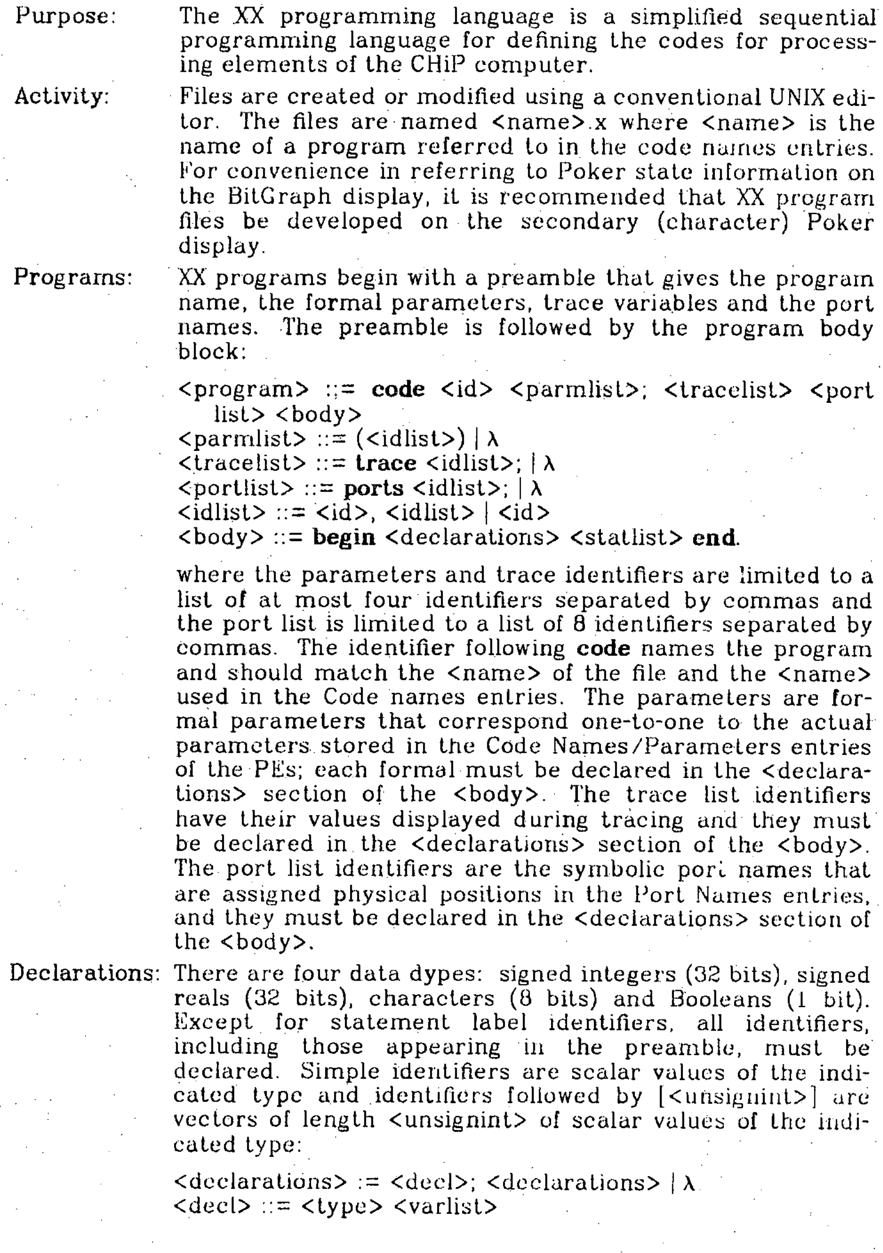
```
5/4                 c_blt.s         lab1
8                   c_mov  #1,found.w      ; found := true;
        ;
        ; 28        lpc(R[], lpcout[M]);
        ;
        lab1:
8                   p_mov.w         #15,DTRDEST.w       ; So all PEs will transfer to PE15
4                   p_mov.w a0,-(a7)              ; push a0 on stack
4                   p_mov.w a1,-(a7)              ; push a1 on stack
4                   c_mov.w d0,-(a7)              ; push d0 on stack

        lpcdo:
4                   p_mov.w         #R,a1  ; The energy is greater than lothresh
10                  c_jsr           lpc    ; find the lpc coefficients
                                           ; Return lpc coef., in D1, one per PE

        lpcdone:
4                   p_mov.w         (a7)+,a1        ; pull a1 off stack
4                   p_mov.w         (a7)+,a0        ; pull a0 off stack
4                   c_mov.w         (a7)+,d0        ; pull d0 off stack

4                   p_mov.w         d1,(a1)+        ; Save lpc coef., one per PE

        ;
        ; 29        M ← M + 1;
        ;
2                   c_addq.w        #1,d0  ; Add one to frame count
5                   c_bra           nextframe       ; Get next frame

        ;
        ; 34        ELSE
        ; 35                IF found
        ;
        isend:
6                   c_tst.w found.w; if found then
5/6                 c_beq           newword

        ;
        ; 40        ltw(lpcout[],ltwout[],M,40);
        ;
        ltwdo:
4                   c_mov.w         #I,d1               ; N (M is in d0)
6                   p_mov.l #lpcout,a5      ; Address of R
6                   p_mov.l #Tout,a6        ; Address of Tout
10                  c_jsr           ltw

        ;
        ; 45        shuffle(ltwout[],shuffout[]);
        ;
        ltwdone:
```

Figure A.13 (Continued)

```
4                    p_mov.w      #Tout,a0       ; Take data in the order put out
4                    p_mov.w      #fixed,a1      ; by ltw and arrange it for the DTW
10                   c_jsr        shuffle        ; program.

4                    p_mov.w      #lib,a0 ; a0 points to known utterance in library
       ;
       ; 46          FOR j ← 0 TO VOCABSIZE−1
       ;
8                    c_clr.w libcount
       ;
       ; 47          score[j] ← dtw(shuffout[],lib[j][]);
       ;
       dtwdo:
4                    p_mov.w      #fixed,a1      ; a1 points to the unknown utterance.
4                    p_mov.w      #scores,a2     ; a2 is the scores from the DTW matches.
10                   c_jsr        dtw

4                    c_add.w#1,libcount
4                    c_cmp.w      #VOCABSIZE,libcount
5/4                  c_bne.s dtwdo

       dtwdone:
5                    c_bra        newword

       mainend:
2                    c_stop       #0000
```

Figure A.13 (Continued)

# APPENDIX B: VLSI Processor Array Assembly Language Programs



Purpose: The XX programming language is a simplified sequential programming language for defining the codes for processing elements of the CHiP computer.

Activity: Files are created or modified using a conventional UNIX editor. The files are named <name>.x where <name> is the name of a program referred to in the code names entries. For convenience in referring to Poker state information on the BitGraph display, it is recommended that XX program files be developed on the secondary (character) Poker display.

Programs: XX programs begin with a preamble that gives the program name, the formal parameters, trace variables and the port names. The preamble is followed by the program body block:

<program> ::= **code** <id> <parmlist>; <tracelist> <port list> <body>
<parmlist> ::= (<idlist>) | λ
<tracelist> ::= **trace** <idlist>; | λ
<portlist> ::= **ports** <idlist>; | λ
<idlist> ::= <id>, <idlist> | <id>
<body> ::= **begin** <declarations> <statlist> **end**.

where the parameters and trace identifiers are limited to a list of at most four identifiers separated by commas and the port list is limited to a list of 8 identifiers separated by commas. The identifier following **code** names the program and should match the <name> of the file and the <name> used in the Code names entries. The parameters are formal parameters that correspond one-to-one to the actual parameters stored in the Code Names/Parameters entries of the PEs; each formal must be declared in the <declarations> section of the <body>. The trace list identifiers have their values displayed during tracing and they must be declared in the <declarations> section of the <body>. The port list identifiers are the symbolic port names that are assigned physical positions in the Port Names entries, and they must be declared in the <declarations> section of the <body>.

Declarations: There are four data dypes: signed integers (32 bits), signed reals (32 bits), characters (8 bits) and Booleans (1 bit). Except for statement label identifiers, all identifiers, including those appearing in the preamble, must be declared. Simple identifiers are scalar values of the indicated type and identifiers followed by [<unsignint>] are vectors of length <unsignint> of scalar values of the indicated type:

<declarations> := <decl>; <declarations> | λ
<decl> ::= <type> <varlist>

Figure B.1 Description of *xx* programming language. (From [Synder82b].)

<type> ::= **real** | **int** | **bool** | **char**
<varlist> ::= <varid>, <varlist> | <varid>
<varid> ::= <id> | <id> [<unsignint>]

where no <id> appears more than once.

Statements: The statements are:

<statlist> ::= <lstatement>; <statlist> | <lstatement>
<lstatement> ::= <id>: <statement> | <statement>
  <statement> ::= <assignment> | <conditional> |
  <while> | <break> | <for> | <compound> | <io>

where <id> is used for tracing rather than the target of **goto**.

Assignment: The Assignment statement

<assignment> ::= <varid> := <expression>

where the coercion to the left-hand side identifier type is provided as described in Table 1.

Conditional: In the Conditional statement

<conditional> ::= **if** <expression> **then** <lstatement>
  **else** <lstatement> | **if** <expression>
  **then** <lstatement>

the <expression> must evaluate to a Boolean value and an **else** is associated with the immediately preceding **then**.

While: In the While statement

<while> ::= **while** <expression> **do** <lstatement>

the expression must evaluate to a Boolean value. To assist in synchronization the compiler recognizes the construction **while true do** <lstatement> as a special case and does not generate the conditional branch code.

Break: The Break statement

<break> ::= **break**

has meaning only within the <lstatement> of a While statement, and causes control to skip to the statement following the immediately surrounding While statement.

For: In the For statement

<for> ::= **for** <id> := <expression> **to** <expression> **do**
  <lstatement>

the two expressions, the lower and upper limits of the iteration, respectively, are evaluated once prior to beginning the loop. If the lower and upper limits are not integers, they are coerced to integers as described in Table 1.

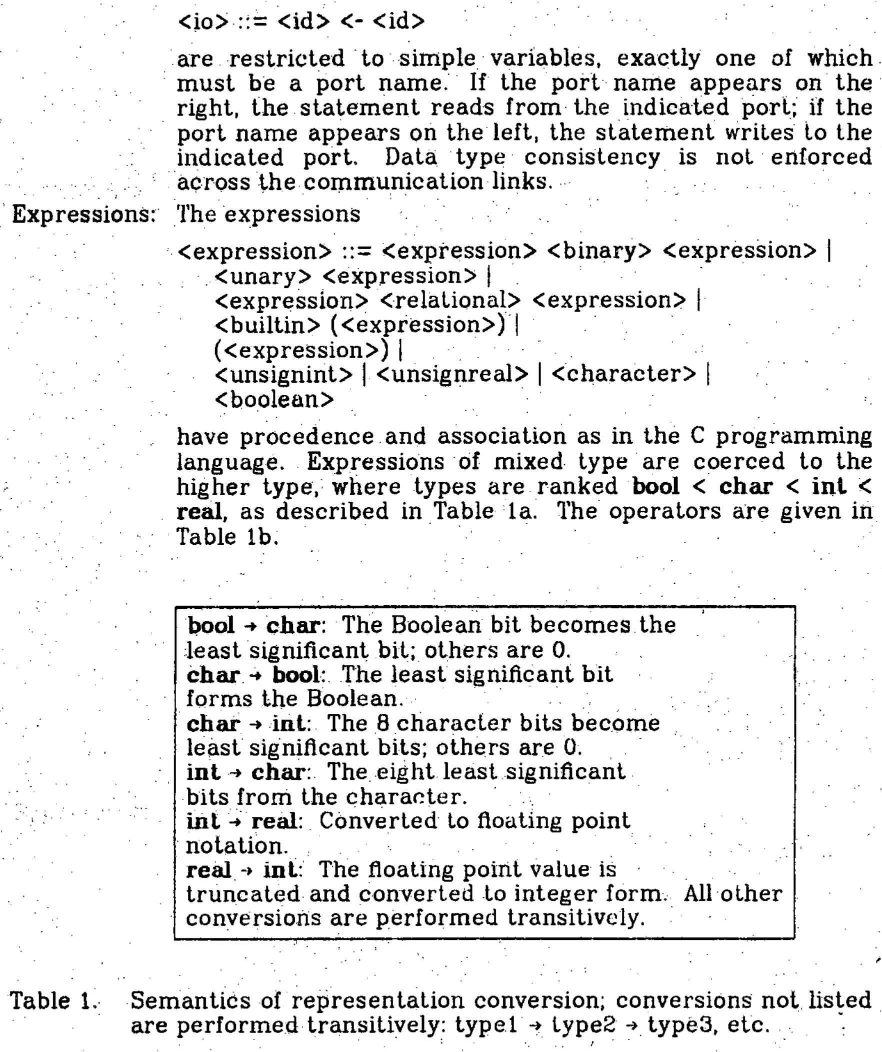
Compound: Notice that the Compound statement

<compound> ::= **begin** <statlist> **end**

is not a block and may not contain declarations.

I/O: The I/O statements

Figure B.1 (Continued)



```
<io> ::= <id> <- <id>
```

are restricted to simple variables, exactly one of which must be a port name. If the port name appears on the right, the statement reads from the indicated port; if the port name appears on the left, the statement writes to the indicated port. Data type consistency is not enforced across the communication links.

Expressions: The expressions

```
<expression> ::= <expression> <binary> <expression> |
   <unary> <expression> |
   <expression> <relational> <expression> |
   <builtin> (<expression>) |
   (<expression>) |
   <unsignint> | <unsignreal> | <character> |
   <boolean>
```

have procedence and association as in the C programming language. Expressions of mixed type are coerced to the higher type, where types are ranked **bool** < **char** < **int** < **real**, as described in Table 1a. The operators are given in Table 1b.

**bool → char**: The Boolean bit becomes the least significant bit; others are 0.
**char → bool**: The least significant bit forms the Boolean.
**char → int**: The 8 character bits become least significant bits; others are 0.
**int → char**: The eight least significant bits from the character.
**int → real**: Converted to floating point notation.
**real → int**: The floating point value is truncated and converted to integer form. All other conversions are performed transitively.

Table 1. Semantics of representation conversion; conversions not listed are performed transitively: type1 → type2 → type3, etc.

Figure B.1 (Continued)

| <unary> | |
|---|---|
| + <real> | no op |
| - <real> | negation |
| ~ <char> | not |

The type indicates the highest type for which the operation is defined; the operation is defined for all lower types.

| <binary> | |
|---|---|
| <real> + <real> | addition |
| <real> - <real> | subtraction |
| <real> * <real> | multiplication |
| <real> / <real> | division |
| <real> **mod** <real> | modulus |
| <real> >= <real> | greater than or equal |
| <real> > <real> | greater than |
| <real> =/ <real> | not equal |
| <real> < <real> | less than |
| <Real> <= <real> | less than or equal |
| <real> = <real> | equal |
| <char> & <char> | and |
| <char> \| <char> | or |
| <char> \|\| <char> | exclusive or |

Table 1b. XX operators.

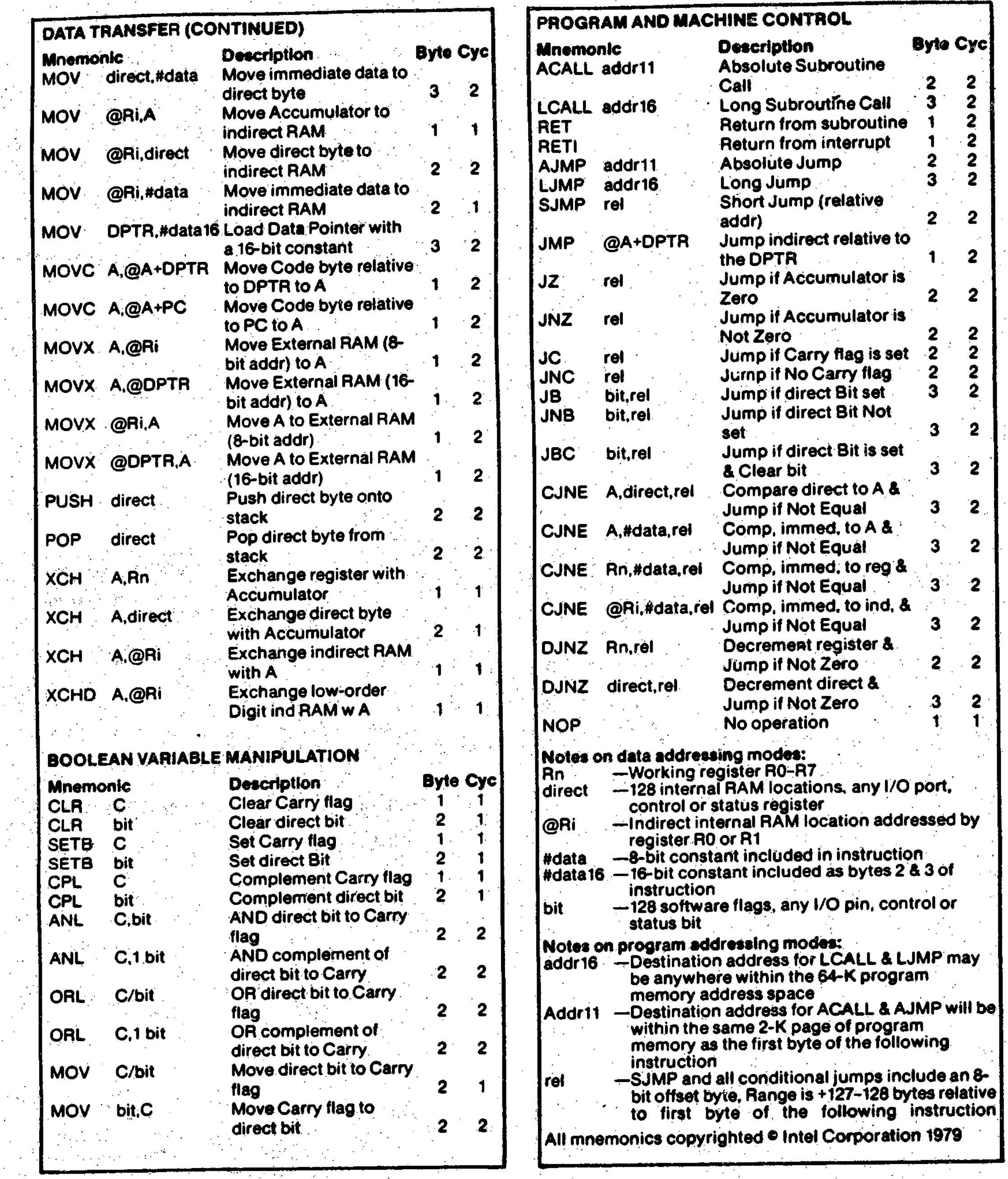
Constants: The constants are unsigned integers and reals in standard formats, quoted (') characters and **true** and **false**.

Identifiers: All identifiers begin with a letter and are followed by any combination of letters and numerals. The maximum length of an identifier is 10 symbols.

Vectors: Vectors can only be subscripted by character or integer types and are referenced using 1 origin.

Built in functions: The built in functions are not yet implemented.

Figure B.1 (Continued)



DATA TRANSFER (CONTINUED)

| Mnemonic | | Description | Byte | Cyc |
|---|---|---|---|---|
| MOV | direct,#data | Move immediate data to direct byte | 3 | 2 |
| MOV | @Ri,A | Move Accumulator to indirect RAM | 1 | 1 |
| MOV | @Ri,direct | Move direct byte to indirect RAM | 2 | 2 |
| MOV | @Ri,#data | Move immediate data to indirect RAM | 2 | 1 |
| MOV | DPTR,#data16 | Load Data Pointer with a 16-bit constant | 3 | 2 |
| MOVC | A,@A+DPTR | Move Code byte relative to DPTR to A | 1 | 2 |
| MOVC | A,@A+PC | Move Code byte relative to PC to A | 1 | 2 |
| MOVX | A,@Ri | Move External RAM (8-bit addr) to A | 1 | 2 |
| MOVX | A,@DPTR | Move External RAM (16-bit addr) to A | 1 | 2 |
| MOVX | @Ri,A | Move A to External RAM (8-bit addr) | 1 | 2 |
| MOVX | @DPTR,A | Move A to External RAM (16-bit addr) | 1 | 2 |
| PUSH | direct | Push direct byte onto stack | 2 | 2 |
| POP | direct | Pop direct byte from stack | 2 | 2 |
| XCH | A,Rn | Exchange register with Accumulator | 1 | 1 |
| XCH | A,direct | Exchange direct byte with Accumulator | 2 | 1 |
| XCH | A,@Ri | Exchange indirect RAM with A | 1 | 1 |
| XCHD | A,@Ri | Exchange low-order Digit ind RAM w A | 1 | 1 |

BOOLEAN VARIABLE MANIPULATION

| Mnemonic | | Description | Byte | Cyc |
|---|---|---|---|---|
| CLR | C | Clear Carry flag | 1 | 1 |
| CLR | bit | Clear direct bit | 2 | 1 |
| SETB | C | Set Carry flag | 1 | 1 |
| SETB | bit | Set direct Bit | 2 | 1 |
| CPL | C | Complement Carry flag | 1 | 1 |
| CPL | bit | Complement direct bit | 2 | 1 |
| ANL | C,bit | AND direct bit to Carry flag | 2 | 2 |
| ANL | C,1 bit | AND complement of direct bit to Carry | 2 | 2 |
| ORL | C/bit | OR direct bit to Carry flag | 2 | 2 |
| ORL | C,1 bit | OR complement of direct bit to Carry | 2 | 2 |
| MOV | C/bit | Move direct bit to Carry flag | 2 | 1 |
| MOV | bit,C | Move Carry flag to direct bit | 2 | 2 |

PROGRAM AND MACHINE CONTROL

| Mnemonic | | Description | Byte | Cyc |
|---|---|---|---|---|
| ACALL | addr11 | Absolute Subroutine Call | 2 | 2 |
| LCALL | addr16 | Long Subroutine Call | 3 | 2 |
| RET | | Return from subroutine | 1 | 2 |
| RETI | | Return from interrupt | 1 | 2 |
| AJMP | addr11 | Absolute Jump | 2 | 2 |
| LJMP | addr16 | Long Jump | 3 | 2 |
| SJMP | rel | Short Jump (relative addr) | 2 | 2 |
| JMP | @A+DPTR | Jump indirect relative to the DPTR | 1 | 2 |
| JZ | rel | Jump if Accumulator is Zero | 2 | 2 |
| JNZ | rel | Jump if Accumulator is Not Zero | 2 | 2 |
| JC | rel | Jump if Carry flag is set | 2 | 2 |
| JNC | rel | Jump if No Carry flag | 2 | 2 |
| JB | bit,rel | Jump if direct Bit set | 3 | 2 |
| JNB | bit,rel | Jump if direct Bit Not set | 3 | 2 |
| JBC | bit,rel | Jump if direct Bit is set & Clear bit | 3 | 2 |
| CJNE | A,direct,rel | Compare direct to A & Jump if Not Equal | 3 | 2 |
| CJNE | A,#data,rel | Comp, immed, to A & Jump if Not Equal | 3 | 2 |
| CJNE | Rn,#data,rel | Comp, immed, to reg & Jump if Not Equal | 3 | 2 |
| CJNE | @Ri,#data,rel | Comp, immed, to ind, & Jump if Not Equal | 3 | 2 |
| DJNZ | Rn,rel | Decrement register & Jump if Not Zero | 2 | 2 |
| DJNZ | direct,rel | Decrement direct & Jump if Not Zero | 3 | 2 |
| NOP | | No operation | 1 | 1 |

**Notes on data addressing modes:**

Rn —Working register R0–R7

direct —128 internal RAM locations, any I/O port, control or status register

@Ri —Indirect internal RAM location addressed by register R0 or R1

#data —8-bit constant included in instruction

#data16 —16-bit constant included as bytes 2 & 3 of instruction

bit —128 software flags, any I/O pin, control or status bit

**Notes on program addressing modes:**

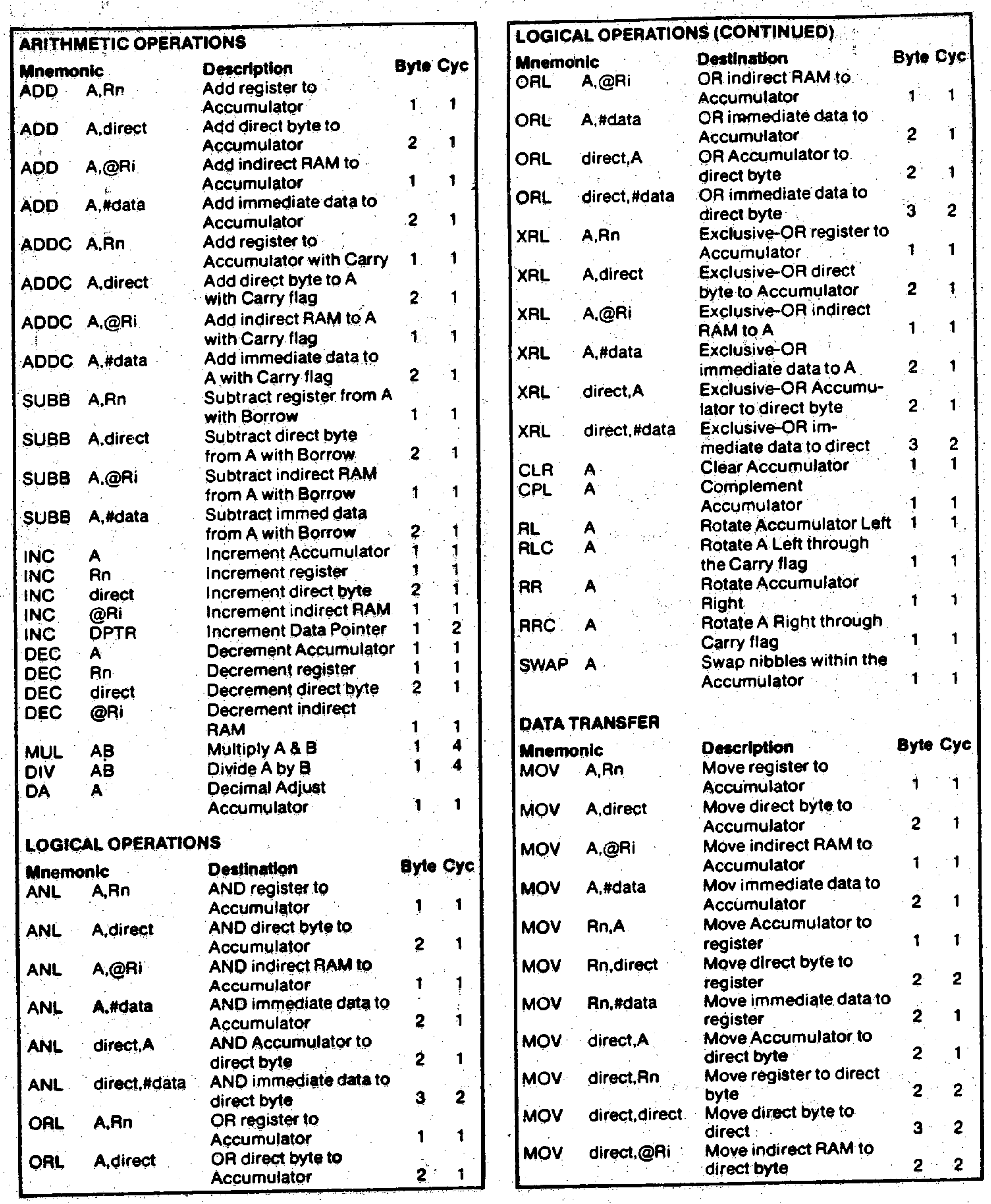
addr16 —Destination address for LCALL & LJMP may be anywhere within the 64-K program memory address space

Addr11 —Destination address for ACALL & AJMP will be within the same 2-K page of program memory as the first byte of the following instruction

rel —SJMP and all conditional jumps include an 8-bit offset byte, Range is +127–128 bytes relative to first byte of the following instruction

All mnemonics copyrighted © Intel Corporation 1979

Figure B.2 8051 instruction set description and timings. (From [Intel].)



**ARITHMETIC OPERATIONS**

| Mnemonic | | Description | Byte | Cyc |
|---|---|---|---|---|
| ADD | A,Rn | Add register to Accumulator | 1 | 1 |
| ADD | A,direct | Add direct byte to Accumulator | 2 | 1 |
| ADD | A,@Ri | Add indirect RAM to Accumulator | 1 | 1 |
| ADD | A,#data | Add immediate data to Accumulator | 2 | 1 |
| ADDC | A,Rn | Add register to Accumulator with Carry | 1 | 1 |
| ADDC | A,direct | Add direct byte to A with Carry flag | 2 | 1 |
| ADDC | A,@Ri | Add indirect RAM to A with Carry flag | 1 | 1 |
| ADDC | A,#data | Add immediate data to A with Carry flag | 2 | 1 |
| SUBB | A,Rn | Subtract register from A with Borrow | 1 | 1 |
| SUBB | A,direct | Subtract direct byte from A with Borrow | 2 | 1 |
| SUBB | A,@Ri | Subtract indirect RAM from A with Borrow | 1 | 1 |
| SUBB | A,#data | Subtract immed data from A with Borrow | 2 | 1 |
| INC | A | Increment Accumulator | 1 | 1 |
| INC | Rn | Increment register | 1 | 1 |
| INC | direct | Increment direct byte | 2 | 1 |
| INC | @Ri | Increment indirect RAM | 1 | 1 |
| INC | DPTR | Increment Data Pointer | 1 | 2 |
| DEC | A | Decrement Accumulator | 1 | 1 |
| DEC | Rn | Decrement register | 1 | 1 |
| DEC | direct | Decrement direct byte | 2 | 1 |
| DEC | @Ri | Decrement indirect RAM | 1 | 1 |
| MUL | AB | Multiply A & B | 1 | 4 |
| DIV | AB | Divide A by B | 1 | 4 |
| DA | A | Decimal Adjust Accumulator | 1 | 1 |

**LOGICAL OPERATIONS**

| Mnemonic | | Destination | Byte | Cyc |
|---|---|---|---|---|
| ANL | A,Rn | AND register to Accumulator | 1 | 1 |
| ANL | A,direct | AND direct byte to Accumulator | 2 | 1 |
| ANL | A,@Ri | AND indirect RAM to Accumulator | 1 | 1 |
| ANL | A,#data | AND immediate data to Accumulator | 2 | 1 |
| ANL | direct,A | AND Accumulator to direct byte | 2 | 1 |
| ANL | direct,#data | AND immediate data to direct byte | 3 | 2 |
| ORL | A,Rn | OR register to Accumulator | 1 | 1 |
| ORL | A,direct | OR direct byte to Accumulator | 2 | 1 |

**LOGICAL OPERATIONS (CONTINUED)**

| Mnemonic | | Destination | Byte | Cyc |
|---|---|---|---|---|
| ORL | A,@Ri | OR indirect RAM to Accumulator | 1 | 1 |
| ORL | A,#data | OR immediate data to Accumulator | 2 | 1 |
| ORL | direct,A | OR Accumulator to direct byte | 2 | 1 |
| ORL | direct,#data | OR immediate data to direct byte | 3 | 2 |
| XRL | A,Rn | Exclusive-OR register to Accumulator | 1 | 1 |
| XRL | A,direct | Exclusive-OR direct byte to Accumulator | 2 | 1 |
| XRL | A,@Ri | Exclusive-OR indirect RAM to A | 1 | 1 |
| XRL | A,#data | Exclusive-OR immediate data to A | 2 | 1 |
| XRL | direct,A | Exclusive-OR Accumulator to direct byte | 2 | 1 |
| XRL | direct,#data | Exclusive-OR immediate data to direct | 3 | 2 |
| CLR | A | Clear Accumulator | 1 | 1 |
| CPL | A | Complement Accumulator | 1 | 1 |
| RL | A | Rotate Accumulator Left | 1 | 1 |
| RLC | A | Rotate A Left through the Carry flag | 1 | 1 |
| RR | A | Rotate Accumulator Right | 1 | 1 |
| RRC | A | Rotate A Right through Carry flag | 1 | 1 |
| SWAP | A | Swap nibbles within the Accumulator | 1 | 1 |

**DATA TRANSFER**

| Mnemonic | | Description | Byte | Cyc |
|---|---|---|---|---|
| MOV | A,Rn | Move register to Accumulator | 1 | 1 |
| MOV | A,direct | Move direct byte to Accumulator | 2 | 1 |
| MOV | A,@Ri | Move indirect RAM to Accumulator | 1 | 1 |
| MOV | A,#data | Mov immediate data to Accumulator | 2 | 1 |
| MOV | Rn,A | Move Accumulator to register | 1 | 1 |
| MOV | Rn,direct | Move direct byte to register | 2 | 2 |
| MOV | Rn,#data | Move immediate data to register | 2 | 1 |
| MOV | direct,A | Move Accumulator to direct byte | 2 | 1 |
| MOV | direct,Rn | Move register to direct byte | 2 | 2 |
| MOV | direct,direct | Move direct byte to direct | 3 | 2 |
| MOV | direct,@Ri | Move indirect RAM to direct byte | 2 | 2 |

Figure B.2 (Continued)

The 8051 has two built-in timers, (0 and 1). The following are the special function register locations used to operate timer 1.

```
tcon        88h         ; timer control register
tmod        89h         ; timer mode register
tl1         8bh         ; timer register LSB
th1         8dh         ; timer register MSB
```

To run a timed loop, first:

```
                mov         tmod,#10h
```

to set timer 1 to no gate and 16 bit mode. The time for the loop is set by

```
          LOOPTIME          equu     150–7
```

where each loop will take 150 $\mu$s and 7 $\mu$s is the overhead to restart the timer. At the beginning of the loop use:

```
loop:       clr         a           ; Clear a register
            mov         tcon,a      ; Stop timer
            mov         tl1,a       ; Clear timer
            mov         th1,a
            setb        tcon.6      ; Start timer
```

At the end of the loop, wait for the timer by using:

```
wait:
        mov     a,#high(LOOPTIME)   ; Wait for MSB of LOOPTIME
        cjne    a,th1,$             ; and timer 1 to match
        mov     a,#low(LOOPTIME)
        xrl     a,tl1               ; xor LSBs to see if they are the same
        rrc     a                   ; move least significant bit
                                    ; into carry bit
        mov     a,#low(LOOPTIME)
        jnc     sync                ; Sync with timer, since the cjne takes
        nop                         ; 2 µs, there is a 50/50 chance the
                                    ; least significant bits of the timer
                                    ; and LOOPTIME will not match, this
                                    ; comparison should sync the program
                                    ; up with the timer so the least
                                    ; significant bits will always match.
sync:
        cjne    a,tl1,$             ; Wait for LSBs of LOOPTIME and timer 1
                                    ; to match
        sjmp    loop
```

Figure B.3 Using a built-in timer to control loop time.

This works for most values of LOOPTIME; of course if LOOPTIME is shorter than the time for one loop, it will not work at all. If LOOPTIME equals 256, for example, it will not work since 256=100h. The MSBs will match at the same time the LSB's will match. But 2 $\mu$s will pass before the LSBs are compared, so they won't match. For this case, only the MSBs need to be compared.

If the LOOPTIME is carefully chosen, the built-in timers can synchronize two cells which are executing different code.

Figure B.3 (Continued)

The following gives examples, written in 8051 assembler code, of how the 8051 writes four bytes of data to the switch and how it reads one byte from the switch.

When writing a byte to the switch, the 8051 first writes the 3 bit direction tag to port 1 (p1) and the 8 bit data to external RAM location *lowSWLat.* The direction tag tells which port is being written to, where 0 is the north port, 1 is northeast, and so on. The Switch hardware polls the output latches on all the cells and when data appears in a given latch, the Switch looks into a table to find where to send the data. Then it writes the data and a tag telling from where it came into the input queue of the destination cell.

Here is an example of how to send four bytes of data, stored in internal RAM, out of the north port. The numbers to the left of the instructions are the execution times in $\mu$s. The syntax for a move is: mov destination,source.

```
 2  mov    dptr,#lowSWLat    ; Have dptr point to the Switch
                             ; Lattice port in external RAM.
 2  mov    p1,#north         ; 8051 builtin port one (p1) is where
                             ; the three bit direction tag is written.
 1  mov    a,byte0           ; Move first byte from internal
                             ; RAM into accumulator (a).
 2  movx   @dptr,a           ; Store accumulator at location that dptr
                             ; points to, which is the switch lattice.
11  lcall  writedelay        ; It takes the switch 12 μs to poll
                             ; all the processors, so wait 12 μs
                             ; to be sure the data have been sent.
 1  mov    a,byte1           ; get next data byte and write to switch.
 2  movx   @dptr,a           ; Notice dptr does not have to be reset,
                             ; nor does p1
11  lcall  writedelay        ; Wait again
 1  mov    a,byte2
 2  movx   @dptr,a           ; Send third byte
11  lcall  writedelay
 1  mov    a,byte3
 2  movx   @dptr,a           ; send last byte
```

Figure B.4 Example of 8051 code for inter-cell communication.

The call to *writedelay* takes 11 $\mu$s, the *mov* instruction takes 1 $\mu$s, giving the 12 $\mu$s delay needed between writes to the switch. The total time to write one 32 bit word is 49 $\mu$s assuming no writes follow (thus the missing call to writedelay after the last mov @dptr,a). The three calls to writedelay give a total of 33 $\mu$s spent waiting on the switch. In some applications, this time can be used doing useful operations. Data, by convention (not hardware restrictions), is sent least significant byte (LSB) first.

To read from the Switch:

```
2   mov   dptr,#lowSWLat        ; Same as writing
2   jnb   p0.7,$                ; Test bit 7 of 8051 port 0.
                                ; If not set, there is no data,
                                ; so keep testing until there is some.
2   mov   a,@dptr               ; Get one byte of data.
1   mov   byte0,a               ; Save
1   mov   a,p0                  ; Read port 0 to see from which
                                ; direction it came.
```

The program will loop on the *jnb* instruction until data arrives in the queue. Reading an empty queue is a fatal error. In some programs, owing to the structure of the program, the data is always in the queue when the switch is read, so checking port 0 bit 7 is not necessary. Otherwise it takes at least 2 $\mu$s to check for the presence of data.

Figure B.4 (Continued)

```
;               Write port directions

north           equ             08h
ne              equ             18h
east            equ             28h
se              equ             38h
south           equ             48h
sw              equ             58h
west            equ             68h
nw              equ             78h

ARG1            equ             87b4h           ; Location of first argument

lowSWLat equ                    0c000h          ; Location of switch lattice
```

Figure B.5 Contents of *ports.h.*

```
;
; Wait 14 microseconds for switch to send data
;
writedelay:
        nop
        nop
writedelay12:           ; Wait 12 microseconds
        nop
        nop             ; Each nop takes 1 microsecond to execute.
        nop             ; The call to writedelay takes 2 microseconds.
        nop             ; The return take 2 microseconds.
        nop             ; Calling and return from "writedelay" takes a
        nop             ; total of 13 microseconds. This leaves one
        nop             ; microsecond for the calling program to do
        ret             ; a register move.

;
; Wait for something to appear in input buffer
;
readwait:
        jnb             p0.7,$   ; Check bit 7 of port 0
        ret                      ; If it is zero there is not data in the queue,
                                 ; so jump back and check again
                                 ; If it is not zero, return
```

Figure B.6 Contents of *util.h.*

Jan 26 10:59 1984  filter.s Page 1

```
        ;           Program Name:     filter (f2)
        ;           Algorithm:        Figure 6.1
        ;           Machine:          VLSI processor array, simulated by Poker.
        ;           Function:         Compute $y_m$ given $x_m$ using
        ;                             $y_m=\sum_{k=0}^{q} b_k x_{m-k} + \sum_{k=1}^{p} a_k y_{m-k}.$
        ;           Precision:        Input:  8-bit unsigned.
        ;                             Coefficients:    8-bit unsigned.
        ;                             Output: 16-bit unsigned.
        ;           Number of PEs:    p+q+1, the number of coefficients.
        ;           Parameters:       p+q+1, the number of coefficients.
        ;           Input:            Arrives at the north port of cell (1,1).
        ;           Output:           Departs from the south port of cell (4,1).
        ;           Loop Time:        33 μs to produce one output sample.
        ;           Max sample Rate:  30 KHz
        #include "ports.h"
                    org     29h             ; Start of readport buffers
        coef:       ds      1               ; Filter coefficient
        sum:        ds      2
        right:      ds      2

                    org     08000h
2                   mov     dptr,#ARG1      ; Get coef value
2                   movx    a,@dptr
1                   mov     coef,a

2                   mov     dptr,#lowSWLat          ; dptr doesn't change after this
2                   mov     p1,#south       ; neither does p1
2                   mov     p0,#0f0h
        ;
        ; 8         sum := 0;
        ;
1                   clr     a
1                   mov     sum,a
1                   mov     sum+1,a
        ;
        ; 9         out <- 0;
        ;
2                   movx    @dptr,a         ; out <- 0
2                   lcall   writedelay
1                   clr     a
2                   movx    @dptr,a
1                   mov     sum,a

        main:
```

Figure B.7  8051 program listing for 8 bit fast filter (f2).

```
Jan 26 10:59 1984  filter.s Page 2

        ;
        ; 14          sum <- top;
        ;
2                     movx   a,@dptr       ; sum <- top
1                     mov    sum+1,a
2                     movx   a,@dptr
1                     mov    sum,a
        ;
        ; 13          in <- right;
        ;
2                     movx   a,@dptr       ; in <- right
        ;
        ; 15          sum := sum +coef * in;
        ;
2                     mov    b,coef
4                     mul    ab
1                     add    a,sum+1               ; sum := sum + in * right
        ;
        ; 16          out <- sum
        ;
2                     movx   @dptr,a       ; out <- sum
        ;
        ; 15          sum := sum +coef * in;     (cont.)
        ;
1                     mov    sum+1,a
1                     mov    a,b
1                     addc   a,sum
1                     mov    sum,a
        ;
        ; 16          out <- sum    (cont.)
        ;
1                     nop
1                     nop
1                     nop
1                     nop                  ; 12 microseconds between writes
1                     nop
1                     nop
1                     nop
1                     mov    a,sum
2                     movx   @dptr,a
        ;
        ; 17          end
        ;
2                     sjmp   main
        #include "util.h"
        end
```

Figure B.7 (Continued)

```
Jan 26 10:59 1984  input.s Page 1

        ;               Program Name:     input (f2)
        ;               Algorithm:        None
        ;               Machine:          VLSI processor array, simulated by Poker.
        ;               Function:         Generate input data for filter program.
        ;               Precision:        Output: 16-bit unsigned.
        ;               Number of PEs:    1
        ;               Output:           Departs from the east port.
        #include "ports.h"
                        org     29h             ; Start of readport buffers
        i:              ds      1               ; Filter coefficient
        sum:            ds      2
        right:          ds      2

                        org     08000h
2                       mov     dptr,#lowSWLat          ; dptr doesn't change after this
2                       mov     p1,#west        ; neither does p1
2                       mov     p0,#0f0h
        ;
        ; 8             i := 1;
        ;
2                       mov     i,#1            ; i := 1
1                       mov     r0,#2           ; wait 6 microseconds so right values
1                       nop
2                       djnz    r0,$            ; follow top values into cell.s
        main:

1                       mov     r0,#4           ; wait 9 microseconds so right values
2                       djnz    r0,$            ; follow top values into cell.s
        ;
        ; 13            out <- i;
        ;
1                       mov     a,i
2                       movx    @dptr,a         ; out <- i
        ;
        ; 12            tmp <- sync;
        ;
2                       jnb     p0.7,$          ; Loop until data arrives
2                       movx    a,@dptr         ; Dummy read on Switch port
        ;
        ; 14            i := i + 1;
        ;
1                       inc     i               ; i := i + 1
        ;
        ; 15            end
        ;
2                       sjmp    main
        end
```

Figure B.7 (Continued)

Jan 26 10:59 1984 output.s Page 1

```
        ;               Program Name:   output (f2)
        ;               Algorithm:      None
        ;               Machine:        VLSI processor array, simulated by Poker.
        ;               Function:       Receive output data from filter program
        ;                               and send it back to some filter cells.
        ;               Precision:      Input:  16-bit unsigned.
        ;               Number of PEs:  1
        ;               Output:         Arrives at the south port.

        #include "ports.h"

                        org     29h             ; Start of readport buffers
        coef:           ds      1               ; Filter coefficient
        sum:            ds      2
        right:          ds      2

                        org     08000h
2                       mov     dptr,#lowSWLat     ; dptr doesn't change after this
2                       mov     p1,#west        ; neither does p1
2                       mov     p0,#0f0h

        main:
        ;
        ; 10            out <- in;
        ;
2                       jnb     p0.7,$          ; Wait for input
2                       movx    a,@dptr         ; sum <- bottom  Read LSB and send out

2                       movx    @dptr,a         ; Send to other cells

1                       mov     sum+1,a

2                       jnb     p0.7,$          ; Wait for input
2                       movx    a,@dptr         ; Read MSB but don't send out
1                       mov     sum,a
        ;
        ; 11            end
        ;
2                       sjmp    main
                        end
```

Figure B.7 (Continued)

```
Jan 26 10:59 1984  output.s Page 2

Jan 26 10:59 1984  zero.s Page 1
         ;             Program Name:     zero (f2)
         ;             Algorithm:        None
         ;             Machine:          VLSI processor array, simulated by Poker.
         ;             Function:         Send zeros to first filter cell after
         ;                               receiving data from the input cell.
         ;             Precision:        Output: 16-bit unsigned zeros.
         ;             Number of PEs:    1
         ;             Output:           Departs from the north port.
         #include "ports.h"
                       org    29h             ; Start of readport buffers
         coef:         ds     1               ; Filter coefficient
         sum:          ds     2
         right:        ds     2

                       org    08000h
2                      mov    dptr,#lowSWLat        ; dptr never changes after this
2                      mov    p1,#north       ; neither does p1
2                      mov    p0,#0f0h
         main:
         ;
         ; 7           z := 0;
         ;
1                      clr    a               ; out <- 0
         ;
         ; 12          Botout <- z;
         ;
2                      movx   @dptr,a
2                      lcall  writedelay
1                      clr    a
2                      movx   @dptr,a
         ;
         ; 11          dumb <- sync;
         ;
2                      jnb    p0.7,$          ; Wait for data in input queue
2                      movx   a,@dptr
         ;
         ; 13          end
         ;
2                      sjmp   main

         #include "util.h"
                       end
```

Figure B.7 (Continued)

Jan 26 14:12 1984 filter.l Page 1

```
        ;               Program Name:     filter (f3)
        ;               Algorithm:        Figure 6.1
        ;               Machine:          VLSI processor array, simulated by Poker
        ;               Function:         Compute y_m given x_m using
        ;                                 $y_m=\sum_{k=0}^{q} b_k x_{m-k} + \sum_{k=1}^{p} a_k y_{m-k}$
        ;               Precision:        Input:  16-bit unsigned.
        ;                                 Coefficients:   8-bit unsigned.
        ;                                 Output: 24-bit unsigned.
        ;               Number of PEs:    p+q+1, the number of coefficients
        ;               Parameters:       p+q+1, the number of coefficients
        ;               Input:            Arrives at the north port of cell 0
        ;               Output:           Departs from the south port of cell p+q
        ;               Loop Time:        63 μs to produce on output sample
        ;               Max sample Rate:  15.8 KHz

        #include "ports.h"

                        org     29h             ; Start of readport buffers
        coef:           ds      1               ; Filter coefficient decimal point
                                                ;         is right of MSB (i.e. coef < 1)
        sum:            ds      3
        right:          ds      2               ; Input data

                        org     08000h
        ;
        ; 1             code    filter(coef);
        ;
2                       mov     dptr,#ARG1      ; Get coef value
2                       movx    a,@dptr
1                       mov     coef,a

2                       mov     dptr,#lowSWLat          ; dptr doesn't change after this
2                       mov     p1,#south       ; neither does p1
2                       mov     p0,#0f0h
        ;
        ; 8             sum := 0;
        ;
1                       clr     a
1                       mov     sum+2,a
1                       mov     sum+1,a
1                       mov     sum+0,a
        ;
        ; 9             out <- 0;
        ;
```

Figure B.8 8051 listing for fast filter program (f3).

```
Jan 26 14:12 1984  filter.l Page 2

2                  movx   @dptr,a        ; out <- 0
2                  lcall  writedelay
1                  clr    a
2                  movx   @dptr,a
2                  lcall  writedelay
1                  clr    a
2                  movx   @dptr,a

      main:
      ;
      ; 14           sum <- top;
      ;
2                  movx   a,@dptr        ; sum <- top
1                  mov    sum+2,a
2                  movx   a,@dptr
1                  mov    sum+1,a
2                  movx   a,@dptr
1                  mov    sum,a
      ;
      ; 13           in <- right;
      ;
2                  jnb    p0.7,$         ; Wait for external input
2                  movx   a,@dptr        ; in <- LSB of right
      ;
      ; 15           sum := sum + coef * in;
      ;
2                  mov    b,coef
4                  mul    ab
1                  add    a,sum+2                ; sum := sum + in * right
1                  mov    sum+2,a
1                  nop                   ; Wait so output.s has a chance
1                  nop                   ; to get data out
1                  nop
1                  nop
1                  nop
1                  nop
1                  nop
      ;
      ; 16           out <- sum;
      ;
2                  movx   @dptr,a        ; out <- LSB of sum
      ;
      ; 15           sum := sum + coef * in; (cont)
      ;
1                  mov    a,b
1                  addc   a,sum+1
1                  mov    sum+1,a
2                  jnc    nocarry        ; This will throw the timing off,
```

Figure B.8 (Continued)

```
Jan 26 14:12 1984  filter.l Page 3

1                        inc     sum            ; but all will resync waiting for the next input
        nocarry:
        ;
        ; 13             in <- right;    (cont)
        ;
2                        movx    a,@dptr        ; in <- MSB of right
        ;
        ; 15             sum := sum + coef * in;        (cont)
        ;
2                        mov     b,coef
4                        mul     ab
1                        add     a,sum+1
        ;
        ; 16             out <- sum;   (cont)
        ;
2                        movx    @dptr,a        ; out <- middle byte of sum
        ;
        ; 15             sum := sum + coef * in;        (cont)
        ;
1                        mov     sum+1,a
1                        mov     a,b
1                        addc    a,sum
1                        mov     sum,a
1                        nop
1                        nop
1                        nop
1                        nop
        ;
        ; 16             out <- sum;   (cont)
        ;
2                        movx    @dptr,a        ; out <- MSB of sum
        ;
        ; 17             end
        ;
2                        sjmp    main

        #include "util.h"

                         end
```

Figure B.8 (Continued)

Jan 26 14:12 1984 input.s Page 1

```
        ;               Program Name:   input (f3)
        ;               Algorithm:      Figure 6.2??
        ;               Machine:        VLSI processor array, simulated by Poker.
        ;               Function:       Generate input for filter program, one
        ;                               sample every LOOPTIME µs
        ;               Precision:      Output: 16-bit unsigned.
        ;               Number of PEs:  1
        ;               Parameters:     LOOPTIME, the time between samples
        ;               Output:         Departs from the east port.

        LOOPTIME equu   100-9           ; Time in micro seconds between outputs

        #include "ports.h"

                        org     29h             ; Start of readport buffers
        i:              ds      2               ; Filter coefficient
        sum:            ds      2
        right:          ds      2

                        org     08000h
2                       mov     dptr,#lowSWLat       ; dptr doesn't change after this
2                       mov     p1,#west        ; neither does p1
2                       mov     p0,#0f0h
2                       mov     tmod,#10h       ; Set timer 1 to no gate, timer, mode 1 (16 bit)
        ;
        ; 8             i := 0;
        ;
2                       mov     i+1,#1          ; i := 1
2                       mov     i,#0

1                       mov     r0,#9           ; wait 20 microseconds so right values
1                       nop
2                       djnz    r0,$

        main:
        ;
        ; 12            tmp <- sync  Rather than use a port read for
        ;                            synchronization, internal timer is used
        ;
1                       clr     a
1                       mov     tcon,a          ; Stop timer
1                       mov     tl1,a           ; clear timer
1                       mov     th1,a
1                       setb    tcon.6          ; Start timer
        ;
        ; 13            out <- i;
        ;
1                       mov     a,i+1
```

Figure B.8 (Continued)

```
Jan 26 14:12 1984  input.s Page 2

2                  movx   @dptr,a          ; out <- LSB of i
        ;
        ; 14       i := i + 1;
        ;
1                  add    a,#1             ; i := i + 1
1                  mov    i+1,a
1                  clr    a
1                  addc   a,i
1                  mov    i,a
        ;
        ; 13       out <- i;      (cont)
        ;
1                  nop
1                  nop
1                  nop
2                  movx   @dptr,a          ; out <- MSB of i

1                  mov    a,#LOOPTIME
2                  cjne   a,tl1,$+3        ; Wait for timer
2                  jnc    $-3
        ;
        ; 15       end
        ;
2                  sjmp   main

                   end
```

Figure B.8 (Continued)

Jan 26 14:12 1984 output.s Page 1

```
      ;            Program Name:   output (f3)
      ;            Algorithm:      Figure 6.2??
      ;            Machine:        VLSI processor array, simulated by Poker.
      ;            Function:       Receive input from filter program.
      ;                            Wait for signal from input cell and send
      ;                            input batck to filter cells.
      ;            Precision:      Input:  24-bit unsigned.
      ;                            Output: 16-bit unsigned.
      ;            Number of PEs:  1
      ;            Input:          Data arrives from south port.
      ;                            Synch signal arrives from north port.
      ;            Output:         Departs from the east port.

      #include "ports.h"

                   org    29h             ; Start of readport buffers
      coef:        ds     1               ; Filter coefficient
      sum:         ds     3
      right:       ds     2

                   org    08000h
2                  mov    dptr,#lowSWLat       ; dptr doesn't change after this
2                  mov    p1,#west        ; neither does p1
2                  mov    p0,#0f0h

      main:
2                  jnb    p0.7,$          ; Wait for buffer to fill
2                  movx   a,@dptr         ; Throw away the LSB of sum
1                  mov    sum+2,a

2                  jnb    p0.7,$          ; Wait for buffer to fill
2                  movx   a,@dptr
1                  mov    sum+1,a                ; get middle byte of sum

2                  jnb    p0.7,$          ; Wait for buffer to fill
2                  movx   a,@dptr
1                  mov    sum,a           ; get MSB of sum

1                  mov    a,sum+1
2                  jnb    p0.7,$          ; Wait for buffer to fill
2                  movx   @dptr,a         ; Send out sum+1 as soon as possible
2                  movx   a,@dptr         ; dummy read
1                  mov    sum+3,a

1                  mov    a,sum
1                  nop                    ; wait for switch network
1                  nop
1                  nop
```

Figure B.8 (Continued)

Jan 26 14:12 1984 output.s Page 2

```
2                   movx   @dptr,a        ; Send out sum
2                   movx   a,@dptr        ; dummy read
1                   mov    sum+4,a

2                   sjmp   main

                    end
```

Figure B.8 (Continued)

Jan 26 14:12 1984  zero.s Page 1

```
        ;               Program Name:   zero (f3)
        ;               Algorithm:      Figure 6.2??
        ;               Machine:        VLSI processor array, simulated by Poker.
        ;               Function:       Generate zero values for filter program
        ;                               every time a synch signal come from the
        ;                               input cell.
        ;               Precision:      Input:  16-bit unsigned.
        ;                               Output: 16-bit unsigned.
        ;               Number of PEs:  1
        ;               Input:          Synch signal arrives from east port.
        ;               Output:         Departs from the north port.
        #include "ports.h"
                        org     29h             ; Start of readport buffers
        coef:           ds      1               ; Filter coefficient
        sum:            ds      2
        right:          ds      2

                        org     08000h
2                       mov     dptr,#lowSWLat          ; dptr never changes after this
2                       mov     p1,#north       ; neither does p1
2                       mov     p0,#0f0h
        main:
        ;
        ; 10            out <- 0;
        ;
1                       clr     a               ; out <- 0
2                       movx    @dptr,a
2                       lcall   writedelay
1                       clr     a
2                       movx    @dptr,a
2                       lcall   writedelay
1                       clr     a
2                       movx    @dptr,a
        ;
        ; 9             dumb <- sync;
        ;
2                       jnb     p0.7,$          ; Wait for input
2                       movx    a,@dptr
2                       jnb     p0.7,$          ; Wait for input
2                       movx    a,@dptr
        ;
        ; 11            end
        ;
2                       sjmp    main

        #include "util.h"
                        end
```

Figure B.8 (Continued)

Jan 31 16:34 1984  auto.s Page 1

```
        ;               Program Name:    auto (a3)
        ;               Algorithm:       Figure 6.12
        ;               Machine:         VLSI processor array, simulated by Poker.
        ;               Function:        Find autocorrelation coefficients R(i)
        ;                                given input signal x(m), using
                                                  k=M-i-1
        ;                                R(i)=      ∑     x(k)x(k+i).
                                                   k=0
        ;
        ;               Precision:       Input:  16-bit unsigned.
        ;                                Output: 32-bit unsigned.
        ;               Number of PEs:   p, the number of coefficients computed.
        ;               Parameters:      p, the number of coefficients computed.
        ;               Input:           Arrives at the north port of cell (1,2).
        ;               Output:          Departs from east port of merge cell.
        ;               Loop Time:       82 µs to process one input sample.
        ;               Max Sample Rate: 12 KHz
        ;
        ;               Sends data through the switch LSB first
        ;
        ;               Use with runrev.o eproms
        ;

        samples         equu    10              ; Number of sample per frame

        #include "ports.h"

                        org     029h
        ;
        ; 18            sint    i;
        ;
        i:              ds      1
        ;
        ; 19            int     sum,left,top
        ;
        sum:            ds      4
        top:            ds      2
        left:           ds      2

                        org     8000h
1                       mov     r0,#18          ; wait 37 microS for input.s to fill buffer
2                       djnz    r0,$
        ;
        ; 22            i := 0;
        ;
1                       clr     a
```

Figure B.9  8051 programs for autocorrelation program a3 using 16-bit inputs and 32-bit sums.

```
Jan 31 16:34 1984  auto.s Page 2

1                    mov    i,a              ; i := 0
      ;
      ; 23           sum := 0;
      ;
1                    mov    sum+3,a                ; sum := 0
1                    mov    sum+2,a
1                    mov    sum+1,a
1                    mov    sum,a
      ;
      ; 25           out <- sum
      ;
2                    mov    p1,#south        ; Write 0 to south port
2                    mov    dptr,#lowSWLat         ; This value will remain in dptr
                                             ; from now on

1                    clr    a
2                    movx   @dptr,a          ; write 0 to switch

1                    mov    r0,#3
2                    djnz   r0,$             ; Wait 8 microseconds for switch

1                    clr    a
2                    movx   @dptr,a          ; write 0 to switch

      main:
      ;
      ; 29           i := i+1;
      ;
1                    inc    i                ; i := i + 1
      ;
      ; 31           left <- in2;
      ;
2                    movx   a,@dptr          ; Read LSB of left from switch
1                    mov    left+1,a
2                    movx   a,@dptr          ; Read MSB of left from switch
1                    mov    left,a
      ;
      ; 30           top <- in1;
      ;
2                    movx   a,@dptr          ; Read LSB of top from switch
1                    mov    top+1,a
2                    movx   a,@dptr          ; Read MSB of top from switch
1                    mov    top,a
      ;
      ; 33           if i < samples then
      ;
1                    mov    a,i              ; load i
2                    cjne   a,#samples,loop; if(i != samples) goto loop
```

Figure B.9 (Continued)

```
2                       ljmp   endloop          ; if( i == samples) goto endloop

        ;               sum := sum + left * top
        ;               Where sum is 32 bits and left and top are 16 bits
        ;
        ;                 30     31    (left)
        ;           X     2e     2f    (top)
        ;          -----------------
        ; +             30x2f  31x2f
        ; +        30x2e 31x2e
        ; + 2a      2b    2c     2d    (sum)
        ;
        loop:
        ;
        ; 36            sum := sum + left * top
        ;
1                       mov    a,left+1         ; LSB of left
2                       mov    b,top+1                  ; LSB of top
4                       mul    ab
1                       add    a,sum+3                  ; LSB of sum (byte 4)
1                       mov    sum+3,a
1                       mov    a,b
1                       addc   a,sum+2                  ; add in byte 3 of sum
1                       mov    sum+2,a
1                       clr    a
1                       addc   a,sum+1                  ; add carry to byte 2 of sum
1                       mov    sum+1,a
1                       clr    a
1                       addc   a,sum            ; add carry to MSB of sum (byte 1)
1                       mov    sum,a

1                       mov    a,top+1                  ; LSB of top
        ;
        ; 35            out <- top;
        ;
2                       movx   @dptr,a          ; Send LSB of top to south port
        ;
        ; 36            sum := sum + top * left       (cont)
        ;
2                       mov    b,left           ; MSB of left
4                       mul    ab
1                       add    a,sum+2                  ; add to byte 3 of sum
1                       mov    sum+2,a
1                       mov    a,b
1                       addc   a,sum+1                  ; add to byte 2 of sum
1                       mov    sum+1,a
1                       clr    a
1                       addc   a,sum            ; add carry to byte 1 of sum
```

Figure B.9 (Continued)

```
Jan 31 16:34 1984  auto.s Page 4

1                     mov     sum,a

1                     mov     a,top             ; MSB of top
        ;
        ; 35          out <- top;
        ;
2                     movx    @dptr,a           ; Send MSB of top to south port
        ;
        ; 36          sum := sum + top * left       (cont)
        ;
2                     mov     b,left+1          ; LSB of left
4                     mul     ab
1                     add     a,sum+2                     ; add to byte 3 of sum
1                     mov     sum+2,a
1                     mov     a,b
1                     addc    a,sum+1                     ; add to byte 2 of sum
1                     mov     sum+1,a
1                     clr     a
1                     addc    a,sum             ; add carry to byte 1 of sum
1                     mov     sum,a

1                     mov     a,top             ; MSB of top
2                     mov     b,left            ; LSB of left
4                     mul     ab
1                     add     a,sum+1                     ; add to byte 2 of sum
1                     mov     sum+1,a
1                     mov     a,b
1                     addc    a,sum             ; add to MSB of sum (byte 1)
1                     mov     sum,a
        ;
        ; 37          end
        ;
2                     ljmp    main

        ;             sum := sum + left * top
        ;             Where sum is 32 bits and left and top are 16 bits
        ;
        ;               30      31      (left)
        ;       X       2e      2f      (top)
        ;       ----------------
        ; +           30x2f   31x2f
        ; +     30x2e 31x2e
        ; + 2a    2b    2c      2d      (sum)
        ;
        endloop:
        ;
        ; 41          results <- sum;
        ;
```

Figure B.9 (Continued)

```
Jan 31 16:34 1984  auto.s Page 5

2                   mov     p1,#east          ; Next write is to east port
        ;
        ; 40        sum := sum  + top * left;
        ;
1                   mov     a,left+1          ; LSB of left
2                   mov     b,top+1                    ; LSB of top
4                   mul     ab
1                   add     a,sum+3                    ; LSB of sum (byte 4)
1                   mov     sum+3,a
        ;
        ; 41        results <- sum;           (cont)
        ;
2                   movx    @dptr,a           ; Send LSB of sum to east port
        ;
        ; 40        sum := sum  + top * left;          (cont)
        ;
1                   mov     a,b
1                   addc    a,sum+2                    ; add in byte 3 of sum
1                   mov     sum+2,a
1                   clr     a
1                   addc    a,sum+1                    ; add carry to byte 2 of sum
1                   mov     sum+1,a
1                   clr     a
1                   addc    a,sum             ; add carry to MSB of sum (byte 1)
1                   mov     sum,a

1                   mov     a,top+1                    ; LSB of top
2                   mov     b,left            ; MSB of left
4                   mul     ab
1                   add     a,sum+2                    ; add to byte 3 of sum
1                   mov     sum+2,a
1                   mov     a,b
1                   addc    a,sum+1                    ; add to byte 2 of sum
1                   mov     sum+1,a
1                   clr     a
1                   addc    a,sum             ; add carry to byte 1 of sum
1                   mov     sum,a

1                   mov     a,top             ; MSB of top
2                   mov     b,left+1          ; LSB of left
4                   mul     ab
1                   add     a,sum+2                    ; add to byte 3 of sum
1                   mov     sum+2,a
        ;
        ; 41        results <- sum;           (cont)
        ;
2                   movx    @dptr,a           ; Send third byte of sum to east port
        ;
```

Figure B.9 (Continued)

```
Jan 31 16:34 1984  auto.s Page 6

        ; 40          sum := sum + top * left;      (cont)
        ;
1                     mov     a,b
1                     addc    a,sum+1                   ; add to byte 2 of sum
1                     mov     sum+1,a
1                     clr     a
1                     addc    a,sum           ; add carry to byte 1 cf sum
1                     mov     sum,a

1                     mov     a,top           ; MSB of top
2                     mov     b,left          ; MSB of left
4                     mul     ab
1                     add     a,sum+1                   ; add to byte 2 of sum
        ;
        ; 41          results <- sum;         (cont)
        ;
2                     movx    @dptr,a         ; Send second byte of sum to east port
        ;
        ; 40          sum := sum + top * left;      (cont)
        ;
1                     mov     sum+1,a
1                     mov     a,b
1                     addc    a,sum           ; add to MSB of sum (byte 1)
1                     mov     sum,a
        ;
        ; 41          results <- sum;         (cont)
        ;
1                     nop                     ; Wait 8 microseconds for switch
1                     nop
1                     nop
1                     nop
2                     movx    @dptr,a         ; Send second byte of sum to east port

        ;
        ;             Initialize i and sum for next autocorrelation calculation
        ;
        ;
        ;
        ;
        ; 44          i := 0;
        ;
1                     clr     a
1                     mov     i,a             ; i := 0
        ;
        ; 42          sum := 0;
        ;
1                     mov     sum+3,a                   ; sum := 0
1                     mov     sum+2,a
1                     mov     sum+1,a
```

Figure B.9 (Continued)

```
Jan 31 16:34 1984  auto.s Page 7

1                       mov     sum,a
        ;
        ; 43            out <- sum;
        ;

1                       clr     a               ; Send a 0 to south port
1                       nop                     ; Wait for switch
2                       mov     p1,#south
2                       movx    @dptr,a         ; write to switch

1                       mov     r0,#3
1                       nop
2                       djnz    r0,$            ; Wait 8 microseconds for switch

2                       movx    @dptr,a         ; write to switch
        ;
        ; 45            end
        ;
2                       ljmp    main

                        end
```

Figure B.9 (Continued)

```
Jan 31 16:34 1984  input.s Page 1

         ;                   This is the warp drive version of input.x
         ;                   It outputs 16 bit integers to port 0 LSB first.
         ;

         #include "ports.h"

                             org    029h
         ;
         ; 14                int    i;
         ;
         i:                  ds     2

                             org    8000h
         ;
         ; 16                i := 100;
         ;
2                            mov    i,#0              ; i := 100
2                            mov    i+1,#100

2                            mov    dptr,#lowSWLat        ; Get switch address
2                            mov    p1,#east          ; Set direction to 2 (east)
2                            mov    p0,#0f0h

         main:
         ;                   Write i to east port
         ;
         ; 20                out <- i;
         ;
1                            mov    a,i+1             ; get LSB of i
2                            movx   @dptr,a           ; write to switch

1                            mov    r0,#3             ; Wait 8 micro seconds for switch
2                            djnz   r0,$

1                            mov    a,i               ; get MSB of i
2                            movx   @dptr,a           ; write to switch
         ;
         ; 22                i = 2*i;
         ;
1                            mov    a,i+1
1                            add    a,acc             ; i := 2i
1                            mov    i+1,a
1                            mov    a,i
1                            addc   a,acc
1                            mov    i,a
         ;
         ; 21                tmp <- sync;
         ;
```

Figure B.9 (Continued)

```
Jan 31 16:34 1984  input.s Page 2

2                       jnb     p0.7,$          ; if p0.7 is 0 there is no data to read
                                                ; so loop until there is
2                       movx    a,@dptr         ; Read sync byte from switch

2                       jnb     p0.7,$          ; if p0.7 is 0 there is no data to read
                                                ; so loop until there is
2                       movx    a,@dptr         ; Read sync byte from switch
        ;
        ; 23            end
        ;
2                       sjmp    main

                        end
```

Figure B.9 (Continued)

```
Jan 31 16:34 1984  pipe.s Page 1

        ;               Warp Drive version of pipe.x
        ;               Does 2 byte transfers holding the first byte until
        ;               the second is received.
        ;

        #include "ports.h"

                        org     8000h
2                       mov     dptr,#lowSWLat
2                       mov     p0,#0f0h
2                       mov     p1,#north
        ;
        ; 16            while true do
        ;
        main:
        ;               Read input byte
        ;
        ; 18            tmp <- in;
        ;
2                       jnb     p0.7,$          ; if p0.7 is 0 there is no data to read
                                                ; so loop until there is
2                       movx    a,@dptr         ; Read input byte from switch
1                       mov     r1,a            ; It'll only come from one direction

2                       jnb     p0.7,$          ; if p0.7 is 0 there is no data to read
                                                ; so loop until there is
2                       movx    a,@dptr         ; Read input byte from switch
1                       mov     r2,a            ; It'll only come from one direction

        ;               Write i to north port
        ;
        ; 19            out <- tmp;
        ;
1                       mov     a,r1            ; send first byte
2                       movx    @dptr,a         ; write to switch

1                       mov     r0,#3           ; Delay before second write
2                       djnz    r0,$

1                       mov     a,r2            ; send second byte
2                       movx    @dptr,a
        ;
        ; 20            end
        ;
2                       sjmp    main

                        end
```

Figure B.9 (Continued)

```
    ;          Program Name:     auto (a4)
    ;          Algorithm:        Figure 6.12
    ;          Machine:          VLSI processor array, simulated by Poker
    ;          Function:         Find autocorrelation coefficients R(i)
    ;                            given input signal x(m), using
                                      k=M-i-1
    ;                            R(i)=   Σ    x(k)x(k+i)
                                        k=0
    ;
    ;          Precision:        Input:  8 bits, unsigned
    ;                            Output: 16 bits, unsigned
    ;          Number of PEs:    p, the number of coefficients computed.
    ;          Parameters:       p, the number of coefficients computed.
    ;          Input:            Arrives at the north port of cell (1,2)
    ;          Output:           Departs from east port of merge cell
    ;          Loop Time:        26 μs to process one input sample
    ;          Max Sample Rate:  38 KHz
    ;
    ;          Sends data through the switch LSB first
    ;
    ;          Use with runrev.o eproms
    ;

    ;          Warp Drive version of auto.s
    ;          Read 8 bit inputs and produces 16 bit outputs
    ;          Must be used with 16 reversed eproms, which were never made

    #include "ports.h"

    ;          Memory map for internal RAM
               org               29h
    ;
    ; 18       sint              i;
    ;
    i:         ds                1
    ;
    ; 19       int               top,left,sum;
    ;
    sum:       ds                2
    top:       ds                1
    left:      ds                1

               org               8000h
    ;
    ; 22       i := 0;
    ;
1              clr               a
1              mov               i,a            ; i := 0
    ;
```

Figure B.10 8051 program for autocorrelation program a4 using 8-bit inputs and 16-bit sums.

```
1                  nop
1                  nop
1                  nop
1                  nop
1                  nop
1                  nop
1                  nop
1                  nop
2                  movx            @dptr,a         ; Send LSB of sum
        ;
        ; 44       i := 0;
        ;
1                  clr             a
1                  mov             i,a             ; i := 0
        ;
        ; 42       sum := 0;
        ;
1                  mov             sum,a           ; sum := 0
1                  mov             sum+1,a
1                  clr             a               ; Send a 0 to port 4
        ;
        ; 43       out <- sum;
        ;
1                  nop                             ; wait for switch to get ready
1                  nop                             ; Total of 12 micro seconds
1                  nop                             ; between writes to switch
1                  nop
1                  nop
1                  nop
1                  nop
2                  mov             p1,#south
2                  movx            @dptr,a         ; write to switch
        ;
        ; 45       end
        ;
2                  sjmp            main

        hop:
        ;
        ; 31       left <- in2;
        ;
2                  movx            a,@dptr         ; Read input byte from switch
1                  mov             b,a             ; Assume it's from left
        ;
        ; 30       top <- in1;
        ;
2                  movx            a,@dptr         ; Read input byte from switch
        ;
        ; 35       out <- top;
        ;
```

Figure B.10 (Continued)

```
        ; 23        sum := 0;
        ;
1                   mov         sum,a           ; sum := 0
1                   mov         sum+1,a
        ;
        ; 25        out <- sum;
        ;
2                   mov         p1,#south       ;       Write sum to south port
2                   mov         dptr,#lowSWLat
2                   movx        @dptr,a         ; write 0 to switch

        main:
        ;
        ; 29        i := i + 1;
        ;
1                   inc         i       ; i := i + 1
        ;
        ; 33        if i < samples then
        ;
1                   mov         a,i             ; load i
2                   cjne        a,#5,hop        ; if(i != 5) goto hop
        ;
        ; 31        left <- in2;
        ;
2                   movx        a,@dptr         ; Read input byte from switch
1                   mov         b,a             ; Assume it's from left
        ;
        ; 32        top <- in1;
        ;
2                   movx        a,@dptr         ; Read input byte from switch
4                   mul         ab
        ;
        ; 40        sum := sum + top * left;
        ;
1                   add         a,sum+1                 ; Add prod to lower byte of sum
        ;
        ; 41        results <- sum;
        ;
2                   mov         p1,#east        ; Write to east port (2)
2                   movx        @dptr,a         ; Send MSB of sum
        ;
        ; 40        sum := sum + top * left; (cont)
        ;
1                   mov         sum+1,a
1                   mov         a,b
1                   addc        a,sum           ; Add prod to upper byte of sum
1                   mov         sum,a
        ;
        ; 41        results <- sum;  (cont)
        ;
```

Figure B.10 (Continued)

```
2                 movx          @dptr,a        ; write to switch<— Write port
        ;
        ; 36      sum := sum + top * left;
        ;
4                 mul           ab
1                 add           a,sum+1               ; Add prod to lower byte of sum
1                 mov           sum+1,a
1                 mov           a,b
1                 addc          a,sum          ; Add prod to upper byte of sum
1                 mov           sum,a
        ;
        ; 37      end
        ;
2                 sjmp          main

                  end
```

Figure B.10 (Continued)

```
Jul 12 12:56  1984 auto.s Page 1

        ;               Program Name:     auto (a5)
        ;               Algorithm:        Figure 6.12
        ;               Machine:          VLSI processor array, simulated by Poker.
        ;               Function:         Find autocorrelation coefficients R(i)
        ;                                 given input signal x(m), using
                                                  k=M-i-1
        ;                                 R(i)=     Σ    x(k)x(k+i).
                                                   k=0
        ;
        ;               Precision:        Input:  16-bit unsigned.
        ;                                 Output: 32-bit unsigned.
        ;               Number of PEs:    p, the number of coefficients computed.
        ;               Parameters:       p, the number of coefficients computed.
        ;               Input:            Arrives at the north port of cell (1,3).
        ;               Output:           Departs from east port of merge cell.
        ;               Loop Time:        90 µs to process one input sample.
        ;               Max Sample Rate:  11 KHz
        ;
        ;               Sends data through the switch in reverse order
        ;               (i.e. LSB first)
        ;               Use with runrev.o eproms
        ;               Quasi synchronous.  i.e. Can take input from external source
        ;                                                      at unknown intervals
        ;

        samples         equ     5               ; Number of sample per frame

        #include "ports.h"

                        org     029h
        ;
        ; 18            sint    i;
        ;
        i:              ds      1
        ;
        ; 19            int     sum,top,left;
        ;
        sum:            ds      4
        top:            ds      2
        left:           ds      2

                        org     8000h
        ;
        ; 25            out <- sum;
        ;
2                       mov     p1,#south       ;       Write 0 to south port
```

Figure B.11 8051 program for autocorrelation program a5, using asynchronous 16-bit input and 32-bit output.

```
Jul 12 12:56  1984 auto.s Page 2
2                   mov     dptr,#lowSWLat        ; This value will remain in dptr
                                                  ; from now one

1                   clr     a
2                   movx    @dptr,a          ; write 0 to switch

1                   mov     r0,#5
2                   djnz    r0,$             ; Wait 12 microseconds for switch

1                   clr     a
2                   movx    @dptr,a          ; write 0 to switch
        ;
        ; 22        i := 0;
        ;
1                   clr     a
1                   mov     i,a              ; i := 0
        ;
        ; 23        sum := 0;
        ;
1                   mov     sum+3,a               ; sum := 0
1                   mov     sum+2,a
1                   mov     sum+1,a
1                   mov     sum,a

        main:
        ;
        ; 29        i := i + 1;
        ;
1                   inc     i                ; i := i + 1
        ;
        ; 30        top <- in1;
        ;
2                   jnb     p0.7,$           ; Wait for input from external program
2                   movx    a,@dptr          ; Read LSB of top from switch
1                   mov     top+1,a
2                   jnb     p0.7,$           ; Wait for input from external program
2                   movx    a,@dptr          ; Read MSB of top from switch
1                   mov     top,a
        ;
        ; 30        left <- in2;
        ;
2                   jnb     p0.7,$           ; Wait for input from external program
2                   movx    a,@dptr          ; Read LSB of left from switch
1                   mov     left+1,a
2                   jnb     p0.7,$           ; Wait for input from external program
2                   movx    a,@dptr          ; Read MSB of left from switch
1                   mov     left,a
        ;
```

Figure B.11 (Continued)

Jul 12 12:56  1984 auto.s Page 3

```
        ; 33            if i < samples then
        ;
1                       mov     a,i                 ; load i
2                       cjne    a,#samples,loop; if(i != samples) goto loop
2                       ljmp    endloop             ; if( i == samples) goto endloop

        loop:

        ;               sum := sum + left * top
        ;               Where sum is 32 bits and left and top are 16 bits
        ;
        ;                  30     31     (left)
        ;          X       2e     2f     (top)
        ;          -----------------
        ; +               30x2f  31x2f
        ; +        30x2e  31x2e
        ; + 2a       2b    2c     2d     (sum)
        ;
        ;
        ; 36            sum := sum + top * left;
        ;
1                       mov     a,left+1            ; LSB of left
2                       mov     b,top+1                     ; LSB of top
4                       mul     ab
1                       add     a,sum+3                     ; LSB of sum (byte 4)
1                       mov     sum+3,a
1                       mov     a,b
1                       addc    a,sum+2                     ; add in byte 3 of sum
1                       mov     sum+2,a
1                       clr     a
1                       addc    a,sum+1                     ; add carry to byte 2 of sum
1                       mov     sum+1,a
1                       clr     a
1                       addc    a,sum               ; add carry to MSB of sum (byte 1)
1                       mov     sum,a
1                       mov     a,top+1                     ; LSB of top
        ;
        ; 35            out <- top;
        ;
2                       movx    @dptr,a             ; Send LSB of top to south port
        ;
        ; 36            sum := sum + top * left;     (cont)
        ;
2                       mov     b,left              ; MSB of left
4                       mul     ab
1                       add     a,sum+2                     ; add to byte 3 of sum
1                       mov     sum+2,a
1                       mov     a,b
```

Figure B.11 (Continued)

```
Jul 12 12:56  1984 auto.s Page 4

1                       addc    a,sum+1                     ; add to byte 2 of sum
1                       mov     sum+1,a
1                       clr     a
1                       addc    a,sum           ; add carry to byte 1 of sum
1                       mov     sum,a

1                       mov     a,top           ; MSB of top
        ;
        ; 35            sum <- top;   (cont)
        ;
2                       movx    @dptr,a         ; Send MSB of top to south port
        ;
        ; 36            sum := sum + top * left;     (cont)
        ;
2                       mov     b,left+1        ; LSB of left
4                       mul     ab
1                       add     a,sum+2                     ; add to byte 3 of sum
1                       mov     sum+2,a
1                       mov     a,b
1                       addc    a,sum+1                     ; add to byte 2 of sum
1                       mov     sum+1,a
1                       clr     a
1                       addc    a,sum           ; add carry to byte 1 of sum
1                       mov     sum,a

1                       mov     a,top           ; MSB of top
2                       mov     b,left          ; LSB of left
4                       mul     ab
1                       add     a,sum+1                     ; add to byte 2 of sum
1                       mov     sum+1,a
1                       mov     a,b
1                       addc    a,sum           ; add to MSB of sum (byte 1)
1                       mov     sum,a
        ;
        ; 37            end
        ;
2                       ljmp    main

        endloop:
        ;
        ; 43            out <- sum;
        ;
1                       clr     a               ; Send a 0 to south port bottom <- 0
2                       movx    @dptr,a         ; write to switch

1                       mov     r0,#5
2                       djnz    r0,$            ; Wait 12 microseconds for switch
```

Figure B.11 (Continued)

```
Jul 12 12:56  1984 auto.s Page 5

1                       clr     a                 ; Send a 0 to port 4
2                       movx    @dptr,a           ; write to switch

        ;               sum := sum + left * top
        ;               Where sum is 32 bits and left and top are 16 bits
        ;
        ;                 30      31    (left)
        ;           X     2e      2f    (top)
        ;
        ;          -----------------
        ; +               30x2f  31x2f
        ; +         30x2e 31x2e
        ; + 2a      2b    2c      2d    (sum)
        ;
        ;
        ; 40            sum := sum + top * left;
        ;
1                       mov     a,left+1          ; LSB of left
2                       mov     b,top+1                   ; LSB of top
4                       mul     ab
1                       add     a,sum+3                   ; LSB of sum (byte 4)
1                       mov     sum+3,a
        ;
        ; 41            results <- sum;
        ;
2                       mov     p1,#east          ; Next write is to east port
2                       movx    @dptr,a           ; Send LSB of sum to east port
        ;
        ; 40            sum := sum + top * left;        (cont)
        ;
1                       mov     a,b
1                       addc    a,sum+2                   ; add in byte 3 of sum
1                       mov     sum+2,a
1                       clr     a
1                       addc    a,sum+1                   ; add carry to byte 2 of sum
1                       mov     sum+1,a
1                       clr     a
1                       addc    a,sum             ; add carry to MSB of sum (byte 1)
1                       mov     sum,a

1                       mov     a,top+1                   ; LSB of top
2                       mov     b,left            ; MSB of left
4                       mul     ab
1                       add     a,sum+2                   ; add to byte 3 of sum
1                       mov     sum+2,a
1                       mov     a,b
1                       addc    a,sum+1                   ; add to byte 2 of sum
1                       mov     sum+1,a
1                       clr     a
```

Figure B.11 (Continued)

```
1                 addc   a,sum           ; add carry to byte 1 of sum
1                 mov    sum,a

1                 mov    a,top           ; MSB of top
2                 mov    b,left+1        ; LSB of left
4                 mul    ab
1                 add    a,sum+2                ; add to byte 3 of sum
1                 mov    sum+2,a
          ;
          ; 41      results <- sum;        (cont)
          ;
2                 movx   @dptr,a         ; Send third byte of sum to east port
          ;
          ; 40      sum := sum + top * left;     (cont)
          ;
1                 mov    a,b
1                 addc   a,sum+1                ; add to byte 2 of sum
1                 mov    sum+1,a
1                 clr    a
1                 addc   a,sum           ; add carry to byte 1 of sum
1                 mov    sum,a

1                 mov    a,top           ; MSB of top
2                 mov    b,left          ; MSB of left
4                 mul    ab
1                 add    a,sum+1                ; add to byte 2 of sum
          ;
          ; 41      results <- sum;        (cont)
          ;
2                 movx   @dptr,a         ; Send second byte of sum to east port
          ;
          ; 40      sum := sum + top * left;     (cont)
          ;
1                 mov    sum+1,a
1                 mov    a,b
1                 addc   a,sum           ; add to MSB of sum (byte 1)
1                 mov    sum,a
          ;
          ; 41      results <- sum;        (cont)
          ;
1                 nop                    ; Wait 12 microseconds for switch
1                 nop
1                 nop
1                 nop
1                 nop
1                 nop
1                 nop
1                 nop
```

Figure B.11 (Continued)

Jul 12 12:56 1984 auto.s Page 7

```
2                   movx   @dptr,a         ; Send second byte of sum to east port

        ;
        ;           Initialize i and sum for next autocorrelation calculation
        ;
        ;
        ; 44        i := 0;
        ;
1                   clr    a
1                   mov    i,a             ; i := 0
        ;
        ; 42        sum := 0;
        ;
1                   mov    sum+3,a                ; sum := 0
1                   mov    sum+2,a
1                   mov    sum+1,a
1                   mov    sum,a

2                   mov    p1,#south
        ;
        ; 45        end
        ;
2                   ljmp   main

                    end
```

Figure B.11 (Continued)

Jul 12 12:56 1984 split.s Page 1

```
        ;          Program Name:    split
        ;          Algorithm:       None
        ;          Machine:         VLSI processor array, simulated by Poker
        ;          Function:        Take the input data from one input port
        ;                           and write it to two output ports, spliting
        ;                           the input data stream
        ;          Number of PEs:   1
        ;          Parameters:      ARG1 as set in the code names file
        ;          Input:           If ARG1 == 100 it inputs two bytes,
        ;                           then outputs the same bytes first
        ;                           to the up port, then the down port
        ;                           If ARG! != 100, it inputs one byte
        ;                           and outputs it to the up port,
        ;                           then the down port.
        ;          Output:          if ARG1 == 1 up is ne    and down is east
        ;                           if ARG1 == 2 up is north and down is south
        ;                           if ARG1 == 3 up is nw    and down is sw
        ;                           if ARG1 ==100up is ne    and down is east

        #include "./ports.h"

                   org    029h
        input:     ds     2
        up:        ds     1
        down:      ds     1

                   org    8000h
2                  mov    dptr,#ARG1+3       ; Check first parameter to see where to send
2                  movx   a,@dptr
2                  mov    dptr,#lowSWLat     ; Get switch address
2                  mov    p0,#0f0h

2                  cjne   a,#1,next
2                  mov    up,#ne          ; param==1, up is ne
2                  mov    down,#east      ;           down is east
2                  sjmp   main
        next:
2                  cjne   a,#2,next2
2                  mov    up,#north       ; param==2, up is north
2                  mov    down,#south     ;           down is south
2                  sjmp   main
        next2:
2                  cjne   a,#3,next4
2                  mov    up,#nw          ; param==3, up is nw
2                  mov    down,#sw        ;           down is sw

        main:
2                  mov    p1,up   ; Set direction up
```

Figure B.11 (Continued)

```
Jul 12 12:56  1984 split.s Page 2

2                      jnb     p0.7,$          ; Wait for input
        ;              dump    4,1
        ;              db      0a5h
        ;              dw      4
        ;              dw      1

2                      movx    a,@dptr         ; Read byte, and write it out again
2                      movx    @dptr,a
2                      lcall   writedelay
1                      nop

2                      mov     p1,down                  ; Set direction to down
2                      movx    @dptr,a         ; Send out second byte
2                      lcall   writedelay

2                      sjmp    main

        next4:
2                      mov     up,#ne          ; This split is for pe1,1
2                      mov     down,#east
        main2:
2                      mov     p1,up           ; Set direction up

2                      jnb     p0.7,$          ; Wait for input
        ;              dump    4,1
        ;              db      0a5h
        ;              dw      4
        ;              dw      1

2                      movx    a,@dptr
1                      mov     input+1,a

2                      jnb     p0.7,$          ; Wait for input
2                      movx    a,@dptr
1                      mov     input,a

1                      mov     a,input+1       ; Send out first byte
2                      movx    @dptr,a
2                      lcall   writedelay

1                      mov     a,input         ; Send out second byte
2                      movx    @dptr,a
2                      lcall   writedelay

1                      mov     a,input+1
2                      mov     p1,down                  ; Set direction to 2 (down)
2                      movx    @dptr,a
2                      lcall   writedelay
```

Figure B.11 (Continued)

```
Jul 12 12:56  1984 split.s Page 3

1                      mov     a,input
2                      movx    @dptr,a
2                      lcall   writedelay

2                      sjmp    main2


        #include "./util.h"

                       end
```

Figure B.11 (Continued)

Jul 12 12:57 1984 merge.x Page 1

```
/*
                This routine will merge two data streams into one by
                taking interlace number of data from the top, then
                interlace number for the bottom.
                Kludge: if interlace is 4, all data is read from
                top is outputed, before reading data from the bottom.
*/
code            merge(interlace);
trace           tmp;
ports           top,bottom,out;
begin
                sint    i;
                int     tmp,
                        top,bottom,out,
                        interlace,
                        bottomhold[4],
                        tophold[4];

                if interlace = 4 then
                        while true do
                                begin
                                for i := 1 to interlace do
                                        begin
                                        tmp <- top;
                                        out <- tmp;
                                        end;
                                tmp <- bottom;
                                out <- tmp;
                                end
                else
                        while true do
                           begin
                           for i := 1 to interlace do
                                   begin
                                   tmp <- top; tophold[i] := tmp;
                                   tmp <- bottom;      bottomhold[i] := tmp;
                                   end;
                           for i := 1 to interlace do
                                        out <- tophold[i];
                                for i := 1 to interlace do
                                        out <- bottomhold[i];
                           end;
                end.
```

Figure B.11 (Continued)

```
Jul 11 10:52 1984 even.s Page 1

        ;             Program Name:   dtw, even.s (d2)
        ;             Algorithm:      Figure 6.23
        ;             Machine:        VLSI processor array, simulated by Poker
        ;             Function:       Match two utterance using dynamic time warping
        ;             Precision:      Input coefficients:      8 bits, unsigned
        ;                             Distances:     16 bits, unsigned
        ;                             Output score:  16 bits, unsigned
        ;             Number of PEs:  2r+1, where r is the width of the warping path
        ;             Parameters:     r, width of the warping path
        ;                             coefs, the number of coefficients per frame
        ;                             I, the number of frames per utterance
        ;             Input:          a vectors enter cells (1,7) and (1,8)
        ;                             b vectors enter cells (7,7) and (6,8)
        ;             Output:         scores appear in cell (4,6)

        #include "dtw.h"
        #include "init.h"

        ;             Start of main loop, read a then b and find distance

        main:
        ;
        ; 46          while true do
        ;
1                     clr     a
1                     mov     tcon,a            ; Stop timer
1                     mov     tl1,a             ; Clear timer
1                     mov     th1,a
1                     setb    tcon.6            ; Start timer
        ;
        ; 49          for l := 1      to coefs do
        ;
1                     mov     r0,#avec          ; r0 points to current a coef.
1                     mov     r1,#bvec          ; r1 points to current b coef.
1                     mov     r2,#COEFS
        ;
        ; 48          d := 0;
        ;
2                     mov     d+1,#0                      ; d := 0
2                     mov     d,#0

        input:
        ;
        ; 51          aout <- a[i];
        ;
```

Figure B.12  8051 routine used for DTW program d2.

Jul 11 10:52 1984 even.s Page 2

```
1                 mov     a,@r0           ; get avec[?]
2                 mov     p1,#south       ; Set direction of write
    #ifdef BOTTOM
1                 nop
1                 nop
    #else
2                 movx    @dptr,a         ; send avec[?] to switch
    #endif
2                 lcall   writedelay
    ;
    ; 51            bout <- b[i];
    ;
1                 mov     a,@r1           ; Send bvec[?]
2                 mov     p1,#nw          ; Set new direction
    #ifdef TOP
1                 nop
1                 nop
    #else
2                 movx    @dptr,a         ; send avec[?] to switch
    #endif
    ;
    ; 53            atmp <- ain;  a[i] := atmp;
    ;
2                 movx    a,@dptr         ; get a coef
1                 mov     @r0,a           ; save for next time
1                 inc     r0
1                 mov     b,a

1                 mov     r3,#4           ; Wait 9 more uS for b values
2                 djnz    r3,$
    ;
    ; 54            btmp <- bin;  b[i] := btmp;
    ;
2                 movx    a,@dptr         ; get b coef
1                 mov     @r1,a
1                 inc     r1

    ;             Find local distance d
    ;
    ; 56            tmp1  := atmp - btmp;
    ;
1                 clr     c
1                 subb    a,b             ; acc := a - b
2                 jnb     acc.7,dist2     ; take absolute value of acc
1                 cpl     a
1                 inc     a
    dist1:
    ;
```

Figure B.12 (Continued)

```
Jul 11 10:52 1984 even.s Page 3

        ; 57            d := d + tmpl * tmpl;
        ;
1                       mov     b,a             ; acc := acc * acc
4                       mul     ab
1                       add     a,d+1
1                       mov     d+1,a           ; add to d
1                       mov     a,b
1                       addc    a,d
1                       mov     d,a
        ;
        ; 58            end
        ;
2                       djnz    r2,input

        ;               Check to see if any vecter is inf
        ;
        ; 61            if (a[1] = inf) | b[1]  inf) then
        ;
1                       mov     a,inf8
2                       cjne    a,avec,bcheck   ; Is avec == inf?
2                       mov     d+1,inf+1       ;   d := inf
2                       mov     d,inf
2                       sjmp    dout
        dist2:
2                       sjmp    dist1           ; dummy jump to kept time the same
        doutdelay:
1                       nop                     ; This is so all paths have the same length
1                       nop
2                       sjmp    dout
        bcheck:
2                       cjne    a,bvec,doutdelay        ; Is bvec == inf?
        ;
        ; 62            d := inf;
        ;
2                       mov     d+1,inf+1       ;   d := inf
2                       mov     d,inf

        ;               Send out d values

        dout:
        ;
        ; 64            DTtop <- d;
        ;
2                       mov     p1,#ne          ; DTtop <- d
1                       mov     a,d+1
        #ifdef TOP
1                       nop
1                       nop
```

Figure B.12 (Continued)

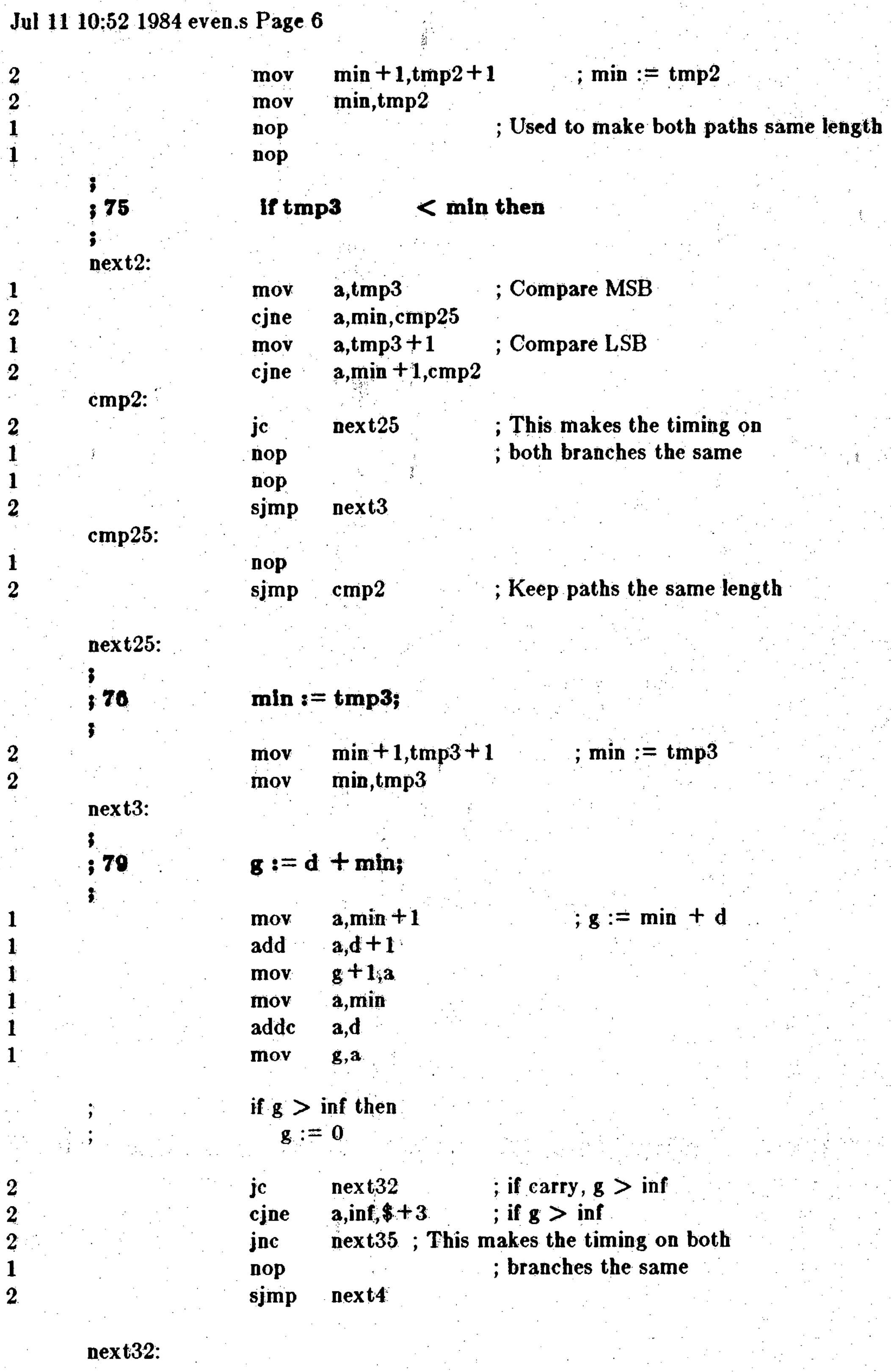
```
Jul 11 10:52 1984 even.s Page 4

2                  lcall   writedelay
1                  mov     a,d
1                  nop
1                  nop
2                  lcall   writedelay
       #else
2                  movx    @dptr,a         ; Send LSB of d to switch
2                  lcall   writedelay
1                  mov     a,d
2                  movx    @dptr,a         ; Send MSB of d to switch
2                  lcall   writedelay
       #endif
       ;
       ; 65        DTbot <- d;
       ;
1                  mov     a,d+1
2                  mov     p1,#east        ; DTbot <- d
       #ifdef BOTTOM
1                  nop
1                  nop
2                  lcall   writedelay
1                  mov     a,d
1                  nop
1                  nop
       #else
2                  movx    @dptr,a         ; Send LSB of d to switch
2                  lcall   writedelay
1                  mov     a,d
2                  movx    @dptr,a         ; Send MSB of d to switch
       #endif

       ;           Find path with shortest total sum

       minpath:
       ;
       ; 67        tmp1   := Gbotold + 2*Dbot;
       ;
1                  mov     a,Dbot+1        ; tmp1 := Gbotold + 2*Dbot
1                  rl      a
1                  mov     tmp1+1,a        ;          tmp1 := 2*Dbot
1                  mov     a,Dbot
1                  rlc     a
1                  mov     tmp1,a
1                  mov     a,tmp1+1
1                  add     a,Gbotold+1
1                  mov     tmp1+1,a
1                  mov     a,Gbotold
1                  addc    a,tmp1
```

Figure B.12 (Continued)

Jul 11 10:52 1984 even.s Page 5

```
1                   mov     tmp1,a
        ;
        ; 68        tmp2    := g + d;
        ;
1                   mov     a,d+1           ; tmp2 := g + d
1                   add     a,g+1
1                   mov     tmp2+1,a
1                   mov     a,d
1                   addc    a,g
1                   mov     tmp2,a
        ;
        ; 69        tmp3    := Gtopold + 2*Dtop;
        ;
1                   mov     a,Dtop+1        ; tmp3 := Gtopold + 2*Dtop
1                   rl      a
1                   mov     tmp3+1,a
1                   mov     a,Dtop
1                   rlc     a
1                   mov     tmp3,a
1                   mov     a,tmp3+1
1                   add     a,Gtopold+1
1                   mov     tmp3+1,a
1                   mov     a,Gtopold
1                   addc    a,tmp3
1                   mov     tmp3,a
        ;
        ; 71        if tmp1         < tmp2 then
        ;
1                   mov     a,tmp1          ; compare MSB
2                   cjne    a,tmp2,cmp15
1                   mov     a,tmp1+1
2                   cjne    a,tmp2+1,cmp1; compare LSB
        cmp1:
2                   jnc     next1
        ;
        ; 72        min := tmp1;
        ;
2                   mov     min+1,tmp1+1          ; min := tmp1
2                   mov     min,tmp1
2                   sjmp    next2
        cmp15:
1                   nop
2                   sjmp    cmp1            ; Keep paths same length

        next1:
        ;
        ; 74        min := tmp2;
        ;
```

Figure B.12 (Continued)



```
Jul 11 10:52 1984 even.s Page 6

2                    mov    min+1,tmp2+1         ; min := tmp2
2                    mov    min,tmp2
1                    nop                   ; Used to make both paths same length
1                    nop
       ;
       ; 75           if tmp3       < min then
       ;
       next2:
1                    mov    a,tmp3         ; Compare MSB
2                    cjne   a,min,cmp25
1                    mov    a,tmp3+1       ; Compare LSB
2                    cjne   a,min+1,cmp2
       cmp2:
2                    jc     next25         ; This makes the timing on
1                    nop                   ; both branches the same
1                    nop
2                    sjmp   next3
       cmp25:
1                    nop
2                    sjmp   cmp2           ; Keep paths the same length

       next25:
       ;
       ; 76          min := tmp3;
       ;
2                    mov    min+1,tmp3+1         ; min := tmp3
2                    mov    min,tmp3
       next3:
       ;
       ; 79          g := d  + min;
       ;
1                    mov    a,min+1              ; g := min  + d
1                    add    a,d+1
1                    mov    g+1,a
1                    mov    a,min
1                    addc   a,d
1                    mov    g,a

       ;             if g > inf then
       ;                g := 0

2                    jc     next32         ; if carry, g > inf
2                    cjne   a,inf,$+3      ; if g > inf
2                    jnc    next35  ; This makes the timing on both
1                    nop                   ; branches the same
2                    sjmp   next4

       next32:
```

Figure B.12 (Continued)

Jul 11 10:52 1984 even.s Page 7

```
1                   nop
1                   nop
1                   nop
1                   nop
      next35:
      ;
      ; 81          g := 0;
      ;
1                   clr     a               ; g := 0
1                   mov     g+1,a
1                   mov     g,a

      next4:
      ;
      ; 89          GTtop <- g;
      ;
2                   mov     p1,#ne          ; DTtop <- g
1                   mov     a,g+1
      #ifdef        TOP
1                   nop
1                   nop
2                   lcall   writedelay
1                   mov     a,g
1                   nop
1                   nop
2                   lcall   writedelay
      #else
2                   movx    @dptr,a
2                   lcall   writedelay
1                   mov     a,g
2                   movx    @dptr,a
2                   lcall   writedelay
      #endif
      ;
      ; 90          DTbot <- g;
      ;
1                   mov     a,g+1
2                   mov     p1,#east        ; DTbot <- g
      #ifdef        BOTTOM
1                   nop
1                   nop
2                   lcall   writedelay
1                   mov     a,g
1                   nop
1                   nop
      #else
2                   movx    @dptr,a
2                   lcall   writedelay
```

Figure B.12 (Continued)

```
Jul 11 10:52 1984 even.s Page 8

1                   mov     a,g
2                   movx    @dptr,a
        #endif
        ;
        ; 83        Gtopold := Gtop;
        ;
2                   mov     Gtopold+1,Gtop+1    ; Gtopold := Gtop
2                   mov     Gtopold,Gtop
        ;
        ; 84        Gbotold := Gbot;
        ;
2                   mov     Gbotold+1,Gbot+1    ; Gbotold := Gbot
2                   mov     Gbotold,Gbot
        ;
        ; 92        Dtop <- DTtop;
        ;
        #ifdef      TOP
2                   mov     Dtop+1,inf+1  ; Dtop <- inf
2                   mov     Dtop,inf
1                   nop
1                   nop
        #else
2                   movx    a,@dptr       ; Dtop <- DTtop
1                   mov     Dtop+1,a
2                   movx    a,@dptr
1                   mov     Dtop,a
        #endif
        #ifdef      OUTPUT
1                   mov     a,g+1         ; Send data to scores cell
1                   mov     p1,#west
2                   mov     @dptr,a
        #endif
        ;
        ; 93        Dbot <- DTbot;
        ;
        #ifdef      BOTTOM
2                   mov     Dbot+1,inf+1  ; Dbot <- inf
2                   mov     Dbot,inf
1                   nop
1                   nop
        #else
2                   movx    a,@dptr       ; Dbot <- DTbot
1                   mov     Dbot+1,a
2                   movx    a,@dptr
1                   mov     Dbot,a
        #endif
        ;
        ; 86        Dtop <- DTtop;
```

Figure B.12 (Continued)

Jul 11 10:52 1984 even.s Page 9

```
        ;
        #ifdef          TOP
2                       mov     Gtop+1,inf+1    ; Gtop <- inf
2                       mov     Gtop,inf
1                       nop
1                       nop
        #else
2                       movx    a,@dptr         ; Gtop <- DTtop
1                       mov     Gtop+1,a
2                       movx    a,@dptr
1                       mov     Gtop,a
        #endif
        #ifdef          OUTPUT
1                       mov     a,g             ; Send data to scores cell
2                       movx    @dptr,a
        #endif
        ;
        ; 87            Gbot <- DTbot;
        ;
        #ifdef          BOTTOM
2                       mov     Gbot+1,inf+1    ; Gbot <- inf
2                       mov     Gbot,inf
1                       nop
1                       nop
        #else
2                       movx    a,@dptr         ; Gbot <- DTbot
1                       mov     Gbot+1,a
2                       movx    a,@dptr
1                       mov     Gbot,a
        #endif
        ;
        ; 95            end
        ;
1                       mov     a,#low(LOOPTIME);   Wait for timer
1                       xrl     a,tl1           ; xor LSBs to see if they are the same
1                       rrc     a               ; move LSB into carry bit
1                       mov     a,#low(LOOPTIME)
2                       jnc     sync            ; Sync with timer, since the cjne takes
1                       nop                     ; 2 uS, there is a 50/50 chance the LSB
                                                ; will not match, this comparison should
                                                ; sync the program up with the timer so
                                                ; with LSB will always match
        sync:
2                       cjne    a,tl1,$

2                       ljmp    main
        #include "util.h"
                        end
```

Figure B.12 (Continued)

Jul 11 10:52 1984 even.s Page 10

```
        ;               Program Name:   dtw, odd.s (d2)
        ;               Algorithm:      Figure 6.16??
        ;               Machine:        VLSI processor array, simulated by Poker
        ;               Function:       Match two utterance using dynamic time warping
        ;               Precision:      Input coefficients:     8 bits, unsigned
        ;                               Distances:      16 bits, unsigned
        ;               Number of PEs:  2r+1, where r is the width of the warping path
        ;               Parameters:     r, width of the warping path
        ;                               coefs, the number of coefficients per frame
        ;                               I, the number of frames per utterance
        ;               Input:          a vectors enter cells (1,7) and (1,8)
        ;                               b vectors enter cells (7,7) and (6,8)
        ;               Output:         scores appear in cell (4,6)

        #include "ports.h"
        #include "dtw.h"
        #include "init.h"

        ;               Start of main loop, read a then b and find distance

        main:
1                       clr     a
1                       mov     tcon,a          ; Stop timer
1                       mov     tl1,a           ; clear timer
1                       mov     th1,a
1                       setb    tcon.6          ; Start timer

        ;
        ; 44            for l := 1 to coefs do
        ;
1                       mov     r0,#avec        ; r0 points to current a coef.
1                       mov     r1,#bvec        ; r1 points to current b coef.
1                       mov     r2,#COEFS
2                       mov     d+1,#0                  ; d := 0
2                       mov     d,#0
        input:
        ;
        ; 46            aout <- a[l];
        ;
1                       mov     a,@r0           ; get avec[?]
2                       mov     p1,#south       ; Set direction of write
        #ifdef          BOTTOM
1                       nop
1                       nop
        #else
2                       movx    @dptr,a         ; send avec[?] to switch
        #endif
2                       lcall   writedelay
```

Figure B.12 (Continued)

Jul 11 11:36 1984 odd.s Page 1

```
        ;
        ; 47          bout <- b[i];
        ;
1                     mov    a,@r1          ; Send bvec[?]
2                     mov    p1,#ne         ; Set new direction
        #ifdef        TOP
1                     nop
1                     nop
        #else
2                     movx   @dptr,a
        #endif
1                     nop

        ;
        ; 48          atmp <- ain
        ;
2                     movx   a,@dptr        ; get a coef
        ;
        ; 48          a[i] := atmp;
        ;
1                     mov    @r0,a          ; save for next time
1                     inc    r0
1                     mov    b,a

1                     mov    r3,#4          ; Wait 9 more uS for b values
2                     djnz   r3,$

1                     nop

        ;
        ; 49          btmp <- bin;
        ;
2                     movx   a,@dptr        ; get b coef
        ;
        ; 49          b[i] := btmp;
        ;
1                     mov    @r1,a
1                     inc    r1
        ;
        ; 51          tmp1 := atmp - btmp;      /* Compute distance between vectors */
        ;
        ;             Find the local distance between avec and bvec

1                     clr    c
1                     subb   a,b            ; acc := a - b
2                     jnb    acc.7,dist2    ; take absolute value of acc
1                     cpl    a
1                     inc    a
```

Figure B.12 (Continued)

Jul 11 11:36 1984 odd.s Page 2

```
        ;
        ; 52            d := d + tmp1 * tmp1
        ;
        dist1:
1                       mov     b,a             ; acc := acc * acc
4                       mul     ab
1                       add     a,d+1
1                       mov     d+1,a           ; add to d
1                       mov     a,b
1                       addc    a,d
1                       mov     d,a
        ;
        ; 53            end;
        ;
2                       djnz    r2,input
        ;
        ; 56            if (a[1] = inf) | (b[1] = inf) then
        ;
        ;               Check for infinity

1                       mov     a,inf8
2                       cjne    a,avec,bcheck   ; Is avec == inf?
2                       mov     d+1,inf+1       ;   d := inf
2                       mov     d,inf
2                       sjmp    din
        dist2:
2                       sjmp    dist1           ; dummy jump to kept time the same
        dindelay:
1                       nop                     ; This is so all pathes have the same length
1                       nop
2                       sjmp    din
        ;
        ; 56            if (a[1] = inf) | (b[1] = inf) then       (cont)
        ;
        bcheck:
2                       cjne    a,bvec,dindelay ; Is bvec == inf?
        ;
        ; 57            d := inf;
        ;
2                       mov     d+1,inf+1       ;   d := inf
2                       mov     d,inf

        ;
        ; 84            DTbot <- d;
        ;
        ;               Get and send d values

        din:
```

Figure B.12 (Continued)

```
Jul 11 11:36 1984 odd.s Page 3

1                     mov     a,d+1
2                     mov     p1,#sw          ; DTbot <- d
2                     movx    @dptr,a
2                     lcall   writedelay
1                     mov     a,d
2                     movx    @dptr,a
2                     lcall   writedelay
        ;
        ; 85          DTtop <- d;
        ;
1                     mov     a,d+1
2                     mov     p1,#west        ; DTtop <- d
2                     movx    @dptr,a
2                     lcall   writedelay
1                     mov     a,d
2                     movx    @dptr,a
        ;
        ; 63          tmp2 := g + d;
        ;
1                     mov     a,d+1           ; tmp2 := g + d
1                     add     a,g+1
1                     mov     tmp2+1,a
1                     mov     a,d
1                     addc    a,g
1                     mov     tmp2,a
        ;
        ; 59                  Dbot <- DTbot;
        ;
2                     movx    a,@dptr         ; Dbot <- DTbot
1                     mov     Dbot+1,a
2                     movx    a,@dptr
1                     mov     Dbot,a
        ;
        ; 62          tmp1 := Gbot+2*Dbot;      /* Find minimum path          */
        ;
1                     mov     a,Dbot+1        ; tmp1 := Gbot + 2*Dbot
1                     rl      a
1                     mov     tmp1+1,a        ;         tmp1 := 2*Dbot
1                     mov     a,Dbot
1                     rlc     a
1                     mov     tmp1,a
1                     mov     a,tmp1+1
1                     add     a,Gbot+1
1                     mov     tmp1+1,a
1                     mov     a,Gbot
1                     addc    a,tmp1
1                     mov     tmp1,a
```

Figure B.12 (Continued)

Jul 11 11:36 1984 odd.s Page 4

```
        ;
        ; 60              Dtop <- DTtop;
        ;
2                         movx    a,@dptr          ; Dtop <- DTtop
1                         mov     Dtop+1,a
2                         movx    a,@dptr
1                         mov     Dtop,a
        ;
        ; 64              tmp3 := Gtop+2*Dtop;
        ;
1                         mov     a,Dtop+1         ; tmp3 := Gtop + 2*Dtop
1                         rl      a
1                         mov     tmp3+1,a
1                         mov     a,Dtop
1                         rlc     a
1                         mov     tmp3,a
1                         mov     a,tmp3+1
1                         add     a,Gtop+1
1                         mov     tmp3+1,a
1                         mov     a,Gtop
1                         addc    a,tmp3
1                         mov     tmp3,a
        ;
        ; 66                      if tmp1 < tmp2 then
        ;
1                         mov     a,tmp1           ; compare MSB
2                         cjne    a,tmp2,cmp15
1                         mov     a,tmp1+1
2                         cjne    a,tmp2+1,cmp1; compare LSB
        cmp1:
2                         jnc     next1
        ;
        ; 67              min := tmp1
        ;
2                         mov     min+1,tmp1+1            ; min := tmp1
2                         mov     min,tmp1
2                         sjmp    next2
        cmp15:
1                         nop
2                         sjmp    cmp1
        ;
        ; 68              else
        ; 69                      min := tmp2;
        ;
        next1:
2                         mov     min+1,tmp2+1            ; min := tmp2
2                         mov     min,tmp2
1                         nop
```

Figure B.12 (Continued)

Jul 11 11:36 1984 odd.s Page 5

```
1                       nop
    ;
    ; 70                if tmp3 < min then
    ;
    ;                                           if tmp3 < min
    next2:
1                       mov     a,tmp3          ; Compare MSB
2                       cjne    a,min,cmp25
1                       mov     a,tmp3+1        ; Compare LSB
2                       cjne    a,min+1,cmp2
    cmp2:
2                       jc      next25          ; This makes the timing on
1                       nop                     ; both branches the same
1                       nop
2                       sjmp    next3
    cmp25:
1                       nop
2                       sjmp    cmp2

    ;
    ; 71                min := tmp3;
    ;
    next25:
2                       mov     min+1,tmp3+1            ; min := tmp3
2                       mov     min,tmp3
    ;
    ; 74                g := d + min
    ;
    next3:
1                       mov     a,min+1                 ; g := min + d
1                       add     a,d+1
1                       mov     g+1,a
1                       mov     a,min
1                       addc    a,d
1                       mov     g,a
1                       mov     a,min

    ;
    ; 73                if min < inf then
    ;
    ;                   if g > inf then
    ;                      g := 0

2                       jc      next32          ; if carry, g > inf
2                       cjne    a,inf,$+3       ; if g > inf
2                       jnc     next35          ; This makes the timing on both
1                       nop                     ; branches the same
2                       sjmp    next4
```

Figure B.12 (Continued)

```
Jul 11 11:36 1984 odd.s Page 6

        next32:
1                       nop
1                       nop
1                       nop
1                       nop
        ;
        ; 75            else
        ; 76                    g := 0;
        ;
        next35:
1                       clr     a               ; g := 0
1                       mov     g+1,a
1                       mov     g,a

        ;
        ; 78            DTbot <- g;
        ;
        next4:
2                       mov     p1,#sw          ; DTbot <- g
1                       mov     a,g+1
2                       movx    @dptr,a
2                       lcall   writedelay
1                       mov     a,g
2                       movx    @dptr,a
2                       lcall   writedelay

        ;
        ; 79            DTtop <- g;
        ;
1                       mov     a,g+1
2                       mov     p1,#west        ; DTtop <- g
2                       movx    @dptr,a
2                       lcall   writedelay
1                       mov     a,g
2                       movx    @dptr,a
2                       lcall   writedelay
        ;
        ; 81            Gbot <- DTbot;
        ;
2                       movx    a,@dptr         ; Gbot <- DTbot
1                       mov     Gbot+1,a
2                       movx    a,@dptr
1                       mov     Gbot,a
        ;
        ; 82            Gtop <- DTtop;
        ;
2                       movx    a,@dptr         ; Gtop <- DTtop
1                       mov     Gtop+1,a
```

Figure B.12 (Continued)

Jul 11 11:36 1984 odd.s Page 7

```
2                   movx    a,@dptr
1                   mov     Gtop,a

1                   mov     a,#low(LOOPTIME);    Wait for timer
1                   xrl     a,tl1               ; xor LSBs to see if they are the same
1                   rrc     a                   ; move LSB into carry bit
1                   mov     a,#low(LOOPTIME)
2                   jnc     sync                ; Sync with timer, since the cjne takes
1                   nop                         ; 2 uS, there is a 50/50 chance the LSB
                                                ; will not match, this comparison should
                                                ; sync the program up with the timer so
                                                ; with LSB will always match
      sync:
2                   cjne    a,tl1,$

      ;
      ; 87            end;
      ;
2                   ljmp    main


      #include "util.h"

      ;
      ; 88            end.
      ;
                    end
```

Figure B.12 (Continued)

```
Jul 11 10:53 1984 dtw.h Page 1

        COEFS           equu    4       ;       Number of coefficients used
        LOOPTIME equ            510-7   ; Total number of microseconds per loop

        #include "ports.h"

                        org     29h             ; Start of readport buffers
        avec:           ds      COEFS
        bvec:           ds      COEFS
                        ds      1       ; number of coefficients used
        d:              ds      2       ; local distance
        Dbot:           ds      2
        Dtop:           ds      2
        g:              ds      2       ; Total accumulated distance
        Gbot:           ds      2
        Gbotoid:ds      2
        Gtop:           ds      2
        Gtopold:ds      2
        inf:            ds      2       ; Infinity 16   bit
        inf8:           ds      1       ; Infinity 8    bit
                        ds      1
        min:            ds      2
        tmp1:           ds      2
        tmp2:           ds      2
        tmp3:           ds      2
```

Figure B.12 (Continued)

Jul 11 10:55 1984 init.h Page 1

```
                        org     08000h
        ;
        ; 28            int := 32786;
        ;
2                       mov     inf8,#0ffh
2                       mov     inf+1,#0
2                       mov     inf,#40h
        ;
        ; 40            for i := 1      to 2*coefs do
        ;
1                       mov     r0,#avec        ; Initialize a and b vectors to inf8
1                       mov     r2,#2*COEFS
1                       mov     a,inf8
        init1:
        ;
        ; 42            a[i] := inf;
        ; 43            b[i] := inf;
1                       mov     @r0,a           ; set avec and bvec to inf8
1                       inc     r0
2                       djnz    r2,init1
        ;
        ; 32            Gbotold := inf;
        ; 33            Gtopold := inf;
        ; 34            Gbot := inf;
        ; 35            Gtop := inf;
        ; 36            Dbot := inf;
        ; 37            Dtop := inf;
        ; 38            g := 0;
        ;
1                       mov     a,inf+1         ; Initize LSB of variables
1                       mov     Gbotold+1,a
1                       mov     Gtopold+1,a
1                       mov     Gbot+1,a
1                       mov     Gtop+1,a
1                       mov     Dbot+1,a
1                       mov     Dtop+1,a
2                       mov     g+1,#0
1                       mov     a,inf           ; Initize MSB of variables
1                       mov     Gbotold,a
1                       mov     Gtopold,a
1                       mov     Gbot,a
1                       mov     Gtop,a
1                       mov     Dbot,a
1                       mov     Dtop,a
2                       mov     g,#0
2                       mov     dptr,#lowSWLat          ; dptr is never changed after this.
2                       mov     p0,#0f0h        ; neither is p0
```

Figure B.12 (Continued)

```
Jul 11 11:42 1984 repeat.s Page 1

        ;               Program Name:   dtw, repeat.s (d2)
        ;               Algorithm:      Figure 6.16??
        ;               Machine:        VLSI processor array, simulated by Poker
        ;               Function:       32 bit coef are input, the upper three
        ;                               bytes are thrown away. The lower byte is
        ;                               stored in aarch until an entire word is
        ;                               received. The the word is output one frame
        ;                               at a time with LOOPTIME time between
        ;                               frames. The word is outputed VOCAB times,
        ;                               i.e. once for every word in the
        ;                               vocabulary.
        ;               Precision:      Input coefficients:     32-bit unsigned integers
        ;                               Output: 8-bit unsigned
        ;               Number of PEs:  1
        ;               Parameters:     coefs, the number of coefficients per frame

        VOCAB           equ     3       ; Words in vocabulary
        COEFS           equ     4       ; coefficients per frame
        FRAMES          equ     2       ; Frames per word
        LOOPTIME equ            510-5

#include "ports.h"

                        org     29h
        count:          ds      1       ; Number of coefs input, or frames output
        vcount:         ds      1       ; Number of vocabulary outputted
        aindex:         ds      2       ; Pointer to next location in aarch in EXRAM
        avec:           ds      COEFS   ; INRAM temp storage for on frame
        inf8:           ds      1       ; 8 bit infinity

                        org     8000h
2                       mov     inf8,#0ffh
2                       mov     dptr,#lowSWLat
2                       mov     p0,#0f0h        ; This is for good luck
2                       mov     p1,#ne          ; All data goes out the ne port

        ;               Start of loop to read in the word and store in EXRAM

        main:
2                       mov     count,#0
2                       mov     aindex+1,#low(aarch)
2                       mov     aindex,#high(aarch)

        moredata:
2                       jnb     p0.7,$          ; Wait for input
2                       movx    a,@dptr
1                       inc     count
2                       mov     dpl,aindex+1    ; Get pointer to aarch
```

Figure B.12 (Continued)

Jul 11 11:42 1984 repeat.s Page 2

```
2               mov     dph,aindex
2               movx    @dptr,a
2               inc     dptr
2               mov     aindex+1,dpl
2               mov     aindex,dph
2               mov     dptr,#lowSWLat          ; Point to switch

1               mov     r0,#3           ; Ignore next 3 input bytes
        dummyread:
2               jnb     p0.7,$          ; Wait for input
2               movx    a,@dptr
2               djnz    r0,dummyread

1               mov     a,count
2               cjne    a,#COEFS*FRAMES,moredata ; Jump if more data to read

        ;
        ;       Turn on timer
        ;
1               clr     a
1               mov     tcon,a          ; Stop timer
1               mov     tl1,a           ; Clear timer
1               mov     th1,a
1               setb    tcon.6          ; Start timer

        ;
        ;       Pad input data with on frame of inf
        ;
2               mov     dpl,aindex+1
2               mov     dph,aindex
1               mov     r2,#COEFS
1               mov     a,inf8
        infpad:
2               movx    @dptr,a
2               inc     dptr
2               djnz    r2,infpad

2               mov     vcount,#0

        nextword:
2               mov     count,#0
2               mov     aindex+1,#low(aarch)
2               mov     aindex,#high(aarch)

        nextframe:
1               mov     r0,#avec
1               mov     r2,#COEFS
2               mov     dpl,aindex+1
```

Figure B.12 (Continued)

Jul 11 11:42 1984 repeat.s Page 3

```
2                   mov     dph,aindex

        transfer:
2                   movx    a,@dptr         ; Move one frame from EXRAM to INRAM
1                   mov     @r0,a
1                   inc     r0
2                   inc     dptr
2                   djnz    r2,transfer

2                   mov     aindex+1,dpl
2                   mov     aindex,dph

        ;
        ;           Wait for timer
        ;
1                   mov     a,#high(LOOPTIME)
2                   cjne    a,th1,$         ; Wait for upper 8 bits to match
1                   mov     a,#low(LOOPTIME);   Wait for timer
1                   xrl     a,tl1           ; xor LSBs to see if they are the same
1                   rrc     a               ; move LSB into carry bit
1                   mov     a,#low(LOOPTIME)
2                   jnc     sync            ; Sync with timer, since the cjne takes
1                   nop                     ; 2 uS, there is a 50/50 chance the LSB
                                            ; will not match, this comparison should
                                            ; sync the program up with the timer so
                                            ; with LSB will always match
        sync:
2                   cjne    a,tl1,$


        ;
        ;           Turn on timer
        ;
1                   clr     a
1                   mov     tcon,a          ; Stop timer
1                   mov     tl1,a           ; Clear timer
1                   mov     th1,a
1                   setb    tcon.6          ; Start timer

2                   mov     dptr,#lowSWLat
1                   mov     r0,#avec
1                   mov     r2,#COEFS
        sendout:
1                   mov     a,@r0
2                   movx    @dptr,a
1                   inc     r0
2                   lcall   writedelay
2                   djnz    r2,sendout
```

Figure B.12 (Continued)

Jul 11 11:42 1984 repeat.s Page 4

```
1                       inc     count
1                       mov     a,count
2                       cjne    a,#FRAMES+1,nextframe

1                       inc     vcount
1                       mov     a,vcount
2                       cjne    a,#VOCAB,nextword

2                       mov     tcon,#0         ; Turn off timer
2                       ljmp    main

#include "util.h"

        aarch:          ds      1

                        end
```

Figure B.12 (Continued)

```
Jul 11 11:58 1984 seq.s Page 1

        ;               Program Name:   dtw, seq.s (d2)
        ;               Algorithm:      Figure 6.16??
        ;               Machine:        VLSI processor array, simulated by Poker
        ;               Function:       For each unknown vector (frame) seq.s
        ;                               receives from repeat.s it outputs the unkown
        ;                               vector to its north port and a known vector
        ;                               from its library to the south port.  The new
        ;                               vectors are output once every LOOPTIME µs.
        ;                               If no vectors come from repeat.s, seq.s will
        ;                               output infinite vectors every LOOPTIME µs.
        ;               Precision:      Input coefficients:     8-bit unsigned integers
        ;                               Output:8-bit unsigned integers
        ;               Number of PEs:  1
        ;               Parameters:     coefs, the number of coefficients per frame

        ;               Warp drive version of DTW (seq.s)
        ;               8               bit coefs       16 bit distances

        COEFS           equ     4
        LOOPTIME equu           510-7

#include "./ports.h"

                        org     29h
        avec:           ds      COEFS
        bvec:           ds      COEFS
                        ds      1
        infvec:         ds      COEFS
        bindex:         ds      2
        inf:            ds      2
        inf8:           ds      1

                        org     08000h
1                       nop

2                       mov     inf8,#0ffh
2                       mov     inf+1,#0
2                       mov     inf,#40h

1                       mov     r0,#infvec      ; Initialize infvec to inf8
1                       mov     r2,#COEFS
1                       mov     a,inf8
        init1:
1                       mov     @r0,a           ; set avec and bvec to inf8
1                       inc     r0
1                       nop                     ; Since this loop runs half as long as
1                       nop                     ; the init1 loop in even.s and odd.s,
1                       nop                     ; it must be twice as long to keep
```

Figure B.12 (Continued)

Jul 11 11:58 1984 seq.s Page 2

```
1                 nop                       ; the timing the same.
2                 djnz    r2,init1

1                 mov     r0,#infvec        ; r0 points to inf vector
1                 mov     r1,#infvec        ; r1 points to inf vector

1                 mov     r3,#7             ; Wait 16 microseconds for other PEs
1                 nop
2                 djnz    r3,$

2                 mov     dptr,#lowSWLat
2                 mov     p0,#0f0h          ; p0 is never changed after this.

        ;         Start of main loop, read a then b and find distance

        main:
1                 clr     a
1                 mov     tcon,a            ; Stop timer
1                 mov     tl1,a             ; clear timer
1                 mov     th1,a
1                 setb    tcon.6            ; Start timer

1                 mov     r2,#COEFS
1                 mov     r3,#2             ; Wait 6 uS
1                 nop
2                 djnz    r3,$

        input:
1                 mov     a,@r0
2                 mov     p1,#north
2                 movx    @dptr,a           ; Send avec[?]
2                 lcall   writedelay

1                 mov     a,@r1             ; Send bvec[?]
2                 mov     p1,#south         ; Set new direction
2                 movx    @dptr,a

1                 inc     r0                ; Point to next element in a vector
1                 inc     r1                ; Point to next element in b vector

1                 mov     r3,#16            ; Wait 33 microseconds
2                 djnz    r3,$

2                 djnz    r2,input

        ;
        ;         Check for input data
        ;
```

Figure B.12 (Continued)

Jul 11 11:58 1984 seq.s Page 3

```
2                   jnb     p0.7,infout         ; Jump if no input data
1                   mov     r0,#avec
1                   mov     r1,#bvec
2                   mov     dpl,bindex+1
2                   mov     dph,bindex

        ;           Get b vector from external RAM
1                   mov     r2,#COEFS           ; for r2 := coefs to 0 step -1
        getb:
2                   movx    a,@dptr
1                   mov     @r1,a               ; bvec[r1] = barch[?]
1                   inc     r1
2                   inc     dptr
2                   djnz    r2,getb

2                   mov     bindex+1,dpl
2                   mov     bindex,dph
2                   mov     dptr,#lowSWLat

        ;           Get a vector from input port
1                   mov     r2,#COEFS           ; for r2 := coefs to 0 step -1
        geta:
2                   jnb     p0.7,$              ; Wait for next input

2                   movx    a,@dptr
1                   mov     @r0,a               ; avec[r0] <- input
1                   inc     r0
2                   djnz    r2,geta

1                   mov     r0,#avec            ; Get pointers ready to output
1                   mov     r1,#bvec

2                   sjmp    timer

        ;
        ;           No more input data, send infinity vectors instead
        ;
        infout:
1                   mov     r0,#infvec
1                   mov     r1,#infvec
2                   mov     bindex+1,#low(barch)  ; Start b vector over again
2                   mov     bindex,#high(barch)

        ;
        ;           Wait for timer
        ;
        timer:
1                   mov     a,#high(LOOPTIME)   ; wait for upper byte to turn to 2
```

Figure B.12 (Continued)

Jul 11 11:58 1984 seq.s Page 4

```
2                     cjne    a,th1,$

1                     mov     a,#low(LOOPTIME)
1                     xrl     a,tl1              ; xor LSBs to see if they are the same
1                     rrc     a                  ; move LSB into carry bit
1                     mov     a,#low(LOOPTIME)
2                     jnc     sync               ; Sync with timer, since the cjne takes
1                     nop                        ; 2 uS, there is a 50/50 chance the LSB
                                                 ; will not match, this comparison should
                                                 ; sync the program up with the timer so
                                                 ; with LSB will always match
        sync:
2                     cjne    a,tl1,$            ; Wait for lower byte to turn to right value

2                     ljmp    main

#include "./util.h"

        barch:
                      db      168                ; all1
                      db      138
                      db      143
                      db      84
                      db      170
                      db      131
                      db      147
                      db      82
                      db      0ffh               ; Put inf vector between words
                      db      0ffh
                      db      0ffh
                      db      0ffh

                      db      152                ; less (the word)
                      db      112
                      db      152
                      db      85
                      db      151
                      db      110
                      db      149
                      db      86
                      db      0ffh               ; Put inf vector between words
                      db      0ffh
                      db      0ffh
                      db      0ffh

                      db      170                ; all1    with frames 1 and 2 reversed
                      db      131
                      db      147
```

Figure B.12 (Continued)

```
Jul 11 11:58 1984 seq.s Page 5

                db      82
                db      168
                db      138
                db      143
                db      84
                db      0ffh            ; Put inf vector between words
                db      0ffh
                db      0ffh
                db      0ffh

                end
```

Figure B.12 (Continued)

Jul 11 12:03 1984 scores.s Page 1

```
        ;               Program Name:   dtw, scores.s (d2)
        ;               Algorithm:      Figure 6.16??
        ;               Machine:        VLSI processor array, simulated by Poker
        ;               Function:       This is routine collects the distance
        ;                               scores from pe 4,7.  The only score that really
        ;                               means something is the score just before a zero
        ;                               value.  All others are intermidiate values.
        ;               Precision:      Input scores:    16-bit unsigned integers
        ;               Number of PEs:  1

        #include "ports.h"

                        org     029h
        input:          ds      2
        lastzero:       ds      1               ; ==1  if last value was a 0
        scores:         ds      2*(2+1)         ; score of all the words

                        org     8000h
2                       mov     dptr,#lowSWLat          ; Get switch address
2                       mov     p0,#0f0h
1                       mov     r0,#scores      ; r0  points to the next hi location in scores
1                       mov     r1,#scores+1    ; r1  points to the next lo location in scores

        main:
2                       jnb     p0.7,$          ; Wait for input
        ;               dump    17,1
                        db      0a5h
1                       dw      17
1                       dw      1

2                       movx    a,@dptr         ; Read byte
1                       mov     input+1,a       ; Save

2                       jnb     p0.7,$          ; Get second byte

2                       movx    a,@dptr
1                       mov     input,a

1                       clr     a               ; jump if input != 0
2                       cjne    a,input,notyet
2                       cjne    a,input+1,notyet
2                       cjne    a,lastzero,notyet2

        ;               The input was zero, so the pervious input was a good score
        ;               increment the scores pointer (r0 and r1) and don't save
        ;               the zero values

1                       inc     r0
```

Figure B.12 (Continued)

```
Jul 11 12:03 1984 scores.s Page 2

1                         inc     r0
1                         inc     r1
1                         inc     r1
2                         mov     lastzero,#1

2                         sjmp    main

        notyet:
2                         mov     lastzero,#0
        notyet2:
2                         mov     @r0,input
2                         mov     @r1,input+1
2                         sjmp    main

                          end
```

Figure B.12 (Continued)

```
        ;           Program Name:   input
        ;           Machine:        VLSI processor array, simulated by Poker
        ;           Function:       Output a stored speech signal at a given
        ;                           sampling rate
        ;           Precision:      Output: 16 bits, sign-magnitude
        ;           Number of PEs:  1
        ;           Parameters:     SAMPLETIME, the time in µs between samples
        ;           Input:          None
        ;           Output:         Departs from west port once every
        ;                           SAMPLETIME µs

        SAMPLETIME equu   160-8          ; Time in microseconds between outputs

        #include "ports.h"

                    org   29h            ; Start of readport buffers
        in:         ds    2              ; pointer to next word of input data

                    org   08000h
2                   mov   in+1,#low(word)      ; point in to start of data
2                   mov   in,#high(word)
2                   mov   p1,#west       ; direction to send data
2                   mov   p0,#0f0h
2                   mov   tmod,#10h      ; Set timer 1 to no gate, timer,
                                         ; mode 1 (16 bit)

1                   mov   r0,#19         ; wait 40 microseconds so right values
1                   nop
2                   djnz  r0,$

        main:
1                   clr   a
1                   mov   tcon,a         ; Stop timer
1                   mov   tl1,a          ; clear timer
1                   mov   th1,a
1                   setb  tcon.6         ; Start timer

2                   mov   dpl,in+1
2                   mov   dph,in
2                   inc   dptr
2                   movx  a,@dptr        ; get LSB of next input sample
2                   mov   dptr,#lowSWLat
2                   movx  @dptr,a        ; send byte to switch

2                   mov   dpl,in+1
2                   mov   dph,in
2                   movx  a,@dptr        ; get MSB of next input sample
2                   inc   dptr
```

Figure B.13 Program to output stored speech signal.

```
2                   inc     dptr
2                   mov     in+1,dpl
2                   mov     in,dph

2                   mov     dptr,#lowSWLat
2                   movx    @dptr,a          ; send byte to switch

        ;
        ;           If you are at the end of the word, start over again
        ;
1                   mov     a,in+1
2                   cjne    a,#low(dataend),wait
1                   mov     a,in
2                   cjne    a,#high(dataend),wait
2                   mov     in+1,#low(word)
2                   mov     in,#high(word)

        ;
        ;           Wait for timer
        ;
        wait:
1                   mov     a,#high(SAMPLETIME)
2                   cjne    a,th1,$
1                   mov     a,#low(SAMPLETIME)
1                   xrl     a,tl1            ; xor LSBs to see if they are the same
1                   rrc     a                ; move LSB into carry bit
1                   mov     a,#low(SAMPLETIME)
2                   jnc     sync             ; Sync with timer, since the cjne takes
1                   nop                      ; 2 uS, there is a 50/50 chance the LSB
                                             ; will not match, this comparison should
                                             ; sync the program up with the timer so
                                             ; with LSB will always match
        sync:
2                   cjne    a,tl1,$

2                   sjmp    main             ; Wait for rest of PEs to work

        word:


        #include "all1.h"                ; The file "all.1h" contains the
                                             ; speech data
        dataend:
                    db      0
                    end
```

Figure B.13 (Continued)

```
        ;               Program Name:     filter
        ;               Machine:          VLSI processor array, simulated by Poker
        ;               Function:         preemphsize the input signal using the
        ;                                 transfer function:
        ;                                         H(z)=1-.95*z^-1
        ;               Precision:        Input:  16 bits, 2's complement
        ;                                 Sum:    16 bits, sign-magLitude
        ;                                 Output: 16 bits, sign-magnitude
        ;               Number of PEs:    1
        ;               Parameters:       COEF, the filter coefficient
        ;               Input:            Arrives at the north port of cell
        ;               Output:           Departs from east port of cell
        ;               Loop Time:        85 µs
        ;               Max Sample Rate:  11 KHz
        ;

        COEF            equ     243     ; 243/256 = .95

        #include "ports.h"

                        org     29h             ; Start of readport buffers
        sign:           ds      1               ; Sign of sum
        sum:            ds      2               ; last value * COEF
        last:           ds      2               ; last value

                        org     08000h

2                       mov     dptr,#lowSWLat          ; dptr doesn't change after this
2                       mov     p1,#east        ; neither does p1
2                       mov     p0,#0f0h

1                       clr     a
1                       mov     sum+1,a                 ; 8     sum := 0
1                       mov     sum+0,a
1                       mov     sign,a
1                       clr     c               ; clear carry flag (no borrow)

        main:
2                       jnb     p0.7,$          ; Wait for external input
2                       movx    a,@dptr         ; in <- LSB of right
1                       mov     last+1,a        ; Save for later

        ;               sum := a - sum;

1                       subb    a,sum+1                 ; sum := a - sum        carry flag was cleared
1                       mov     sum+1,a                 ;                       at end of loop

2                       jnb     p0.7,$          ; Wait for next byte
```

Figure B.14  Assembly language program for preemphasis filtering.

```
2                       movx    a,@dptr
1                       mov     last,a          ; Save for later

1                       subb    a,sum
1                       mov     sum,a

2                       jb      sum.7,negative
1                       mov     a,sum+1                 ; Value is positive, no changes needed
2                       movx    @dptr,a         ; out <- a
1                       mov     a,sum

1                       mov     r0,#6           ; Wait 13 more microS
2                       djnz    r0,$

2                       movx    @dptr,a
2                       sjmp    multiply

        negative:
1                       mov     a,sum+1                 ; Convert to sign/magnitude
1                       xrl     a,#0ffh
1                       add     a,#1
2                       movx    @dptr,a         ; out <- a

1                       mov     a,sum
1                       xrl     a,#0ffh
1                       addc    a,#0            ; Add in carry
1                       orl     a,#80h          ; Set sign bit to negative

1                       mov     r0,#4           ; Wait 10 more microS
1                       nop
2                       djnz    r0,$

2                       movx    @dptr,a         ; out <- a

        multiply:
2                       jnb     last.7,positive ; Jump if last input value positive

1                       mov     a,last+1        ; Convert to Sign/Magnitude
1                       xrl     a,#0ffh
1                       add     a,#1
1                       mov     last+1,a

1                       mov     a,last
1                       xrl     a,#0ffh
1                       addc    a,#0
1                       mov     last,a
2                       mov     sign,#0ffh

        ;               sum := sum * COEF
```

Figure B.14 (Continued)

```
        positive:
2                       mov     b,#COEF            ; Since COEF is a fraction,
1                       mov     a,last+1
4                       mul     ab          ; the result in the A register
2                       mov     sum+1,b            ; is thrown away, too small.

2                       mov     b,#COEF
1                       mov     a,last
4                       mul     ab
1                       add     a,sum+1
1                       mov     sum+1,a
1                       mov     a,b
1                       addc    a,#0        ; add carry
1                       mov     sum,a

2                       jnb     sign.7,pos  ; if not set, the number is positive
1                       mov     a,sum+1            ; convert back to 2's complement
1                       xrl     a,#0ffh
1                       add     a,#1
1                       mov     sum+1,a

1                       mov     a,sum+0
1                       xrl     a,#0ffh
1                       addc    a,#0
1                       mov     sum+0,a

        pos:
1                       clr     a
1                       mov     sign,a
1                       clr     c           ; Get ready to subtract sum from

2                       ljmp    main

                        end
```

Figure B.14 (Continued)

```
;               Program Name:       auto
;               Algorithm:          Figure 6.12
;               Machine:            VLSI processor array, simulated by Poker
;               Function:           Find autocorrelation coefficients R(i)
;                                   given input signal x(m), using
;                                             k=N-i-1
;                                   R(i)=       ∑     x(k)x(k+i)
;                                              k=0
;               Precision:          Input:  16 bits, sign/magnitude
;                                   Output: 32 bits, signed
;               Number of PEs:      p, the number of coefficients computed.
;               Parameters:         p, the number of coefficients computed.
;               Input:              Arrives at the north port of cell (1,3)
;               Output:             Departs from east port of merge cell
;               Loop Time:          160 μs to process one input sample
;               Max Sample Rate:    6.25 KHz
;
;               Sends data through the switch in reverse order
;               (i.e. LSB first)
;               Use with runrev.o eproms
;               Quasi synchronous.  i.e. Can take input from external source
;                                                    at unknown intervals
;

samples         equ     100             ; Number of sample per frame

#include "ports.h"

                org     029h

;
; 18            sint    i;
;
i:              ds      1
;
; 19            int     sum,top,left;
;
sign:           ds      1
sum:            ds      4
top:            ds      2
left:           ds      2

;               sign  = ff if product is negative
;                     = 00 if positive
;               The results of each mul is xlr with sign, so if the result is
;               negative the product will be complemented.  Also the carry bit
;               is set to sign, to if the result is negative, the product will have
;               1 added to it.  The make the product in 2 comp notation.

                org     08000h
```

Figure B.15 Assembly language program for autocorrelation.

```
        ;
        ; 25            out <- sum;
        ;
2                       mov     p1,#south       ;         Write 0 to south port
2                       mov     dptr,#lowSWLat            ; This value will remain in dptr from now on

1                       clr     a
2                       movx    @dptr,a         ; write 0 to switch

1                       mov     r0,#6
2                       djnz    r0,$            ; Wait 14 microseconds for switch

1                       clr     a
2                       movx    @dptr,a         ; write 0 to switch
        ;
        ; 22            i := 0;
        ;
1                       clr     a
1                       mov     i,a             ; i := 0
        ;
        ; 23            sum := 0;
        ;
1                       mov     sum+3,a                   ; sum := 0
1                       mov     sum+2,a
1                       mov     sum+1,a
1                       mov     sum,a

        main:
        ;
        ; 29            i := i + 1;
        ;
1                       inc     i               ; i := i + 1
2                       mov     sign,#0         ; assume result is positive
        ;
        ; 30            top <- in1;
        ;
2                       jnb     p0.7,$          ; Wait for input from external program
2                       movx    a,@dptr         ; Read LSB of top from switch
1                       mov     top+1,a
2                       jnb     p0.7,$          ; Wait for input from external program
2                       movx    a,@dptr         ; Read MSB of top from switch
1                       mov     top,a
        ;
        ; 30            left <- in2;
        ;
2                       jnb     p0.7,$          ; Wait for input from external program
2                       movx    a,@dptr         ; Read LSB of left from switch
1                       mov     left+1,a
2                       jnb     p0.7,$          ; Wait for input from external program
2                       movx    a,@dptr         ; Read MSB of left from switch
```

Figure B.15 (Continued)

```
1                   mov    left,a

1                   xrl    a,top          ; exclusive or signs to see what sign result is
2                   jnb    acc.7,there
2                   mov    sign,#0ffh     ; result is negative
        ;
        ; 33        if i < samples then
        ;
        there:
        ;           dump   1,1
                    db     0a5h
1                   dw     1
1                   dw     1

1                   mov    a,left         ; remove sign bit from left
1                   anl    a,#07fh
1                   mov    left,a

1                   mov    a,i            ; load i
2                   cjne   a,#samples,loop; if(i != samples) goto loop
2                   ljmp   endloop        ; if( i == samples) goto endloop

        loop:

        ;           sum := sum + left * top
        ;           Where sum is 32 bits and left and top are 16 bits
        ;
        ;              30    31    (left)
        ;       X      2e    2f    (top)
        ;       ----------------
        ; +          30x2f 31x2f
        ; +     30x2e 31x2e
        ; + 2a    2b    2c    2d   (sum)
        ;
        ;
        ; 36        sum := sum + top * left;
        ;
1                   mov    a,left+1       ; LSB of left
2                   mov    b,top+1               ; LSB of top
4                   mul    ab
1                   xrl    a,sign         ; change sign if needed
1                   mov    c,sign.7
1                   addc   a,sum+3               ; LSB of sum (byte 4)
1                   mov    sum+3,a
1                   mov    a,b
1                   xrl    a,sign         ; change sign if needed
1                   addc   a,sum+2               ; add in byte 3 of sum
1                   mov    sum+2,a
1                   mov    a,sign
1                   addc   a,sum+1               ; add carry to byte 2 of sum
```

Figure B.15 (Continued)

```
1                mov    sum+1,a
1                mov    a,sign
1                addc   a,sum          ; add carry to MSB of sum (byte 1)
1                mov    sum,a
1                mov    a,top+1                  ; LSB of top
       ;
       ; 35      out <- top;
       ;
2                movx   @dptr,a        ; Send LSB of top to south port
       ;
       ; 36      sum := sum + top * left;      (cont)
       ;
2                mov    b,left         ; MSB of left
4                mul    ab
1                xrl    a,sign         ; change sign if needed
1                mov    c,sign.7
1                addc   a,sum+2                  ; add to byte 3 of sum
1                mov    sum+2,a
1                mov    a,b
1                xrl    a,sign         ; change sign if needed
1                addc   a,sum+1                  ; add to byte 2 of sum
1                mov    sum+1,a
1                mov    a,sign
1                addc   a,sum          ; add carry to byte 1 of sum
1                mov    sum,a

1                mov    a,top          ; MSB of top
       ;
       ; 35      sum <- top;   (cont)
       ;
2                movx   @dptr,a        ; Send MSB of top to south port
       ;
       ; 36      sum := sum + top * left;      (cont)
       ;
1                anl    a,#7fh         ; Remove sign bit from top
2                mov    b,left+1       ; LSB of left
4                mul    ab
1                xrl    a,sign         ; change sign if needed
1                mov    c,sign.7
1                addc   a,sum+2                  ; add to byte 3 of sum
1                mov    sum+2,a
1                mov    a,b
1                xrl    a,sign         ; change sign if needed
1                addc   a,sum+1                  ; add to byte 2 of sum
1                mov    sum+1,a
1                mov    a,sign
1                addc   a,sum          ; add carry to byte 1 of sum
1                mov    sum,a

1                mov    a,top          ; MSB of top
```

Figure B.15 (Continued)

```
1               anl     a,#7fh          ; Remove sign bit from top
2               mov     b,left          ; LSB of left
4               mul     ab
1               xrl     a,sign          ; change sign if needed
1               mov     c,sign.7
1               addc    a,sum+1                 ; add to byte 2 of sum
1               mov     sum+1,a
1               mov     a,b
1               xrl     a,sign          ; change sign if needed
1               addc    a,sum           ; add to MSB of sum (byte 1)
1               mov     sum,a
    ;
    ; 37            end
    ;
2               ljmp    main

    endloop:
    ;
    ; 43            out <- sum;
    ;
1               clr     a               ; Send a 0 to south port bottom <- 0
2               movx    @dptr,a         ; write to switch

1               mov     r0,#6
2               djnz    r0,$            ; Wait 14 microseconds for switch

1               clr     a               ; Send a 0 to port 4
2               movx    @dptr,a         ; write to switch

    ;               sum := sum + left * top
    ;               Where sum is 32 bits and left and top are 16 bits
    ;
    ;                  30     31     (left)
    ;          X       2e     2f     (top)
    ;          ------------------
    ; +             30x2f  31x2f
    ; +        30x2e 31x2e
    ; + 2a      2b     2c     2d     (sum)
    ;
    ;
    ; 40            sum := sum + top * left;
    ;
1               mov     a,left+1        ; LSB of left
2               mov     b,top+1                 ; LSB of top
4               mul     ab
1               xrl     a,sign          ; change sign if needed
1               mov     c,sign.7
1               addc    a,sum+3                 ; LSB of sum (byte 4)
1               mov     sum+3,a
    ;
```

Figure B.15 (Continued)

```
        ; 41          results <- sum;
        ;
2                     mov     p1,#east        ; Next write is to east port
2                     movx    @dptr,a         ; Send LSB of sum to east port
        ;
        ; 40          sum := sum + top * left;     (cont)
        ;
1                     mov     a,b
1                     xrl     a,sign          ; change sign if needed
1                     addc    a,sum+2                 ; add in byte 3 of sum
1                     mov     sum+2,a
1                     mov     a,sign
1                     addc    a,sum+1                 ; add carry to byte 2 of sum
1                     mov     sum+1,a
1                     mov     a,sign
1                     addc    a,sum           ; add carry to MSB of sum (byte 1)
1                     mov     sum,a

1                     mov     a,top+1                 ; LSB of top
2                     mov     b,left          ; MSB of left
4                     mul     ab
1                     xrl     a,sign          ; change sign if needed
1                     mov     c,sign.7
1                     addc    a,sum+2                 ; add to byte 3 of sum
1                     mov     sum+2,a
1                     mov     a,b
1                     xrl     a,sign          ; change sign if needed
1                     addc    a,sum+1                 ; add to byte 2 of sum
1                     mov     sum+1,a
1                     mov     a,sign
1                     addc    a,sum           ; add carry to byte 1 of sum
1                     mov     sum,a

1                     mov     a,top           ; MSB of top
1                     anl     a,#7fh          ; Remove sign bit from top
2                     mov     b,left+1        ; LSB of left
4                     mul     ab
1                     xrl     a,sign          ; change sign if needed
1                     mov     c,sign.7
1                     addc    a,sum+2                 ; add to byte 3 of sum
1                     mov     sum+2,a
        ;
        ; 41          results <- sum;         (cont)
        ;
2                     movx    @dptr,a         ; Send third byte of sum to east port
        ;
        ; 40          sum := sum + top * left;     (cont)
        ;
1                     mov     a,b
1                     xrl     a,sign          ; change sign if needed
```

Figure B.15 (Continued)

```
1               addc   a,sum+1               ; add to byte 2 of sum
1               mov    sum+1,a
1               mov    a,sign
1               addc   a,sum         ; add carry to byte 1 of sum
1               mov    sum,a

1               mov    a,top         ; MSB of top
1               anl    a,#7fh        ; Remove sign bit from top
2               mov    b,left        ; MSB of left
4               mul    ab
1               xrl    a,sign        ; change sign if needed
1               mov    c,sign.7
1               addc   a,sum+1               ; add to byte 2 of sum
        ;
        ; 41    results <- sum;      (cont)
        ;
2               movx   @dptr,a       ; Send second byte of sum to east port
        ;
        ; 40    sum := sum + top * left;     (cont)
        ;
1               mov    sum+1,a
1               mov    a,b
1               xrl    a,sign        ; change sign if needed
1               addc   a,sum         ; add to MSB of sum (byte 1)
1               mov    sum,a
        ;
        ; 41    results <- sum;      (cont)
        ;
1               mov    r0,#4         ; Wait 14 microseconds for switch
2               djnz   r0,$

2               movx   @dptr,a       ; Send second byte of sum to east port

        ;
        ;       Initialize i and sum for next autocorrelation calculation
        ;
        ;
        ;
        ; 44    i := 0;
        ;
1               clr    a
1               mov    i,a            ; i := 0
        ;
        ; 42    sum := 0;
        ;
1               mov    sum+3,a               ; sum := 0
1               mov    sum+2,a
1               mov    sum+1,a
1               mov    sum,a

2               mov    p1,#south
```

Figure B.15 (Continued)

```
       ;
       ; 45            end
       ;
2                      ljmp     main

                       end
```

Figure B.15 (Continued)