ESD-TR-86-102

Technical Report
759

An Array Computer for Digital Signal Processing

M.A. Zissman

5 January 1987

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Lexington, Massachusetts

Prepared for the Department of the Air Force
under Electronic Systems Division Contract F19628-85-C-0002.

ADA178335

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, with the support of the Department of the Air Force under Contract F19628-85-C-0002.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LINCOLN LABORATORY

AN ARRAY COMPUTER FOR DIGITAL SIGNAL PROCESSING

M.A. ZISSMAN
Group 24

TECHNICAL REPORT 759

5 JANUARY 1987

LEXINGTON MASSACHUSETTS

ABSTRACT

This report describes the implementation of a MIMD array computer designed and built at the Lincoln Laboratory for signal processing. Some of the software tools needed to successfully use such an array are discussed, and the software package written to allow debugging of the array from a host computer is described. The first application of the array, a 12-channel filter bank front-end for a speech recognition system, is discussed. Finally, a block diagram compiler is described. This compiler converts block diagrams, entered at a CAE work-station, into efficient assembly code for all cells in the array.

TABLE OF CONTENTS

Abstract iii
List of Illustrations vii

1. INTRODUCTION 1

2. HARDWARE OVERVIEW 5

2.1 The Cell 5
2.2 The Interfaces 7
2.2.1 Host Network Interface 7
2.2.2 Analog Interfaces 7
2.2.3 MULTIBUS Interface 7
2.3 Implementation 9

3. BASIC SOFTWARE DEVELOPMENT TOOLS 11

3.1 Control Schemes 11
3.1.1 In-Circuit Emulation Method 11
3.1.2 Host Control Method 11
3.1.3 PROMs 12
3.2 Implementation of Host Control 12
3.2.1 Host Control Software 12
3.2.1.1 Host Computer 13
3.2.1.2 Host Network Interface 13
3.2.1.3 TMS32010 Cells 13
3.2.2 Interprocessor Communication 13
3.2.2.1 Intercell Protocol 14
3.2.2.2 HNI/Cell Communication 17
3.2.2.3 Host/HNI Communication 17
3.2.3 Command Example 18
3.2.4 Running and Stopping 19
3.3 Testing and Evaluation 19

4. TWELVE-CHANNEL FILTER BANK 23

4.1 Application Software 23
4.2 Application Hardware 23
4.3 Evaluation 26

5. HIGH-LEVEL SOFTWARE DEVELOPMENT TOOLS 27

5.1 The Block Diagram 31
5.1.1 Primitive Bodies 31
5.1.2 Hierarchy — Nonprimitive Bodies 35
5.1.3 Interconnection of Bodies — Signals 35
5.1.4 The Synchronous vs Asynchronous Issue 35
5.1.5 Partitioning Assignment 39

5.2 Block Diagram Compiler 41
5.2.1 Graphics Modules 44
5.2.1.1 Graphic Data Entry System 44
5.2.1.2 Graphic Sub-Compiler 44
5.2.2 Data Base Generation Modules 44
5.2.2.1 Signal Table Generator 44
5.2.2.2 Primitive Table Generator 47
5.2.3 Partitioning Module — The Splitting and Routing Program 47
5.2.3.1 Splitting a Multi-Cell Application 47
5.2.3.2 Routing Intercell Data Transfers 49
5.2.4 Code Generation Modules 49
5.2.4.1 Ordering Program 49
5.2.4.1.1 C1 — The Simplest Constraint 49
5.2.4.1.2 C2 — Feedback Handling Constraint 50
5.2.4.1.3 C3 — The Delay Handling Constraint 50
5.2.4.1.4 The Asynchronous/Synchronous Constraint 51
5.2.4.2 Assembly Code Generator 57

5.3 Task Assignment Tool 57

5.4 Efficiency and Evaluation 57
5.4.1 Easing the Programming Task 60
5.4.2 Efficiency 60
5.4.3 Conclusion 61

6. CONCLUSIONS 63

Acknowledgments 65

References 67

APPENDIX — USER COMMANDS 69

LIST OF ILLUSTRATIONS

Figure No.		Page
1	IIR and FIR in Series	2
2	Cell Block Diagram	5
3	Array Examples	6
4	Array Examples with Interfaces	8
5	Prototype Test Stand	10
6	Five-Channel Filter Bank Block Diagram	20
7	Twelve-Channel Filter Bank Block Diagram	24
8	Two-Cell MULTIBUS Card	25
9	Second-Order Filter	28
10	Simple Application Block Diagrams — 1	29
11	Simple Application Block Diagrams — 2	30
12	ADDER1, ADDER2, GAIN	32
13	DELAY and GAIN_SPEC	33
14	(a) SOS Body, and (b) Definition	34
15	(a) Two Drawings, Same Meaning; (b) Asynchronous Primitives	36
16	(a) ADAPTIVE_GAIN, and (b) DOWN	37
17	Asynchronous Example	38
18	Diagram with PROC_NUMs	40
19	IIR and FIR in Series	42
20	BDC Flowchart	43
21	IIR/FIR Parsed Expansion File	45
22	Part of the IIR/FIR Signal and Primitive Tables	46
23	Splitting and Routing	48
24	Asynchronous Block Diagram to Flowchart Conversion	52
25	Block Diagram with Asynchronous Primitives	54
26	An Ordered Process Table	55

Figure No.		Page
27	IIR/FIR Process Table; Primitive Macro Programs	56
28	Assembly Code for Cell 0000	58
29	BDC vs Manually Written Code	59

AN ARRAY COMPUTER FOR DIGITAL SIGNAL PROCESSING

1. INTRODUCTION

Many problems in the field of real-time digital signal processing can be solved efficiently using parallel and pipelined architectures. One type of architecture which has been exploited is the array processor. Generally, an array processor consists of (1) a grid of cells where a cell is the basic processing element, and (2) a network of data paths through which the cells communicate with each other. The structure of the cells and the manner in which they are interconnected depend on the particular application.

Besides differing in hardware, each array processor can be categorized by the type of software which it runs. Single instruction stream-multiple data stream (SIMD) machines require that each cell be running the same program.[1] Another class of machine, the pipelined linear array, requires lock-step data communication between cells. A linear pipelined array with adjustable length has been built, with each cell consisting of a multiply-accumulate chip and memory.[2] Finally, multiple instruction stream-multiple data stream (MIMD)[3,4] machines have been suggested.

While SIMD architectures are well suited for highly structured problems, like matrix operations, they are not easily adapted to less-structured problems which arise in digital signal processing. Because each cell in the MIMD machine can execute a different program, the MIMD architecture has a far greater range of applications than the SIMD architecture. This fact was our major motivation for building a MIMD computer. It should be noted that a MIMD machine can emulate a SIMD system if all cells are running the same program.

A general-purpose systolic MIMD digital signal processing architecture has been designed at Lincoln Laboratory.[5] The architecture calls for an arbitrary number of processing cells to be connected in a rectangular array. Each cell can interface up to four other cells. All cells are identical. A cell is composed of a TMS32010 processor, RAM and PROM, four I/O ports, and support logic. The array as a whole communicates with the outside world through (1) A/D and D/A converters which can be connected at an edge of the array, (2) digital interfaces which convert, for example, data from the array into MULTIBUS protocol, and (3) a host network interface (HNI), containing an 8085 microprocessor, which allows array communication with a host computer via an RS-232-C port.

The Lincoln array, being built of discrete "off-the-shelf" components, cannot match the throughput of some of the array processors which have cells on a chip.[6] However, the effectiveness of a system is oftentimes better measured not by its speed in "bits-per-microsecond," but rather by its speed in "answers-per-month".[7] In the past, the drawback of MIMD systems has been that programming them has been difficult. In general, each cell in the array was programmed separately. The purpose of our project was not merely to build hardware, but to build a system which would take as input a high-level description of a task and would output a complete hardware and software design of an array processor which would implement that task in

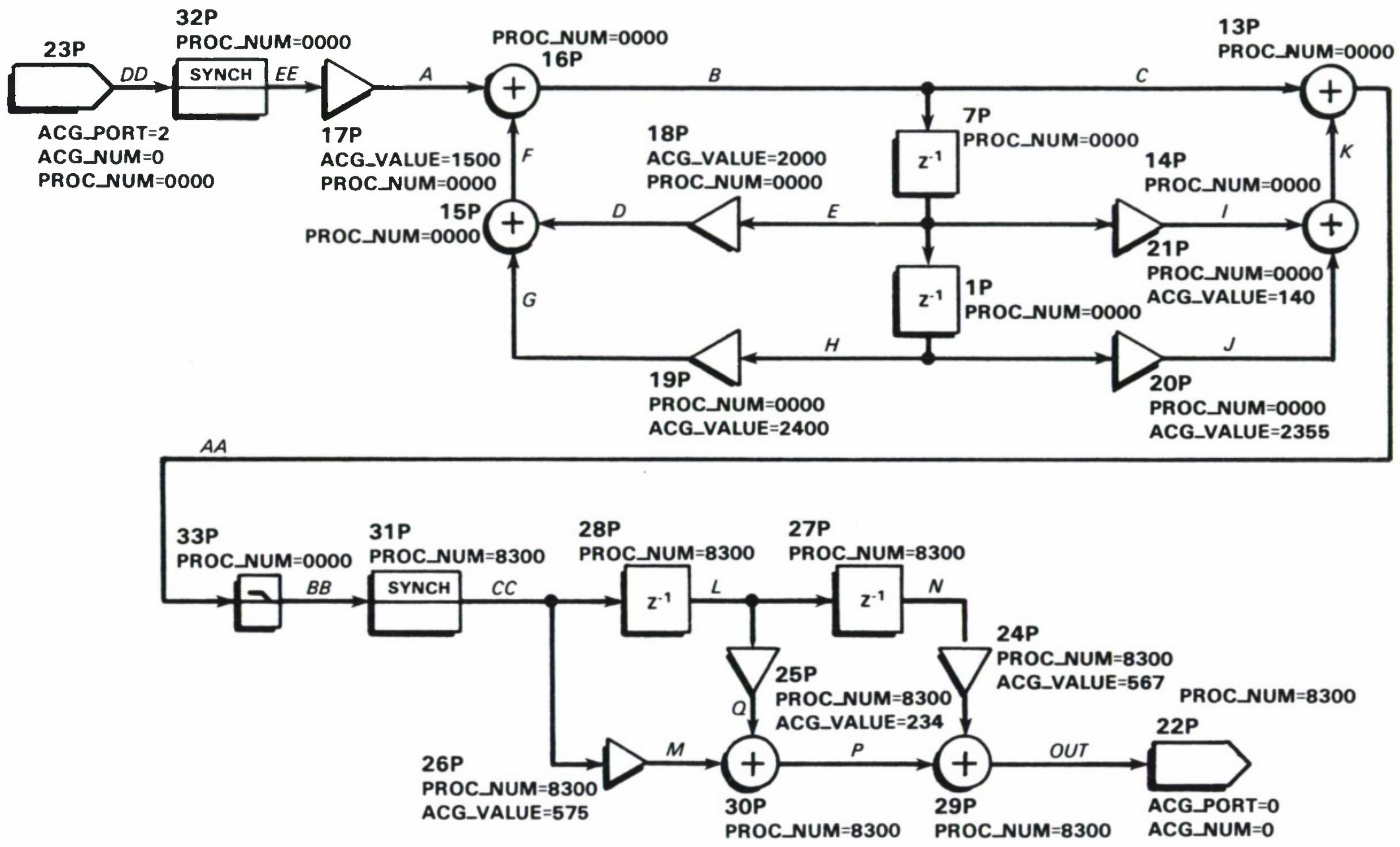


Figure 1. IIR and FIR in series.

76418-1

real time. Blackmer[2] has suggested this end-to-end approach for synchronous linear arrays, and Barnwell has been pursuing this goal for a nonsystolic multi-microprocessor system.[8]

To better understand the motivation for the work presented in this report, consider the user who wants to manually implement an application like an infinite impulse response (IIR) filter in series with a finite impulse response (FIR) filter on a MIMD machine. Ordinarily, the user would first draw a block diagram, such as that shown in Figure 1. Next, he would partition the block diagram among the cells in his machine, perhaps assigning the IIR filter to cell 1 and the FIR filter to cell 2. His third task would be to convert the block diagram into two programs, one for cell 1 and one for cell 2. While writing the programs, he would have to route intercell communication, which would be easy in this example, since data are being passed only from cell 1 to cell 2, but which would be difficult if the application were more complex. After the programs were written, each cell would be downloaded with the proper program. This might be done with In-Circuit Emulation (ICE). Finally, the user would be ready to enter the run-analyze-modify phase until the program was debugged.

As an alternative, the system described in this report can be used to automate and/or simplify most of these software development tasks. The *high-level software development tools* are able to convert a graphic block diagram description of an application, such as that shown in Figure 1, into source code for each cell in the array. This process is called *Block Diagram Compilation.* Using the Block Diagram Compiler (BDC), the user is shielded from the details of assembly coding each cell in the array. In addition, the system automatically routes communication between cells. Since the array is meant to run real-time applications, the code generated by the BDC must be efficient. Once the source code has been generated, *basic software development tools* are used to control the downloading of programs to the array and to debug the application programs. ICE is unnecessary. Both sets of tools are very useful in programming the MIMD hardware described herein.

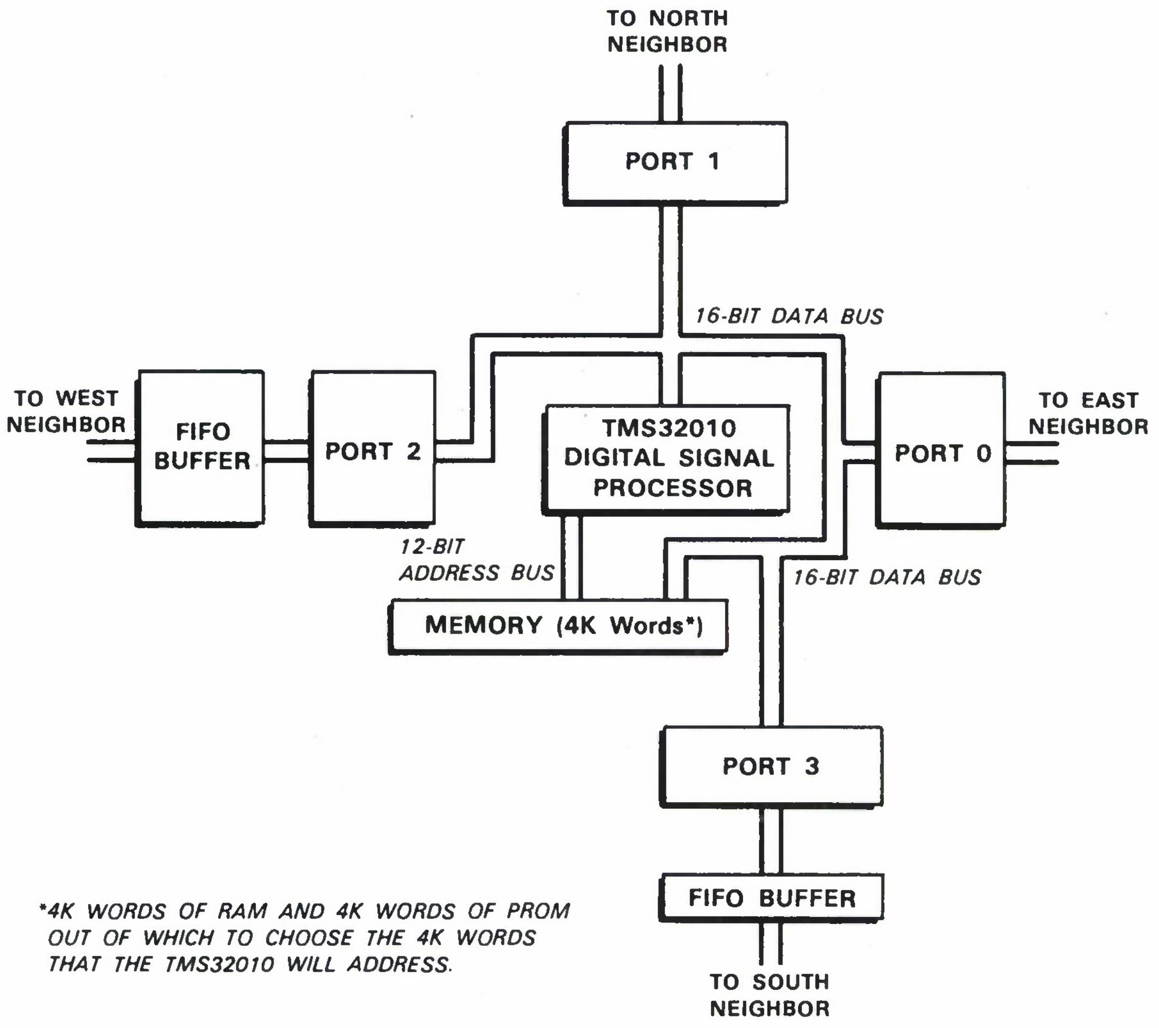
The rest of this report describes: (1) the hardware design of the MIMD computer, the design of the interfaces, and the prototype implementation; (2) the software control system and debugger, including the method of controlling the array from a host computer; (3) the first application of the computer, a 12-channel filter bank for a speech recognition system; and (4) the high-level tools written to ease software development, including the block diagram compiler and the automatic code generator. Finally, some conclusions are drawn and further work is suggested.

2. HARDWARE OVERVIEW

This section describes the hardware comprising the array computer. Beginning with a discussion of the basic processing element, the cell, and continuing with a description of the various array interfaces, the section concludes with a brief description of the prototype system.

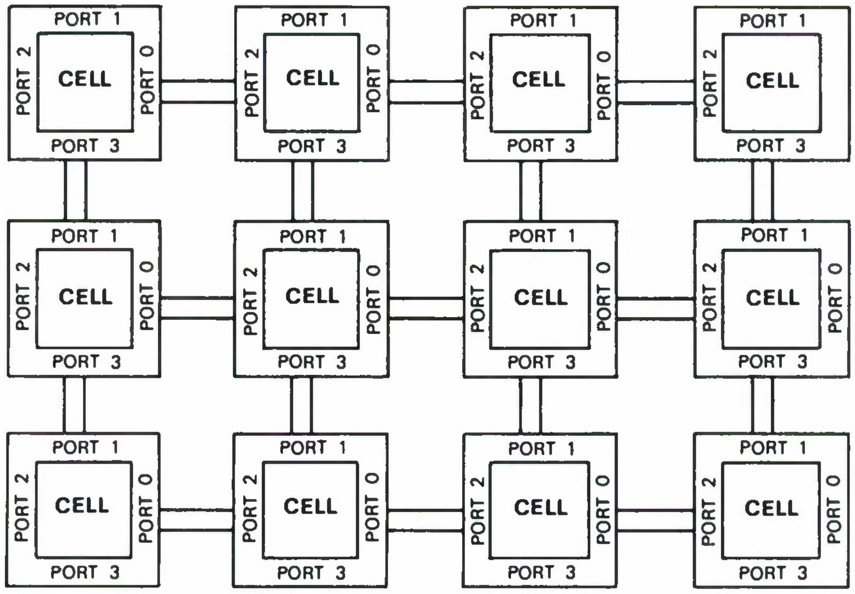
2.1 The Cell

As mentioned previously, the computer designed consists of a grid of identical asynchronous microprocessor-based cells. Each cell in the array consists of a single processing unit with four ports configured so that the cells can be connected into a rectangular grid. Figure 2 is a block diagram of the cell.

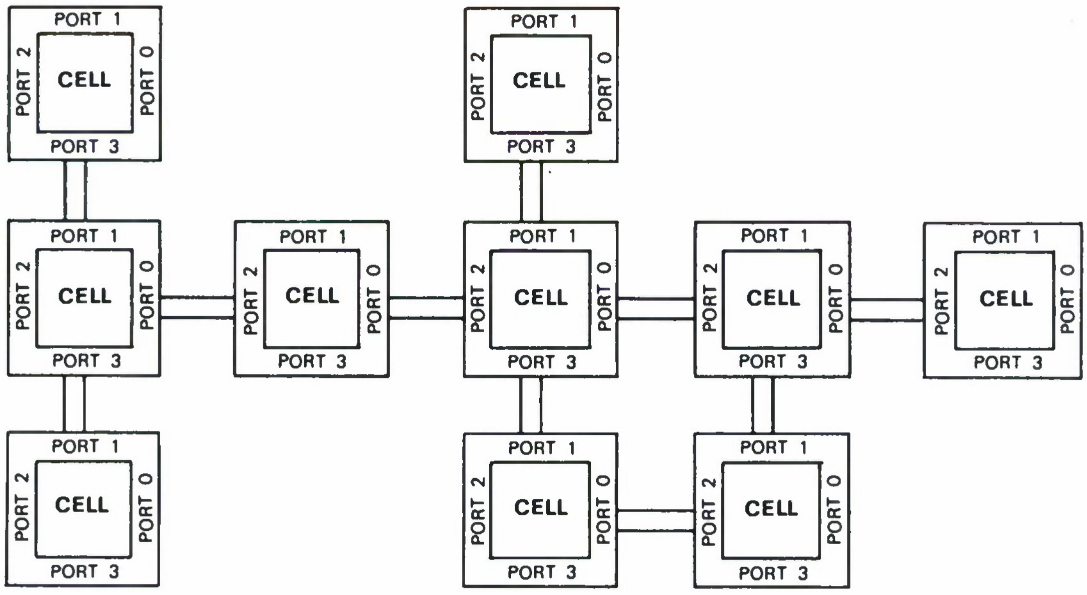


76418-2

Figure 2. Cell block diagram.



EXAMPLE 1 — STANDARD 3×4 CONFIGURATION



EXAMPLE 2 — NONSTANDARD CONFIGURATION

76418-3

Figure 3. Array examples.

The cell processing unit is centered around a TMS32010 digital signal processor, a 16-bit fixed-point processor capable of doing a multiply/accumulate in 200 ns. The chip has 128 words of on-board memory and the ability to access 4K words of external memory. A full complement of RAM and PROM are provided in each cell. The cell also contains four ports that allow it to communicate with its neighbors. Port-to-port data communication takes place over a 16-bit bus which is buffered by 16-word first-in-first-out (FIFO) memories, allowing the cells to operate asynchronously.

Figure 3 shows some examples of how cells are typically connected to form an array. The shape and size of the array are arbitrary and would be chosen by the user with a particular task in mind.

2.2 The Interfaces

There are three interfaces which have been designed and implemented to interface the outside world. The Host Network Interface is used for control of the array. The Analog Interface and MULTIBUS Interface are used for data input and output. Figure 4 shows some array examples with the interfaces connected. The following three sections describe the interfaces.

2.2.1 Host Network Interface

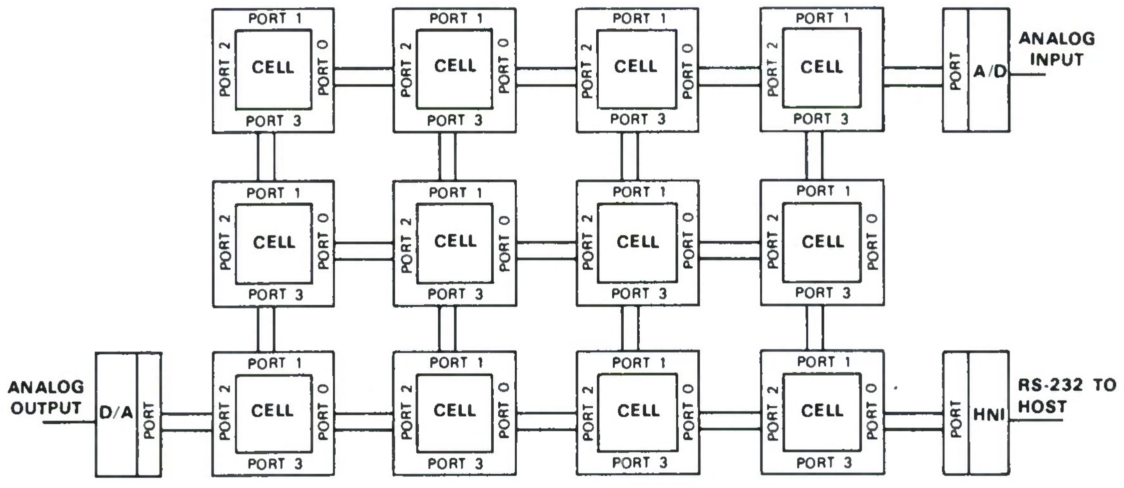
The Host Network Interface (HNI) allows a user access to the array from a host computer. This allows the user to control the array from the host, as discussed in Section 3. The HNI consists of a host interface, a network interface, and a microprocessor-based controller. The host interface, provided by a UART, is a standard RS-232-C connection. The network interface is similar to the standard cell port. Control of the HNI is provided by an Intel 8085 microprocessor, which is programmed to translate the hexadecimal packet data format of the array into the ASCII format used for communication with the host and vice versa.

2.2.2 Analog Interfaces

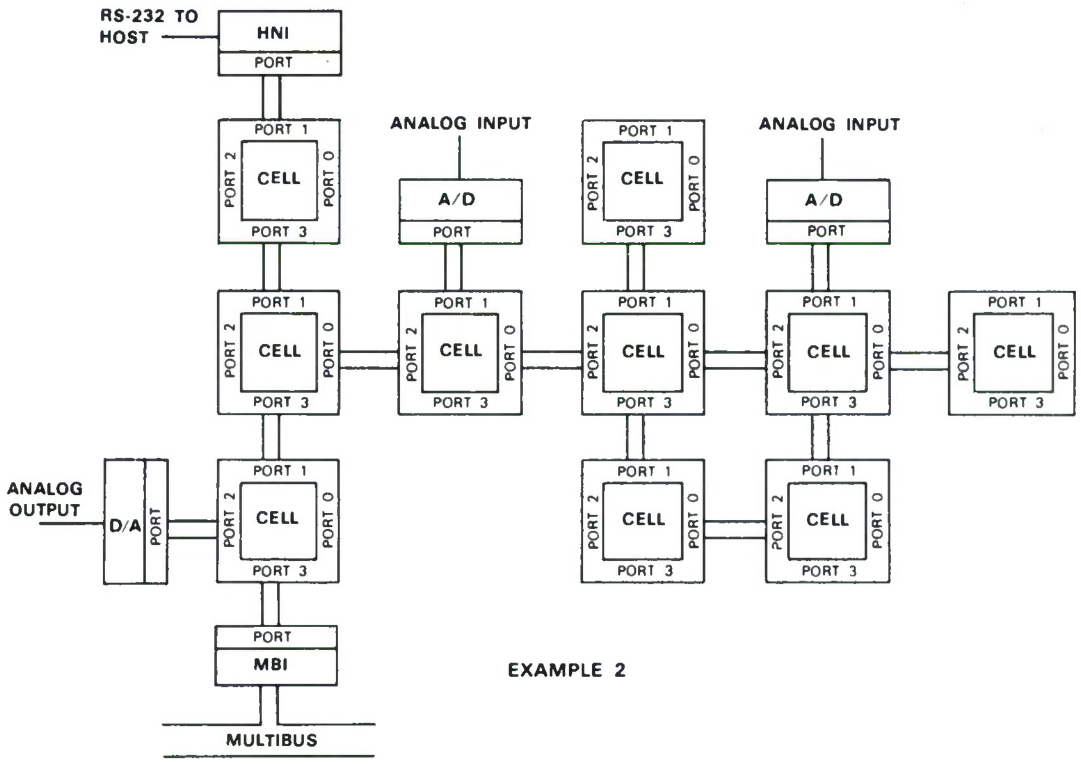
The two types of analog interfaces, A/D and D/A converters, have 12-bit precision and sample at a 10-kHz rate. The analog interfaces also connect to the array through cell-like ports. The A/D allows analog input data entrance to the array, while the D/A permits output of analog data.

2.2.3 MULTIBUS Interface

The MULTIBUS Interface (MBI) was designed to allow array communication with a microcomputer. The MBI is a MULTIBUS slave which interrupts the MULTIBUS controller indicating readiness to transmit or receive data. Once again, the MULTIBUS interface connects to the array through a cell-like port, but it is used to allow digital I/O with the array.



EXAMPLE 1



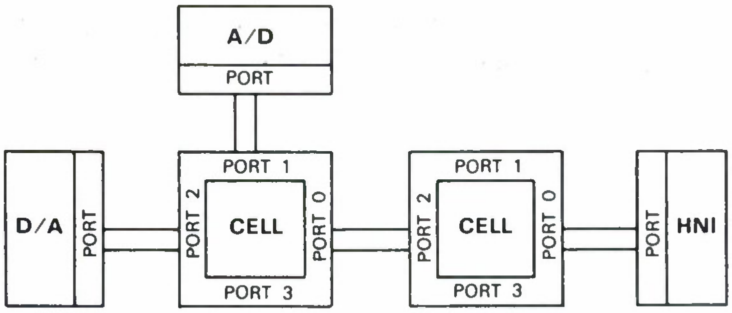
EXAMPLE 2

76418-4

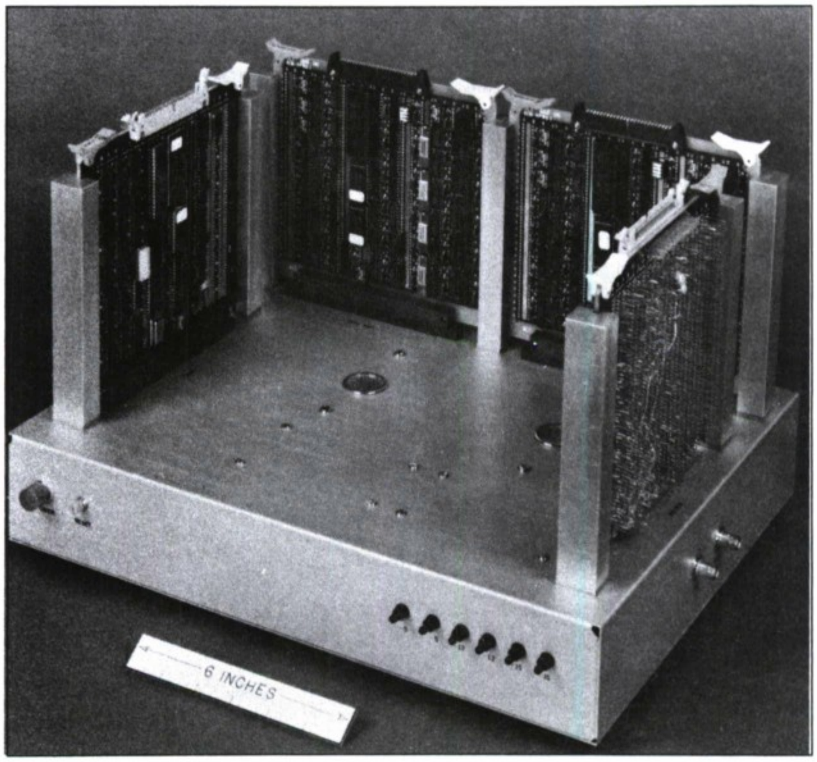
Figure 4. Array examples with interfaces.

2.3 Implementation

The cell, complete with four ports, consists of about 60 chips, and fits on an Augat HPG10 high-density wire-wrap board. For the prototype system, two cells were built, along with a HNI and an analog interface. These four boards and the special-purpose test stand are shown in Figure 5. Connections were made through the 120-pin edge connectors at the bottom of each card.



PROTOTYPE INTERCONNECTIONS



76418-5

TEST STAND PHOTOGRAPH

Figure 5. Prototype test stand.

3. BASIC SOFTWARE DEVELOPMENT TOOLS

This section describes the control and debugging system designed and implemented for the array processor. The first subsection outlines the three control schemes; the second subsection details the most useful of the three schemes, which also happened to be the most difficult of the three to implement; the final subsection offers an evaluation of the complete control and debugging package.

3.1 Control Schemes

Three methods of controlling the array exist, with each method appropriate to a different phase of development. First, the In-Circuit Emulation scheme is valuable during the hardware verification phase. Next, the Host Control method is useful during software development. Finally, the PROM method is helpful after software verification, when an array is to be used for one special-purpose application. A description and evaluation of each of these schemes follow.

3.1.1 In-Circuit Emulation Method

The first method of control is In-Circuit Emulation (ICE). Texas Instruments markets an emulator for the TMS32010 called an Evaluation Module (EVM). Each module has a cable which plugs into a cell in place of the TMS32010. Through this cable, the EVM controls the operation of the cell. The EVM contains its own memory, two RS-232-C ports for communication to a host and to a terminal, as well as most of the common microprocessor development debugging tools. In addition to the Texas Instruments product, Hewlett Packard has made available a TMS32010 adapter for its Model 64000 microprocessor development workstation.

For small arrays (less than four cells), two emulators could be used to debug the whole array. However, larger arrays would require more and more emulators, which would become expensive. Therefore, use of emulators is best restricted to the hardware verification phase, when the ability to control one or two cells at a time is all that is necessary.

3.1.2 Host Control Method

A second control scheme is the Host Control method. The goal of the Host method of control is to provide the user with an interactive method of controlling the array during the debugging phase of software development, after the hardware has been verified operational. The user is allowed to issue commands, through a host computer and HNI, to the array. These commands are as simple as ***run*** and ***stop*** or as complicated as ***modify data memory 100 5 5A34***, where the first number specifies the address of a cell, the second number specifies a memory address within the cell, and the third number specifies the new value to be written at that address. In addition, a facility for the downloading of programs from the host to the array is provided. This method of control allows the greatest flexibility and most powerful debugging.

The Host method of control necessitates two separate array modes. While the array is in *application* mode, the individual cells are doing some sort of signal processing application. Contrasting with application mode is *command* mode, during which individual cells are expecting commands to be coming from the host. The array is toggled between application mode and command mode by using the ***run*** and ***stop*** commands. Commands cannot be issued in application mode, i.e., the array must be stopped before command processing can begin.

3.1.3 PROMs

The third method of control calls for a set of PROMs to be burned for each cell in the array. Each set of PROMs contains the application program that the particular cell is going to run, along with a standard group of routines for intercell communication. This method of control is only appropriate when an array is to be used for one special-purpose application for which error-free software already exists. This control scheme is a "final" phase control scheme, to be used after the hardware and software have been verified using the other two control schemes.

3.2 Implementation of Host Control

This section outlines the design and implementation issues of Host Control. While each of the control methods is useful during different development stages, and while all three have been demonstrated, the ICE modules and PROMs are commercially available, so their associated control methods did not require new implementations. On the other hand, the Host method of control was designed and implemented from scratch. Host Control requires that three different kinds of processors work together: (1) the host computer, (2) the HNI, and (3) the cells. These processors split the task of interpreting and executing the user commands. In addition, three different types of interprocessor communication protocols are necessary: (1) intercell, (2) HNI/cell, and (3) host/HNI. These protocols specify the rules for interprocessor communication. The next few sections outline the task of each of the three processors during Host Control and detail the interprocessor protocols. Finally, an example is presented to help clarify the discussion.

3.2.1 Host Control Software

The goal of the Host Control software is to enable the user to issue commands from the host which the array is able to understand, execute, and acknowledge. The set of instructions includes the ability to read and write, for any cell in the network, the following:

TMS32010 Register Set,

TMS32010 Program Counter,

TMS32010 Stack,

TMS32010 Data Memory,

Cell Offchip Memory.

The Host Control software design attempts to divide the tasks of interpretation and execution of the user commands in an "intelligent" way among the host, HNI, and cell. By "intelligent" we mean that the division of labor was made after considering the strengths and weaknesses of each processor. The next three paragraphs describe the function of each processor during Host Control.

3.2.1.1 Host Computer

The host computer, a VAX 11/780 running UNIX, provides an interface between the user and the rest of the system. Commands are issued at a video display terminal, and acknowledgments from the array are displayed thereon. The user is able to see results of a command in progress as well as a completed command. Because the host computer provides high-level programming languages, and because there is a great deal of memory available, as much of the command interpretation as possible takes place on the host computer. The host computer is connected to the HNI through a 9600-b/s RS-232-C link.

3.2.1.2 Host Network Interface

The HNI provides the interface between the host computer and the array. Its primary function is to convert serial ASCII data from the host computer into the binary data format of the array, and vice versa. The HNI has only 4K bytes of memory, so its program is fairly compact. In order to maximize memory use, the HNI program is written in assembly language. Its output to the cell network is simple for the cells to interpret.

3.2.1.3 TMS32010 Cells

Each cell in the network has a "kernel" program containing instructions for the interpretation of commands coming from the HNI. This entire program for data communication is less than 1K words (25 percent of available program memory). The program, written in assembly language, allows the cell to decode the command arriving through the network from the HNI, perform the specified action, and respond accordingly.

3.2.2 Interprocessor Communication

While the preceding sections briefly described the Host Control software for each of the three processor types (i.e., host, HNI, and cell), the next few sections describe the communication scheme implemented for the command mode of the Host method of control. These sections do not apply to application mode issues, nor do they deal with ICE or PROM control methods. Communication in these other modes and control methods are left up to the user or to the high-level software development system described later in this report. To summarize, this section explains the communication protocols used when the user types a command such as ***modify data memory 100 5 5A34***.

There were four major design specifications imposed on the command mode communication system. First, the following scenario was proposed for all commands. The array would be attached to the host by exactly one HNI. Commands typed by the user to the host would be processed on the host and sent to the HNI. The HNI would do further processing and then send the command to the cell to which it was connected. In turn, each cell would read the command header, and determine whether it was the destination of the command. If it was the destination, it would execute the command and return an acknowledgment toward the direction from which the command came. If it was not the destination, it would forward the command in the direction of the destination cell and wait for an acknowledgment from that same direction. Upon receiving the acknowledgment, the cell would forward the acknowledgment message back toward the direction from which it received the command initially. Eventually, the HNI would receive the acknowledgment and forward it to the host. The communication channel would remain open until the command had been executed by the destination cell and an acknowledgment had been returned from the destination cell, through the array, through the HNI, to the host. Only one communication channel would be open at any one time.

The second design constraint was that the system had to be fast enough to respond to the user's commands within a reasonable time frame. While no time limits were specified, it seemed realistic that simple commands (***start*** and ***stop***) should be almost immediate (less than 1 s), while more complicated commands (***load*** and ***display data memory***) could take somewhat longer (e.g., 5 s).

The third design goal was to implement relative addressing of cells rather than absolute addressing. Absolute addressing would have required a switch pack on each cell for specifying its address. Relative addressing alleviated this complication without affecting the software complexity.

Finally, the last design specification was that the cell portion of the communication software had to be as compact as possible, because cell external data memory is quite limited (4K words). A limit of 1K words, 25 percent of available external memory, was placed on the cell communication software length.

With these four design specifications in mind, a summary of the three types of communication (i.e., intercell, HNI/cell, and host/HNI) is presented. The three protocols were chosen to match the strengths of the processors which use them.

3.2.2.1 Intercell Protocol

The intercell protocol calls for data to be transferred from one cell to another in packets. Each packet contains a header indicating (1) the final destination of the data, (2) the number of words of data in the packet, and (3) some information regarding the nature of the data. The header itself is from one to three words long, with the length of the header specified in the first header word. The header contains enough information for the receiving cell to determine what to do with the rest of the packet. An initial scheme for intercell data transfer was proposed earlier.[5]

A modified version of that initial scheme is used in the current implementation, and its description follows.

The header information preceding a data block transmission can be up to three words long (1 word = 2 bytes). The first word is mandatory and contains various control bits. The second word is optional and, if present, contains either block length or routing information. The third word is also optional and, if present, contains routing information. Header word 1 is decoded as follows:

Header Word 1 — Upper Byte							
15	14	13	12	11	10	9	8
len	*rout*	*comm*	*ack*	*rsv*	*rsv*	*rsv*	*rsv*

len:	1 if a length word is present in the header. 0 otherwise.
rout:	1 if a routing word is present in the header. 0 otherwise.
comm:	1 if this header is the header for a command. 0 otherwise.
ack:	1 if this header is the header for an acknowledgment. 0 otherwise.
rsv:	reserved for further command information.

Header Word 1 — Lower Byte							
7	6	5	4	3	2	1	0
len7	*len6*	*len5*	*len4*	*len3*	*len2*	*len1*	*len0*

len7 to *len0* is the length of the data block following; ignored if *len* = 1. If *len* = 0 and *rout* = 0, then the data block length can be found in header word 1 and the cell receiving the data block is the destination. If *len* = 0 and *rout* = 1, then the data block length can be found in header word 1 and the second header word contains the routing information. If *len* = 1 and *rout* = 0, then the second header word contains the number of words in the data block and the cell receiving the data block is the destination. If *len* = 1 and *rout* = 1 then the second header word contains the number of words in the data block and the third header word contains the routing information. Shown in tabular form:

Header Composition					
len	*rout*	Number of Header Words	Contents†		
			Word 1	Word 2	Word 3
0	0	1	Block Length	x	x
0	1	2	Block Length	Routing	x
1	0	2	‡	Block Length	x
1	1	3	‡	Block Length	Routing

† Besides the contents listed here, word 1 always contains control information.

‡ Word 1 contains *only* control information in this case.

The word representing the number of words in the transmission is simply the 16-bit two's-complement binary representation of the number of words. Thus, 32K — 1 is the largest possible block size. A TMS32010 can only store 4K words, so this block size constraint is not a problem. For block sizes of 256 bytes or greater, the header must contain a block length word.

The word representing the routing directions can be decoded as follows:

Routing Instructions — Upper Byte							
15	14	13	12	11	10	9	8
east	*horz6*	*horz5*	*horz4*	*horz3*	*horz2*	*horz1*	*horz0*

east:	1 = east, 0 = west
horz6 to horz0:	binary representation of how many cells to the east (or west) the destination is.

Routing Instructions — Lower Byte							
7	6	5	4	3	2	1	0
north	*vert6*	*vert5*	*vert4*	*vert3*	*vert2*	*vert1*	*vert0*

north:	1 = north, 0 = south
vert6 to vert0:	binary representation of how many cells north (or south) the destination is.

When a cell receives a header with a routing word, it modifies the routing word before passing the header to its neighbor. Each cell decrements either the upper or lower byte of the routing word, depending on the direction of the data. For example, if a cell receives the routing word:

1000 0011 0000 0010

from its west neighbor, it will send:

1000 0010 0000 0010

to its east neighbor. By convention, cells attempt horizontal transmissions before vertical transmissions; i.e., in this example, the data will be passed two more cells east before being passed two cells south. Using a relative addressing scheme such as this has the advantage that a specific cell does not have to know its position in the network. If absolute addressing were used, each cell in the network would have to know its own position, requiring either a different program for each cell or a switch pack.

3.2.2.2 HNI/Cell Communication

The second type of communication is that between the HNI and the cell. As noted in the discussion of the hardware, the HNI is connected to one cell in the network. That cell does not treat its connection to the HNI any different from a connection to a cell. Therefore, the HNI/cell communication protocol is identical to the intercell packet protocol described above.

3.2.2.3 Host/HNI Communication

The final communication link is that between the host and HNI. Similar to the intercell communication, the host/HNI communication also uses packets. In this case, however, the packets are transmitted in ASCII bytes. ASCII characters are used because (1) the operating system of the host is designed to normally send and receive ASCII characters, (2) the line analyzer applied to the RS-232-C cable during the debug phase is easier to read if ASCII characters are being transmitted, and (3) the higher speed afforded by binary data transmission is not necessary for this application. The header contains 17 bytes. The first 8 bytes contain the command name. The next 4 bytes contain the hexadecimal representation of the number of bytes in the data portion of the packet. The next 4 bytes contain the address of the destination cell, also represented in hexadecimal notation. The last byte is a line feed. Shown in a table form, we have:

Host/HNI Packet Header — 17 Bytes			
0-7	8-11	12-15	16
Command Name	Block Length	Cell Address	Line Feed

3.2.3 Command Example

Instead of delving into the details of what interpretation and translation occurs where, an example command trace is illustrated for the reader. A complete list of commands available is contained in the Appendix.

Suppose the user would like to display the data memory of the cell at location 0100H (a digit string followed by an "H" represents a hexadecimal number; a digit string followed by a "B" represents a binary number). This cell is immediately west of the cell connected to the HNI. He would issue the command:

display data memory 0100 .

The host interprets this command, and sends the following data over the RS-232-C connection to the HNI:

```
getdmem<sp>00000100<lf>
```

where <sp> is the ASCII space character, and <lf> is the ASCII line feed character. The first eight characters are the command name. The next eight are the length and routing information. The last character sent is a line feed. All command names must be eight characters long, hence the need for a space in this case. The first four zeros following the command specify the number of words following the 17-character header, and in this case indicate that no other information is coming from the host. The next four characters form the address of the cell for whom this command is intended. In this case, the address is 0100H.

The HNI receives this packet, which in this case is just the 17-character header, and translates the ASCII data into binary data which can be understood by the cell array. In this example, it sends a two-word header with no data:

First Word — 6200H = 0110 0010 0000 0000 B

Second Word — 0100H = 0000 0001 0000 0000 B.

The first header word indicates that (1) there won't be a length word (bit 15 = 0B), (2) there will be a routing word (bit 14 = 1B), (3) this header is the header of a command (bit 13 = 1B), and (4) the length of the data portion of the packet is 0B (lower byte = 00H). In addition, bits 8-11 of the first word indicate that the command is a ***get data memory*** (command code 0010H = get data memory). The second header word is the routing word indicating that the relative address of the destination cell is 0100H. The cell connected to the HNI reads this header, realizes that the command is not intended for it, decrements the routing word, and passes the command to its neighbor to the west:

First Word — 6200H = 0110 0010 0000 0000 B

Second Word — 0000H = 0000 0000 0000 0000 B.

The next cell reads the header, and realizes that it is the destination. After some decoding, it recognizes the command as a ***get data memory*** and sends an acknowledgment, which contains a header and the desired data memory information, back to the cell from which the command came. Specifically, the acknowledgment contains a two-word header and 90 words of data:

First Word — 5290H = 0101 0010 1001 0000 B

Second Word — 0000H = 0000 0000 0000 0000 B

90 Words of Data Memory Contents.

The cell attached to the HNI, which has been waiting for the acknowledgment, increments the routing word and forwards the acknowledgment to the port from which it received the original command:

First Word — 5290H = 0101 0010 1001 0000 B

Second Word — 0100H = 0000 0001 0000 0000 B

90 Words of Data Memory Contents.

The HNI translates the binary data into ASCII and sends it to the host:

getdmem<sp>00000100<lf>{360 characters of data} .

Acknowledgments from the HNI to the host have headers identical to the last command issued by the host. The host reads the data and displays it on the screen of the user's terminal.

3.2.4 Running and Stopping

Running and stopping the array, i.e., toggling from command mode to application mode and application mode to command mode, is done using a hardwired signal available to all cells. This signal, ATN*, is generated by the HNI. On reception of either a ***stop*** or ***run*** command from the user, a pulse is transmitted on ATN* which interrupts the TMS32010's. By convention, the first pulse on the ATN* line is a ***start***. After that, the meaning of the pulse alternates between ***stop*** and ***start***. The cell interrupt routine causes the cell to begin its application program on interpretation of a ***start***. On a ***stop***, the interrupt routine causes the cell to enter command mode.

Notice that these two commands are "broadcast," making them different from all the other commands which must propagate along the array data paths.

3.3 Testing and Evaluation

The software and communication schemes described above were first tested on an array of two cells. Using the EVMs, PROM code was developed for the kernel program. The RAMs of the cells were successfully loaded with application programs using the load instruction of the controller. The array was started, stopped, probed, modified, restarted, etc. In this manner, the application programs were debugged. Eventually, the two cells working in tandem were able to perform a five-channel filter bank with pre-emphasized input and companded output. Figure 6 shows a block diagram of this application.

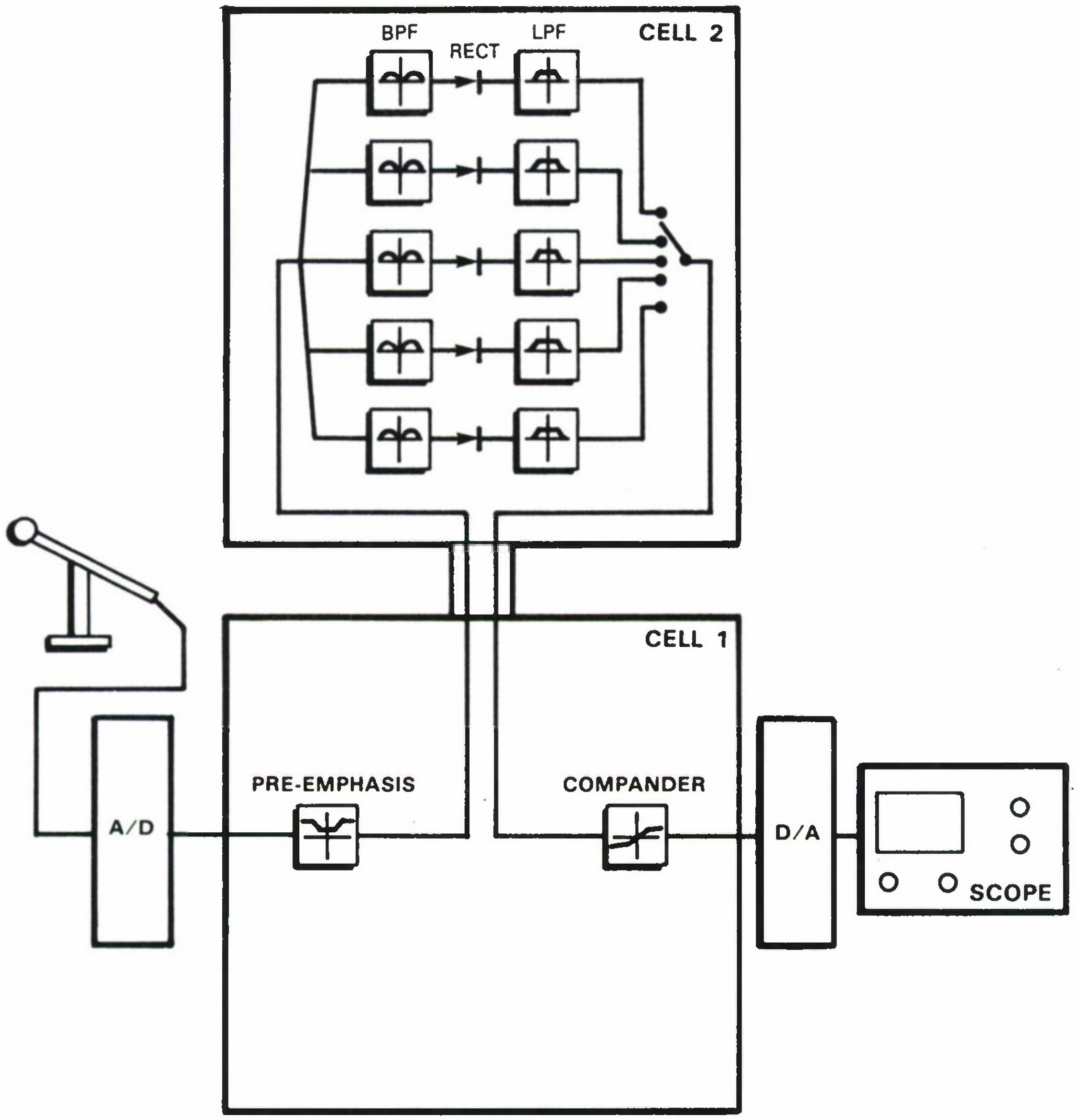


Figure 6. Five-channel filter bank block diagram.

The implemented array met all four of the original design constraints, namely, (1) the command-execute-acknowledge procedure, (2) fast response, (3) relative addressing, and (4) cell communication program of less than 1K words. In this regard, the host/HNI/array software package was deemed a success.

One unfortunate aspect of the cell program memory restriction is that a completely general cell communication program would not fit within 1K words. Thus, the program run by each cell will work only properly in a rectangular array. Arrays with strange snakelike configurations would pose a problem for the current version of the cell software. While the control system worked well and was very helpful in debugging the application software, there are two limitations which are worthy of mention. First, there is no provision in the array for hardware breakpoints. The reason for not implementing full hardware breakpoints was the added hardware complexity cost. A full hardware breakpoint would have allowed a user to (1) specify a breakpoint for any cell, (2) start the array, and (3) have all cells in the array enter command mode on any cell's entrance into a breakpoint. This entrance into command mode in unison would be accomplished through another common signal similar to ATN*, called BRK*. Hardware contained on each cell would decode the address used to fetch program instructions and would assert BRK* if and when the breakpoint address was identified. A scheme which would be less intensive hardware-wise would be to reassemble a cell's application program with an instruction to assert this new breakpoint signal, BRK*, after entrance into the breakpoint. Future versions of the array might implement this feature.

The second limitation of the controller is that the array cannot be single-stepped. Since any "apparent" single-step would require a number of real steps in every processor to allow packet communication among the cells, it seemed that single-stepping would not aid in identifying errors. In addition, the amount of software which would have had to be resident on the cell to handle a single-step feature would have increased the command software well beyond its 25-percent limit.

4. TWELVE-CHANNEL FILTER BANK

The first application run by the array was a front end for the Lincoln Laboratory's Dynamic Time Warping (DTW) Speech Recognition System.[9] This system needed a real-time 12-channel filter bank front end which could output average power estimates for each channel of analog speech input. While there were other available means for implementing this front end, each had drawbacks. An analog filter bank could have been built, but changing filter parameters would have required modifying the hardware, which would have been undesirable. A digital filter bank could have been programmed on a Lincoln Digital Signal Processor (LDSP), which is a 50-ns instruction cycle special-purpose processor. The LDSPs, however, are housed in large racks (6 ft high, 3 ft wide, 3 ft deep), which would have made the DTW system immobile. For these reasons, as well as for experimental purposes, the array processor described in this report was built to implement the filter bank. The next few sections describe the software and hardware design decisions made in this first application.

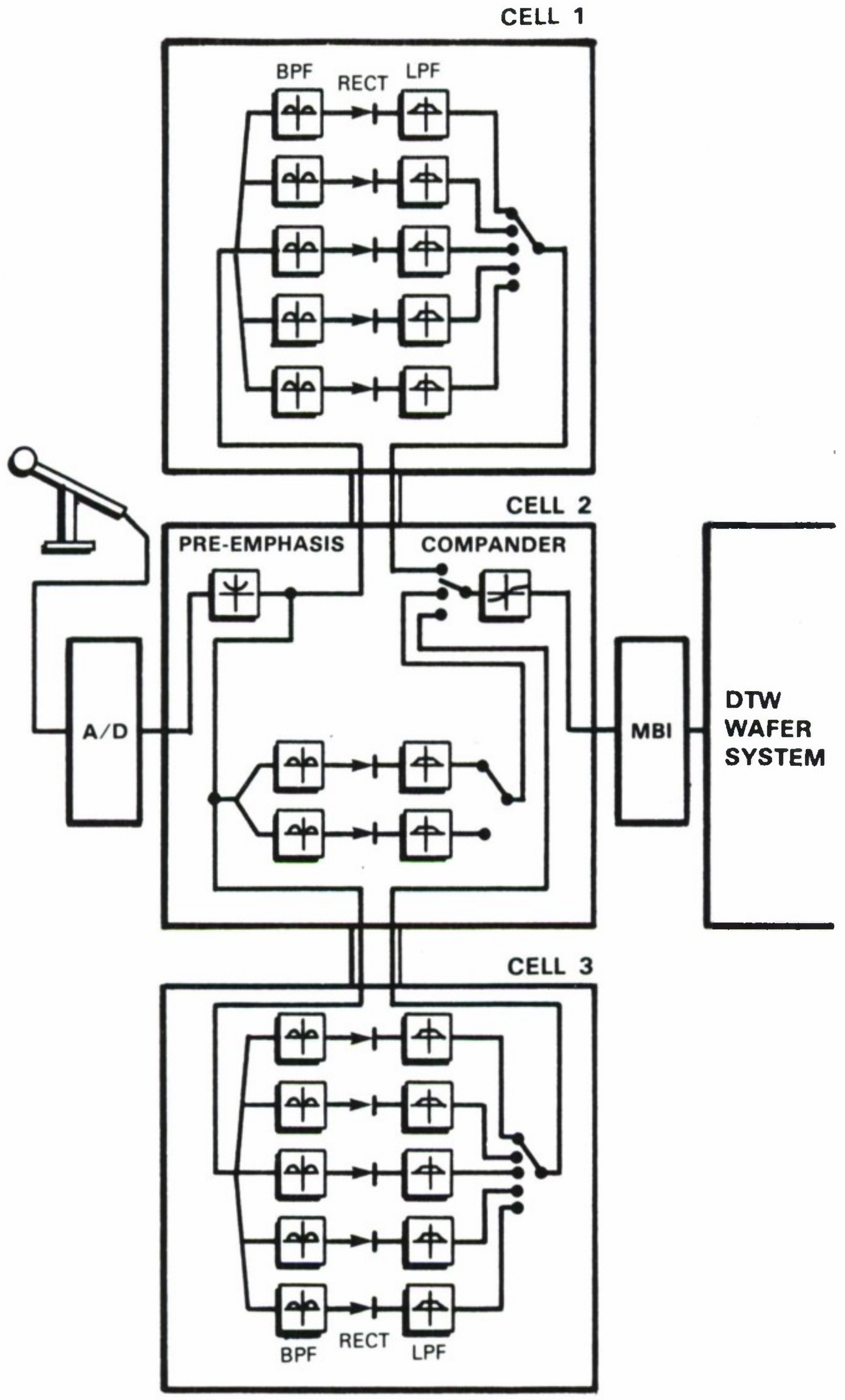
4.1 Application Software

Each channel of the filter bank produces a power estimate for a specified band every 10 ms. Three cells are required for the 12-channel front end. Each channel runs the same software, except for the filter specification constants. The input is pre-emphasized, boosting the high end for speech recognition purposes. In each channel, the pre-emphasized signal is fed through a two-section band-pass filter, where each section is a second-order Butterworth filter. The filtered output is rectified, adding a D.C. component to the spectrum. Following rectification, the signal is low-pass filtered and down-sampled. After down-sampling, μ-law compression is performed. It is this compressed output which is transmitted over the MULTIBUS to the DTW wafer.

After writing some blocks of code to do these various functions, it was estimated that three cells would be needed to run the application in real time. Figure 7 shows the layout of cell interconnects as well as cell function. The application code is about 1/2K-word long in each cell, so program memory is not constraining. The actual constraint is the real-time restriction. Using three cells, each cell is processing about 80 percent of the time. I/O overhead is held below 10 percent. It would have been impossible to use only two cells and still meet the real-time restriction.

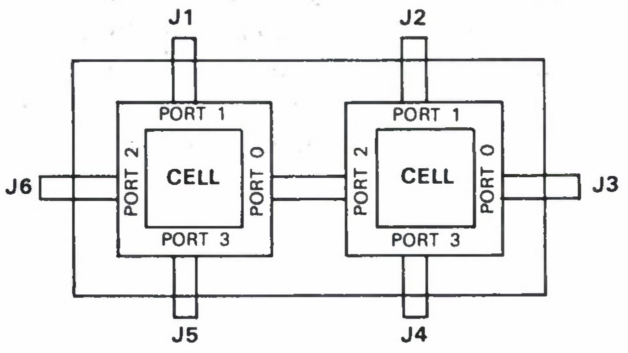
4.2 Application Hardware

In order to integrate the array into the DTW card cage, the prototype design was repackaged on two MULTIBUS wirewrap cards. Card 1 contains two cells and is shown in Figure 8. Card 2, which looks similar to card 1, contains one cell, one HNI, an A/D, and an MBI. Both cards have been built and tested.

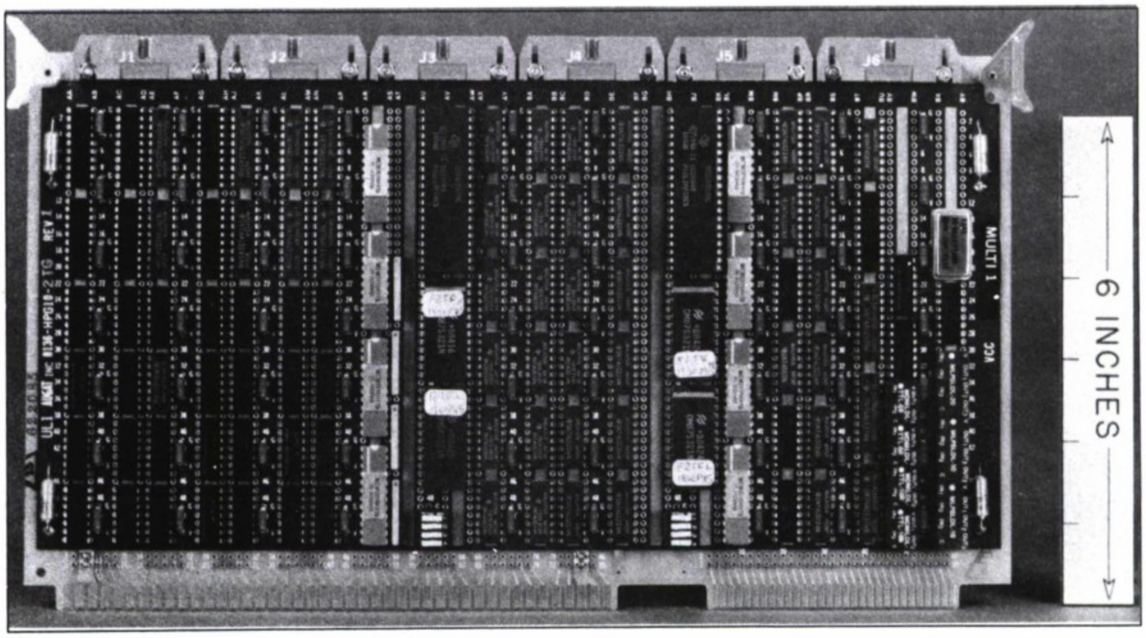


76418-7

Figure 7. Twelve-channel filter bank block diagram.



TWO-CELL MULTIBUS CARD
INTERCONNECTIONS



TWO-CELL MULTIBUS CARD PHOTOGRAPH

Figure 8. Two-cell MULTIBUS card.

76418-8

4.3 Evaluation

The low-level software development tools described in Section 3 performed admirably during the debug phase of the filter bank system. Debugging the DTW front end consisted mainly of (1) tracking down noise in the inter-card MULTIBUS hardware, (2) reviewing the TMS32010 fixed-point overflow mechanism, and (3) establishing communication between the array and the MULTIBUS system CPU. As of this writing, the hardware and software serving as the DTW front end have been completely tested.

5. HIGH-LEVEL SOFTWARE DEVELOPMENT TOOLS

MIMD systems, such as the one described in this report, are often very difficult to program since the programmer must (1) partition the problem among the cells, (2) route intercell data transfers, and (3) write different code for each cell in the array. To make this MIMD array processor attractive to use, development tools were created to automate steps (2) and (3) listed above. In addition, an automatic partitioning tool, to ease step (1) of the programming problem, was developed outside the scope of this report and is briefly described at the end of this section. While the system I implemented requires manual partitioning of the problem, future versions will incorporate an automatic partitioning tool, resulting in a completely automatic software development system.

The procedure for developing software is based on the system block diagram, since block diagrams are the natural means of describing most signal processing applications. The user begins by drawing a block diagram of his application on a computer aided engineering (CAE) workstation. Figure 9 is an example of a block diagram for a second-order filter section. The ADDER, GAIN, and DELAY blocks represent functions, while the lines connecting the blocks represent data paths. A block diagram compiler (BDC) was written which (1) converts each of the blocks on such a drawing into pieces of TMS32010 code, and (2) links the individual pieces of code into a complete program. The data paths represent input arguments to functions and output values from functions. The BDC converts these data paths into TMS32010 data memory locations. The final output of the software development system is a "ready to be assembled" source program for each cell in the array.

There are a number of issues which complicate the seemingly straightforward block diagram compilation. First, there must be a provision for assignment of block diagram bodies (blocks on block diagrams are called "bodies") to physical cells, since, in general, the user will want to partition his application among the available cells in his array. Furthermore, intercell data transfer routing should be automatic. Figure 10(a) shows two GAINS, one assigned to cell 0000 and one assigned to cell 8283, connected by signal A. Since cell 0000 is not adjacent to cell 8283, the actual path of the data out of the first GAIN and into the second may be fairly complicated, e.g., 0000 may pass the value to 8100, which will pass it to 8200, etc. The BDC should automatically route such intercell transfers.

The second, and most difficult, problem facing the BDC is the fact that a block diagram cannot be directly translated into source code since the block diagram may specify completely parallel processing while the actual hardware consists of processors which run sequential programs. Figure 10(b) shows a simple application. The signal A is needed for both GAINs, and both GAINs are assigned to processor 0000. Since processor 0000 contains a single TMS32010, it cannot execute both GAINs in parallel, i.e., only one GAIN can be executed at a time. In this simple case, the ordering of the two GAINS is arbitrary, since they do not depend on each other. In Figure 10(c), an ADDER has been added to the drawing. Again, all three bodies have been assigned to processor 0000. Clearly, once signal A has arrived, both GAINs must be executed

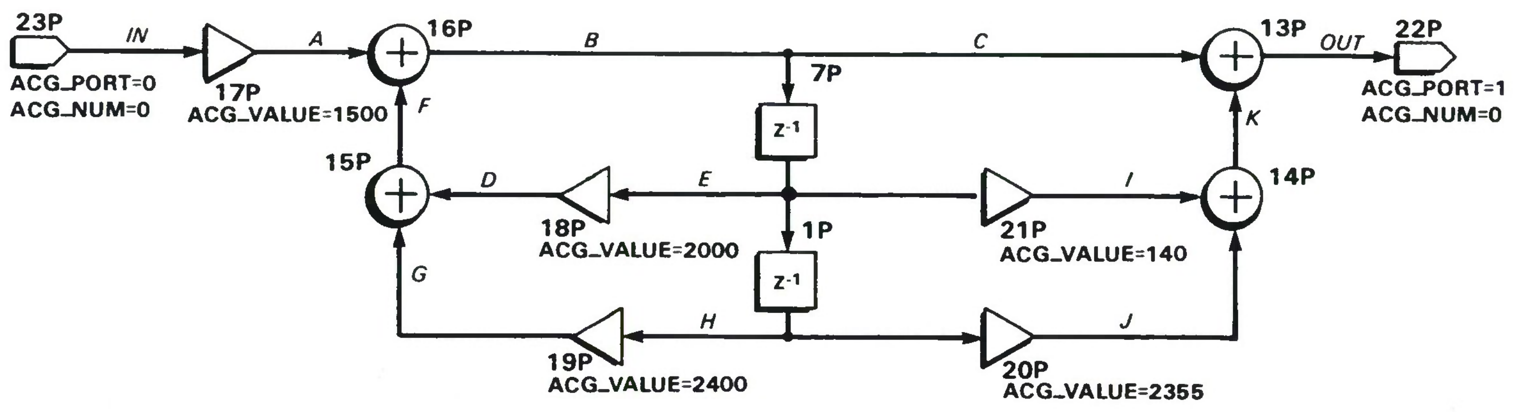


Figure 9. Second-order filter.

76418-9

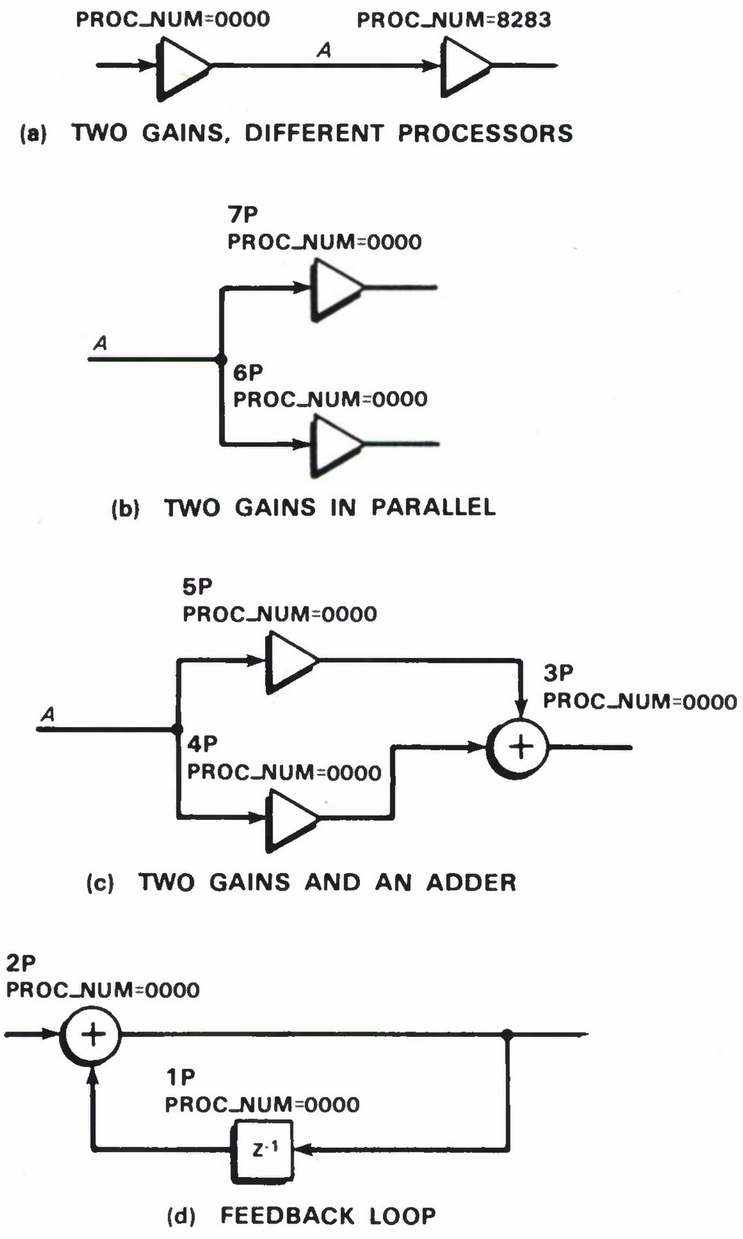


Figure 10. Simple application block diagrams — 1.

before the ADDER is executed. DELAYs further complicate the issue. Figure 10(d) shows a simple feedback loop. The ADDER cannot be executed until both of its inputs are valid, but the bottom input will not be valid until the DELAY has executed. The DELAY cannot be executed until its input is valid. Since the DELAY input is the ADDER output, the DELAY cannot execute before the ADDER. This example, which seems to cause deadlock, must be properly handled by the BDC. The solution is to allow every DELAY to have an initial value which it can output before it has received its first input. Another problem is caused by a body which either (1) does not need all of its inputs to generate one of its outputs, or (2) does not necessarily generate one of each of its outputs on reception of all of its inputs. As one example of the first class, consider the drawing in Figure 11(a) which shows the ADAPTIVE_GAIN body. This body multiplies its two inputs and places this value at the output. The ADAPTIVE_GAIN needs a new value of its first input to generate each output, but the second input is stored and may be only irregularly updated, i.e., ADAPTIVE_GAIN does not need a new value of its second input in order to generate an output. Figure 11(b) shows the ADAPTIVE_GAIN used in an application. The ordering process must correctly account for the special nature of the ADAPTIVE_GAIN, i.e., allowing it to execute even if its second input has not been given a new value. These examples show that an ordering algorithm must be developed for the BDC to correctly convert a block diagram into sequential code.

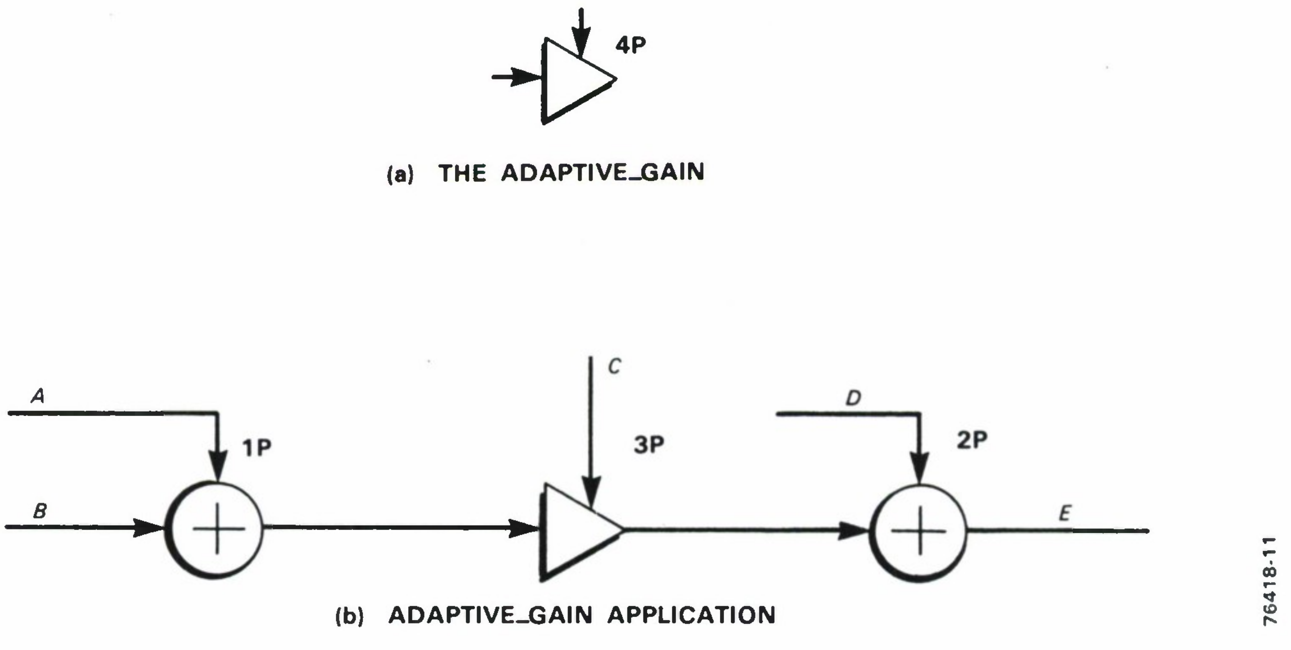


Figure 11. Simple application block diagrams — 2.

The final complication to the BDC design problem is the body-to-code conversion task. A library containing short TMS32010 programs corresponding to primitive bodies, such as the ADDER and GAIN, must be created. Hierarchy should be supported so that a user can define a new body in terms of already existing bodies, much the way a software designer uses subroutines.

The BDC solves these three problems: (1) intercell communication routing, (2) ordering, and (3) assembly code generation. The rest of this section describes the operation of the BDC and, in the process, provides a solution for each of the three problems listed. Section 5.1 outlines the creation of a block diagram, including several common examples. In Section 5.2 we describe the tools designed and implemented for analyzing the block diagram, creating the various data bases, routing intercell communication, and synthesizing source code. A brief description of the automatic partitioning tool is presented in Section 5.3. Section 5.4 concludes the discussion and evaluates the efficiency of the system.

The tools described in the rest of the section are aimed at signal processing problems which are easily described by a signal flow graph. These tools are not directly aimed at simplifying block data processing problems, such as the overlap-add method for FFT calculations, although it is possible to ease this task as well using the hierarchical methods discussed below. Furthermore, the only data types supported are 16-bit fixed point and 1-bit flags. These are the only two types of data for which the TMS32010 provides reasonable support.

5.1 The Block Diagram

This section describes the block diagram, which is the interface between the programmer of the array processor and the software development tools. The block diagram is a complete description of the desired application and is also the input to the system. This is analogous to the computer program which, containing a complete description of an application, is the interface between the computer programmer and the compiler.

A block diagram is composed of a group of blocks (called bodies) connected by lines (called wires). The bodies represent functions, and the wires represent data paths. Block diagrams containing bodies such as ADDERs and GAINs are drawn much the same way digital circuit diagrams are drawn containing AND gates and inverters. Figure 11 shows how bodies such as ADDERs, GAINs, and DELAYs can be interconnected, using wires, to form a second-order filter. The following subsections describe the various bodies available to the user, the method of interconnecting the bodies, and some of the issues involving their use.

5.1.1 Primitive Bodies

The primitive bodies are the lowest-level bodies available. They are atomic, in the sense that they cannot be split into bodies of simpler nature. An example of a primitive body is the ADDER shown in Figure 12(a). Each primitive body has a set of pins used for I/O. Each pin has a property called a PINNAME, which has an alphanumeric string as its value. Pins are often referred to only by their PINNAME property, as a given primitive cannot have two pins with the same value for PINNAME. In addition, each pin has an IOTYPE property, which can have the value INPUT or OUTPUT. The ADDER has three pins: ADDEND0 has IOTYPE=INPUT, ADDEND1 has IOTYPE=INPUT, and SUM has IOTYPE=OUTPUT. Figure 12(b) shows another "version" of ADDER. Each version of a primitive has the same pins, but their orientation may be different. The version of a given primitive used on a drawing depends on what makes the drawing easiest to read. The functionality of the drawing remains the same, regardless of the version chosen.

The GAIN is another primitive body, shown in Figure 12(c). It has two I/O pins. The first has PINNAME=IN and IOTYPE=INPUT. The second has PINNAME=OUT and IOTYPE=OUTPUT. In addition, the GAIN property has the property ACG_VALUE. This property must be given a value specifying the value of the gain.

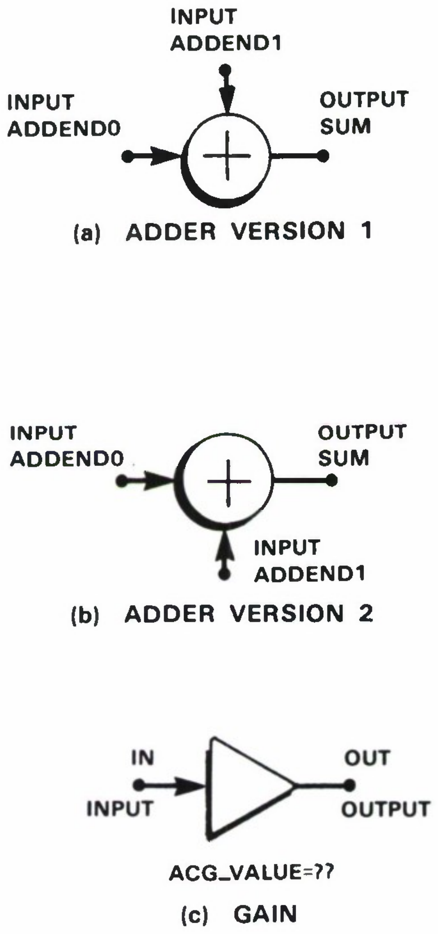


Figure 12. ADDER1, ADDER2, GAIN.

Figure 13(a) shows the DELAY primitive, which has one input and one output. Its output pin has the special property INITIAL_VALUE, indicating that the DELAY primitive is able to output its first value before it receives its first input. The importance of the INITIAL_VALUE property is more fully explored in succeeding sections.

Bodies can have an arbitrary number of outputs. For example, a special body called GAIN_SPEC, shown in Figure 13(b), has two outputs: OUT1 is the input doubled, and OUT2 is the input tripled.

When primitives appear on drawings, pin properties such as PINNAME, IOTYPE, and INITIAL_VALUE are not displayed on the drawing. Body properties, such as ACG_VALUE, are displayed. In addition, each body on a drawing has a unique identifier, called its path name. This path name distinguishes each of the four ADDERs seen in Figure 9. A part of the path name,

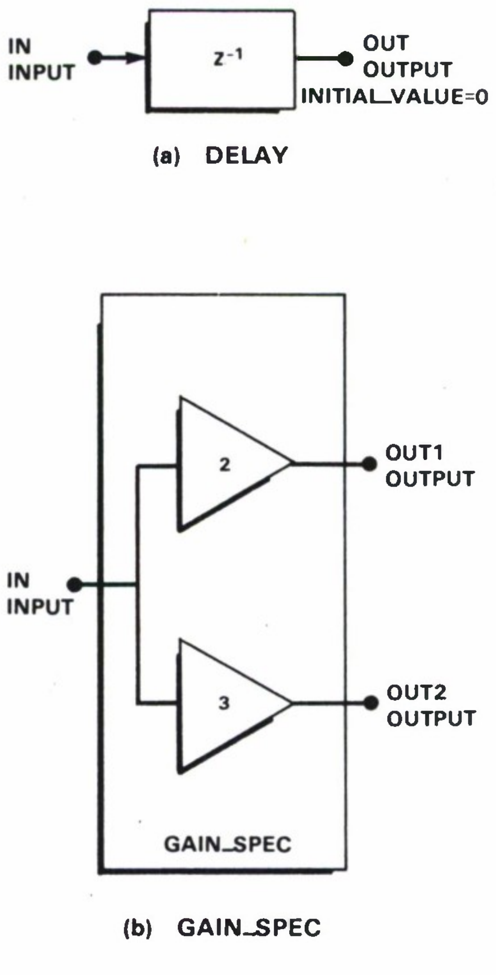


Figure 13. DELAY and GAIN_SPEC.

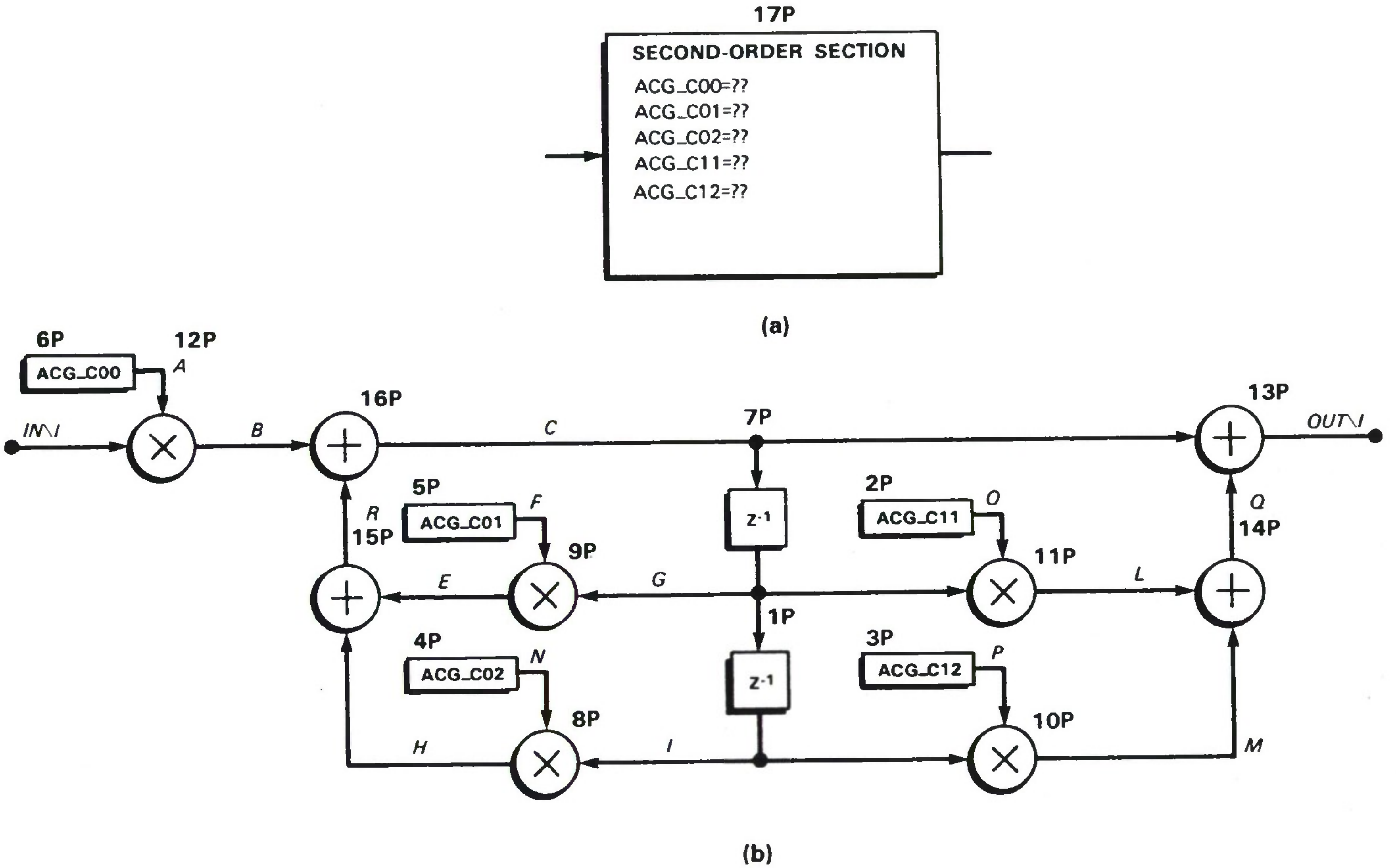


Figure 14. (a) SOS body, and (b) definition.

76418-14

called the path number, is displayed near the body. In the example drawing (Figure 9), there is an ADDER which has path name EX16 ADDER16P and path number 16P. Only the 16P appears on the drawing (the EX16 comes from an abbreviation to the title of the drawing). Path names and path numbers are generated by the system and need not be specified by the user.

5.1.2 Hierarchy — Nonprimitive Bodies

Much as the programmer defines subroutines to avoid unnecessary code repetition and to help modularize, the block diagram user can create nonprimitive bodies for similar reasons. A nonprimitive body is defined by a group of bodies, interconnected by wires. Figure 14 shows the SOS body, a second-order section, along with its definition. The compact SOS body can be used in place of the complicated SOS definition on any drawing. A new body, the FILTERBANK, could be defined using several SOSs.

5.1.3 Interconnection of Bodies — Signals

Bodies are interconnected by signals, which are wires whose nodes are body pins. Signals have the SIGNAME property, which is given an alphanumeric value. In the second-order section of Figure 9, the signal connecting the OUT pin of the GAIN body at 16P to the ADDEND0 pin of the ADDER at 15P has SIGNAME=D. Signals which are not explicitly given SIGNAMEs will be assigned a SIGNAME by the system. All signals having the same SIGNAME are the same signal. Thus, the two drawings in Figure 15(a) represent the same application. Each signal must have exactly one node with IOTYPE=OUTPUT.

5.1.4 The Synchronous vs Asynchronous Issue

We now address the classes of primitives which (1) do not need all their inputs to generate an output, or (2) might not generate any of their outputs even on reception of all their inputs. By affixing the READY property, with value either SYNCHRONOUS or ASYNCHRONOUS, to output pins of primitives, the user specifies whether a primitive falls into either of these two classes.

The READY property with value SYNCHRONOUS is given to an output pin of a body if, in the steady state, that output is generated *if and only if* a new value for each input to the body has arrived. The READY property with value ASYNCHRONOUS is given to an output pin of a body when (1) that output can be generated without new values for all the inputs to the body available, or (2) that output might not be generated even if new values for all the inputs to the body are available. Output pins which are not explicitly given the READY property are SYNCHRONOUS by default.

All the examples shown so far have had SYNCHRONOUS outputs. The SUM output on the ADDER body requires exactly one ADDEND0 and one ADDEND1 to generate one SUM.

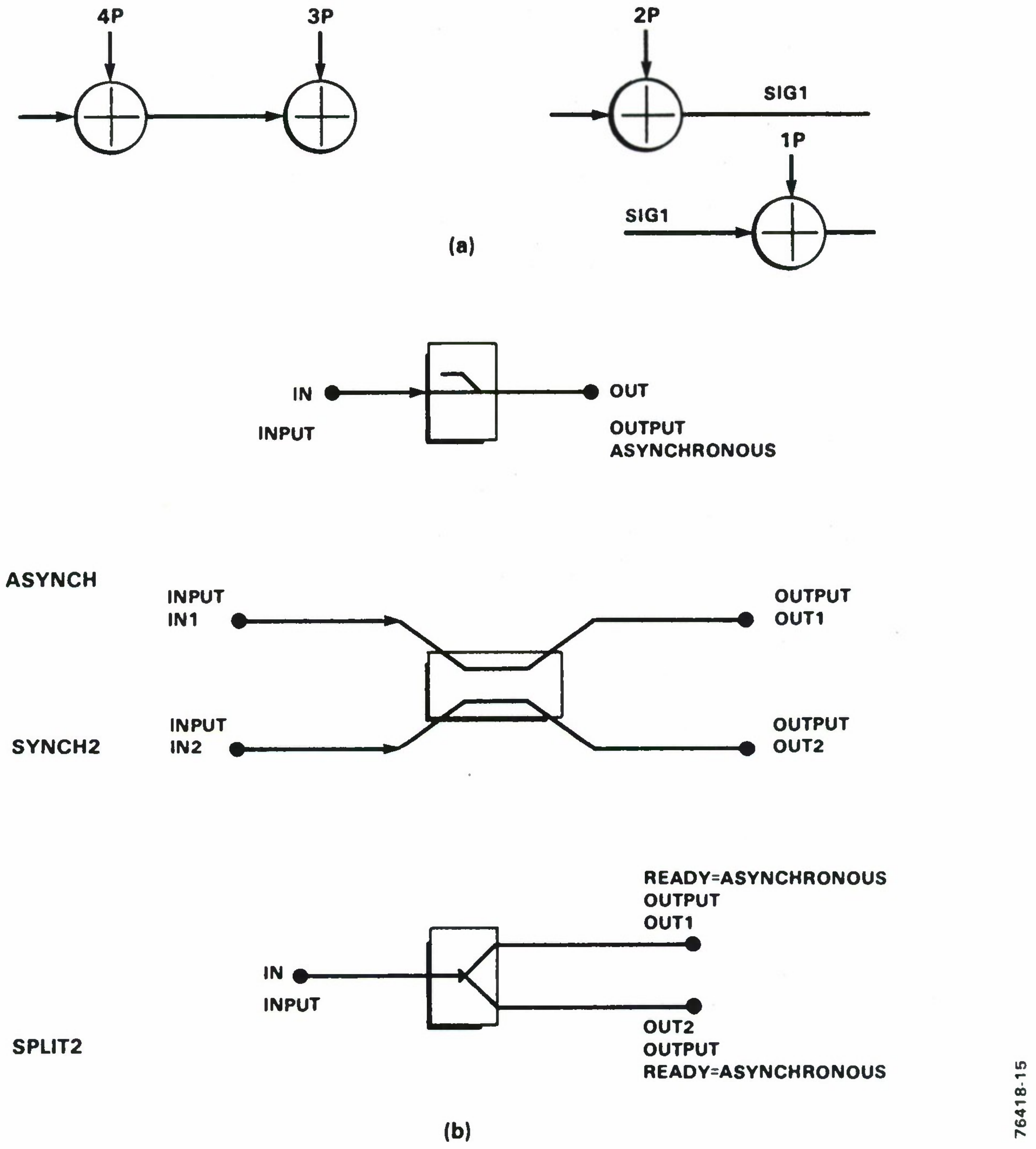


Figure 15. (a) Two drawings, same meaning; (b) asynchronous primitives.

SUM cannot be generated without both inputs, and it must be generated if both inputs are available. Thus, the SUM output is SYNCHRONOUS. Similarly, since the INITIAL_VALUE property affects only the first output of the DELAY body, and since the OUT output of the DELAY body is generated exactly once for each IN input thereafter, the OUT output of the DELAY body is SYNCHRONOUS.

An example of a body having an asynchronous output is the ADAPTIVE_GAIN, shown in Figure 16(a), which has two inputs (IN and GAIN) and one output (OUT). OUT is defined as IN multiplied by A_GAIN. The ADAPTIVE_GAIN does not need a new value of A_GAIN for each new value of IN. Rather, A_GAIN values arrive sporadically. The ADAPTIVE_GAIN uses the most recent value of A_GAIN in calculating OUT. Thus, OUT has READY=ASYNCHRONOUS. The down-sampler body, DOWN, is another example of a body having an asynchronous output. It is shown in Figure 16(b). DOWN has one input pin (FAST) and one output pin (SLOW). Having the body property ACG_RATIO, the decimation factor, it must receive ACG_RATIO values of FAST before generating one SLOW output. Therefore, SLOW has READY=ASYNCHRONOUS.

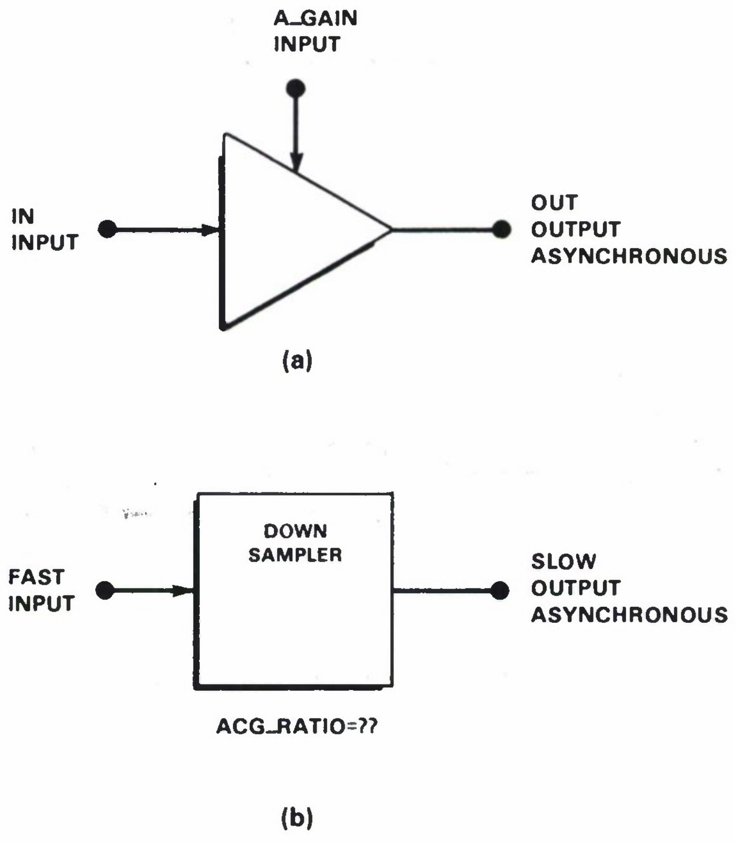


Figure 16. (a) ADAPTIVE_GAIN, and (b) DOWN.

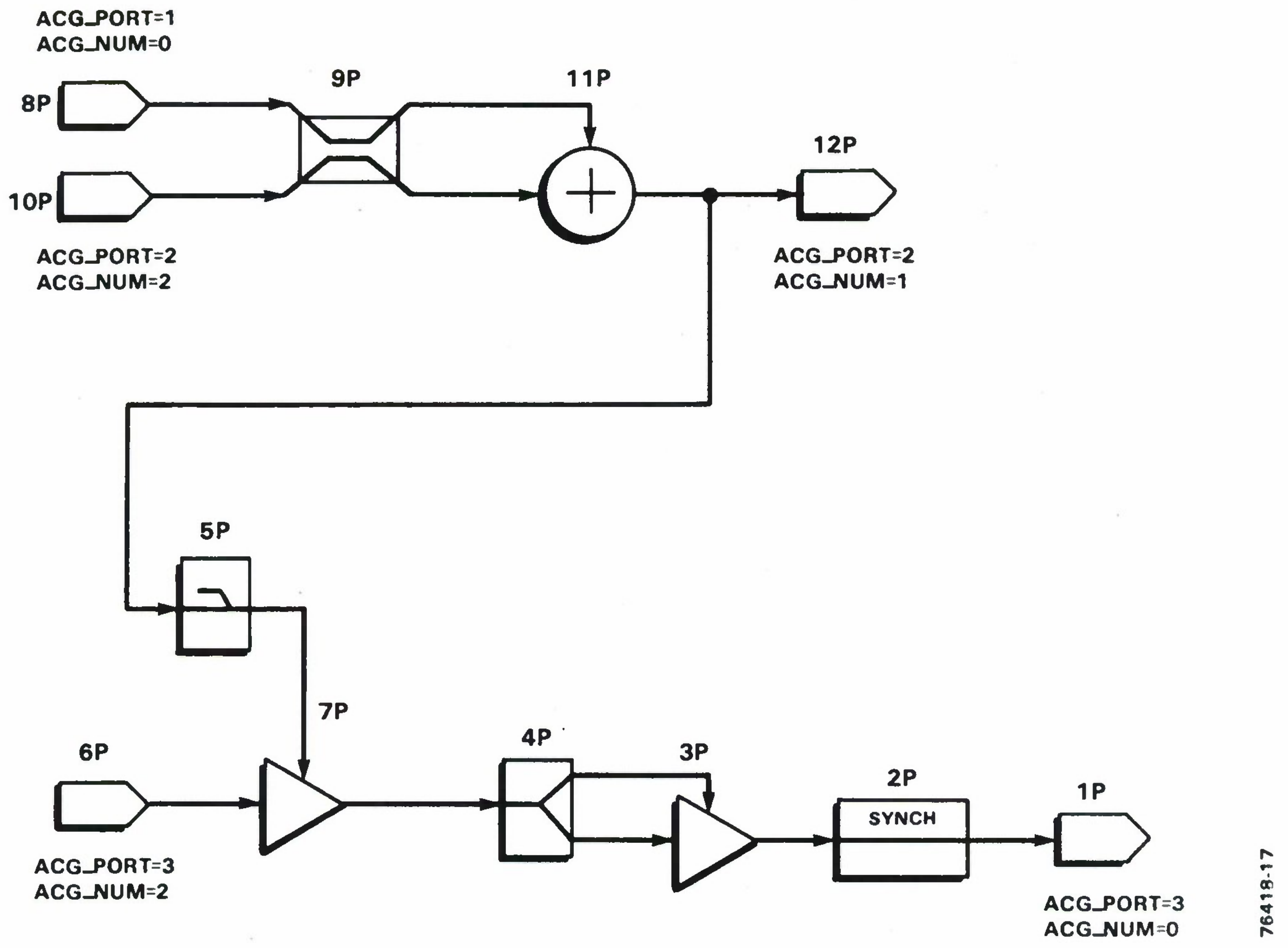


Figure 17. Asynchronous example.

A synchronous *body* is defined as one whose outputs all have READY=SYNCHRONOUS. A synchronous *signal* is one whose output node has READY=SYNCHRONOUS. A synchronous *group* is a collection of interconnected synchronous bodies and signals. Similarly, an asynchronous body is one which has at least one output with READY=ASYNCHRONOUS. An asynchronous signal is one whose output node has READY=ASYNCHRONOUS.

There are three restrictions on the design and interconnection of synchronous and asynchronous bodies and signals:

(1) Any signal having a node on an asynchronous body is asynchronous by definition.

(2) All input signals to a synchronous body or group must be synchronized with each other.

(3) An asynchronous signal must have exactly one node which has IOTYPE=INPUT and one node which has IOTYPE=OUTPUT.

There are three types of bodies provided to allow the programmer to meet the three restrictions. The first is the ASYNCH primitive which converts a synchronous signal to an asynchronous signal. The second type of body is the SYNCH primitive, which synchronizes signals with each other. The third type of body is the SPLIT primitive, which splits an asynchronous signal into several asynchronous signals. These three types of bodies are shown in Figure 15(b). Both the SYNCH and SPLIT primitives are better thought of as families of primitives. For example, the SYNCH primitives have names such as SYNCH2, SYNCH3, etc., depending on how many signals are being synchronized. Similarly, the SPLIT family contains primitives such as SPLIT2 and SPLIT3, where the last character of the name indicates the number of output signals of the splitter.

An example of the use of synchronous and asynchronous signals and bodies is shown in Figure 17. The output of an INPUT primitive is asynchronous, by definition. In order to interface the two inputs at 8P and 10P to the synchronous ADDER at 11P, a SYNCH2 is needed (Restriction 2). This SYNCH2 causes polling of the two INPUTS until they both have valid data on their output lines. When this occurs, the ADDER is allowed to execute. The ASYNCH at 5P is used to convert the synchronous output of the ADDER at 11P into an asynchronous signal (Restriction 1), for input into the ADAPTIVE_GAIN at 7P. The SPLIT2 effectively splits the output of the ADAPTIVE_GAIN at 7P for input into the ADAPTIVE_GAIN at 3P (Restriction 3).

5.1.5 Partitioning Assignment

The final issue involving the creation of a block diagram is the assignment of each block in the drawing to a given processor. Currently, the user makes this assignment manually by attaching the PROC_NUM property to each body in the drawing. For each body, the PROC_NUM property is given a value representing the physical processor on which the body is to run. Figure 18 shows an example of a block diagram whose bodies have been assigned to physical processors through the use of the PROC_NUM property. In future versions of the BDC, the automatic partitioning tool (alluded to at the beginning of this section, and described at the end of this section) will be used to make this assignment of bodies to physical processors.

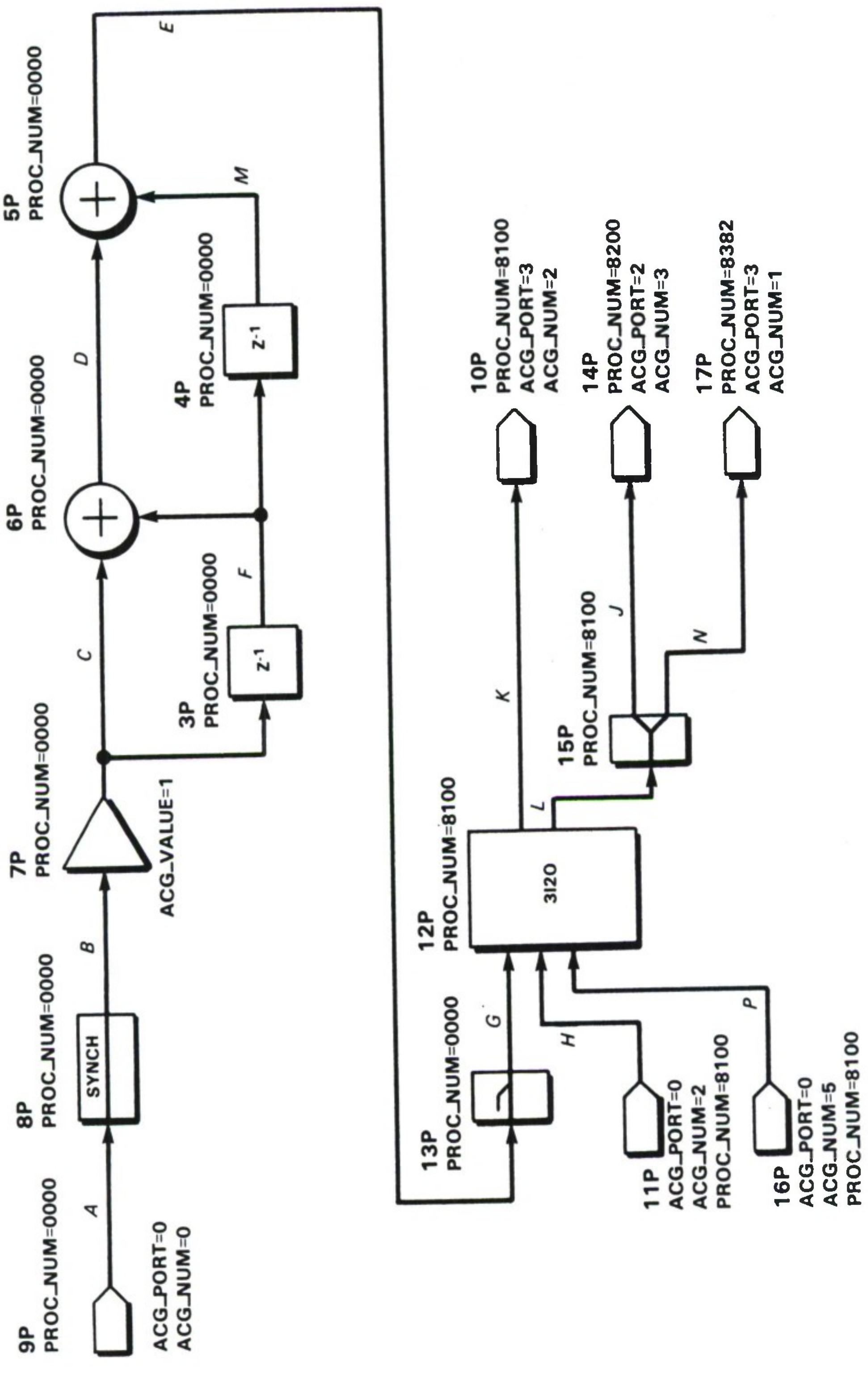


Figure 18. *Diagram with PROC_NUMs.*

76418-18

5.2 Block Diagram Compiler

The block diagram compiler (BDC) is a tool developed to automate the code generation and intercell communication routing. The BDC takes a block diagram as input, and outputs source code for each cell in the array. The process closely parallels the automatic circuit design packages currently available, in which the user draws the circuit on a CAE workstation and the system converts the drawing into appropriate net lists and parts lists needed for fabrication. In fact, some of the BDC is part of an automatic circuit design package adapted to this particular application.

As described above, the user draws a block diagram with the aid of a CAE workstation. The BDC groups the bodies by their physical processor assignments and converts the block diagram into programs for each sequential processor. Bodies are eventually converted to code, and wires become memory locations. Bodies can be simple (e.g., adders or subtractors) or complicated (e.g., second-order filters). In addition, the BDC handles all intercell communication routing.

Figure 19 shows the block diagram for an example of a digital signal processing task, a second-order infinite impulse response filter (IIR) in series with a three-tap finite impulse response filter (FIR). The filters are to be run on different processors, using a pipeline, to increase throughput. The picture created on the CAE system is input to the BDC. The BDC is also informed of the physical array geometry characteristics, e.g., number of cells, arrangement of the cells within the array, cell interconnections. The output of the BDC is one assembly code program for each cell, including the intercell communication software. Thus, the BDC has spared the user the time-consuming job of converting the block diagram into source code.

The operation of the BDC is split into seven modules:

Graphics Modules

1. The Graphic Data Entry System
2. The Graphic Sub-Compiler

Data Base Generation Modules

3. The Signal Table Generator
4. The Primitive Table Generator

Partitioning Module

5. The Splitting and Routing Program

Code Generation Modules

6. The Ordering Program
7. The Assembly Code Generator

which are described in the succeeding sections. A flowchart description of the BDC is provided in Figure 20.

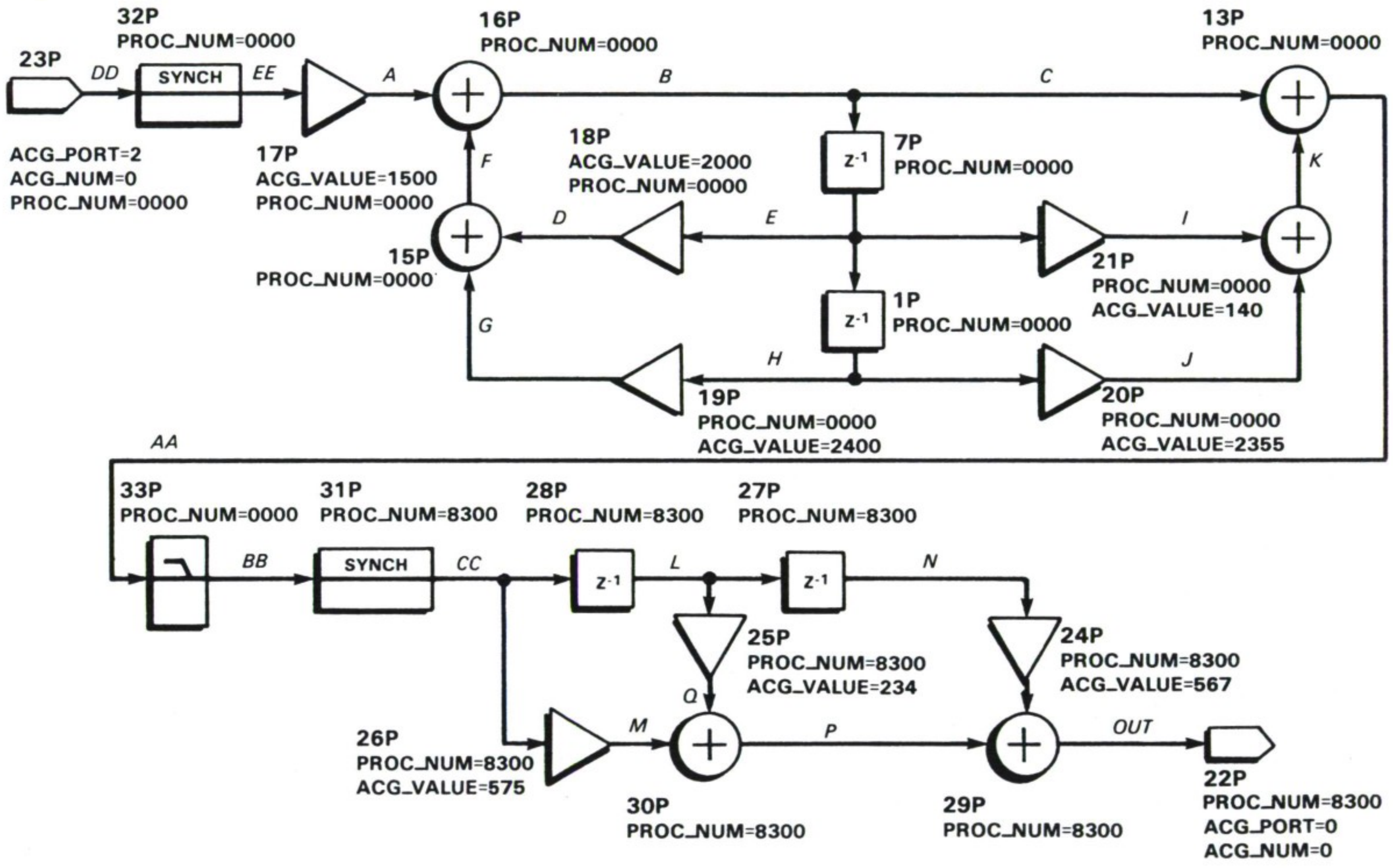
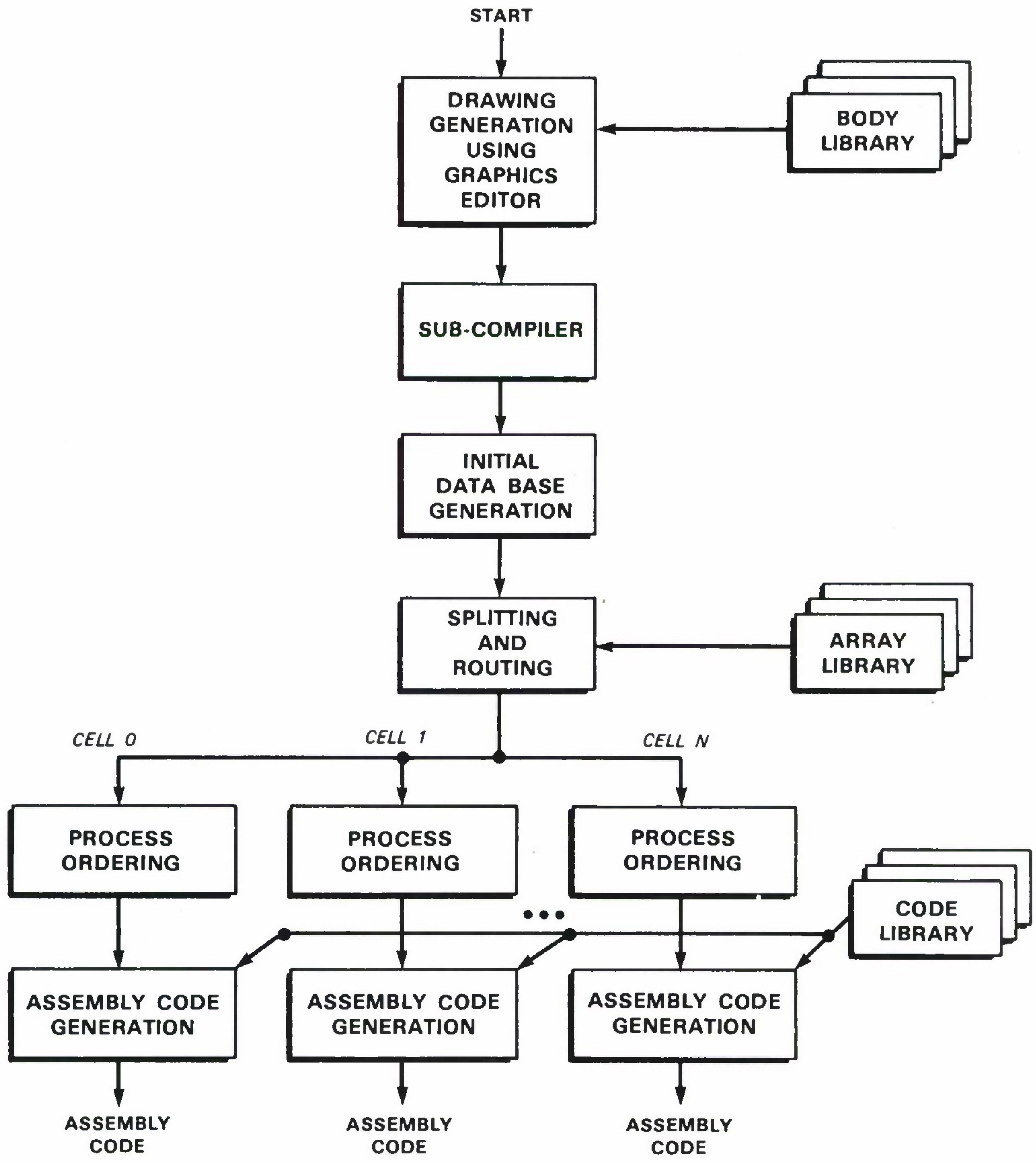


Figure 19. IIR and FIR in series.

76418-19



76418-20

Figure 20. BDC flowchart.

5.2.1 Graphics Modules

The graphics modules described below are implemented on a VALID SCALDsystem CAE workstation. The system used consists of two workstations, a CPU, a 400-Mbyte disk drive, and two printer/plotters.

5.2.1.1. Graphic Data Entry System

The graphics editor is used to "draw" the block diagram. The workstation is equipped with a puck and a magnetic tablet which are used to enter the drawing. Bodies are added to the drawing, placed in the proper positions, and connected with wires using VALID's graphics editor (GED). Text is added to the drawing using the workstation's keyboard.

5.2.1.2 Graphic Sub-Compiler

When the user has finished adding and connecting bodies, he issues a command which writes his drawing to the mass storage device (a disk in this case). The form in which the drawing is stored, a so-called "vector representation," is one which makes it easy for a workstation to re-display the block diagram on the screen at some later time. This representation of the block diagram is not suitable for the task we want to perform, namely block diagram compilation. However, VALID provides a sub-compiler (they call it the "compiler," but in this report it will be referred to as the "sub-compiler") which converts the less-than-useful vector description into an easily readable ASCII file containing all the information about the bodies, their connectivities, and special properties necessary to perform the compilation. The sub-compiler "flattens" the hierarchy of the drawing by expanding all nonprimitive bodies into their all-primitive equivalents. In order to eliminate some of the extraneous information, the compiler expansion file is run through a "filter" program, leaving only relevant data. The beginning of the filtered expansion file for the IIR/FIR example is shown in Figure 21.

5.2.2 Data Base Generation Modules

Once the sub-compilation is finished, the data base generators begin the task of converting the long ASCII compiler expansion file into two compact data bases. The reason for making this conversion is that the BDC can run much faster if the drawing can be represented by a number of special-purpose linked lists, as opposed to one single file. The next two paragraphs briefly describe the data base generators.

5.2.2.1 Signal Table Generator

A data base of the signals, called the signal table, is generated for the entire application. The signal table is implemented as a hash table, where the hashing function extracts the first character in the signal name and uses it as an index into the signal table array. For each signal, a doubly

```
FILE_TYPE=PARSED_LOGIC_EXPANSION;
 ROOT_DRAWING='IIRFIR';
PRIMITIVE 'GAIN':
 BODY
  PATH_NAME='(IIRFIR GAIN24P)';
  PROC_NUM='B3ØØ';
  ACG_VALUE='567';
 END_BODY:
 BINDINGS
  'OUT'='UN$I$ADDER$29P$ADDENDI'<Ø>\IOTYPE='OUTPUT';
  'IN'='N'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'GAIN':
 BODY
  PATH_NAME='(IIRFIR GAIN25P)';
  PROC_NUM='B3ØØ';
  ACG_VALUE='234';
 END_BODY;
 BINDINGS
  'OUT'='O'\IOTYPE='OUTPUT';
  'IN'='L'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'GAIN';
 BODY
  PATH_NAME='(IIRFIR GAIN26P)';
  PROC_NUM='B3ØØ';
  ACG_VALUE='575';
 END_BODY;
 BINDINGS
  'OUT'='M'\IOTYPE='OUTPUT';
  'IN'='CC'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'DELAY';
 BODY
  PATH_NAME='(IIRFIR DELAY27P)';
  PROC_NUM='B3ØØ';
 END_BODY;
 BINDINGS
  'OUT'='N'\INITIAL_VALUE='Ø'\IOTYPE='OUTPUT';
  'IN'='L'\IOTYPE='INPUT';
 END_BINDINGS:
END_PRIMITIVE:
PRIMITIVE 'DELAY';
 BODY
  PATH_NAME='(IIRFIR DELAY28P)';
  PROC_NUM='B3ØØ';
 END_BODY:
 BINDINGS
  'OUT'='L'\INITIAL_VALUE='Ø'\IOTYPE='OUTPUT';
  'IN'='CC'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'ADDER';
 BODY
  PATH_NAME='(IIRFIR ADDER29P)';
  PROC_NUM='B3ØØ';
 END_BODY;
 BINDINGS
  'SUM'='OUT'\IOTYPE='OUTPUT';
  'ADDENDI'='UN$I$ADDER$29P$ADDENDI'<Ø>\IOTYPE='INPUT';
  'ADDENDØ'='P'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'ADDER';
 BODY
  PATH_NAME='(IIRFIR ADDER3ØP)';
  PROC_NUM='B3ØØ';
 END_BODY;
 BINDINGS
  'SUM'='P'\IOTYPE='OUTPUT';
  'ADDENDI'='O'\IOTYPE='INPUT';
  'ADDENDØ'='M'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'ADDER';
 BODY
  PATH_NAME='(IIRFIR ADDER16P)';
  PROC_NUM='ØØØØ';
 END_BODY;
 BINDINGS
  'SUM'='B'\IOTYPE='OUTPUT';
  'ADDENDI'='F'\IOTYPE='INPUT';
  'ADDENDØ'='A'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'ADDER';
 BODY
  PATH_NAME='(IIRFIR ADDER15P)';
  PROC_NUM='ØØØØ';
 END_BODY;
 BINDINGS
  'SUM'='F'\IOTYPE='OUTPUT';
  'ADDENDI'='D'\IOTYPE='INPUT';
  'ADDENDØ'='G'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'ADDER';
 BODY
  PATH_NAME='(IIRFIR ADDER14P)';
  PROC_NUM='ØØØØ';
 END_BODY;
 BINDINGS
  'SUM'='K'\IOTYPE='OUTPUT';
  'ADDENDI'='I'\IOTYPE='INPUT';
  'ADDENDØ'='J'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'ADDER';
 BODY
  PATH_NAME='(IIRFIR ADDER13P)';
  PROC_NUM='ØØØØ';
 END_BODY;
 BINDINGS
  'SUM'='AA'\IOTYPE='OUTPUT';
  'ADDENDI'='K'\IOTYPE='INPUT';
  'ADDENDØ'='B'\IOTYPE='INPUT';
 END_BINDINGS;
END_PRIMITIVE;
PRIMITIVE 'DELAY';
 BODY
  PATH_NAME='(IIRFIR DELAY7P)';
  PROC_NUM='ØØØØ';
 END_BODY;
 BINDINGS
  'OUT'='E'\IOTYPE='OUTPUT'\INITIAL_VALUE='Ø';
```

76418-21

Figure 21. IIR/FIR parsed expansion file.

```
FILE_TYPE=SIGNAL_TABLE:

--------------------

Signal Name:  A

Does not have initial value.

Synchronous.

Inputs on Net:
(IIRFIR ADDER16P) : ADDENDØ

Outputs on Net:
(IIRFIR GAIN17P) : OUT

Previous:
NULL

Next:
AA

--------------------

Signal Name:  AA

Does not have initial value.

Synchronous.

Inputs on Net:
(IIRFIR ASYNCH33P) : IN

Outputs on Net:
(IIRFIR ADDER13P) : SUM

Previous:
A

Next:
NULL

--------------------

Signal Name:  B

Does not have initial value.

Synchronous.

Inputs on Net:
(IIRFIR DELAY7P) : IN
(IIRFIR ADDER13P) : ADDENDØ

Outputs on Net:
(IIRFIR ADDER16P) : SUM

Previous:
NULL

Next:
BB

--------------------
```

```
FILE_TYPE=PRIMITIVE_TABLE:

Pathname: (IIRFIR ADDER15P) is a (an) ADDER
Run on processor: ØØØØ
With inputs:
     Pin ADDENDØ is connected to G
     Pin ADDEND1 is connected to D
With outputs:
     Pin SUM is connected to F
Has Properties:
     NONE

Pathname: (IIRFIR ADDER16P) is a (an) ADDER
Run on processor: ØØØØ
With inputs:
     Pin ADDENDØ is connected to A
     Pin ADDEND1 is connected to F
With outputs:
     Pin SUM is connected to B
Has Properties:
     NONE

Pathname: (IIRFIR ADDER3ØP) is a (an) ADDER
Run on processor: 83ØØ
With inputs:
     Pin ADDENDØ is connected to M
     Pin ADDEND1 is connected to Q
With outputs:
     Pin SUM is connected to P
Has Properties:
     NONE

Pathname: (IIRFIR ADDER29P) is a (an) ADDER
Run on processor: 83ØØ
With inputs:
     Pin ADDENDØ is connected to P
     Pin ADDEND1 is connected to UNØ
With outputs:
     Pin SUM is connected to OUT
Has Properties:
     NONE

Pathname: (IIRFIR DELAY28P) is a (an) DELAY
Run on processor: 83ØØ
With inputs:
     Pin IN is connected to CC
With outputs:
     Pin OUT is connected to L
Has Properties:
     NONE

Pathname: (IIRFIR DELAY27P) is a (an) DELAY
Run on processor: 83ØØ
With inputs:
     Pin IN is connected to L
With outputs:
     Pin OUT is connected to N
Has Properties:
     NONE

Pathname: (IIRFIR GAIN26P) is a (an) GAIN
Run on processor: 83ØØ
```

76418-22

Figure 22. Part of the IIR/FIR signal and primitive tables.

linked list entry is created containing (1) the signal name, (2) the path name of each primitive which has a pin on the signal net, (3) an indication of whether the signal has an initial value, and (4) an indication of whether the signal is synchronous or asynchronous. The entries for signals A and AA in the IIR/FIR application have been printed in Figure 22 (left column).

5.2.2.2 Primitive Table Generator

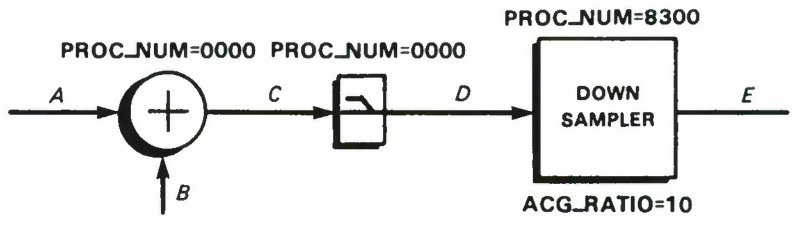
A second data base, called the primitive table, is created in tandem with the signal table. The primitive table contains an entry for each primitive in the array. An entry consists of (1) a path name, (2) a primitive type, (3) a physical processor number, (4) each pin name on the primitive, along with the name of the signal to which it is connected, and (5) the primitive properties. The primitive table is implemented as a singly linked list. The entries for some of the ADDERS in the IIR/FIR application have been printed in Figure 22 (right column).

5.2.3 Partitioning Module — The Splitting and Routing Program

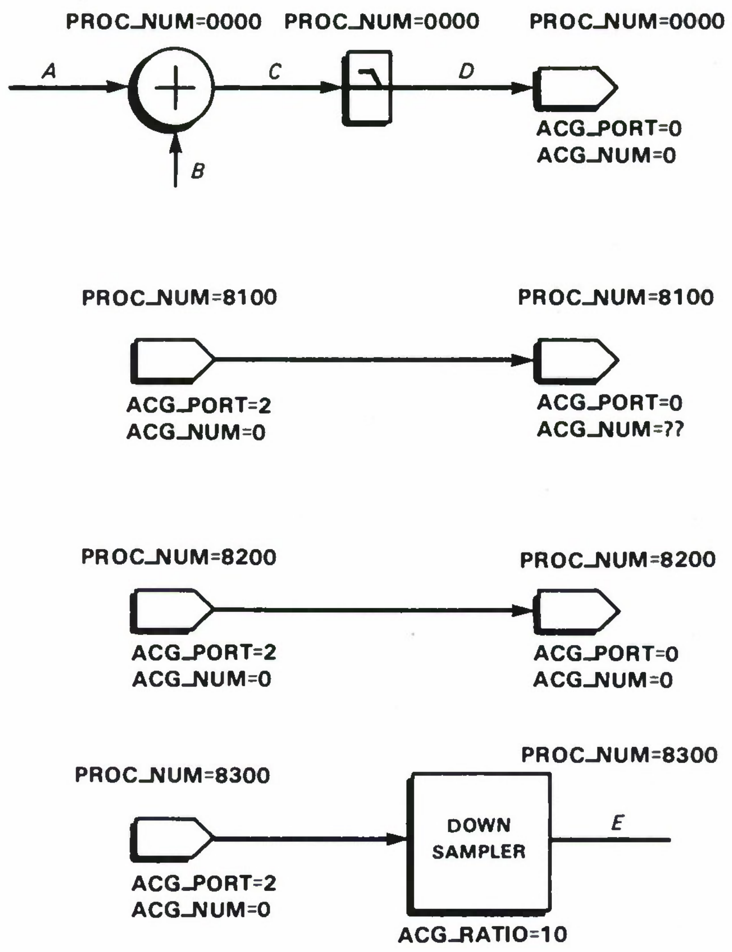
The drawing which is input to the system by the user specifies an application which, in general, is to be executed using many cells. The partitioning module divides the multi-cell problem, as specified by the signal table and primitive table, into many single-cell problems which are simpler for the BDC to process. Next, intercell data transfers are routed through the array. The rest of this subsection describes the algorithms implemented to accomplish both these tasks.

5.2.3.1 Splitting a Multi-Cell Application

The algorithm for splitting the multi-cell application into many single-cell applications is straightforward. For each signal in the signal table, a routine is run which determines whether all primitives using the signal are to be run on the same cell. If so, the routine exits. If not, the intercell signal is split into many single-cell signals, connected by INPUTs and OUTPUTs. Consider the application shown in Figure 23(a). The ADDER output is used as an input to a DOWN, with an ASYNCH between them. The ADDER and ASYNCH run on cell 0000, while the DOWN runs on cell 8300. The signal D, therefore, is an intercell signal. The splitting and routing program (1) determines that D is an intercell signal, (2) finds all the primitives attached to it, (3) routes paths from the primitive using the signal as an output to all the cells using it as an input, and (4) adds the appropriate INPUT and OUTPUT primitives. Figure 23(b) shows the result of this conversion. Note that since cell 0000 is not adjacent to cell 8300, there are a number of intermediate cells which are used in forwarding the data. The splitting and routing program is given information describing the geometry of the array being used so that it knows how to route data properly. Once all the intercell signals have been converted in this manner, the program continues by splitting the large primitive table into many primitive tables, one for each cell.



(a) DRAWING INPUT BY USER



(b) DRAWING EXPANDED BY SPLITTING AND ROUTING PROGRAM

76418-23

Figure 23. Splitting and routing.

5.2.3.2 Routing Intercell Data Transfers

The last task performed by the splitting and routing program is some further processing of the intercell data transfers. Specifically, for each cell all INPUTs and OUTPUTs are grouped on the basis of which port they will be using. If more than one INPUT or OUTPUT is using the same port, the multiple INPUTs and OUTPUTs are converted into multiple-input and multiple-output bodies. In the code generation phase of the BDC, each multiple-input and multiple-output body is converted into assembly code which implements a polled I/O scheme. Each data word to be transmitted is preceded by a header, indicating its I/O number. For example, if cell 0000 expects two different types of data from cell 8100 (over port 0), cell 8100 must precede each data word sent to cell 0000 by a header indicating which of the two types of data it is about to send. This header overhead is only suffered when there is more than one INPUT or more than one OUTPUT using the same port. The user is free to define his own INPUT_BLOCK or OUTPUT_BLOCK bodies which would be capable of block transfers, thereby reducing the header overhead. Future versions of the BDC may incorporate more efficient routing schemes including automatic block data transfer handling and interrupt driven I/O.

5.2.4 Code Generation Modules

The code generation modules use the data bases described above to generate assembly code for each cell in the array. The ordering program is run once for each cell in the array taking the global signal table and the cell's primitive table as input. The sections below describe the use of the code generation modules for one single cell's code generation, since their operation is identical for each cell in the array.

5.2.4.1 Ordering Program

The ordering program uses the signal table and the primitive table to generate a control flowchart for each cell. This conversion from a block diagram containing parallel flow patterns into a flowchart containing only sequential flow patterns is needed because each cell is really a sequential processor, i.e., ADDERs which look like they can run in parallel on the block diagram can only be run one at a time on a TMS32010.

The description of the ordering program begins with a step-by-step explanation of the constraints on the ordering processes. The constraints are presented in order of increasing complexity and robustness.

5.2.4.1.1 C1 — The Simplest Constraint

An ordering constraint is used to generate an ordered list of execution for each cell. After forming this ordered list, an assembly language program is generated consisting of the code necessary to execute each primitive in the list in proper order. After execution of the last primitive in the list, the loop is restarted with the first primitive. This ordered primitive list contains each primitive to be processed by the cell exactly once. Clearly, we cannot arbitrarily enter the primitives into the ordered primitive list. The simplest constraint, C1, on the ordering process would be:

> *A primitive is allowed to run only after each of its input pins has been satisfied. By "satisfied," we mean that given an input pin, the primitive which contains the pin of type OUTPUT on the signal of the input pin has already been executed. Primitives having no input pins, such as INPUT primitives, can be executed immediately, with no constraints.*

For example, consider the second-order section of Figure 9. The ADDER at 15P would not be allowed to execute until both the GAINs at 18P and 19P had been executed. Using this constraint dictates that the GAIN at 17P should be executed first. Now, we are stuck. The ADDER at 16P cannot be executed because while A has been calculated, F has not been calculated. The problem is that the feedback loop caused by the DELAY primitives has not been addressed. The solution is found in C2.

5.2.4.1.2 C2 — Feedback Handling Constraint

C2 differs from C1 by specially handling bodies with INITIAL_VALUE properties on their outputs:

> *A primitive is allowed to run only after each of its input pins has been satisfied. By "satisfied," we mean that given an input pin, the primitive which contains the pin of type OUTPUT on the signal of the input pin has already been executed* ***or the pin of type OUTPUT on the signal of the input pin has the property INITIAL_VALUE.*** *Primitives having no input pins can be executed immediately, with no constraints.*

In our example, the GAINs 18P, 19P, 21P, and 20P have all their input pins satisfied and can be executed immediately. Unfortunately, whereas C2 does not get fooled by feedback, it can yield erroneous orderings. Assume that the ADDER at 16P has been executed, yielding B. If the DELAY were considered a normal primitive, we would execute 7P and then execute 1P. This would be a mistake, though, because we would be setting signal E equal to signal B and then setting H equal to signal E. Thus, H would equal B. This is not the effect that was desired when those two DELAYs were drawn in series. In fact, we wanted signal H to be set to E *first*, then E set to B. A better constraint is needed.

5.2.4.1.3 C3 — The Delay Handling Constraint

C3 correctly handles the problem of DELAYs:

> *Any primitive* ***except*** *a DELAY is allowed to run only after each of its input pins has been satisfied. By "satisfied," we mean that given an input pin, the primitive which contains the pin of type OUTPUT on the signal of the input pin has already been executed or the pin of type OUTPUT on the signal of the input pin has the property INITIAL_VALUE. Primitives having no input pins can be executed immediately, with no constraints.* ***After all***

the non-DELAY primitives have been executed, the DELAY primitives are executed in reverse order.

By "reverse" order, we mean that a DELAY cannot be executed until all its outputs are reverse satisfied. By "reverse satisfied," we mean that given an output pin, all the primitives which contains pins of type INPUT on the signal of the output pin have been executed.

Constraint C3 insures that DELAYs are the last primitives executed and that, for example, 7P cannot occur before 1P.

Closely examining C3 reveals some unwanted implicit restrictions which we have made on each of the primitives. First, we have assumed that each time a primitive is executed, it will generate a new output. This is a reasonable assumption for an ADDER, but is not reasonable for an INPUT or a DOWN_SAMPLER. Second, we have assumed that a primitive needs all of its inputs to generate each and every output. This is reasonable for a MULT, but not for an ADAPTIVE_GAIN which might have inputs appearing at different rates. Thus, we discover that C3 correctly orders synchronous primitives only.

5.2.4.1.4 C4 — The Asynchronous/Synchronous Constraint

A new constraint, C4, is needed to handle a mixture of synchronous and asynchronous bodies and signals. Before stating C4, the synchronous/asynchronous issues must be studied more closely.

A synchronous signal, i.e., a signal whose lone output pin has the READY property with value SYNCHRONOUS, contains only data. An asynchronous signal, i.e., a signal whose lone output pin has the property READY=ASYNCHRONOUS, must contain not only data, but also a flag indicating whether the data are valid. This flag is a single-bit value called the READY bit. Upon output of an asynchronous signal, the signal's READY bit is asserted. When the asynchronous signal is used, the signal's READY bit is de-asserted. Since there is only one READY bit per asynchronous signal, an asynchronous signal can be an input to exactly one body. A synchronous signal must pass through an asynchronizer before being used by the rest of the (asynchronous) system. The beginning of a synchronous group is designated by (1) a SYNCH primitive or (2) a synchronous primitive with exactly one input. The end of a synchronous block is designated by an ASYNCH primitive or a primitive which has an output having its READY property equal to ASYNCHRONOUS.

For each primitive p which has no inputs, the program orders all the primitives which depend on p. Where synchronous groups are encountered, the primitives are ordered according to the constraint C3. Where asynchronous primitives are encountered, a modified constraint (C4) is used:

An asynchronous primitive is allowed to run whenever ***any one of its input pins has been satisfied.*** *By "satisfied," we mean that given an input pin, (1) the primitive which contains*

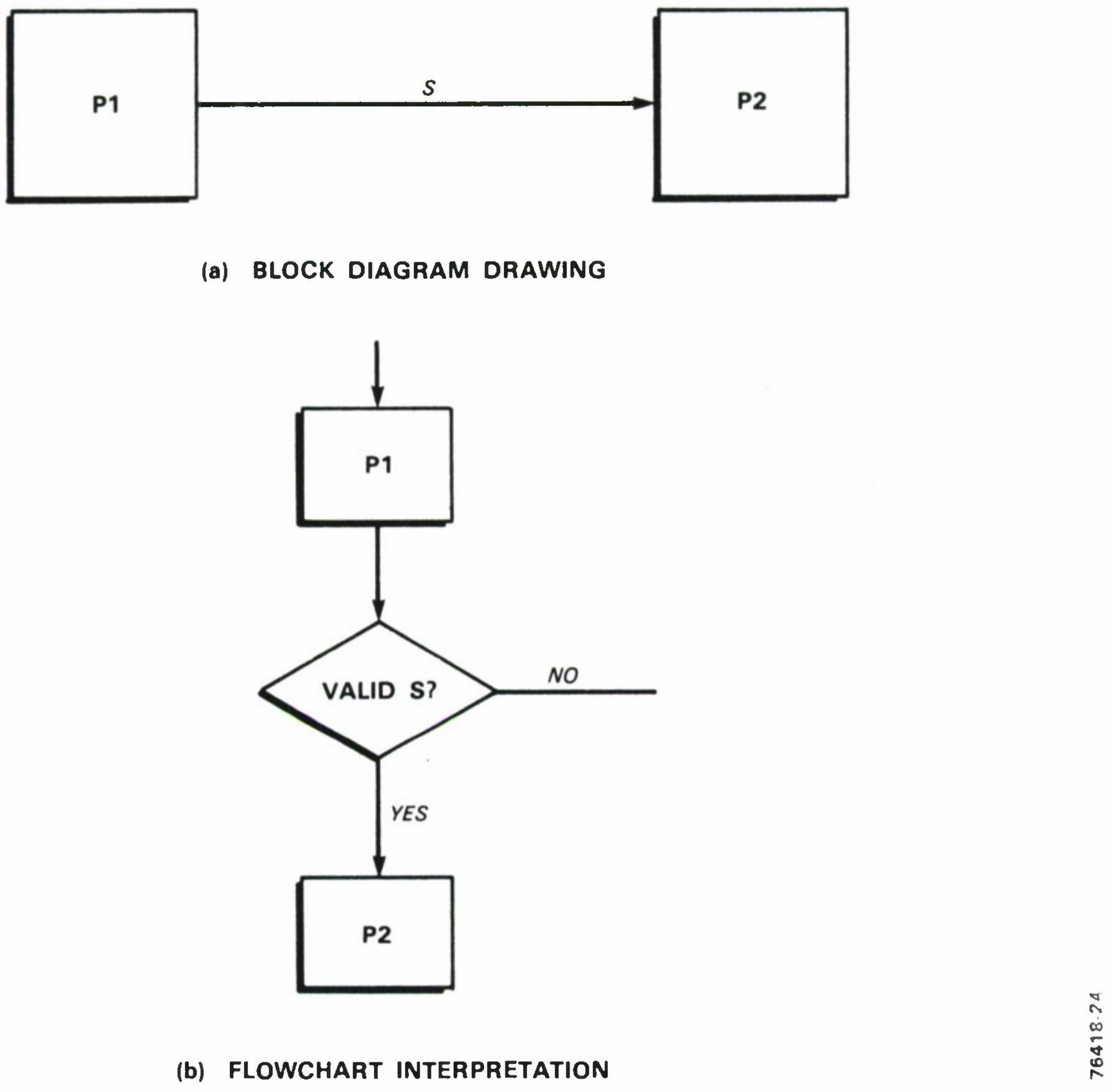


Figure 24. Asynchronous block diagram to flowchart conversion.

the pin of type OUTPUT on the signal of the input pin has already been executed ***and*** *the READY bit is set on that signal, or (2) the signal of the input pin has the property INITIAL_VALUE. Primitives having no input pins can be executed immediately, with no constraints.*

Using C4 and considering Figure 24(a) and (b), any primitive p2 which has an input signal s, which in turn was output by p1, will be executed every time p1 generates an output on s. Notice that p2 is not necessarily executed every time p1 is executed, but that execution is conditional on the presence of valid data on s. Thus, the block diagram shown in Figure 24(a) has the flowchart shown in Figure 24(b).

By using the two constraints, C3 for synchronous groups and C4 for all primitives not in a synchronous group, it is possible to correctly order any block diagram comprised of the primitives described so far. An example block diagram with asynchronous and synchronous primitives is shown in Figure 25. 1I2O is an arbitrary one-input two-output asynchronous primitive. Similarly, 3I2O is an arbitrary three-input two-output asynchronous primitive. The INPUTs are asynchronous, meaning that there are two possible outcomes of executing the INPUT primitive: (1) data were available, meaning that a value is placed on the output signal and the READY bit is set; or (2) no data were available, meaning that no value is placed on the output signal and the READY bit is not set. The asynchronous primitive at 12P is a SPLIT2, i.e., a one-to-two-signal splitter. 11P is a SYNCH2, i.e., a two-signal synchronizer. 13P, 14P, and 15P are ASYNCHS. The rest of the primitives are "normal" synchronous primitives.

Figure 26 shows the output, called a process table, available from the ordering program given the input of Figure 25. Each line contains either a label, beginning with the letter "L," or a statement. Comments, provided by the system, are separated from the statements by semicolons. Statements can be "imperatives" or "interrogatives." The imperatives correspond to primitive executions, while the interrogatives correspond to READY bit checks of SYNCH checks. The process table begins with the execution of the INPUT at 1P. After execution of the INPUT, there might or might not be a valid data value contained in signal D. If D is valid, the INPUT primitive will set the READY bit of signal D. Otherwise, the bit will remain unset. The next line of the process table questions whether that bit is set. If it is not set, execution of primitives depending on D and following 1P will not be attempted at this time. In that case, a jump to label L0 is made. If the READY bit of signal D was set, execution continues with the SYNCH2 primitive. In this respect, the SYNCH primitive is an exception because it is a combination of an imperative statement and an interrogative statement. A SYNCH primitive tests whether all its inputs are ready. If so, it copies its inputs to its outputs and evaluates as ready. Otherwise, it evaluates as not ready. In this example, if both C and D are ready, the SYNCH2 copies C to H and D to I. Execution continues with the ADDER at 2P. If the SYNCH2 either C or D had not been ready, execution would have continued at L1.

Instead of continuing this detailed analysis of the example, some of the important characteristics of the output will be highlighted. Notice the two NEGs, 7P and 6P. These are synchronous primitives, so once M is ready, both 7P and 8P can be run without checking the status of F.

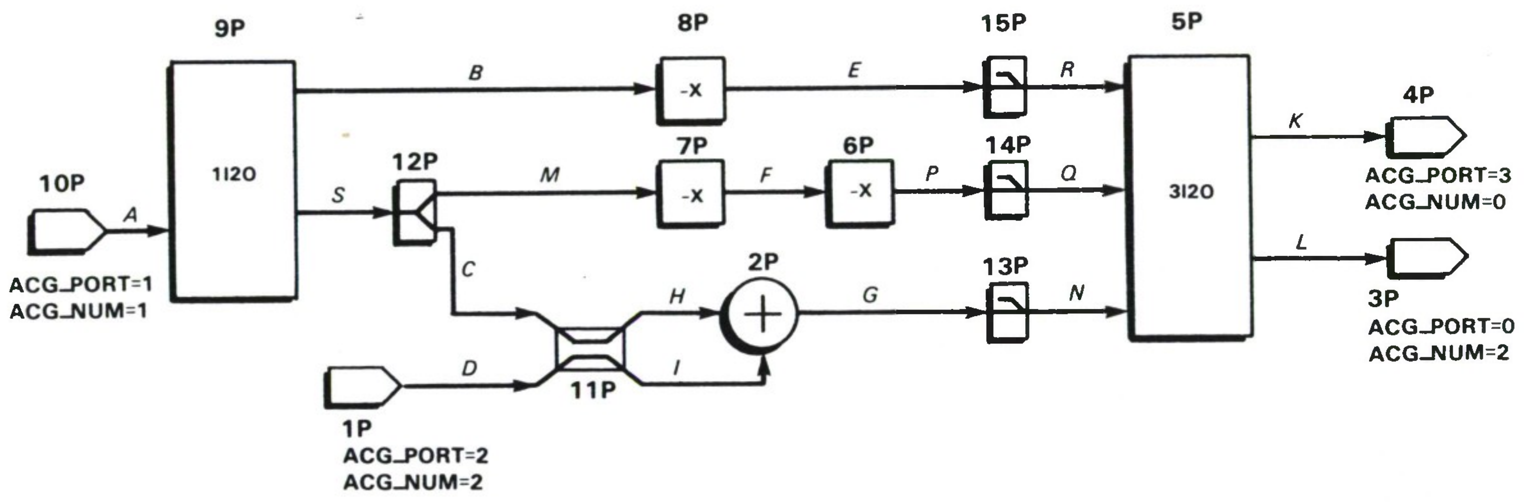


Figure 25. Block diagram with asynchronous primitives.

```
FILE_TYPE=ORDERED_PROCESS_TABLE:
         INPUT (D,2,2)  ; (EX12 INPUT1P)
         IF NOT D JUMP TO LØ
          IF NOT SYNCH2 (C,D,H,I) JUMP TO L1     :  (EX12 SYNCH2.11P)
           ADDER (H,I,G)          ; (EX12 ADDER2P)
           ASYNCH (G,N) ; (EX12 ASYNCH13P)
           IF NOT N JUMP TO L2
            3I2O (R,O,N,K,L)      ; (EX12 .3I2O5P)
            IF NOT K JUMP TO L3
             OUTPUT (K,Ø,3)       ; (EX12 OUTPUT4P)

            IF NOT L JUMP TO L4
             OUTPUT (L,2,Ø)       ; (EX12 OUTPUT3P)



         INPUT (A,I,I)  ; (EX12 INPUT1ØP)
         IF NOT A JUMP TO L5
          I12O (A,B,S)  ; (EX12 .I12O9P)
          IF NOT B JUMP TO L6
           NEG (B,E)    ; (EX12 NEG8P)
           ASYNCH (E,R) ; (EX12 ASYNCH15P)
            IF NOT R JUMP TO L7
             3I2O (R,Q,N,K,L)     ; (EX12 .3I2O5P)
             IF NOT K JUMP TO L8
              OUTPUT (K,Ø,3)      ; (EX12 OUTPUT4P)

             IF NOT L JUMP TO L9
              OUTPUT (L,2,Ø)      ; (EX12 OUTPUT3P)



          IF NOT S JUMP TO L1Ø
           SPLIT2 (S,M,C)         ; (EX12 SPLIT2.12P)
           IF NOT M JUMP TO L11
            NEG (M,F)   ; (EX12 NEG7P)
            NEG (F,P)   ; (EX12 NEG6P)
            ASYNCH (P,O)          ; (EX12 ASYNCH14P)
             IF NOT O JUMP TO L12
              3I2O (R,Q,N,K,L)   ; (EX12 .3I2O5P)
              IF NOT K JUMP TO L13
               OUTPUT (K,Ø,3)     ; (EX12 OUTPUT4P)

              IF NOT L JUMP TO L14
               OUTPUT (L,2,Ø)     ; (EX12 OUTPUT3P)



           IF NOT C JUMP TO L15
            IF NOT SYNCH2 (C,D,H,I) JUMP TO L16 ;  (EX12 SYNCH2.11P)
             ADDER (H,I,G)        ; (EX12 ADDER2P)
             ASYNCH (G,N)         ; (EX12 ASYNCH13P)
             IF NOT N JUMP TO L17
              3120 (R,Q,N,K,L)   ; (EX12 .31205P)
              IF NOT K JUMP TO L18
               OUTPUT (K,Ø,3)     ; (EX12 OUTPUT4P)

              IF NOT L JUMP TO L19
               OUTPUT (L,2,Ø)     ; (EX12 OUTPUT3P)
```

76418-26

Figure 26. An ordered process table.

This situation is correctly handled in the ordered process table. Next, notice that the 3I2O at 5P is found in the ordered process table four times. This accounts for the many different scenarios preceding its execution. Asynchronous primitives require this type of special handling since they may execute without receiving all of their inputs. This code repetition/speed trade-off is discussed at the end of this section. The process table for the IIR/FIR filter example is shown in Figure 27 (left column).

```
FILE_TYPE=ORDERED_PROCESS_TABLE;
PROCESSOR 'B3ØØ':
  INPUT (YY2,ZZ2,2) ; (RTPG INPUT29P)
  IF NOT YY2 JUMP TO LØ
   SYNCH1 (YY2,CC) ; (IIRFIR SYNCH1.31P)
   GAIN (L,O,234) ; (IIRFIR GAIN25P)
   GAIN (N,UNØ,567) ; (IIRFIR GAIN24P)
   GAIN (CC,M,575) ; (IIRFIR GAIN26P)
   ADDER (M,O,P) ; (IIRFIR ADDER3ØP)
   ADDER (P,UNØ,OUT) ; (IIRFIR ADDER29P)
   OUTPUT (OUT,Ø,Ø) ; (IIRFIR OUTPUT22P)
   DELAY (L,N) ; (IIRFIR DELAY27P)
   DELAY (CC,L) ; (IIRFIR DELAY28P)
LØ:
END_PROCESSOR;
PROCESSOR 'B2ØØ':
  INPUT (YY1,ZZ1,2) ; (RTPG INPUT27P)
  IF NOT YY1 JUMP TO LØ
   OUTPUT (YY1,ZZ2,Ø) ; (RTPG OUTPUT28P)
LØ:
END_PROCESSOR;
PROCESSOR 'B1ØØ':
  INPUT (YYØ,ZZØ,2) ; (RTPG INPUT25P)
  IF NOT YYØ JUMP TO LØ
   OUTPUT (YYØ,ZZ1,Ø) ; (RTPG OUTPUT26P)
LØ:
END_PROCESSOR;
PROCESSOR 'ØØØØ':
  INPUT (DD,Ø,2) ; (IIRFIR INPUT23P)
  IF NOT DD JUMP TO LØ
   SYNCH1 (DD,EE) ; (IIRFIR SYNCH1.32P)
   GAIN (E,D,2ØØØ) ; (IIRFIR GAIN18P)
   GAIN (E,I,14Ø) ; (IIRFIR GAIN21P)
   GAIN (H,G,24ØØ) ; (IIRFIR GAIN19P)
   GAIN (H,J,2355) ; (IIRFIR GAIN2ØP)
   GAIN (EE,A,15ØØ) ; (IIRFIR GAIN17P)
   ADDER (J,I,K) ; (IIRFIR ADDER14P)
   ADDER (G,D,F) ; (IIRFIR ADDER15P)
   ADDER (A,F,B) ; (IIRFIR ADDER16P)
   ADDER (B,K,AA) ; (IIRFIR ADDER13P)
   ASYNCH (AA,BB) ; (IIRFIR ASYNCH33P)
   DELAY (E,H) ; (IIRFIR DELAY1P)
   DELAY (B,E) ; (IIRFIR DELAY7P)
    IF NOT BB JUMP TO L1
     OUTPUT (BB,ZZØ,Ø) ; (RTPG OUTPUT24P)
L1:
LØ:
END_PROCESSOR;
END.
```

```
                MACRO EXAMPLES

adder      $macro    a,b,c
           zalh      :a.s:
           addh      :b.s:
           sach      :c.s:
           $end

delay      $macro    a,b
           lac       :a.s:
           sacl      :b.s:
           $end
```

76418-27

Figure 27. IIR/FIR process table; primitive macro programs.

5.2.4.2 Assembly Code Generator

The last step in the block diagram compilation is the translation of the output of the ordering program into TMS32010 assembly code. This is easily accomplished by creating a library of macro programs which roughly correspond to each graphical primitive. Figure 27 (right column) shows the macro programs corresponding to the ADDER and DELAY primitives. Information from the ordered process table, along with some additional information from the signal and primitive tables, is used during assembly code generation. Figure 28 shows part of the assembly code for cell 0000 in the IIR/FIR example.

5.3 Task Assignment Tool

An automated procedure has been developed to assign parts of a large problem, described in a data flow language, to cells in the array. This procedure uses an algorithm called "simulated annealing"[10] to find an assignment with minimum "cost." The cost of an assignment is determined by summing the charges for "penalties" incurred by the assignment. The group of penalties is flexible, but might include: (1) "Excessive Idle," i.e., too many processors in the array are idle too often; or (2) "Excessive Intercell Communication," i.e., the processing required for intercell communication is too large a fraction of total processing time. After associating a charge for each penalty incurred, and by following the six steps listed below, the assignment with the minimum cost can be identified:

(1) Randomly assign tasks to cells.

(2) Compute cost of current assignment.

(3) Randomly move one task from one cell to another.

(4) Recompute cost.

(5) Accept this change with probability p.

(6) Decrease p by some percentage.

(7) Decrement count. If zero, finished. Else, go to (3).

Future plans call for integration of the task assignment tool with the rest of the BDC.

5.4 Efficiency and Evaluation

The high-level software tools described above can be evaluated in two areas. First, we must judge how effective the tools are at reducing the user's software development task. Second, we must determine whether the software which is output by the tools makes efficient use of the array hardware. Evaluation of the high-level software in these two areas is the subject of the rest of this section. The chart shown in Figure 29 shows some example evaluations.

```
Jul  7 15:08 1986  iirfir.list Page 1

0037 040E        start
0038                    INPUT   DD,0,2  ; (IIRFIR INPUT23P)
0001 040E
0002 040E
0003 040E
0004 040E
0005 040E
0006 040E
0007 040E 447E   IN RAMINT,INTRPT
0008 040F 2470   LAC ONE,2*2
0009 0410 797E   AND RAMINT
0010 0411 FF00   BZ X1038
     0412 0417
0011 0413
0012 0413 4206   IN DD,2 ; GET DATA
0013 0414
0014 0414 2170   LAC ONE,DDB  ; SET THE READY BIT
0015 0415 7A04   OR DDW
0016 0416 5004   SACL DDW
0017 0417
0018 0417        X1038
0019 0417
0039                    IFNOT   DDW,DDB,LO
0001 0417 2170   LAC  ONE,DDB
0002 0418 7904   AND  DDW
0003 0419 FF00   BZ LO
     041A 0454
0040                    SYNCH1  DD,EE   ; (IIRFIR SYNCH1.32P)
0001 041B
0002 041B
0003 041B 2170   LAC ONE,DDB  ; CLEAR THE INPUT
0004 041C 7872   XOR HIGH16   ; READY BIT
0005 041D 7904   AND DDW
0006 041E 5004   SACL DDW
0007 041F
0008 041F 2006   LAC DD
0009 0420 5008   SACL EE
0010 0421
0041                    GAIN    E,D,2000         ; (IIRFIR GAIN18P)
0001 0421 6A07   LT E
0002 0422 87D0   MPYK 2000
0003 0423 7F8E   PAC
0004 0424 5C05   SACH D,4
0042                    GAIN    E,I,140 ; (IIRFIR GAIN21P)     ;
0001 0425 6A07   LT E
0002 0426 808C   MPYK 140
0003 0427 7F8E   PAC
0004 0428 5C0C   SACH I,4
0043                    GAIN    H,G,2400         ; (IIRFIR GAIN19P)
0001 0429 6A0B   LT H
0002 042A 8960   MPYK 2400
0003 042B 7F8E   PAC
0004 042C 5C0A   SACH G,4
0044                    GAIN    H,J,2355         ; (IIRFIR GAIN20P)
0001 042D 6A0B   LT H
0002 042E 8933   MPYK 2355
```

```
Jul  7 15:08 1986  iirfir.list Page 2

0003 042F 7F8E   PAC
0004 0430 5C0D   SACH J,4
0045                    GAIN    EE,A,1500        ; (IIRFIR GAIN17P)
0001 0431 6A08   LT EE
0002 0432 85DC   MPYK 1500
0003 0433 7F8E   PAC
0004 0434 5C00   SACH A,4
0046                    ADDER   J,I,K   ; (IIRFIR ADDER14P)
0001 0435 200D   LAC J
0002 0436 000C   ADD I
0003 0437 500E   SACL K
0047                    ADDER   G,D,F   ; (IIRFIR ADDER15P)
0001 0438 200A   LAC G
0002 0439 0005   ADD D
0003 043A 5009   SACL F
0048                    ADDER   A,F,B   ; (IIRFIR ADDER16P)
0001 043B 2000   LAC A
0002 043C 0009   ADD F
0003 043D 5002   SACL B
0049                    ADDER   B,K,AA  ; (IIRFIR ADDER13P)
0001 043E 2002   LAC B
0002 043F 000E   ADD K
0003 0440 5001   SACL AA
0050                    ASYNCH  AA,BB   ; (IIRFIR ASYNCH33P)
0001 0441 2001   LAC AA
0002 0442 5003   SACL BB
0003 0443
0004 0443 2070   LAC ONE,BBB
0005 0444 7A04   OR BBW
0006 0445 5004   SACL BBW
0007 0446
0051                    DELAY   E,H     ; (IIRFIR DELAY1P)
0001 0446 2007   LAC E
0002 0447 500B   SACL H
0052                    DELAY   B,E     ; (IIRFIR DELAY7P)
0001 0448 2002   LAC B
0002 0449 5007   SACL E
0053                    IFNOT   BBW,BBB,L1
0001 044A 2070   LAC  ONE,BBB
0002 044B 7904   AND  BBW
0003 044C FF00   BZ L1
     044D 0454
0054                    OUTPUT  BB,ZZ0,0         ; (RIPC OUTPUT24P)
0001 044E
0002 044E
0003 044E
0004 044E
0005 044E
0006 044E
0007 044E 447E   IN RAMINT,INTRPT
0008 044F 2170   LAC ONE,0*2+1
0009 0450 797E   AND RAMINT
0010 0451 FF00   BZ X1102
     0452 0454
0011 0453
0012 0453 4803   OUT BB,0
```

Figure 28.   Assembly code for cell 0000.

76418-28

6-CHANNEL FILTER BANK
(Not Including I/O)

GENERATOR	LOOP TIME (μs)	PROGRAM LENGTH (in Words)	DATA MEMORY LOCATIONS NEEDED
BDC (Simple Primitives)	37	186	72
BDC (Extended Primitives)	14	34	30
HAND CODED	14	34	30

8-POINT FFT
(Complex Data, No Scaling, No I/O)

GENERATOR	LOOP TIME (μs)	PROGRAM LENGTH (in Words)	DATA MEMORY LOCATIONS NEEDED
BDC (Simple Primitives)	26	1024	48
BDC (Extended Primitives)	26	12B	4B
HAND CODED	26	128	20

76418-29

*These charts show the relative performances of the three methods of code generation. The "BDC Simple Primitive" method limits the block diagram to adders, mults, etc. The "BDC Extended Primitive" method allows the use of higher-level primitives, e.g., a **FILTER_BANK** primitive. The "Hand Coded" method is self-explanatory.*

Figure 29. BDC vs manually written code.

5.4.1 Easing the Programming Task

Since the primary design goal of the BDC was to automate the MIMD software development process, we would expect that for any implementation of a BDC to be judged a success, it must substantially reduce the amount of time the user spends programming. Using this criterion, preliminary results indicate that the BDC implementation described in this report is a success. First, the fact that the experienced user does not get bogged down in TMS32010 assembly code generation causes a marked decrease in the amount of time spent developing a program. A novice user need not learn the TMS32010 assembly language at all. Second, the BDC is, for the most part, processor independent (i.e., should a new MIMD array be developed in the future, only the assembly code generation module would need to be substantially modified). The interface to the user could remain virtually identical. This is the same sort of criterion which one uses to judge standard programming languages, namely, source code transportability among various hardware implementations.

5.4.2 Efficiency

The second area of evaluation of the BDC is efficiency. Efficiency encompasses many notions of performance; so, for the purposes of this report, we will consider the following comparisons:

(1) Speed of compiled code vs speed of manually written code,

(2) Size of compiled code vs size of manually written code, and

(3) Data memory allocation in compiled code vs data memory allocation in manually written code.

We will find that in all three of these comparisons BDC code, while not as efficient as manually written code, performs reasonably.

In the speed comparison, we find that applications drawn by the user containing primitive elements such as ADDERs and MULTs, or nonprimitive elements which are defined in terms of primitive elements such as ADDERs and MULTS, take almost twice as long to run as their hand-coded counterparts. The reason for this is twofold. First, the compiler is unable to make use of the index registers. Thus, special TMS32010 pipelining instructions which use the index registers as pointers are never generated by the BDC output. Second, the compiler is unable to use the accumulator as a storage location between bodies. This means that many intermediate values, which might have been stored in the accumulator, must be written to and then read from data memory. However, if the user has more-complex primitive bodies available to him (e.g., a primitive body that represents a filter section), we find that the efficiency of the BDC output can approach 90 percent of the efficiency of hand-coded programs. Thus, depending on the size of the library of primitive bodies, the speed of BDC code ranges anywhere from 50 to 90 percent of the speed of hand-coded programs.

In the program length comparison, the results depend greatly on the particular application. Since the BDC generates in-line code, rather than subroutines, any application which makes use of the same code module many times will be much longer if generated by the BDC than if generated by hand. One reason for inefficiency is that in-line code runs considerably faster than subroutines and, since most real-time applications are constrained by the speed of the TMS32010 (rather than the size of its program memory), this trade-off seems justified. In future versions of the BDC, the user would be given the option of in-line code or subroutine generation. A second reason for inefficiency is the need for data memory access when hand-coded programs can use the accumulator. This inefficiency was also seen in the speed comparison.

In the data memory use comparison, the results show that the BDC is quite inefficient compared with the hand-coded programs. Once again, we see that the reason for this inefficiency is the fact that the accumulator cannot be used to store intermediate results. For this reason, future versions of the BDC will make more efficient use of the accumulator as a temporary storage location. However, when the primitive library is expanded to include more-complex modules, the use of the data memory decreases dramatically.

5.4.3 Conclusion

Concluding, we see that if the BDC makes use of an expanded primitive library, its output is comparable to hand-coded programs in many areas of efficiency. When the added BDC benefit of ease of use is also considered, the BDC appears to be a success.

There are many areas for improvement of the BDC. First, data-type support would be valuable in order to use the BDC with hardware supporting different data types. If an array were built out of cells centered around a floating-point processor, BDC data-type support would be quite valuable. Second, the BDC should be given more knowledge about the type of cell for which it is generating code. In the TMS32010 case, an improvement would be to pass variable values in the accumulator, instead of in data memory. This would increase speed and decrease memory use at the same time. Third, as already mentioned, program length might be significantly reduced without an unreasonable effect on speed if subroutines were generated for common-code blocks instead of in-line code. Fourth, extended signal and body properties could be provided to allow more-sophisticated applications. These properties could support more-complex data structures, such as circular buffers and multiple time scales. Fifth, some of the explicit boundaries, such as SYNCHs and ASYNCHs, which the user must place between asynchronous and synchronous blocks could be inferred by the system. Finally, the BDC could be merged with the task assignment module forming a more-complete system. Part of the task assignment module would be an algorithm for determining the percentage of real time that each processor was running.

6. CONCLUSIONS

In conclusion, the work undertaken in this report has encompassed many areas. First, a complete MIMD hardware system was designed, implemented, and debugged. Second, software was provided to make the hardware controllable from a Host Computer. Third, the hardware was demonstrated in a particular application, the 12-channel filter bank. Finally, high-level software tools were designed and implemented to make the hardware more attractive to use by easing the software development task.

What has been created is a complete system which can be used as a general digital signal processing facility. The fact that the MIMD system can be programmed automatically is novel, and forms the basis of the original research of the report. The existing system is modular enough that a new hardware design having different cell structure could still make use of many of the low- and high-level tools already existing.

A 16-cell array has recently been built and debugged. This new hardware will afford us the opportunity to test and, if necessary, upgrade both the low- and high-level software tools.

Future research and development would be concentrated in two areas. First, the existing system could be upgraded by implementing any of the minor upgrades described at the end of Sections 3 and 5. Basically, these upgrades were (1) hardware modifications to support breakpoints and single stepping, and (2) various modifications in the BDC to improve efficiency.

The second area of future research would be in major hardware and software modifications. In the area of hardware, "cell on a chip" architectures which would be supported by the type of low-level tools described in this report could be designed. In the software field, a BDC which could naturally handle block-type data transfers would be a more general-purpose tool than the BDC described in this report.

ACKNOWLEDGMENTS

I would like to thank Gerald O'Leary and Bruce Musicus for supervising the writing of my Master's thesis, on which this report is based. Gerry, once again, was very helpful in providing both technical advice and general guidance; his help was invaluable. In addition, I'd like to thank Ben Gold, Roger Hale, Al Huntoon, and Joe Tierney for answering my technical questions and Ed Adamczyk and Al Davison for their expert technical assistance.

REFERENCES

1. M.J. Flynn, "Very High-Speed Computing Systems," Proc. IEEE **54**, 1901-1909 (1966).
2. J. Blackmer, G. Frank, and P. Kuekes, "A 200 Million Operations Per Second (MOPS) Systolic Processor," in *Real-Time Signal Processing IV,* Proc. SPIE **298**, 10-18 (1981).
3. A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer," IEEE Trans. Comput. **C-32**, 175-189 (1983).
4. M.C. Pease, III, "The Indirect Binary *n*-Cube Microprocessor Array," IEEE Trans. Comput. **C-26**, 458-473 (1977).
5. M.A. Zissman, "An Array of Digital Signal Processors," Bachelor's Thesis, MIT (1985).
6. A.L. Fisher, H.T. Kung, L.M. Monier, and Y. Dohi, "The Architecture of a Programmable Systolic Chip," J. VLSI Comput. Syst. **1**, 153-169 (Fall 1984).
7. G.M. Amdahl, G.A. Blaauw, and F.P. Brooks, Jr., "Architecture of the IBM System/360," IBM J. Res. Dev. **8**, 87-100 (1964).
8. T.P. Barnwell, III and C.J.M. Hodges, "A Synchronous Multi-Microprocessor System for Implementing Digital Signal Processing Algorithms," SOUTHCON/82, Orlando, Florida, Vol. 21, No. 4, 23-25 March 1982, pp. 1-6.
9. J.R. Mann and F.M. Rhodes, "A Wafer Scale DTW Multiprocessor," ICASSP '86, Vol. 3, 8-11 April 1986, pp. 1557-1560.
10. S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," Science **220**, 671-680 (1983).

APPENDIX — USER COMMANDS

While in the Host Control mode, the user can issue instructions to read and write the following:

TMS32010 Register Set

TMS32010 Stack

TMS32010 Data Memory

Cell Offchip Memory

Commands which display, modify, or load memory can only be executed while the array is in Command mode. Command mode is entered when the reset button is pushed or when a ***stop*** command is issued. Following is a list of commands which are currently available to the user.

Argument Definitions

filename: A file specification recognizable by UNIX, e.g., /u0/maz/tms320/bpf.obj (or just bpf.obj if the array control program was called from /u0/maz/tms320). The file should be the object code resulting from the assembly of a TMS32010 assembly language program in the Texas Instruments SDSMAC format.†

cell_address: The hexadecimal representation of the relative address of a cell. Relative addressing begins from the cell which is connected to the HNI. Addressing follows the same rules as those for intercell communication, e.g., the cell which is one cell to the east and one cell to the south of the cell connected to the HNI has address 8101 (=1000 0001 0000 0001).

memory_address: The hexadecimal representation of a memory address. Valid data memory and addresses run from 0000 to 008F. Valid external memory addresses run from 0000 to 0FFF. Leading zeros can be dropped.

new_value: Any 4-digit (16-bit) hexadecimal number, e.g., A8C4.

† "TMS32010 Assembly Language Programmer's Guide," Texas Instruments (1983).

Commands (Square Brackets Contain Optional Part of a Command)

l[oad] filename cell_address

Loads the file specified by filename into the cell at relative address *cell_address.*

d[isplay] d[ata] m[emory] cell_address

Displays the entire 144-word data memory of the cell at relative address *cell_address.*

d[isplay] e[xternal] m[emory] cell_address memory_address

Displays 128 words of external memory, beginning at memory address *memory_address*, of the cell at relative address *cell_address.*

m[odify] d[ata] m[emory] cell_address memory_address new_value

Sets the location specified by *memory_address* to *new_value* of the data memory of the cell located at relative address *cell_address.*

r[un]

Sends the ***run*** command to the array.

s[top]

Sends the ***stop*** command to the array.

c[lear]

Clears the CRT screen.

h[elp]

Lists the available user commands on the screen below the command line.

n[ohelp]

Removes the list of user commands from below the command line.

q[uit]

Exits the array control program back to UNIX exec.

Notice that there are no ***display register*** or ***modify register*** commands. These are not necessary because each cell automatically writes the contents of most of the TMS32010 registers to data memory when entering Command mode, as shown in the following table:

Memory Location	TMS32010 Register
88	Low Word of Accumulator
89	High Word of Accumulator
8A	Auxiliary Register 0
8B	Auxiliary Register 1
8C	Top of Stack — PC Register
8D	Second on Stack
8E	Third on Stack
8F	Bottom of Stack

In addition, modifying the data memory locations listed above will modify the corresponding registers when a ***run*** command is issued.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-86-102	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* An Array Computer for Digital Signal Processing		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER Technical Report 759
7. AUTHOR(s) Marc A. Zissman		8. CONTRACT OR GRANT NUMBER(s) F19628-85-C-0002
9. PERFORMING ORGANIZATION NAME AND ADDRESS Lincoln Laboratory, MIT P.O. Box 73 Lexington, MA 02173-0073		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Element Nos. 33401F and 64754F
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Systems Command Andrews AFB Washington, DC 20334		12. REPORT DATE 5 January 1987
		13. NUMBER OF PAGES 82
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Electronic Systems Division Hanscom AFB, MA 01731		15. SECURITY CLASS. *(of this Report)* Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

block diagram compiler
digital signal processing
MIMD architecture
multiprocessor
processor arrays

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes the implementation of a MIMD array computer designed and built at the Lincoln Laboratory for signal processing. Some of the software tools needed to successfully use such an array are discussed, and the software package written to allow debugging of the array from a host computer is described. The first application of the array, a 12-channel filter bank front-end for a speech recognition system, is discussed. Finally, a block diagram compiler is described. This compiler converts block diagrams, entered at a CAE workstation, into efficient assembly code for all cells in the array.

DD FORM 1473 1 Jan 73 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*