PART THREE. PARALLEL PROCESSING

I posted a 'fig -- FORTH INTEREST GROUP' logo on the door of my office. About two years ago, a fellow knocked the door and introduced himself, saying that he saw the poster and was glad to find another Forth enthusiast. He told me that he had programmed in STOIC, which is a early variant of Forth, and was working on a parallel processor project. He also asked me if I would be interested in this project. He spoke in an Eastern European accent, with every vowel and consonant meticulously pronounced. It was not surprising to find out in a short while that he was a Polish mathematician. His name was Wlodzimierz Holsztynski, which I still cannot spell without looking at my notebook.

So, I started working on this parallel processor project. The processor was produced by NCR, with a strange name-- Geometric Arithmetic Parallel Processor-- or GAPP for short. The central processing unit in it was a strange one bit ALU. It was very difficult to think in bits, after cutting the teeth on 8 bit processors, and gradually migrating to 16 bit processors and 32 bit processors. However, Dr. Holsztynski was able to show how complicated problems could be broken down and solved very elegantly using bit-serial algorithms. When you have a very large array of these simple processors working coherently together, suddenly you have a extremely powerful computing structure with the throughput of CRAY supercomputers at a very small fraction of the costs.

After working with him for several months, a colleague told me that he was no less than the inventor of the GAPP chip. No wonder that he was so much at ease in thinking at the bit and gate level. In many ways, Dr. Holsztynski and Chuck Moore are very similar. In their programs, they would not allow a single instruction, even a single bit in an instruction, or a single machine cycle to be wasted by not serving useful function. They are computer poetry in the purest form. These are the people who push the state of the art.

Parallelism seems to be the only way to achieve the computation throughput needed for large and complicated applications. GAPP chip is a very unique way to link a large amount of processing units together with the least amount of silicon and interconnections. In this section I am presenting the limited experience in learning to use it. Lots of material are still under development and are proprietary information. However, what's published here can give us a glimpse of the potential of this technology.

Once we build the processor array and some form of a controller to operate the array, Forth can be of great help in the use and the programming of the processor array. Two papers were presented at the 1987 Rochester Forth Conference, and they are reprinted here showing the status of this parallel processor project. A GAPP array simulator was developed before the array

was built, using an image processor to simulator a 512 by 512 GAPP array. Code for this simulator is also included here. GAPP machine code is very similar to bit slice microcontroller code. Forth is a natural language to implement an assembler for GAPP array. The fourth paper in this section shows how to assemble GAPP code.

The last paper is of a different nature. It was a paper I presented at the FPS Array Processor Users Conference in May, 1984 at Denver, Co. It shows how one can put a large floating point math processor under interactive control for program development. It turns a floating point processor into a huge vector stack machine to process large arrays of floating point vectors. This proves two assertions: that Forth can handle floating point numbers in a grand style, and that floating point number is an I/O problem.

IX. HIGH DENSITY PARALLEL PROCESSING

I. The Processor Array and Macro Controller

A GAPP processor array of 11520 processors and its associated controller were built and tested. It allows the processor array to be programmed conveniently using high level languages without sacrificing speed or code efficiency. The system is fully functional. The hardware structure and special features of this system are presented in this report.

1. The Parallel Processor System.

As part of the process of evaluating parallel processor algorithms with emphasis on image processing we have developed a complete parallel array processor system. This is a necessary tool because large programs require unacceptable long time to execute on a software simulator, and because algorithm optimization ultimately requires testing with real time data. Included in the system are a 96 by 120 array of GAPP processors(Geometric Arithmetic Parallel Processor, NCR 45CG72) and an MIMD (Multiple Instruction Multiple Datapaths) controller optimized for program compression and fast program flow control.

Rather than inventing a new operating environment, the system was designed to operate as an external coprocessor to an IBM AT personal computer as a host. Paths are provided from the host to the parallel processor system for program and data loading, run-time operation, and status monitoring. In addition, high speed 12 bit parallel input and output ports are provided which are capable of 10 megawords per second syncrhronous data transfers, and slower asynchronous transfers.

Processing within the parallel processor system is completely self-contained so that once started by the host, program execution can proceed independently. Our present system loads data via a DMA channel in the host computer and the results can be unloaded similarly to host or to a real time video display unit. A software console program was developed for use in the host computer to control the parallel processor system interactively.

2. Processor Array

The SIMD (Single Instruction Multiple Datapaths) data processing section consists of an array of GAPP devices and is constructed from four circuit board assemblies, each of which has a 60 by 48 array of single bit GAPP processors. The boards, which were specially designed for this application, contain extensive signal buffering to allow array expansion in all four directions by using multiple boards. This versatility allows altering the array aspect ratio for experimentation with various classes of problems. Array expansion with these components is feasible up to approximately 256 by 256 cells at which point it is worthwhile to design a unique board package for each case so as to optimize density and area efficiency.

The processors are organized as two arrays as show in Figure 1: one of 12 by 96 processors for input/output corner-turning, and a main array of 108 by 96 processors. The two arrays may optionally execute from independent instruction and address streams. However, in the present system only one stream is used. In the main array, the EW and NS register planes are connected in a cylindrical surface topology in both the east-west and north-south directions, although spiral and other interconnections are jumper selectable. The corner-turning array is cylindrical in the

north-south direction for the NS register plane, and uses the east edges for input and the west edge for output. The two arrays are connected, also in a cylindrical manner, by the CM register plane in the GAPP devices.

3. The Distributed Macro Controller (DMC)

The controller, dubbed DMC, addresses the critical issue in parallel processing computers like the GAPP with high processor density and limited memory and instruction sets. The controller allows ready implementation of adaptive programming decisions made by the host; that is, without loss of machine cycles. The top level architectural innovation is that the controller is a MIMD machine that processes three different instructions streams simultaneously as show in Figure 2. A Flow Control Unit feeds several (here two) Macro Generator Units. The instruction streams from the Macro Generator Units are combined to feed the control lines of the GAPP array. Each of these units will be a single chip in VLSI. All Macro Generator Units are identical.

Both the Flow Control and Macro Generator Units use externally writable control stores to store instruction streams. The Flow Control Unit supervises program flow within DMC while the Macro Generator Units produce output instructions for the GAPP array. The MIMD architecture is hierarchical; i.e., the Flow Control Unit directs the production of the programs from the Macro Generator Units. The final output stream consists of two 15 bit words, combined to form a single 20 bit instruction and address stream for the GAPP. The controller offers a very high degree of program compression. Existing sequencers have wide microcode words but little program optimization or compression.

The Flow Control Unit allows eight levels of nested subroutines and eight levels of nested loops. While loops increment, the loop counts of interior loops can be changed. Subroutine calls and returns are performed in 3 clock cycles. With the provision that subroutines are at least three instructions long, this allows penalty-free macroprogramming. External inputs may be tested for conditional operatings (branching, looping, calling, and returning). The Flow Control Unit uses a 32 bit wide instruction format. It is designed to be a single chip unlike existing sequencers, although its primary function in the present controller is to direct the internal flow (within the DMC) of the program.

The Macro Generator Units, which are physically identical and each designed to be a single chip, have several novel features:

(1) Callable macro and address routines
(2) Automatic memory management
(3) Static and dynamic reinterpretation logic
(4) A rich set of stack operation.

Feature (1) is for program compression. Pre-loaded instruction streams can be called by specifying a pointer and length. Typically, these are not GAPP instructions but instructions which cause the Macro Generator Units to produce GAPP code indirectly.

Memory management calculates physical addresses given logical ones. Thus all of the memory addressing is indirect and penalty free. A linked list of memory segments with "occupied" and "free" areas is maintained. This handles allocation of memory and makes the task of the GAPP programmer much easier. Also, if these functions were to be performed in software, the processing system would not be able to operate at full speed.

Reinterpretation is a method of program compression useful from both an op-code and memory point of view. There are both dynamic and static reinterpretations available. Reinterpretation involves performing an exclusive-OR operation on the output with a mask pattern. A number of patterns may be stored. Dynamic reinterpretation allows an external constant to be loaded in where the bits of the constant can be used to modify the output with one of several masks. Static reinterpretation is only selectable at the macro-instruction level. Thus, the if-then-else construction becomes available to parallel processors without penalty.

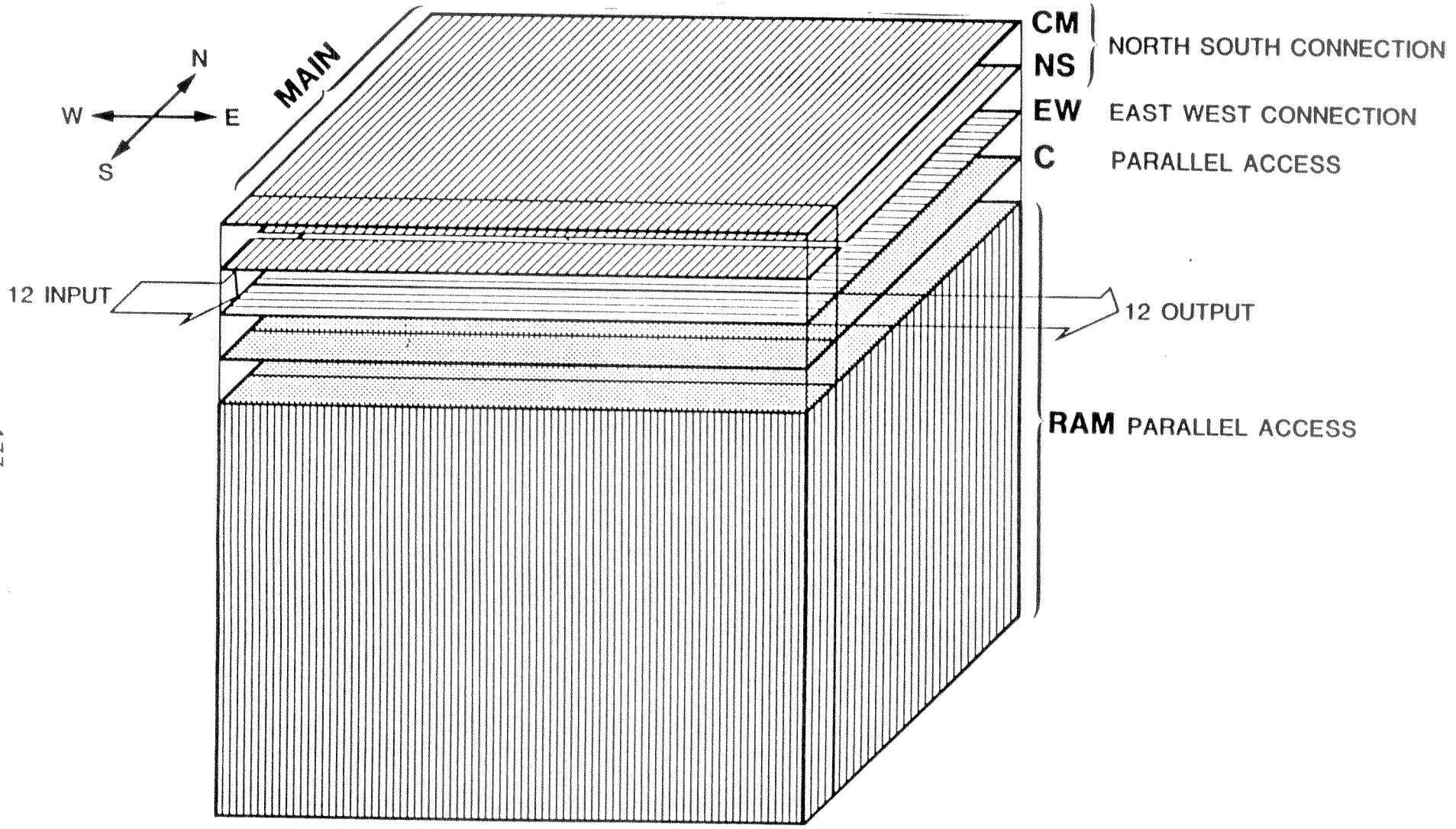


Figure 16. GAPP processor array

The usefulness of reinterpretation is that applications natural to geometric SIMD machines tend to be highly patterned. Addition, subtraction and template matching to either a zero or one differ only in the selection of CARRY and BORROW, loops proceed by alternately selecting one stack or another, etc. A study of the class of transformations natural to GAPP-like machines reveals the frequent occurrence of such patterns allowing switching between instructions with the use of reinterpretation bits. Reinterpretation of address bits allows symmetrical operations within address space.

The controller also provides a rich set of stack operators, operating simultaneously on two stacks holding address pointers to the GAPP memory. Stacks offer a way of changing the instruction sequence to the GAPP in nonconsecutive or nonlinear ways. Two stacks with two top elements cached in the address Macro Generator Unit give the programmer convenient access to four different memory areas to implement complex arithmetic and logic operations.

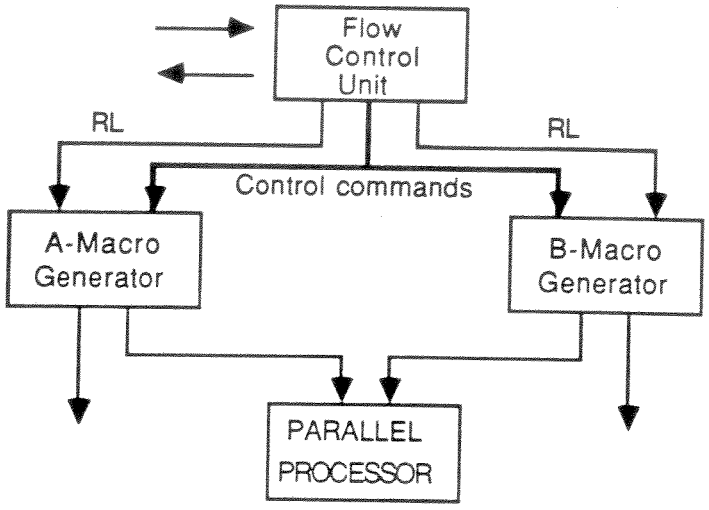
4. Conclusions

This parallel processor system is currently fully functional with the GAPP processor array. The system was designed to run at the maximum clock rate of 10 Mhz. Currently it is running at 2.5 Mhz. The computational throughput is thus 28.8 Giga instructions per second. We are developing software tools and programs to evaluate its performance for many different classes of problems, such as image processing and understanding, real time signal processing and analysis, translation invariant and non-invariant problems, and the general studies on the using and programming of parallel processing systems. Some of the results will be show in a video tape, demostrating the real time image processing capability of this system.

The controller is rich in ways that optimize the programming of GAPP-like arrays. The detailed architecture will be presented in a patent application. We believe that the study of its concepts will allow better understanding of the maximum efficiency obtainable from geometric SIMD arrays and lead to distinct developments in this branch of computer science.

5. Acknowledgements

Dr. Wlodzimierz Holsztynski, the inventor of the GAPP device, provided the architectural design of the Distributed Macro Controller. Integrated Test Systems of Santa Barbara, CA constructed both the GAPP processor boards and the Distributed Macro Controller system.



RL: Reinterpretation Logic

Figure 17. Functional units within the DMC

X. HIGH DENSITY PARALLEL PROCESSING

II. Software and Programming

Tools, utilities, assemblers, and compilers are needed to develop programs which can be run on our parallel processor system, making use of the full power of the GAPP processor array and the Distributed Macro Controller. Some of the software tools are described here and a few examples are also given to illustrated the process of software development on this system.

In order to make the parallel processor system generally useful for experimentation by those not familiar with the hardware, a set of software tools is being developed. A host-resident console program was produced that allows program loading, program execution at arbitrary starting addresses, program halting, and examination of status and error flags.

An assembler program has been developed to assemble GAPP instructions and address macros for the Macro Generator Units. This program has a unique structure as it must deal with three concurrent instruction streams and keep track of relative timing or program lengths among them. This assembler has also to take features of a high level compiler so as to generate appropriate instruction streams for the Flow Control Unit, which is the focal point of the entire parallel processor system, coordinating the GAPP processor arrary with the two Macro Generator Units.

1. The Console Monitor

The Console Monitor is a program which runs on the IBM AT host computer. It allows a user to perform some primitive operations on the Distributed Macro Controller (DMC) system, such as initiation, loading code into the Flow Control Unit and the Macro Generator Units, running a flow program, and monitoring the status of the DMC-GAPP system while it is running.

The DMC is designed so that all its writable control store memory areas can be accessed by the host computer. In fact, all the writable control store memory and many of the important internal registers in both the Flow Control Unit and the Macro Generator Units are mapped to a contiguous 128 Kbytes of memory. The mapped memory locations can be interrogated by the host, and new code or data can be loaded into these registers and memory areas through the Console Monitor. Effective use of this feature allows a user to assemble GAPP programs and flow programs directly. It is extremely valuable during testing and debugging phases of program development.

The Console Monitor also has many built in high level functions. One is loading programs in either binary form or hexadecimal form. The binary form of a program is simply the image of the 128 Kbyte mapped memory, which can be saved as a binary file on the disk of the host computer. The saved binary file can be loaded back into DMC to restore DMC to the state when the saved file was generated. A hexadecimal file format was defined so that code written in hexadecimal numbers can be translated and downloaded into various selected parts of DMC. This file format also specifies the output file produced by GAPP assemblers and DMC compilers which can be run on other host computers for off line program development.

The other important feature of the Console Monitor is that it can load data into the GAPP corner-turn or input/output processor array and unload results from the corner-turn array, through a DMA board inside the IBM AT. This is necessary to test GAPP and DMC programs and to verify

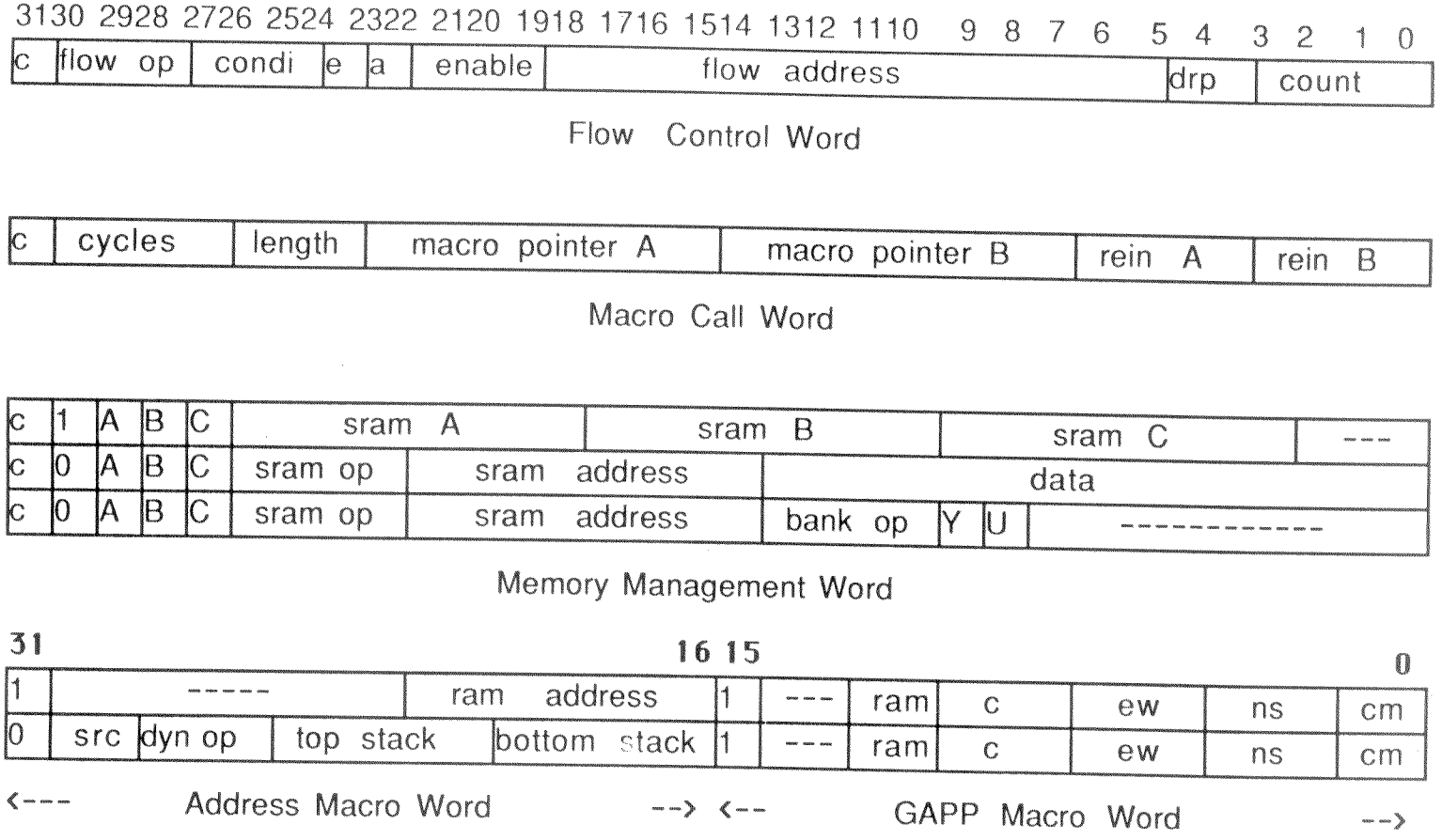


Figure 18. Instruction format for distributed macro controller

that the hardware and software are in working condition. It is also useful in testing various algorithms and evaluating their performance. The Console Monitor thus serves as the major user interface to the parallel processor system.

2. GAPP and Address Macro Assembler

GAPP is a Single-Instruction-Multiple-Datapath (SIMD) machine. Its ALU is a one bit full-adder-subtractor, and there are many data paths around the ALU and the internal registers and memory. A GAPP instruction is used to specify precisely the data paths to and from the registers and the memory. A GAPP instruction is 20 bit wide, 13 bits for data multiplexers and 7 bits for memory selection, as shown in Figure 1. A GAPP program is thus a sequence of these 20 bit instructions commanding the GAPP array how to route data and results inside the processor array. It is fully programmable in the sense that any function that does not exceed the memory capacity of the machine can be executed. However, all processing units execute identical instruction so that a general MIMD (Multiple Instruction Multiple Datapath) command is executed at lower efficiency.

To facilitate programming efforts, a set of GAPP mnemonics is defined to specify the source and destination of data in each clock cycle. The Assembler translates these mnemonics into GAPP code and memory address specifications, and constructs macro routines which are callable by flow programs. Since there are many data paths independently controllable by a single GAPP instruction, the assembler allows the user to specify multiple data paths in a single instruction, as well as the detailed function which has to be carried out by the address macro generator, such as pushing, popping or incrementing the memory pointers on the stacks.

The Macro Assembler generates two concurrent macro's from a set of mnemonic code sequence, terminated by the special operator '_$_'. The address macro takes the most significant 16 bits and the GAPP macro takes the least significant 16 bits of an assembler 32 bit code. The two parts will be separated and downloaded to their respective writable control store memory in the Macro Generator Units.

A library of macro routines is assembled and kept in the Macro Generator Units for the flow control program to call. For specific applications, special macro routines can be define by the user and downloaded to be used together with the library macros. Figure 2 shows a sample of the macro routines required by the Game of Life flow program.

3. Flow Control Assembler

The Flow Control Unit stores its instructions in 32 bit words, each flow instructions requires 3 to 18 flow words. The first word in a flow instruction is called Flow Control Word which specifies the program flow, such as JUMP, LOOP, CALL, and RETURN. The second flow word specifies the macros to be executed from the Macro Generator Units and the clock cycles and number of macro instructions to be executed in this macro. It is called Macro Call Word. The third and following flow words are Memory Management Words, which controls the memory management mechanism in the Address Macro Generator Unit. The formats and the functional fields in these flow words are show in Figure 1.

The Flow Control Assembler compiles the flow control mnemonic code, similar to those commonly used in high level programming languages, and generate sequences of flow words. The flow words will then be downloaded into the Flow Control Unit. Any of the flow instruction, starting with a Flow Control Word, can be executed and the DDP will perform algorithm specified by this and subsequent flow instructions until completion.

Since the Flow Control Unit supports all the fundamental programming structures, such as conditional and unconditional branching, looping, subroutine calling and returning, flow programs can be modularized and written in highly structured form. This practice greatly enhances program readability and eases debugging and maintainence. The Flow Control Assembler produces efficient code within this structured framework.

An example of flow program is show in Figure 3, implementing the Game of Life. It uses the macros assembled by the macro assembler. The core or the Game of Life program is a 25 machine cycle sequence of GAPP instructions, assembled into 4 macros which are called by the main program. The most time consuming part of the program is to dump the map of lifes out to the display device, which is isolated as a subroutine called from the main program loop.

4. Conclusion

We have built enough tools and utilities to program and use the high density SIMD processor arrays efficiently. Complicated algorithms can be broken into address macros, op-code macros, and flow control sequences, which can be assembled from high level source program into machine code loaded into the parallel processor system and executed. Effective use of macros thus tremendously simplifies the programming efforts and greatly compresses the program. The limited experience we have gained in a very short time exceeds our expectation that the DMC controller allows massively parallel processors to be conveniently programmed using high level language without compromising the performance. The program compression will allow much more complicated algorithms to be expressed concisely for the eventual execution on huge processor arrays.

Several articles of this project, as well as research results on parallel computations and on applications have been prepared in more detail for publication.

```
g: init-life   8005 gadr c=p _$_   _$_   _$_

g: ew1sum   8005 gadr p:ns:ew=c _$_
      ew=w _$_
      ns=ew ew=ns _$_
      ew=e _$_
      8000 gadr ns:ew=plus carry _$_
      8001 gadr p=c _$_

g: ns2sum   c=0 ns=s _$_
      8002 gadr plus carry _$_
      8001 gadr ns:ew=p _$_
      ns=s _$_
      8003 gadr plus carry _$_
      8004 gadr p=c _$_


g: ns3sum   8000 gadr ns=p c=1 _$_
      8002 gadr ew=p ns=n _$_   8002 gadr plus carry _$_
      8001 gadr ns=p _$_
      8003 gadr ew=p ns=n _$_
      8003 gadr plus carry _$_
      8004 gadr ns=0 ew=p _$_   8004 gadr ew=plus _$_

g: final   8003 gadr ns=p c=0 _$_
      8002 gadr ns=p borrow _$_
      8002 gadr p=c c=1 _$_
      8005 gadr ew=p _$_
      8002 gadr borrow ew=p ns=0 _$_
      carry _$_


: scratch   807f gadr ;

g: c-to-scratch   scratch   p=c _$_   _$_   _$_

g: shift-scratch   scratch cm=p _$_   south _$_
      scratch p=cm _$_

g: scratch-to-ew   scratch ew=p _$_   _$_   _$_

g: ew-to-scratch   c=ew _$_   scratch p=c _$_   _$_

g: mm-to-scratch   top c=p _$_   scratch p=c _$_   _$_

g: scratch-to-mm   scratch c=p _$_   top p=c _$_   _$_
```

Listing 11. Game of life with GAPP

```
flow-block output-life    ( from gadr 7f scratch plane)
   loop 9 times    4 gre ( frame mark)  _$$$_
   loop 0c times _$$$_
      noop ram-en _$_    shift-scratch macro _$$_
      noop ct-en _$_    shift-scratch macro _$$_ ( 2 msb bits)
      loop 4 times _$_     scratch-to-ew macro _$$_
      loop 10 times _$$$_
         noop _$_     shift-w macro _$$_
      end-loop _$$$_
      end-loop _$$$_
   end-loop _$$$_
   end-loop _$$$_
   return _$$$_
end-block

flow-block game-of-life
   noop mm-en _$_    init-life macro _$$_
flow-block repeat-life
   noop mm-en _$_    6 cycles 6 instructions ew1sum macro _$_ _$_
   noop mm-en _$_    6 cycles 6 instructions ns2sum macro _$_ _$_
   noop mm-en _$_    8 cycles 0 instructions ns3sum macro _$_ _$_
   noop mm-en _$_    6 cycles 6 instructions final macro _$_ _$_
   call output-life adr mm-en _$_    c-to-scratch macro _$$_
   jump repeat-life adr _$$$_
end-block
   stops
end-block
```

Listing 11. Game of life with GAPP (cont'd)

XI. SIMULATOR OF A GAPP PROCESSOR ARRAY

1. THE GAPP CHIP

GAPP (Geometrical Arithmetic Parallel Processor) is a CMOS chip with 72 processors on a single chip. It was invented by Dr. Wlodzimierz Holsztynski, a Polish mathematician, when he was with Martin-Marietta in Florida. The chip is now manufactured by NCR as NCR45CG72. The processors are arranged as a 6x12 array. Each processor is connected to its four nearest neighbors. Processors on the edges of the chip have their connection brought to I/O pins, so that many chips can be connected to form a very large processor array. As the processors in a large array form a 2D array with nearest neighbor connections, it is a very efficient structure to handle two dimensional problems like image processing.

The internal structure of a processor is very simple. It is basically a one bit processor with 4 internal registers and 128 bits of memory. There is a simple ALU which takes the contents of three registers as input and generates a one bit sum, a carry and a borrow in every clock cycle. There are five multiplexers, one in front of each internal register and one in front of the memory. These multiplexers can be programmed to route data among the registers, the ALU, and the memory. It thus belongs to the SIMD (Single Instruction Multiple Datapath) architecture, because the ALU is performing one and the same function every machine cycle. It is a full, one bit adder/subtractor. However, by configuring the registers properly, the ALU can perform all the two bit binary logic functions; therefore, it serves as the basis of a very powerful computing structure. Figure 19 shows the contents of a GAPP processor element.

Instructions to the processor array is used to control the multiplexers to define the datapath in a clock cycle. All processors in the processor array execute the same instruction at any given cycle. The instruction is broadcast to all processors and it is executed synchronously. An instruction consists of 20 bits, 13 of them are used to control the multiplexers, and 7 bits are used to select a memory plane to read or write. To run this processor array, a specialized controller is required to generate sequences of the 20 bit instruction patterns at the clock rate of the array.

It is difficult to convince people that this type of simple computing device is even useful, not to mention the possibility to compete against the modern powerful processors. Dr. Holsztynski made the following interesting observation to illustrate the power of SIMD device. In a conventional

processor, the CPU must perform a host of different functions. Each function requires resources in both silicon area and in machine cycles. Only one instruction can be executed at any time, consequently the other logic devices are idle. A processor with 100 instructions shows an instruction efficiency of only 1%. The GAPP processor, on the other hand, has an ALU which is active always and thus shows the highest silicon efficiency. The problem is how to program the GAPP array so that the ALU will perform useful work all the time.

2. GAPP SIMULATOR

As the GAPP array requires a high speed, programmable controller to supply instruction streams, not much one can do at the beginning of the GAPP project. To demonstrate realistically the usefulness of GAPP array to solve practical problems, a good simulator would be of great help. Here an old image processor made by DeAnza/Gould was available to us for this simulation. Because GAPP array consists of a large number of processors connected as a planar matrix, this structure maps very well to an image processor which stores and processes two dimensional images. The DeAnza/Gould IP5500 has one megabytes of memory, which can be thought of as 32 planes of 512x512 bits stacked together. Physically these bitplanes are grouped into 4 channels, with 8 bitplanes to a channel. Among the 32 bitplanes, we have to use the top 12 planes to simulator the GAPP registers and ALU. Only 20 bitplanes are available to simulate GAPP memory planes. For most of the simulation work, only the lower 16 bitplanes are simulated using channels 0 and 1 in the image processor.

The GAPP chip has four registers: CM, EW, NS, and C. The CM register handles communication in the north-south direction. The EW and NS registers are used for nearest neighbor data transfer, and the C register is used to store carry or borrow from the ALU. The ALU receives inputs from the EW, NS, and C registers, and generates the sum, carry, and borrow as the results of the three bit addition/subtraction. The logic function of this ALU is best described by the following truth table in Figure 20.

This set of input/output relationship can be best simulated in the image processor by table look-up technique. This truth table is used to generate an ITT (Intensity Transformation Table) in the image processor. When the image memory simulating the EW, NS, and C registers are read through this ITT, the plus, carry, and borrow bits are automatically generated and then stored back to the appropriate memory planes. Thus in a single TV frame time, we can produce the ALU results of all 512x512 simulated GAPP processors. In the next TV frame, the ALU results as well as other data can be routed back to the proper destination registers or memory through another table look-up operation.

Although GAPP is basically a one bit full adder/subtractor, by fetching and storing data in consecutive memory and use the carry or borrow stored in the C register, it is very straightforward to implement multiple bit integer arithmetics. A few examples will be show later. The more interesting property of the ALU is that by setting or clearing one or more registers among EW, NS, and C, all the binary logic function can be performed by this ALU, making it is truly general purpose computing device for large arrays of digital information. The conditions to perform logic operations are shown in Figure 21.

3. GAPP SIMULATOR IN FORTH

The DeAnza/Gould image processor IP5500 is controlled by a LSI-11 microprocessor which runs a very early version of LSI-11 polyForth from Forth, Inc. The image processing software system was described in some detail in my 'Forth Notebook,' pp. 78-113. Since the GAPP simulator uses only the image transfer operation through intensity transformation table, very little knowledge about the image processor is required to understand the GAPP simulator.

The source code and shadow comments are shown in Listing 12. Screens 1 through 10 are the source code of the simulator proper. Screens 11 to 15 are examples of elementary GAPP functions, such as multiple bit addition, subtraction, absolute values, and image dilations and erosions. Screen 21 to 25 show the game of life implemented in this GAPP simulator. As most source code have fairly detailed comments in the corresponding shadow screens, we shall only discussed some global features of this simulator.

The syntax of a GAPP instruction is as follows:

```
dest1: dest2: ... =src1   dest3: ... =src2   ...   _$_
```

One register can serve as the source for many registers. A destination register can only receive data from one source. Source registers are prefixed with an equal sign, and destination registers are appended with a colon. In any one GAPP instruction, only one memory plane can be referred to for either reading or writing, but not both. Memory plane used for source is coded as n =P, and for destination n P: . The bitplane number n must be put on the stack before the memory code.

_$_ terminates a GAPP instruction. In a real GAPP machine, all the code before this terminator up to the last _$_ code are executed in a single cycle. In this simulator, the destination code like =EW actually performs the date transfer, because the image processor cannot process input from many different registers at the same TV frame.

In real GAPP, the 7 bit field specifying the GAPP memory plane is an integral part of the GAPP instruction. In the simulator, we take advantage of the Forth system to pass the address as a parameter on the data stack. This short-cut greatly simplifies the structure of the simulator, but is not quite realistic. Nevertheless, in a real GAPP system, the address generation has to be handled by rather complicated logic circuits, which cannot be simulated easily.

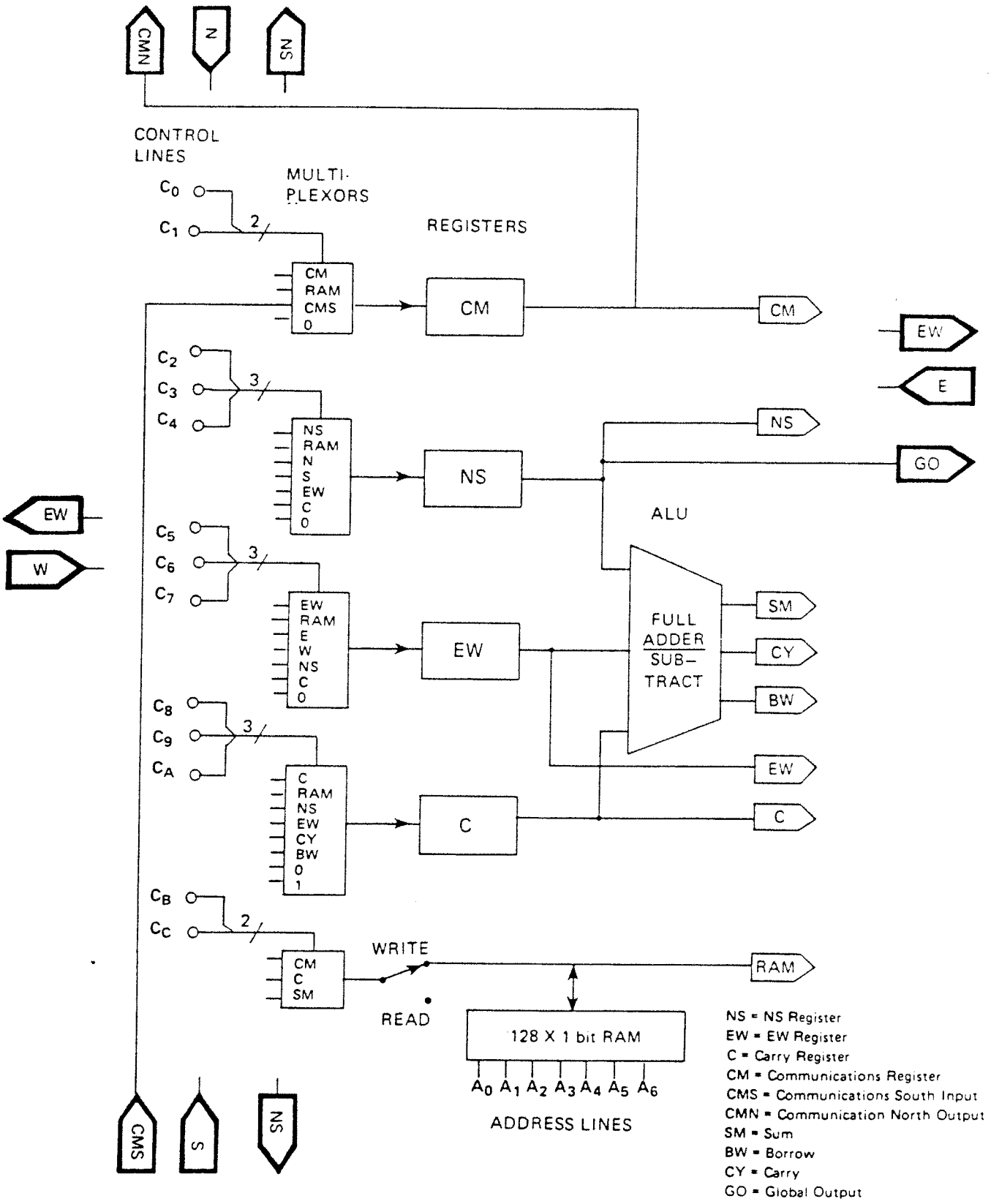


Figure 19. Schematic diagram of a GAPP processor unit

Figure 20. Truth Table of GAPP ALU

Input			Output		
NS	EW	C	plus	carry	borrow
0	0	0	0	0	0
0	1	0	1	0	1
1	0	0	1	0	0
1	1	0	0	1	0
0	0	1	1	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	1	1

Figure 21. Logic Operations of GAPP ALU

Logical Operation	Description	Condition
INV	SM=/NS	EW=0, C=1
	SM=/EW	NS=0, C=1
	SM=/C	NS=0, EW=1
AND	CY=NS*EW	C=0
	CY=EW*C	NS=0
	CY=NS*C	EW=0
	BW=/NS*EW	C=0
OR	CY=NS+EW	C=1
	BW=/NS+EW	C=1
	BW=EW+C	NS=0
XOR	SM=NSxC	EW=0
	SM=NSxEW	C=0
	SM=EWxC	NS=0
XNOR	SM=/(NSxEW)	C=1

Note: SM: plus, CY: carry, BW: borrow, /: negate, x: exclusive or

```
      0
 0 GAPP SIMULATOR USING IP5000 IMAGE PROCESSOR
 1 A GAPP chip has 72 processors arranged in a 6 by 12 matrix.
 2 Many GAPP chips can be connected to form a large 2D matrix,
 3 suitable for image processing and many other types of
 4 applications.
 5 The IP5000 image processor from DeAnza/Gould can simulate a
 6 GAPP array of 512x512 processors.  However, the depth of memory
 7 in IP5000 is only 32 bits deep, instead of 128 bits in GAPP.
 8 The GAPP ALU function is simulated not by arithmetic unit in
 9 IP5000, but by the look-up tables.  This is a unique feature
10 of this simulator.
11 To complete a GAPP cycle, IP5000 needs two TV frames.  One
12 frame to make ALU look-up, and the second frame to copy results
13 back to memory and registers.
14
15
```

```
      1
 0 ( DEANZA REGISTERS, 30JAN85CHT )
 1 : THRU 1+ SWAP DO I LOAD LOOP ;
 2 OCTAL
 3 117450 CONSTANT TSTOP     117452 CONSTANT OPOP
 4 117440 CONSTANT LOMASK    117442 CONSTANT HIMASK
 5 117426 CONSTANT CSR       117600 CONSTANT CBCR
 6 120000 CONSTANT IMAGE     104000 CONSTANT ITT0
 7 117424 CONSTANT MEMPORT
 8 117446 CONSTANT INPDP
 9 DECIMAL
10 A-ON
11 2 10 THRU ( GAPP SIMULATOR)
12 20 25 THRU ( GAME OF LIFE)
13 CONTEXT GOLDEN 20 MOVE  HERE H 2+ !
14
15
```

```
      2
 0 ( GAPP SIMULATION, 29JAN86CHT )
 1 OCTAL
 2 : ITT-INIT ( INITIALIZE LOOKUP TABLES FOR UNITY TRANSFORM )
 3    SETUP   377 MEMPORT !   A-ON   0 FRAMES
 4    ITT0 2000   OVER + SWAP DO
 5       I ( 377 AND )   I C!   LOOP ;
 6 : BITPLANE ( CH# MASK - , DISPLAY A CHANNEL THROUGH THE MASK )
 7    SWAP 400 * ITT0 +   400 OVER + SWAP DO
 8       DUP I AND   377 AND   IF 377 ELSE 0 THEN   I   C!
 9    LOOP DROP ;
10 DECIMAL
11
12
13
14
15
```

```
     50
GAPP SIMULATOR USING IP5000 IMAGE PROCESSOR
For detailed description of IP5000 Image Processor, one has to
go back to the reference manuals from DeAnza/Gould:
   IP5000 Programmer's Manual

For the image processing software, I had published a Forth
image processing system program in:
   Forth Notebook, Offete Enterprises, pp. 79-135, 1987

For GAPP chip and array, refer to manuals and spec sheets
from NCR.
```

```
     51
( DEANZA REGISTERS, 30JAN85CHT )                04:58 03Oct87cht
TSTOP     Test ALU operation
OPOP      Main ALU operation
LOMASK HIMASK   Two 16 bit masks to enable writing to bitplanes.
CSR       Control and status register of IP5000.
CBCR      Control and base coordinate registers
IMAGE     Image data window address
ITT0      Base address of Channel 0 image data look-up table
MEMPORT   Memory port control register
INPDP     Input data path register
A-ON      Turn on alphanumeric overlay
```

```
     52
( GAPP SIMULATION, 29JAN86CHT )                 05:07 03Oct87cht

ITT-INIT   Write straight diagonal line into the look-up table
           to initialize the ALU to noop operation.


BITPLANE   Write a mask function to the look-up table so that
           one bitplane is displayed on the monitor.
```

Listing 12. GAPP simulator using IP5500 image processor

```
      3
 0 ( COPY BLOCK TO CHANNEL 3 FOR DISPLAY, 29JAN86CHT
 1 OCTAL
 2 CODE LINE-MOVE ( ADDR # - , COPY # CELLS TO IMAGE MEMORY )
 3    2 S )+ MOV   1 S )+ MOV
 4    BEGIN IMAGE 1 )+ MOV   2 SOB   NEXT
 5 : WX-CBCR 101000 CSR ! 100000 CBCR ! 66377 CBCR 6 + !
 6    -1 CBCR 4 + !   4777 CBCR 2+ ! ;
 7 : WY-CBCR 101000 CSR ! 142000 CBCR ! 66377 CBCR 6 + !
 8    -1 CBCR 4 + !   0 CBCR 2+ ! ;
 9 DECIMAL
10 : BLOCK-WRITE ( BLOCK# LIM START - )
11    DO DUP BLOCK 512 LINE-MOVE LOOP DROP ;
12 : Y-SHOW ( BLOCK# - )   239   WY-CBCR 256 0 BLOCK-WRITE ;
13 : X-SHOW ( BLOCK# - )   239   WX-CBCR 256 0 BLOCK-WRITE ;
14
15
```

```
      53
( COPY BLOCK TO CHANNEL 3 FOR DISPLAY, 29JAN86CH05:12 030ct87cht
LINE-MOVE   Copy a range of memory into the IP5000 image memor.
            for displaying.

WX-CBCR     Initialize the CBCR registers to write data in
            horizontal rows.
WY-CBCR     Initialize CBCR registers to write data in columns.


BLOCK-WRITE Copy 1024 bytes from a block to fill the displa.
            image memory.
Y-SHOW      Copy data in block 239 to display columnwise.
X-SHOW      Copy data in block 239 to display rowwise.
```

```
      4
 0 ( PLUS, CARRY, BORROW OF GAPP, 29JAN86CHT )
 1 HEX
 2 CREATE ARU ( ARITHMETIC UNIT LUT )   100 , 205 , 205 , 70e ,
 3 : C-ARU ( NIBBLE - BYTE , COMPUTE THROUGH ARU )
 4    0F AND DUP 2/   ARU + C@ 2* 2* 2* 2*   + ;
 5 OCTAL
 6 : >ITT ( WRITE ITT2 TO PERFORM ARU FUNCTION )
 7    ITT0 1000 +   400 OVER + SWAP DO   I C-ARU I C!   LOOP ;
 8 : CPYM ( CH# LO-MASK HI-MASK - , COMPUTE AND STORE TO CH3 )
 9    377 MEMPORT ! 52 TSTOP   HIMASK ! LOMASK !   INPDP !
10    FMOPER ;
11 DECIMAL
12 EXIT
13 >ITT
14 2 0 -256 CPYM
15
```

```
      54
( PLUS, CARRY, BORROW OF GAPP, 29JAN86CHT )     10:42 04Oct87cht
ARU         Look-up table of the GAPP ALU functions.
C-ARU       Compute the ALU look-up table entry.  The assignments
            of bits in channel 2 is: x, x, x, x, C, EW, NS, and CM
            Bits in Channel 3 from LSB up are:
            x, borrow, carry, plus, C', EW', NS', CM'.
            C-ALU takes bit pattern in C, EW, and NS and deposits
            ALU results in borrow, carry, and plus bitplanes,
            while preserves C, EW, NS, CM in C', EW', NS', CM'.
>ITT        Write the ALU function into Channel 3 look-up table.
            An ALU cycle copies Channel 3 memory back to itself
            through the look-up table.
CPYM        A generalized image memory copying operation.  On the
            stack is the source Channel number and the two masks
            which selects any of the 32 bitplanes to write into.
```

```
      5
 0 ( DISPLAY IMAGE DATA, 30JAN86CHT )
 1 OCTAL
 2 CODE MOVE-LINE 2 S )+ MOV   1 S )+ MOV
 3    BEGIN 1 )+ IMAGE MOV   2 SOB NEXT
 4 : DX-CBCR ( CH# Y )
 5    WX-CBCR   SWAP 22000 * 377 + CBCR 6 + ! ( CHANNEL SELECT )
 6    777 AND 4777 SWAP -   CBCR 2+ ! ( LINE SELECT ) ;
 7 : IDUMP ( CH# LINE # )
 8    >R   DX-CBCR   PAD R>  OVER OVER MOVE-LINE   DUMP ;
 9 DECIMAL
10
11
12
13
14
15
```

```
      55
( DISPLAY IMAGE DATA, 30JAN86CHT )              10:33 04Oct87cht

MOVE-LINE ( source bytes -- )   Given a source address and t.
        count, move that many bytes to the image memory.
        This allows initializing image bitplanes for debugging.

DX-CBCR  ( channel row -- )  Initialize the CBCR registers for
        writing into the image memory.  On the stack is the
        channel to be written and the starting row in that
        channel of image.
IDUMP    ( channel row bytes -- )
        Write a number of bytes from the PAD buffer to a
        channel at the row specified.  The contents of PAD
        buffer is also dumped to console for verification.
```

Listing 12. GAPP simulator using IP5500 image processor (cont'd)

```
      6
 0 ( COPY INTO GAPP REGISTERS, 30JAN86CHT )
 1 OCTAL
 2 : COMPUTE ( COPY CH2 TO CH3 WITH PLUS, CARRY AND BORROW )
 3    >ITT   2 0 177400 CPYM   ;
 4 : MASK ( N - MASK, CHANGE BIT# TO BIT MASK )
 5    1 SWAP   ?DUP IF   0 DO 2* LOOP  THEN ;
 6 : =IN ( N -, 0<=N<=31 SELECTS INPUT BITPLANE )
 7    10 /MOD DUP ( CH# ) INPDP ! ( INPUT SELECTOR )
 8    SWAP ; CH# BIT# ;   MASK BITPLANE ( WRITE PROPER ITT )
 9    52 TSTOP ! ( NOP )   377 MEMPORT ! ( TURN ON ITT )
10    FMOPER ;
11 DECIMAL
12 : DISPLAY ( N )   31 AND 8 /MOD   SWAP MASK BITPLANE ;
13
14
15
```

```
      7
 0 ( NORTH, SOUTH, EAST AND WEST, 30JAN86CHT )
 1 OCTAL
 2 117474 CONSTANT XSCRZM   117476 CONSTANT YSCRZM
 3 : >NORTH      0 XSCRZM !     0 YSCRZM ! ;
 4 : >SOUTH      0 XSCRZM !   776 YSCRZM ! ;
 5 : >WEST     777 XSCRZM !   777 YSCRZM ! ;
 6 : >EAST       1 XSCRZM     777 YSCRZM ! ;
 7 : >CENTER     0 XSCRZM !   777 YSCRZM ! ;
 8 DECIMAL
 9
10
11
12
13
14
15
```

```
      8
 0 ( OUTPUT FILTER, 31JAN86CHT )
 1 VARIABLE HFILTER   VARIABLE LFILTER
 2 : ZFILTER   0 HFILTER !   0 LFILTER ! ;
 3 : P: ( N )    MASK   ZFILTER LFILTER ! ;
 4 : _$_    COMPUTE   ZFILTER    FRAME @ 1+ FRAMES !
 5 : CM:    HFILTER @ 1 OR   HFILTER ! ;
 6 : NS:    HFILTER @ 2 OR   HFILTER ! ;
 7 : EW:    HFILTER @ 4 OR   HFILTER ! ;
 8 : C:     HFILTER @ 8 OR   HFILTER ! ;
 9
10
11
12
13
14
15
```

```
      5a
( COPY INTO GAPP REGISTERS, 30JAN86CHT )        10:46 04Oct87chr

COMPUTE   From source registers in channel 2, compute ALU
          results and deposit them in channel 3.
MASK      ( bit# -- mask )    Set a bit in a 16 bit mask word.

=IN       ( n -- )   Select one of the 32 bit planes in the
          memory and display the binary image in the image
          channel the bitplane belong.


DISPLAY   Display one bitplane in the image memory.
```

```
      57
( NORTH, SOUTH, EAST AND WEST, 30JAN86CHT )     10:57 04Oct87chr

XSCRZM   X scroll and zoom register.
YSCRZM   Y scroll and zoom register.
>NORTH   Scroll to obtain data from the north neighbor.
>SOUTH   Scroll to south.
>WEST    Scroll to west.
>EAST    Scroll to east.
>CENTER  Restore image plane to center.
```

```
      58
( OUTPUT FILTER, 31JAN86CHT )                   11:03 04Oct87chr
HFILTER   Write enable mask for channels 2 and 3.
LFILTER   Write enable mask for channels 0 and 1.
ZFILTER   Reset both masks to disable writing to image memor..
P:        Select one bitplane in GAPP memory (channels 0 and 1
          as the destination of a GAPP memory operation.
_$_       Perform one GAPP cycle.  Increment cycle count in
          FRAME and display the cycle count on image overlay.
CM:       Select CM register as the destination.
NS:       Select NS regsiter as the destination.
EW:       Select EW register as the destination.
C:        Select C register as the destination.
```

Listing 12. GAPP simulator using IP5500 image processor (cont'd)

```
 9
 0 ( SOURCE TO TARGET, 31JAN86CHT )
 1 : SOURCE TARGET ( CH# BIT# )
 2   OVER >R   BITPLANE   R> LFILTER @ MFILTER @  00:48 030ct87cht
 3   ZFILTER   CPYM    ;
 4 : =CM   >CENTER 3 1 SOURCE>TARGET ;
 5 : =NS   >CENTER 3 2 SOURCE>TARGET ;
 6 : =EW   >CENTER 3 4 SOURCE>TARGET ;
 7 : =C    >CENTER 3 8 SOURCE>TARGET ;
 8 : =N    >NORTH 3 2 SOURCE>TARGET ;
 9 : =S    >SOUTH 3 2 SOURCE>TARGET ;
10 : =E    >EAST  3 4 SOURCE>TARGET ;
11 : =W    >WEST  3 4 SOURCE>TARGET ;
12 : =CARRY   >CENTER 3 32 SOURCE>TARGET ;
13 : =BORROW  >CENTER 3 64 SOURCE>TARGET ;
14 : =DPLUS   >CENTER 3 16 SOURCE>TARGET ;
15
```

```
10
 0 ( RAM OR CONSTANT TO TARGET, 31JAN86CHT)
 1 OCTAL
 2 : SITT ( FILL CH3 ITT WITH 255 )
 3   ITT0 1400 +   400 OVER + SWAP DO   377 I C!   LOOP ;
 4 : ZITT ( ZERO CH3 ITT )
 5   ITT0 1400 +   400 OVER + SWAP DO   0 I C!   LOOP ;
 6 DECIMAL
 7 : =P ( N )   15 AND 8 /MOD SWAP   MASK SOURCE>TARGET ;
 8 : =1   SITT   3 LFILTER @ MFILTER @ ZFILTER   CPYM   ;
 9 : =0   ZITT   3 LFILTER @ MFILTER @ ZFILTER   CPYM   ;
10 : CARRY   C:  =CARRY ;
11 : BORROW  C:  =BORROW ;
12 : DPLUS   P:  =DPLUS ;
13
14
15
```

```
11
 0 ( GAPP EXAMPLE, DILATION, 19MAR86CHT )
 1 ( INPUT IN C, OUTPUT IN C )
 2 : SDIL      NS: EW: =C              _$_
 3               NS: =S   C: =1        _$_
 4               CARRY                 _$_   ;
 5 : EDIL      NS: EW: =C              _$_
 6               EW: =E   C: =1        _$_
 7               CARRY                 _$_   ;
 8 : NDIL      NS: EW: =C              _$_
 9               NS: =N   C: =1        _$_
10               CARRY                 _$_   ;
11 : WDIL      NS: EW: =C              _$_
12               EW: =W   C: =1        _$_
13               CARRY                 _$_   ;
14 : #DIL   EDIL WDIL   NDIL SDIL   ;
15
```

```
 59
( SOURCE TO TARGET, 31JAN86CHT )                    11:10 040ct87cht
SOURCE>TARGET ( channel bit -- )
        Select the source of data to different registers.

=CM     Select CM as source.
=NS     Select NS as source.
=EW     Select EW as source.
=C      Select C as source.
=N      Select north neighbor as source.
=S      Select south as source.
=E      Select east as source.
=W      Select west as source.
=CARRY  Select carry as source.
=BORROW Select borrow as source.
=DPLUS  Select plus as source.
```

```
 60
( RAM OR CONSTANTS TO TARGET, 31JAN86CHT)           11:16 040ct87cht

SITT      Fill channel 3 look-up table with 255 to set all bits.

ZITT      Fill channel 3 look-up table with 0 to clear it.

=P        Select one memory plane as source3.
=1        Select 1 as source.
=0        Select 0 as source.
CARRY     Copy carry from ALU to C register.
BORROW    Copy borrow from ALU to C register.
DPLUS     Copy plus from ALU to a memory plane.
```

```
 61
( GAPP EXAMPLE, DILATION, 19MAR86CHT )              15:52 040ct87cht
Dilation is to OR a pixel with a neighboring pixel and store
        the results back.  C is used both as source and result.
SDIL    Copy C into both EW and NS registers.
        Get south neighbor to NS and set C.
        Carry contains result of ORing, which is moved back to C
EDIL    Dilation with east neighbor.
NDIL    Dilation with north neighbor.
WDIL    Dilation with west neighbor.
#DIL    Dilation with four nearest neighbors.
```

Listing 12. GAPP simulator using IP5500 image processor (cont'd)

```
     12
 0 ( GAPP EXAMPLE, EROSION, 19MAR86CHT )
 1 ( INPUT IN C, OUTPUT IN C )
 2 : SERO      NS: EW: =C                _$_
 3               NS: =S   C: =0          _$_
 4               CARRY                   _$_   ;
 5 : EERO      NS: EW: =C                _$_
 6               EW: =E   C: =0          _$_
 7               CARRY                   _$_   ;
 8 : NERO      NS: EW: =C                _$_
 9               NS: =N   C: =0          _$_
10               CARRY                   _$_   ;
11 : WERO      NS: EW: =C                _$_
12               EW: =W   C: =0          _$_
13               CARRY                   _$_   ;
14 : #ERO   EERO WERO   NERO SERO   ;
15 : #EROS   0 DO #ERO LOOP ;
```

```
     13
 0 ( GAPP EXAMPLE, VERTICAL SUBTRACTION, 04FEB86CHT )
 1 : V-SUB   INPUT1 INPUT2 OUTPUT BITS )
 2             C: =0                   _$_
 3    ( BITS ) 0 DO
 4    ROT DUP   NS: =P                 _$_   1+
 5    ROT DUP   EW: =P                 _$_   1+
 6    ROT DUP   DPLUS   BORROW         _$_   1+
 7    LOOP
 8             P: =C
 9    DROP DROP   ;
10
11
12
13
14
15
```

```
     14
 0 ( GAPP EXAMPLE, VERTICAL ADDITION, 04FEB86CHT )
 1 : V-ADD ( INPUT1 INPUT2 OUTPUT BITS )
 2             C: =0                   _$_
 3    ( BITS ) 0 DO
 4    ROT DUP   NS: =P                 _$_   1+
 5    ROT DUP   EW: =P                 _$_   1+
 6    ROT DUP   DPLUS   CARRY          _$_   1+
 7    LOOP
 8             P: =C                   _$_
 9    DROP DROP   ;
10
11
12
13
14
15
```

```
     62
( GAPP EXAMPLE, EROSION,  19MAR86CHT )          16:12 04Oct87---
Erosion is very similar to dilation.  The only difference is
        the C register is cleared instead of being set.
SERO    Copy C to both EW and NS registers.
        Copy south neighbor to NS and clear C.
        The results from ANDing is move back to C.
EERO    Erosion in east direction.
NERO    Erosion in north direction.
WERO    Erosion in west direction.
#ERO    Square erosion.
#EROS   Repeat square erosions n times.
```

```
     63
( GAPP EXAMPLE, VERTICAL SUBTRACTION, 04FEB86CHT )
V-SUB ( input1 input2 output bits -- )
        Subtract input 2 from input 1 and store results to
        output.  Bits are number of bits to be processed.
        Two's compliment subtraction.  Sign bit is stored in
        output+bits+1.
  C: =0                         Initialize C.
  ( BITS ) 0 DO                 Repeat n times.
    ROT DUP   NS: =P              Get input1.
    ROT DUP   EW: =P              Get input2.
    ROT DUP   DPLUS   BORROW      Store result to output and
                                     save carry in C.
  LOOP                          Loop back.
  P: =C                         Copy sign to output+bits+1
  DROP DROP                     Clear stack.
```

```
     64
( GAPP EXAMPLE, VERTICAL ADDITION, 04FEB86CHT ) 16:27 04Oct87CHT
V-ADD ( INPUT1 INPUT2 OUTPUT BITS )
        Like V-SUB except taking carry back to C instead of
        borrow.
```

Listing 12. GAPP simulator using IP5500 image processor (cont'd)

```
 15
 0 ( ABSOLUTE VALUES, 10FEB86CHT )
 1 : ABSOLUTE ( INPUT OUTPUT SIGN )
 2    SWAP   C: =P   EW: =0   _$_
 3    0 DO
 4    SWAP DUP   NS: =P   _$_   1+
 5    SWAP DUP   DPLUS    _$_   1+
 6    LOOP   DROP DROP   ;
 7
 8
 9
10
11
12
13
14
15
```

```
 65
( ABSOLUTE VALUES, 10FEB86CHT )                16:30 04Oct87cht
ABSOLUTE ( INPUT OUTPUT SIGN BITS -- )
   SWAP   C: =P   EW: =0        Get sign bit to C register.
   0 DO                         Repeat n times.
      SWAP DUP   NS: =P           Get a bit from input.
      SWAP DUP   DPLUS            Store absolute to output.
   LOOP   DROP DROP             Loop back.
   DROP DROP                    Clear up.
```

```
 21
 0 ( SETUP SAMPLE IMAGES, 05FEB86CHT )
 1 : GAPP-INIT
 2    239 X-SHOW   3 255 0 CPYMEM
 3    239 Y-SHOW   3 -256 0 CPYMEM
 4    NS: 4 =P  _$_
 5    EW: 12 =P  _$_
 6    C: =0 _$_
 7    3 [ HEX ] 70 0 CPYMEM [ DECIMAL ]
 8    C:  13 =P _$_
 9    0 P: =DPLUS _$_
10    1 P: =CARRY _$_
11    2 P: =BORROW _$_
12    ;
13
14
15
```

```
 71
( SETUP SAMPLE IMAGES, 05FEB86CHT )            16:21 04Oct87cht
GAPP-INIT       Initialize memory for game-of-life demonstratio
       Initialize channel 0 to a horizontal ramp.
       Initialize channel 1 to a vertical ramp.
       AND bitplanes 4 and 12 to create a checkerboard pattern.


       Save the checkerboard in channel 1.
       Save C in channel 1, bitplane 13.
       Save plus in bitplane 0.
       Save carry in bitplane 1.
       Save borrow in bitplane 2.
```

```
 22
 0 ( HANDY DEBUGGING TOOL, 28FEB86CHT )
 1 : DEPTH  S0 @  'S - 2/ 2 - ;
 2 : .S   CR  DEPTH IF
 3    'S S0 @ 4 -   DO I @ . -2 +LOOP
 4    ELSE ." Empty "   THEN ;
 5
 6 : ?DISPLAY   CONSTANT   DOES> @ DISPLAY ;
 7 16 ?DISPLAY ?CM   17 ?DISPLAY ?NS   18 ?DISPLAY ?EW
 8 19 ?DISPLAY ?C    24 ?DISPLAY ?CM   25 ?DISPLAY ?NS
 9 26 ?DISPLAY ?EW   27 ?DISPLAY ?C    28 ?DISPLAY ?PLUS
10 29 ?DISPLAY ?CARRY   30 ?DISPLAY ?BORROW
11 : ?P   DISPLAY ;
12
13
14
15
```

```
 72
( HANDY DEBUGGING TOOL, 28FEB86CHT )           17:36 04Oct87cht
DEPTH   Old polyForth had no depth.
.S      Stack displaying must also be defined, bringing
        polyForth up to date.


?DISPLAY   Define words which will show a GAPP register.
?CM   ?NS   ?EW
?C    ?CM   ?NS
?EW   ?C    ?PLUS
?CARRY      ?BORROW
?P         Display a memory plane.
```

Listing 12. GAPP simulator using IP5500 image processor (cont'd)

```
      23
 0 ( GAME OF LIFE, 03MAR86CHT )
 1 : 1STEP ( INPUT 0, SUM OF E W 0 IN C AND EW/NS )
 2    0   NS:  EW:  =P    _$_   ( COPY INPUT INTO EW/NS )
 3             EW:  =E    _$_   ( GET EAST )
 4         C:  =EW  EW:  =NS   _$_   ( RESTORE EW FROM NS )
 5             EW:  =W    _$_   ( GET WEST )
 6    2   P:   EW:  NS:  =DPLUS   ( SAVE SUM IN PLANE2 EW/NS )
 7             CARRY      _$_
 8    ;
 9 : 2STEP ( SUM IN EW/NS AND PLANE2, CARRY IN C )
10    3   P:  =C   NS:  =N    _$_   ( GET NORTH )
11        C:  =NS  NS:  =EW   _$_   ( RESTORE NS FROM EW )
12                 NS:  =S    _$_   ( GET WEST LSB )
13    2   DPLUS    CARRY      _$_   ( CAL LSB OF N S AND O
14    ;
15
```

```
      73
 ( GAME OF LIFE, 03MAR86CHT )                    17:29 04Oct87cht
This is my game of life, which is slightly different from the
standard Conway's.  The rules are:
1.  A new life is born to a cell if it is empty and has
    three life cells around it.
2.  A life cell will live if it has two life neighbors.
3.  In all other cases, the cell will be empty in the next
    generation.
Life is represented by a set bit in a bitplane.  0 represents
an empty cell.  The program sums the number of life cells in a
3 by 3 array.  If the sum is exactly 3, the central cell will
have life in the next generation.  Otherwise, the central cell
will be empty.
```

```
      24
 0 ( GAME OF LIFE, 03MAR86CHT )
 1 : 3STEP ( LSB IN PLANE2, CARRY IN C )
 2    3   NS:  EW:   =P    _$_   ( GET MSB OF E+W+0 )
 3             NS:   =N    _$_   ( MSB OF NORTH )
 4    3   P:  EW:  =DPLUS    ( SAVE SUM TO EW )
 5        CARRY   NS:  =EW    _$_    ( RESTORE NS FROM EW )
 6    1   P:  =C   NS:  =S   C:  =C   _$_   ( GET MSB SOUTH
 7    3   DPLUS   CARRY    _$_    ( SUM MSB N+S+0 )
 8    ;
 9
10
11
12
13
14
15
```

```
      74
 ( GAME OF LIFE, 03MAR86CHT )                    17:29
Initial life pattern is stored in bitplane 0.
Results of the next generation is copied back to bitplane 0.
```

```
      25
 0 ( GAME OF LIFE, 04MAR86CHT )
 1 : 4STEP
 2    2  EW:  =P   NS:  =C   C:  =0   _$_
 3    3  EW:  =P   NS:  =0   BORROW   _$_
 4    1           NS:  =P   CARRY    _$_
 5       EW:  =C             C:  =0   _$_
 6                           BORROW   _$_
 7    0   P:  =C                      _$_
 8    ;
 9 : LIFE   1STEP 2STEP 3STEP 4STEP 0=?P ;
10 : LIFES   0 DO LIFE LOOP ;
11
12
13
14
15
```

```
      75
 ( GAME OF LIFE, 04MAR86CHT )                    17:30
```

LIFE Compute one generation of life.
LIFES Repeat the game of life for n generations.

Listing 12. GAPP simulator using IP5500 image processor (cont'd)

INTERPRET

XII. GAPP ASSEMBLER

1. INTRODUCTION

GAPP OPCODES AND MNEMONICS

GAPP is a SIMD (Single Instruction Multiple Datapath) parallel processing device. Many GAPP devices can be connected to form a two-dimenional array with east-west and north-south nearest neighbor connections. All GAPP devices in an array executes the same instruction in one machine cycle. A GAPP instruction consists of 13 bits of opcode and 7 bits of RAM address, which can be represented in the following format:

GAPP Opcode

12	11	10	9	8	7	6	5	4	3	2	1	0
RAM			C-REG			EW			NS		CM	

RAM Address

6	5	4	3	2	1	0
address						

The assembler will thus generate two 16 bit words for each macro instruction. They are stored in memory as one 32 bit word. The most significant 16 bits contain the RAM address, and the least significant 16 bits contain the GAPP opcode.

THE F83 ENVIRONMENT

F83 is a language and operating system based on the computer language FORTH according to the most recent FORTH-83 Standard. It is extremely powerful as a meta-language because it contains all the necessary tools to build new languages according to specified syntax and grammar. However, if the syntax requirements are specified in terms of the reversed Polish or post-fixed style common to most FORTH system, the new language can be defined very conveniently while retaining all the utilities resident in the host FORTH operating system. This GAPP assembler is defined in this fashion to take advantage of the conciseness and

interactiveness of FORTH.

The GAPP assembler is a collection of FORTH words or instructions which are then be used to describe or define GAPP macro instructions in mnemonic form. These descriptions are the cource code of GAPP macro instructions stored in files. When the descriptions are processed by the FORTH interpreter, the mnemonics are translated into macro code which are stored temporarily at the end of the source file. A large number of data blocks at the end of the source file are dedicated to the microcode. These microcode blocks should not be sent to a printer.

A number of FORTH utilities words are also provided to let the user to examine and modify the macro instructions either in the file. User can interact closely with both the assembler and the microcode, making the programming activity high productive and shorten considerably the time to program and debug the GAPP code.

USING THE GAPP ASSEMBLER

The GAPP assembler and a collection of macro routines are defined in a file named GAPPASSM.BLK. To execute the assembler and down load all these macro routines to the macro generators, the user should type the following commands:

```
F83    GAPPASSM.BLK                (load F83 and open file)
OK                                 (load assembler and
                                   assemble macro routines)
BYE                                (exit assembler)
```

After typing 'OK', the computer will load the GAPP assembler and assemble all the macro routines defined in this file. While the assembler is translating the source code, a list of macro names and their assembler code will be displayed on the console screen. With appropriate printer commands or output redirection, this listing can be printed on a line printer or saved in a separate file for later reference.

2. THE GAPP ASSEMBLER

THE ASSEMBLER COMMANDS

The assembler commands are a collection of FORTH words which cause GAPP instructions to be translated and stored at the end of the file. They are used to control the process of assembly and manage other activities such as outputing assembled code to a line printer or to the console. These words are listed and their functions explained as follows:

INIT	Clear the temprary file space. Initiate the assembling process.
_$_	Terminate the GAPP instruction assembling process and store the GAPP instruction. Begin assembling the next GAPP instruction.
G: <name>	Mark the address of the current GAPP instruction, and assign it a label <name>. This label will be used to call the macro routine by the flow program compiler.
DOWNLOAD	Download all the GAPP instructions so far assembled to the GAPP array.
GDUMP	Dump a range of GAPP instructions on the console for inspection.
DUMP-DMC	Dump a range of the Program Memory in the GAPP array to inspect the microcode.

GAPP MNEMONICS AND OPCODES

GAPP mnemonics are of the general format:

dest1[:dest2[:dest3[:dest4]]]=src

where dest represent register or memory as destination of data and src is a register or memory providing data in a GAPP machine cycle.

dest	CM, NS, EW, C, P (for RAM memory).
src	CM, NS, EW, C, P, N, S, E, W, 0 and 1

Only those combinations which are valid in a GAPP processor array are defined. Invalid combinations will generate an error message and cause the assembling process to abort.

Besided these explicit GAPP opcode mnemonics, there are a set of implicit mnemonics which express the functions more clearly:

CARRY	Copy carry output to C register.
BORROW	Copy borrow output to C register.
PLUS	Copy the sum output to a RAM memory plane.
SOUTH	Copy CM or south neighbor to CM register.
-dest=0	Copy 0 to all registers except dest. Examples: -CM=0, -NS=0, -EW=0, -C=0

-dest=P	Copy RAM memory to all registers but dest. Examples: -CM=P, -NS=P, -EW=P, -C=P
4R=0	Copy 0 to all registers.
4R=P	Copy RAM memory to all registers.

GAPP ADDRESS OPCODE

The GAPP address mnemonics will assemble code into the 16 bit address field in a GAPP instruction, which is processed in parallel with the GAPP opcode. Within one GAPP instruction or between two consecutive _$_ commands, the order of address opcode and GAPP mnemonics is not important. GADR is the general address opcode command which can be used to put any 16 bit pattern into the address field:

GADR	Assemble the 16 bit number before GADR into the 16 bit GAPP address field in the current GAPP instruction. Only the lower 7 bits are taken by the GAPP array as RAM address. Other bits are ignored. However, unused bits can be used to carry information to control the GAPP array.

3. MACRO ROUTINE LIBRARY

An extensive collection of macro routines are coded and available in the macro routine library, as the result of our experimentation with the GAPP array. The source of this library is also contained in the file GAPPMAC.BLK. It is assembled when GAPPMAC.BLK is loaded with F83 and interpreted. User defined macro routines can be entered into this file and assembled with the library. In eventual applications, those macro routines not used by the program can be commented out so that they will not be assembled into the application program.

The source code of this assembler and some examples of the macro routine library are shown in Listing 13.

Source code of the assembler and the macro routines are stored in block files, which are divided into blocks of 1024 bytes. Each block of text can be entered, edited, and interpreted under the FORTH operating system. F83 provides a very simple and easy to use screen editor for these purposes. Blocking the source code into 1024 byte segments imposed on the programmer a discipline of modularization, which makes the macro routines easy to enter and to be debugged.

Excellent references are available on how the F83 functions and how to use the screen editor in it for programming.

```
      1
 0 \ GAPP Compiler for PC                                  11oct87cht \ GAPP Compiler for PC                    23:02 01Jan80cht
 1 2 7 THRU ( GAPP ASSEMBLER )                                        GAPP assembler and simple opcode
 2 8 11 THRU ( COMPOUND MNEMONICS )                                   Compound opcode involving multiple registers
 3 14 40 THRU ( GAPP MACROS )                                         GAPP macros and sample code
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
      2
 0 \ Pointers                                              06mav87cht \ Pointers                                23:10 01Jan80cht
 1 0 CONSTANT ORIGIN ( starting GAPP instruction pointer )            ORIGIN  Starting GAPP instruction pointer.
 2 140 CONSTANT STORAGE                                               STORAGE Starting block to store GAPP code.
 3 139 CONSTANT MACRO#                                                MACRO#  Block to store GAPP macro pointers.
 4 VARIABLE MACRO-POINTER                                             MACRO-POINTER   Pointer to the macro currently being assembled.
 5 VARIABLE POINTER ( current GAPP instruction count )                POINTER Current GAPP instruction pointer.
 6 CODE BYTES ( n -- low-byte#4 high-byte )                           BYTES   The byte pointer to the current GAPP instruction in the
 7    AX POP   BX BX XOR   AH BL MOV   BH AH MOV                              disk buffer.  Returns (n MOD 256)*4 the byte offset in a
 8    AX SHL   AX SHL   AX PUSH   BX PUSH   NEXT END-CODE                    block, and n/256 the block number.
 9 : ADDRESS ( -- buffer-adr , address of virtual memory )            ADDRESS ( -- buffer-adr , address of virtual memory )
10    POINTER @ BYTES   STORAGE +   BLOCK + ;                               Returns the address of the current GAPP instruction in
11                                                                          the disk buffer.
12
13
14
15
```

```
      3
 0 \ Utility                                               26mar87cht \ Utility                                 23:15 01Jan80cht
 1 : 2OR ( D D -- D )   ROT OR >R   OR R> ;                           2OR ( D D -- D )     OR two 32 bit numbers.
 2 : 2AND ( D D -- D )   ROT AND >R   AND R> ;                        2AND ( D D -- D )    AND two 32 bit numbers.
 3 : 2NOT ( D -- D )   -1 XOR   SWAP -1 XOR SWAP ;                    2NOT ( D -- D )      Compliment a 32 bit number.
 4 : 2SHIFT ( D N -- D )                                              2SHIFT ( D N -- D )
 5    ?DUP IF 0 DO D2* LOOP THEN  ;                                         Shift a 32 bit number n bits to the left, to align
 6                                                                          the LSB to a field in the 32 bit number.
 7 : G.   POINTER @ U.    ADDRESS 2@  UD. ;                           G.      Print the current GAPP code.
 8 : GDUMP ( pointer # -- )                                           GDUMP   ( pointer # -- )
 9    POINTER @ >R   SWAP POINTER !   0 DO                                   Like DUMP, but dump a range of GAPP code in the disk
10       CR POINTER @ 8 U.R   ADDRESS 2@ 20 UD.R                             storage.
11    1 POINTER +!   LOOP   R> POINTER ! ;
12
13
14
15
```

Listing 13. GAPP assembler on PC

```
      4                                                    104
 0 \ Fields                                 21apr86cht \ Fields                                  23:42 01Jan80
 1 : FIELD ( START WIDTH -- NEXT )                      FIELD   ( START WIDTH -- NEXT )
 2    CREATE   OVER .   +                                       Create a field in a 32 bit GAPP instruction, with START
 3    DOES> @   0 SWAP 2SHIFT                                   as the LSB position and WIDTH as the field width.
 4       ADDRESS DUP >R 2@ 2OR   R> 2! ;                        At run time, insert the pattern on stack into the field
 5 0 2 FIELD CM   3 FIELD NS   3 FIELD EW   3 FIELD CY          thus defined.
 6    2 FIELD RAM DROP                                  CM      Put a code into the CM multiplexer.
 7 16 7 FIELD gadr DROP                                 NS      Put a code into the NS multiplexer.
 8                                                      EW      Put a code into the EW multiplexer.
 9                                                      CY      Put a code into the CY multiplexer.
10                                                      RAM     Put a code into the RAM multiplexer.
11                                                      GADR    Put the top of stack into the GAPP RAM address field.
12
13
14
15

      5                                                    105
 0 \ Initiate                               15apr87cht \ Initiate                                23:49 01Jan80
 1 HEX
 2 : INIT   ORIGIN POINTER !   8000 0 ADDRESS 2!                INIT    Initialize the instruction pointer and macro pointer.
 3    0 MACRO-POINTER !   ;
 4 : $   CR POINTER @ 10 U.R   ADDRESS 2@ 12 UD.R                $       Terminate a GAPP instruction by advance the instruction
 5    1 POINTER +!   0.00   ADDRESS 2!   UPDATE ;                        pointer and also initialize the next instruction.
 6 : G:   >IN @ R   BLK @ >R                                     G:      Define a new GAPP macro routine.  This macro does not
 7    CR   MACRO-POINTER @ 3 U.R   3 SPACES                              assemble GAPP code at this moment.  When the macro is
 8    POINTER @   MACRO# BLOCK   MACRO-POINTER @ 2+ + ! UPDATE           later executed, the GAPP instructions compiled into
 9    1 MACRO-POINTER +!                                                 this macro will be assembled.
10    BL WORD COUNT TYPE   3 SPACES                                      The assembler also prints the name and the contents
11    R> BLK !   R> >IN !   MACRO-POINTER @ 1- CONSTANT   ;              of the macro routine on the console.
12 DECIMAL
13
14
15

      6                                                    106
 0 \ Elementary instructions                05jan87cht \ Elementary instructions                 23:53 01Jan80
 1 : CM:   CREATE .   DOES> @   CM :                    CM:     Define a word assembling code into CM field.
 2 : NS:   CREATE .   DOES> @   NS :                    NS:     Define a word assembling code into NS field.
 3 : EW:   CREATE .   DOES> @   EW :                    EW:     Define a word assembling code into EW field.
 4 : C:    CREATE .   DOES> @   CY :                    C:      Define a word assembling code into CY field.
 5 : RAM:  CREATE .   DOES> @   RAM :                   RAM:    Define a word assembling code into RAM field.
 6
 7
 8
 9
10
11
12
13
14
15
```

Listing 13. GAPP assembler on PC (cont'd)

```
      7
 0 \ Simple GAPP instructions
 1 1 CM: CM=P    2 CM: SOUTH    3 CM: CM=0
 2 1 NS: NS=P    2 NS: NS=N     3 NS: NS=S    4 NS: NS=EW
 3 5 NS: NS=C    6 NS: NS=0
 4 1 EW: EW=P    2 EW: EW=E     3 EW: EW=W    4 EW: EW=NS
 5 5 EW: EW=C    6 EW: EW=0
 6 1  C:  C=P    2  C:  C=NS    3  C:  C=EW   4  C: CARRY
 7 5  C: BORROW  6  C:  C=0     7  C:  C=1
 8 1 RAM: P=CM   2 RAM: P=C     3 RAM: PLUS
 9
10
11
12
13
14
15
```

```
      107
05Jan87cht \ Simple GAPP instructions                    23:59 01Jan8
CM=P    SOUTH    CM=0
        CM register instructions.
NS=P    NS=N     NS=S    NS=EW     NS=C    NS=0
        NS register instructions.
EW=P    EW=E     EW=N    EW=NS     EW=C    EW=0
        EW register instructions.
C=P     C=NS     C=EW    CARRY     BORROW  C=0     C=1
        CY register instructions.
P=CM    P=C      PLUS
        RAM memory instructions.
```

```
      8
 0 ( COMPOUND GAPP INSTRUCTIONS, 24NOV86CHT )
 1 : CM:NS=PLUS   CM=P  NS=P  PLUS ;
 2 : CM:EW=PLUS   CM=P  EW=P  PLUS ;
 3 : CM:C=PLUS    CM=P  C=P   PLUS ;
 4 : NS:EW=PLUS   NS=P  EW=P  PLUS ;
 5 : NS:C=PLUS    NS=P  C=P   PLUS ;
 6 : EW:C=PLUS    EW=P  C=P   PLUS ;
 7 : -CM=PLUS     EW=P  NS=P  C=P   PLUS ;
 8 : -NS=PLUS     CM=P  EW=P  C=P   PLUS ;
 9 : -C=PLUS      CM=P  NS=P  EW=P  PLUS ;
10 : CM=PLUS      CM=P  PLUS ;
11 : NS=PLUS      NS=P  PLUS ;
12 : EW=PLUS      EW=P  PLUS ;
13 : C=PLUS       C=P   PLUS ;
14 : 4R=PLUS      CM=P  NS=P  EW=P  C=P   PLUS ;
15
```

```
      108
( COMPOUND GAPP INSTRUCTIONS, 24NOV86CHT )      00:00 01Jan8
CM:NS=PLUS
CM:EW=PLUS
CM:C=PLUS
NS:EW=PLUS
NS:C=PLUS
EW:C=PLUS
-CM=PLUS
-NS=PLUS
-C=PLUS
CM=PLUS
NS=PLUS
EW=PLUS
C=PLUS
4R=PLUS
```

```
      9
 0 ( COMPOUND GAPP INSTRUCTIONS, 24NOV86CHT )
 1 : P:CM:NS=C    P=C   CM=P   NS=P ;
 2 : P:CM:EW=C    P=C   CM=P   EW=P ;
 3 : P:NS:EW=CM   P=CM  NS=P   EW=P ;
 4 : P:NS:EW=C    P=C   NS=P   EW=P ;
 5 : P:NS:C=CM    P=CM  NS=P   C=P  ;
 6 : P:EW:C=CM    P=CM  EW=P   C=P  ;
 7 : P:3R=CM      P=CM  NS=P   EW=P   C=P   ;
 8 : P:3R=C       P=C   CM=P   NS=P   EW=P  ;
 9 : P:CM=C       P=C   CM=P   ;
10 : P:NS=CM      P=CM  NS=P   ;
11 : P:NS=C       P=C   NS=P   ;
12 : P:EW=CM      P=CM  EW=P   ;
13 : P:EW=C       P=C   EW=P   ;
14 : P:C=CM       P=CM  C=P    ;
15
```

```
      109
( COMPOUND GAPP INSTRUCTIONS, 24NOV86CHT )      00:01 01Jan8
P:CM:NS=C
P:CM:EW=C
P:NS:EW=CM
P:NS:EW=C
P:NS:C=CM
P:EW:C=CM
P:3R=CM
P:3R=C
P:CM=C
P:NS=CM
P:NS=C
P:EW=CM
P:EW=C
P:C=CM
```

Listing 13. GAPP assembler on PC (cont'd)

```
      10
 0 ( COMPOUND GAPP INSTRUCTIONS. 24NOV86CHT )
 1 : CM:NS=P    CM=P   NS=P   ;
 2 : CM:EW=P    CM=P   EW=P   ;
 3 : CM:C=P     CM=P   C=P    ;
 4 : NS:EW=P    NS=P   EW=P   ;
 5 : NS:C=P     NS=P   C=P    ;
 6 : EW:C=P     EW=P   C=P    ;
 7 : NS:EW=C    NS=C   EW=C   ;
 8 : NS:C=EW    NS=EW  C=EW   ;
 9 : EW:C=NS    EW=NS  C=NS   ;
10 : CM:NS=0    CM=0   NS=0   ;
11 : CM:EW=0    CM=0   EW=0   ;
12 : CM:C=0     CM=0   C=0    ;
13 : NS:EW=0    NS=0   EW=0   ;
14 : NS:C=0     NS=0   C=0    ;
15 : EW:C=0     EW=0   C=0    ;
```

```
      11
 0 ( COMPOUND GAPP INSTRUCTIONS. 24NOV86CHT )
 1 : -CM=0      NS=0   EW=0   C=0    ;
 2 : -NS=0      CM=0   EW=0   C=0    ;
 3 : -EW=0      CM=0   NS=0   C=0    ;
 4 : -C=0       CM=0   NS=0   EW=0   ;
 5 : 4R=0       CM=0   NS=0   EW=0   C=0    ;
 6 : -CM=P      NS=P   EW=P   C=P    ;
 7 : -NS=P      CM=P   EW=P   C=P    ;
 8 : -EW=P      CM=P   NS=P   C=P    ;
 9 : -C=P       CM=P   NS=P   EW=P   ;
10 : 4R=P       CM=P   NS=P   EW=P   C=P    ;
11
12
13
14
15
```

```
      14
 0 \ Simple operations
 1 HEX 6: NOOOP   _$_ _$_ _$_ _$_ _$_ _$_ _$_ _$_ _$_ _$_
 2 6: W-SHIFT    EW=E   _$_
 3 6: E-SHIFT    EW=W   _$_
 4 6: N-SHIFT    NS=S   _$_
 5 6: S-SHIFT    NS=N   _$_
 6 6: EXCHANGE   DUP 6ADR C=EW   EW=P   _$_   6ADR P=C   _$_
 7 6: EW-RESET   C=0    _$_   EW=C   _$_
 8 6: EW-SET     C=1    _$_   EW=C   _$_
 9 6: NS-RESET   NS=0   _$_
10 6: NS-SET     C=1    _$_   NS=C   _$_
11 6: C-GET      6ADR C=P     $_
12 6: C-PUT      6ADR P=C     _$_
13 DECIMAL
14
15
```

```
     110
( COMPOUND GAPP INSTRUCTIONS. 24NOV86CHT )      00:03 01Jan80CHT
CM:NS=P
CM:EW=P
CM:C=P
NS:EW=P
NS:C=P
EW:C=P
NS:EW=C
NS:C=EW
EW:C=NS
CM:NS=0
CM:EW=0
CM:C=0
NS:EW=0
NS:C=0
EW:C=0
```

```
     111
( COMPOUND GAPP INSTRUCTIONS. 24NOV86CHT )      00:04 01Jan80CHT
-CM=0
-NS=0
-EW=0
-C=0
4R=0
-CM=P
-NS=P
-EW=P
-C=P
4R=P
```

```
     114
\ Simple operations                              00:42 01Jan80CHT
NOOOP      A sequence of 6APP noop instructions. Extremely useful.
W-SHIFT    Copy east neighbor, move data to west.
E-SHIFT    Copy west neighbor, move date to east.
N-SHIFT    Copy south neighbor, move date to north.
S-SHIFT    Copy north neighbor, move data to south.
EXCHANGE   Exchange data between EW and a memory plane.
EW-RESET   Clear EW register.
EW-SET     Set EW register.
NS-RESET   Clear NS register.
NS-SET     Set NS register.
C-GET      Copy a memory plane into CY register.
C-PUT      Copy CY register to a memory plane.
```

Listing 13. GAPP assembler on PC (cont'd)

```
     15
 0 \ Image moves
 1 HEX
 2 G: EW-GET   GADR EW=P   _$_
 3 G: CM-GET   GADR CM=P   _$_
 4 G: CM-MOVE      SOUTH   _$_
 5 G: CM-PUT   GADR P=CM   _$_
 6 DECIMAL
 7
 8
 9
10
11
12
13
14
15
```

```
  115
\ Image moves                                   00:12 01Jan80
EW-GET   Copy a memory plane into EW register.
CM-GET   Copy a memory plane into CM register.
CM-MOVE  Move data in CM register to north by reading CM of
         south neighbor.
CM-PUT   Copy CM register to a memory plane.
```

```
     16
 0 \ add and subtract
 1 HEX
 2 G: VADD   DUP GADR 1+ ROT   NS=P   C=0    _$_
 3           DUP GADR 1+ ROT   EW=P    _$_
 4           DUP GADR 1+ ROT   PLUS   CARRY    _$_
 5 G: VSUB   DUP GADR 1+ ROT   NS=P   C=0    _$_
 6           DUP GADR 1+ ROT   EW=P    _$_
 7           DUP GADR 1+ ROT   PLUS   BORROW   _$_
 8 G: UNSIGN  DROP DROP GADR   P=C     _$_
 9 G: SIGNED  DROP DROP GADR   PLUS    _$_
10 DECIMAL
11
12
13
14
15
```

```
  116
\ add and subtract                              00:18 01Jan80
VADD    Get source1 to NS.
        Get source2 to EW.
        Add and save sum in memory, carry to CY.
VSUB    Get source1 to NS.
        Get source2 to EW.
        Subtract.  Save difference in memory, and borrow in CY.
UNSIGN  Last step for unsigned add or subtract.
SIGNED  Last step for signed add or subtract.
```

```
     17
 0 \ increments and decrements
 1 HEX
 2 G: DEC   DUP GADR   EW=0   NS=P    _$_
 3          DUP GADR 1+   PLUS   BORROW    _$_
 4 G: DEC-C DUP GADR   C=1   EW=0   NS=P    _$_
 5          DUP GADR 1+   PLUS   BORROW    _$_
 6 G: INC   DUP GADR   EW=0   NS=P    _$_
 7          DUP GADR 1+   PLUS   CARRY     _$_
 8 G: INC-C DUP GADR   C=1   EW=0   NS=P    _$_
 9          DUP GADR 1+   PLUS   CARRY     _$_
10 G: SIGN-INC/DEC   GADR   PLUS    _$_
11 G: UNSIGN-INC/DEC   GADR   P=C    _$_
12 DECIMAL
13
14
15
```

```
  117
\ increments and decrements                     00:24 01Jan80
DEC   Decrement a memory plane according to contents of CY.

DEC-C Decrement a memory plane unconditionally.

INC   Increment a memory plane according to contents of CY.

INC-C Increment a memory plane unconditionally.

SIGN-INC/DEC   Last step of signed increment or decrement.
UNSIGN-INC/DEC Last step of unsigned increment or decrement.
```

Listing 13. GAPP assembler on PC (cont'd)

```
    18                                                    118
 0 \ dilation and erosions                                \ dilation and erosions                  00:26 01Jan80
 1 HEX
 2 6: DILW   EW=P DUP GADR   C=1   _$_  NS=EW  EW=W   _$_      DILW   Dilate in the west direction.
 3           CARRY   _$_  GADR P=C   _$_
 4 6: DILE   EW=P DUP GADR   C=1   _$_  NS=EW  EW=E   _$_      DILE   Dilate in the east direction.
 5           CARRY   _$_  GADR P=C   _$_
 6 6: DILS   NS=P DUP GADR   C=1   _$_  EW=NS  NS=S   _$_      DILS   Dilate in the south direction.
 7           CARRY   _$_  GADR P=C   _$_
 8 6: DILN   NS=P DUP GADR   C=1   _$_  EW=NS  NS=N   _$_      DILN   Dilate in the north direction.
 9           CARRY   _$_  GADR P=C   _$_
10 DECIMAL
11
12
13
14
15

    19                                                    119
 0 \ dilation and erosions                                \ dilation and erosions                  00:27 01Jan80
 1 HEX
 2 6: EROW   EW=P DUP GADR   C=0   _$_  NS=EW  EW=W   _$_      EROW   Erode to west.
 3           CARRY   _$_  GADR P=C   _$_
 4 6: EROE   EW=P DUP GADR   C=0   _$_  NS=EW  EW=E   _$_      EROE   Erode to east.
 5           CARRY   _$_  GADR P=C   _$_
 6 6: EROS   NS=P DUP GADR   C=0   _$_  EW=NS  NS=S   _$_      EROS   Erode to south.
 7           CARRY   _$_  GADR P=C   _$_
 8 6: ERON   NS=P DUP GADR   C=0   _$_  EW=NS  NS=N   _$_      ERON   Erode to north.
 9           CARRY   _$_  GADR P=C   _$_
10 DECIMAL
11
12
13
14
15

    20                                                    120
 0 \ dilation and erosion                    2            \ dilation and erosion                   00:29 01Jan80
 1 HEX
 2 6: DILBOX   NS:EW=P DUP GADR   C=1   _$_  EW=W   _$_      DILBOX   Dilate into a 3x3 box.
 3             CARRY   EW=NS   _$_  EW=E   NS=C   C=1   _$_
 4             CARRY   _$_  NS:EW=C   C=1   _$_  NS=N   _$_
 5             CARRY   NS=EW   _$_  NS=S   EW=C   C=1   _$_
 6             CARRY   _$_  GADR P=C   _$_
 7 6: DILCROSS NS:EW=P DUP GADR   C=1   _$_  EW=W   _$_      DILCROSS Dilate only to the four nearest neighbors.
 8             CARRY   NS:EW=P   _$_  EW=E   NS=C   C=1   _$_
 9             CARRY   _$_  NS=P   EW=C   C=1   _$_  NS=S   _$_
10             CARRY   NS=P   _$_  NS=N   EW=C   C=1   _$_
11             CARRY   _$_  GADR P=C   _$_
12 DECIMAL
13
14
15
```

Listing 13. GAPP assembler on PC (cont'd)

```
    21
 0 \ dilation and erosion                                    2
 1 HEX
 2 G: EROBOX   DUP GADR   NS:EW=P   C=0   _$_   EW=W   _$_
 3            CARRY   EW=NS   _$_   EW=E   NS=0   _$_
 4            CARRY   _$_   NS:EW=C   C=0   _$_   NS=N   _$_
 5            CARRY   NS=EW   _$_   NS=S   EW=0   _$_
 6            CARRY   _$_   GADR P=C   _$_
 7 G: EROCROSS DUP GADR   NS:EW=P   C=0   _$_   EW=W   _$_
 8            CARRY   NS:EW=P   _$_   EW=E   NS=C   C=0   _$_
 9            CARRY   _$_   NS=P   EW=C   C=0   _$_   NS=S   _$_
10            CARRY   NS=P   _$_   NS=N   EW=C   C=0   _$_
11            CARRY   _$_   GADR P=C   _$_
12 DECIMAL
13
14
15
```

```
   121
\ dilation and erosion                          00:31 01Jan80
```

EROBOX Erode within the 3x3 box.

EROCROSS Erode only to the four nearest neighbors.

```
    24
 0 \ image mask, invert, merge
 1 HEX
 2 G: GET-MASK     GADR   EW=P   _$_
 3 G: MASK-IMAGE   GADR   NS=P   C=0   _$_   CARRY   _$_
 4                GADR   P=C   _$_
 5 G: INVERT-IMAGE GADR   NS=P   C=1   EW=0   _$_
 6                GADR   PLUS   _$_   _$_
 7 G: MERGE-IMAGE  GADR   NS=P   C=1   _$_
 8                GADR   EW=P   _$_   CARRY   _$_
 9                GADR   P=C   _$_
10 DECIMAL
11
12
13
14
15
```

```
   124
\ image mask, invert, merge                     00:37 01Jan80
```

GET-MASK Initialize the masking operation.

MASK-IMAGE AND memory plane with a mask.

INVERT-IMAGE Invert a memory plane.

MERGE-IMAGE ORing two memory plane. The condition of merging is in another memory plane. The stack is (destination source1 source2 --)

```
    25
 0 \ bit mask, merge and duplicate
 1 HEX
 2 G: BIT-MASK   GADR   EW=P   _$_   GADR   NS=P   C=0   _$_
 3              CARRY   _$_   GADR   P=C   _$_
 4 G: BIT-MERGE  GADR   EW=P   _$_   NS=P   C=1   _$_
 5              CARRY   _$_   GADR   P=C   _$_
 6 G: DUPLICATE  DUP GADR   C=P   _$_   DUP GADR   P=C   _$_
 7              DUP GADR   P=C   _$_   DUP GADR   P=C   _$_
 8              DUP GADR   P=C   _$_   DUP GADR   P=C   _$_
 9              DUP GADR   P=C   _$_   GADR   P=C   _$_
10 DECIMAL
11
12
13
14
15
```

```
   125
\ bit mask, merge and duplicate                 00:39 01Jan80
```

BIT-MASK ANDing two memory planes.

BIT-MERGE ORing two memory planes.

DUPLICATE Duplicate a memory plane into the next 7 planes.

Listing 13. GAPP assembler on PC (cont'd)

NCR

NCR45CG72

GEOMETRIC ARITHMETIC PARALLEL PROCESSOR

■ APPLICATIONS

- PATTERN RECOGNITION
  - Correlation
  - Sobel Transform
  - Spoke Filter
  - Template Matching
  - Automated Inspection
  - Machine Vision
- PARALLEL DATA PROCESSING
  - Convolution
  - Matrix Operations
  - Histogram
  - Search and Sort
- IMAGE PROCESSING
  - Image Enhancement
  - Edge Detection
  - 2-Dimensional Convolution
  - Compression
  - Spatial Filtering
  - Differential Imaging
- ASSOCIATIVE PROCESSOR
  - Content Addressable Memory
  - Limit Search
  - Hamming Distance

■ GENERAL DESCRIPTION

The NCR45CG72 is a two-dimensional systolic array processor chip. It is a mesh-connected six by twelve arrangement of 1 bit processor elements. Each processor element can communicate with four neighbors: N,E,S, and W. Each processor element is composed of a bit serial ALU, 128 X 1 bit RAM and 4 single bit latches: Three latches hold inputs to the ALU and the fourth latch allows I/O through the cell without interrupting the ALU, i.e. I/O operations are overlapped with computation.

The cascadeability of the GAPP allows system designers to implement arrays of processors of arbitrary size in multiples of 6 X 12 elements.

■ FEATURES

- CMOS systolic array with 72 processors per chip
- 6 X 12 array of bit serial processor elements
- Single instruction multiple data stream architecture – all processor elements operate in parallel
- GAPP devices are fully cascadeable
- System throughput increases linearly with number of processor elements in the system
- Broadcast global input and output
- Separate I/O bus = overlapped I/O and computation
- 128 Bits of static RAM per processor
- VLSI double layer metal CMOS technology
- 500 milliwatts power at 10 MHz

Figure 22. NCR GAPP brochure

■ ABSOLUTE MAXIMUM RATINGS

Supply Voltage, V_{DD} . . . . . + 7V

Voltage on any pin with respect to ground . . . . . −0.3 to V_{DD} + 0.3V

Storage temperature . . . . . −65°C to 150°C

Stresses above "absolute maximum ratings" may result in damage to the device. Functional operation of devices at the "absolute maximum ratings" or above the recommended operation conditions stipulated elsewhere in this specification is not implied.

CAUTION

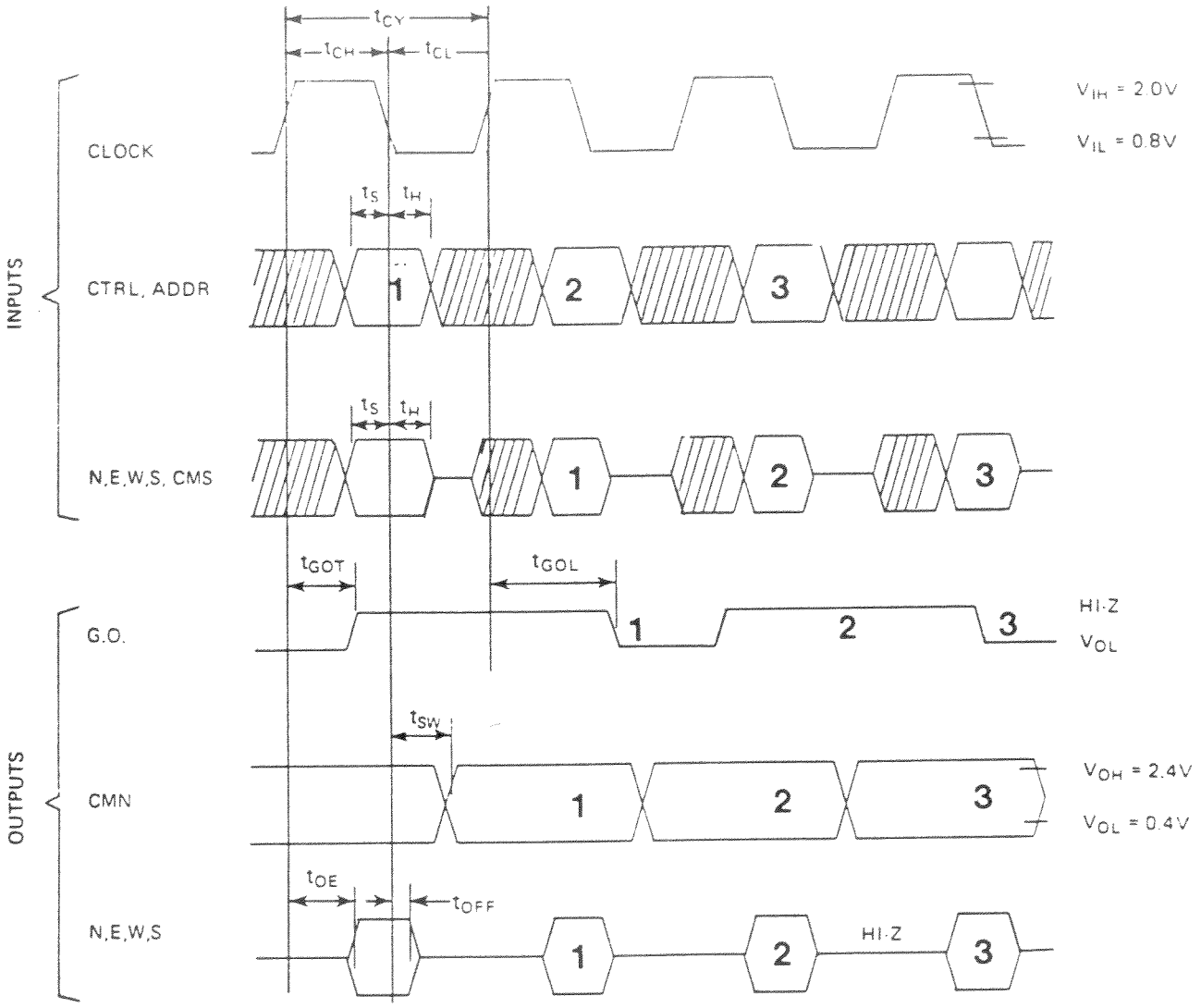
1. CMOS Devices are damaged by high energy electrostatic discharge. Devices must be stored in conductive foam or with all pins shunted. Precautions should be taken to avoid application of voltages higher than the maximum rating.
2. Remove power before insertion or removal of this device.

■ OPERATING CHARACTERISTICS

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS
Supply Voltage	V_{DD}	4.5	5.0	5.5	V
Supply Current (10 pF loads)					
45CG72-2		–		80	mA
45CG72-1	I_{DD}	–		100	mA
Input Low Voltage	V_{IL}	0.0		0.8	V
Input High Voltage	V_{IH}	2.0		V_{DD}	V
Output Low Voltage (I_{OL} = 2 mA)	V_{OL}	–		0.4	V
Output High Voltage (I_{OH} = 1 mA)	V_{OH}	2.4		–	V
Temperature	T_A	0		70	°C
Input Capacitance	C_{IN}	–		8	pF
Output Capacitance	C_O	–		8	pF
Leakage Current on any Input or I/O Pin	I_{IN}	–		10	μA

Figure 22. NCR GAPP brochure (cont'd)

■ TIMING DIAGRAM



NOTE: 1,2,3 refer to the staging sequence of instruction, data in and data out.

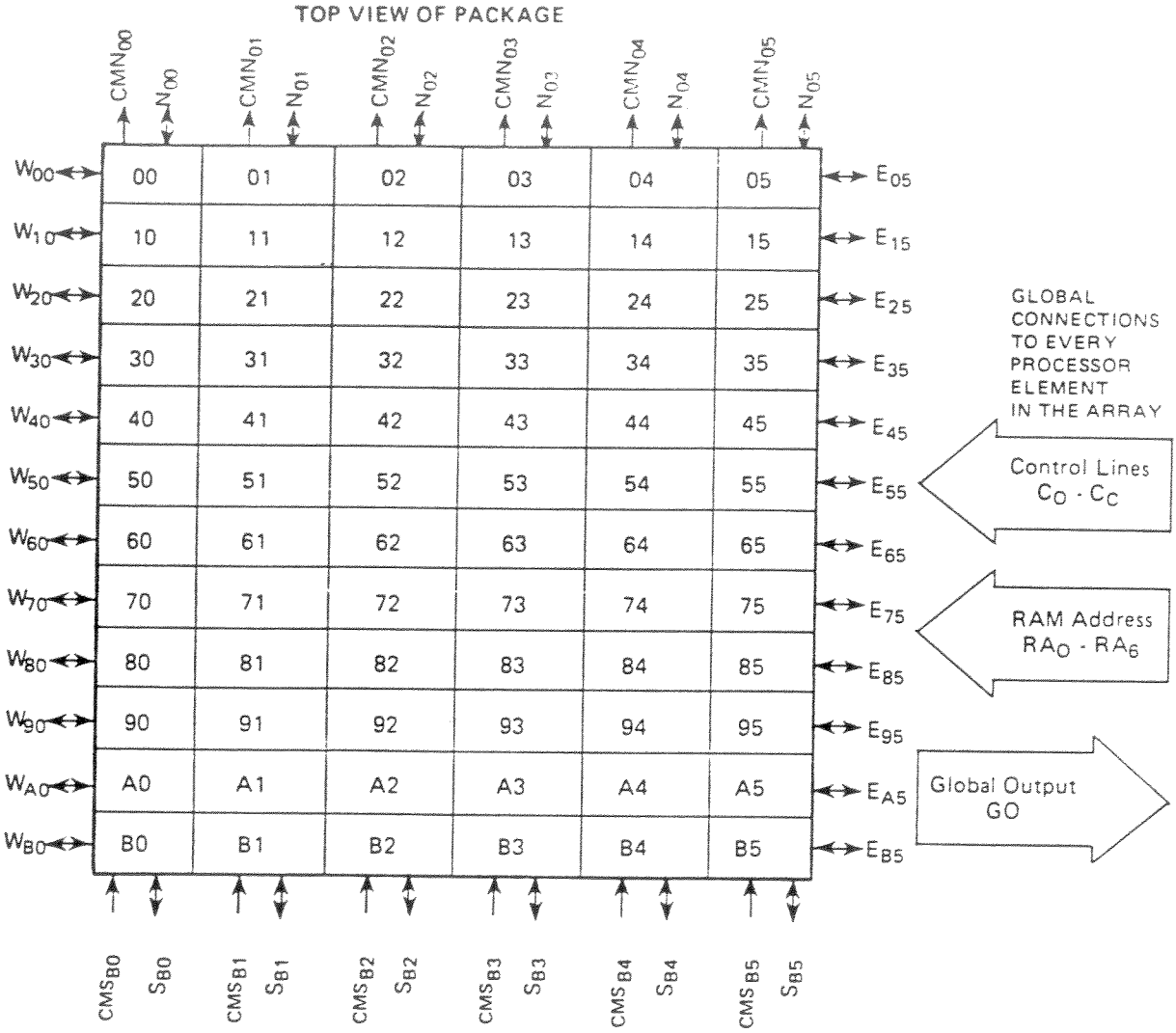
■ AC CHARACTERISTICS

PARAMETER	SYMBOL	NCR45CG72-2		NCR45CG72-1		
		MIN	MAX	MIN	MAX	UNITS
CYCLE TIME	t_{CY}	200		100		ns
CLOCK LOW	t_{CL}	100	5000	50	5000	ns
CLOCK HIGH	t_{CH}	100	5000(1)	50	5000(1)	ns
SETUP TIME	t_S	20		10		ns
INPUT HOLD TIME	t_H	10		10		ns
OUTPUTS ENABLED	t_{OE}	10	50	10	35	ns
OUTPUT HOLD TIME	t_{OFF}	10	50	10	30	ns
GLOBAL OUTPUT LOW	t_{GOL}	20	100	10	70	ns
GLOBAL OUTPUT TRISTATE	t_{GOT}	10	50	10	35	ns
CMN OUTPUT	t_{SW}	20	120	10	85	ns

NOTE: (1) d.c. by design; tested at 5 μsec.

Figure 22. NCR GAPP brochure (cont'd)

■ PROCESSOR ELEMENT AND DATA BUS IDENTIFICATION

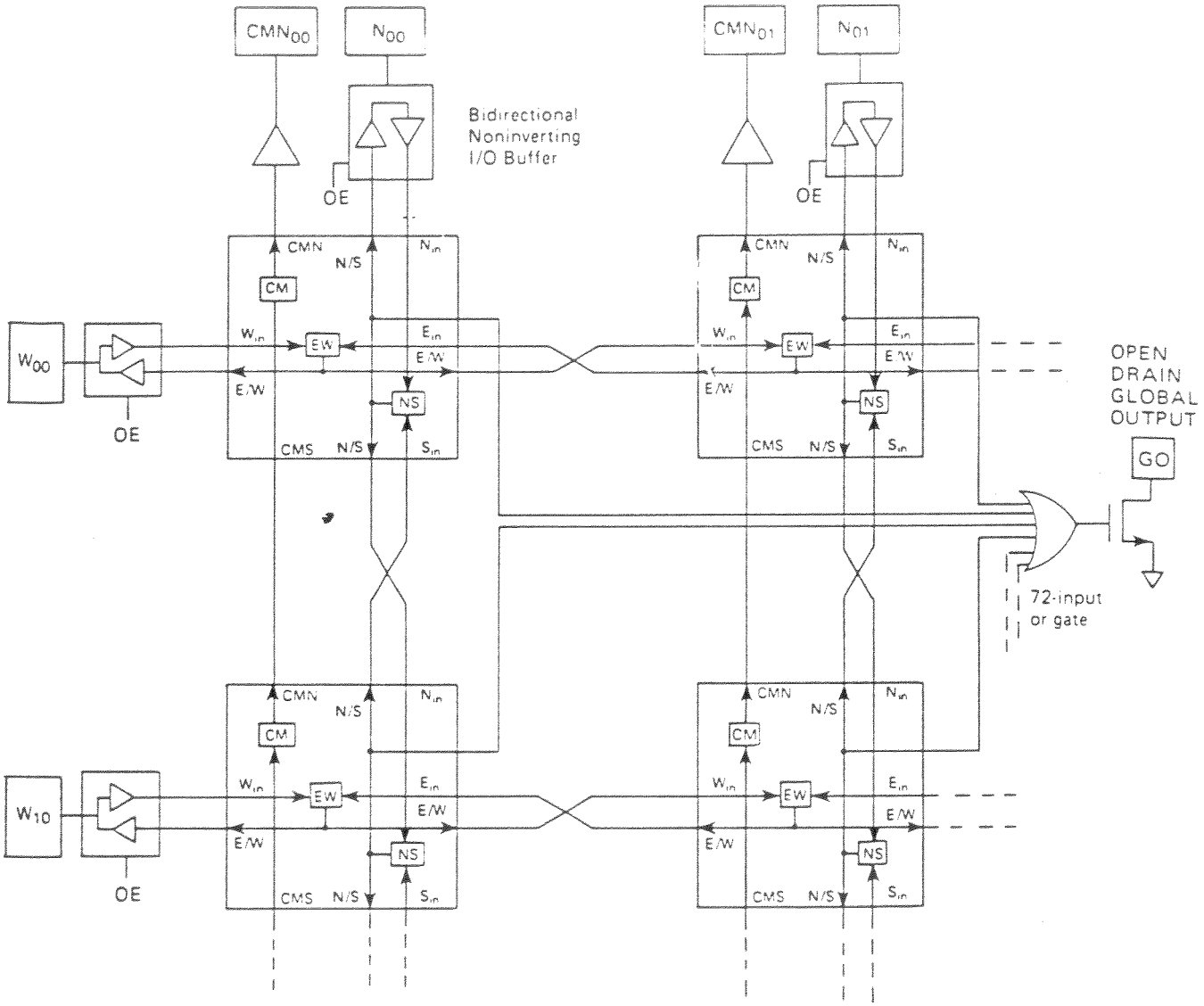


NOTE: This numbering scheme may be extended in systems which contain more than one GAPP device.

PIN LABELS	
$W_{00} - W_{B0}$	WEST DATA BUS
$E_{05} - E_{B5}$	EAST DATA BUS
$N_{00} - N_{05}$	NORTH DATA BUS
$S_{B0} - S_{B5}$	SOUTH DATA BUS
$CMS_{B0} - CMS_{B5}$	INPUT BUS
$CMN_{00} - CMN_{05}$	OUTPUT BUS
$RA_0 - RA_6$	RAM ADDRESS BUS
$C_0 - C_C$	CONTROL LINES – INSTRUCTION BUS – GLOBAL DATA INPUT BUS
GO	GLOBAL OUTPUT LINE

Figure 22. NCR GAPP brochure (cont'd)

▪ BLOCK DIAGRAM OF CONNECTIONS BETWEEN FOUR PROCESSOR ELEMENTS



OE = Output Enable is an internal connection.
East Outputs enabled whenever EW:=W
West Outputs enabled whenever EW:=E
North Outputs enabled whenever NS:=S
South Outputs enabled whenever NS:=N
GO is pulled low whenever any NS register contains 1

Figure 22. NCR GAPP brochure (cont'd)

▪ SCHEMATIC DIAGRAM OF ONE PROCESSOR ELEMENT

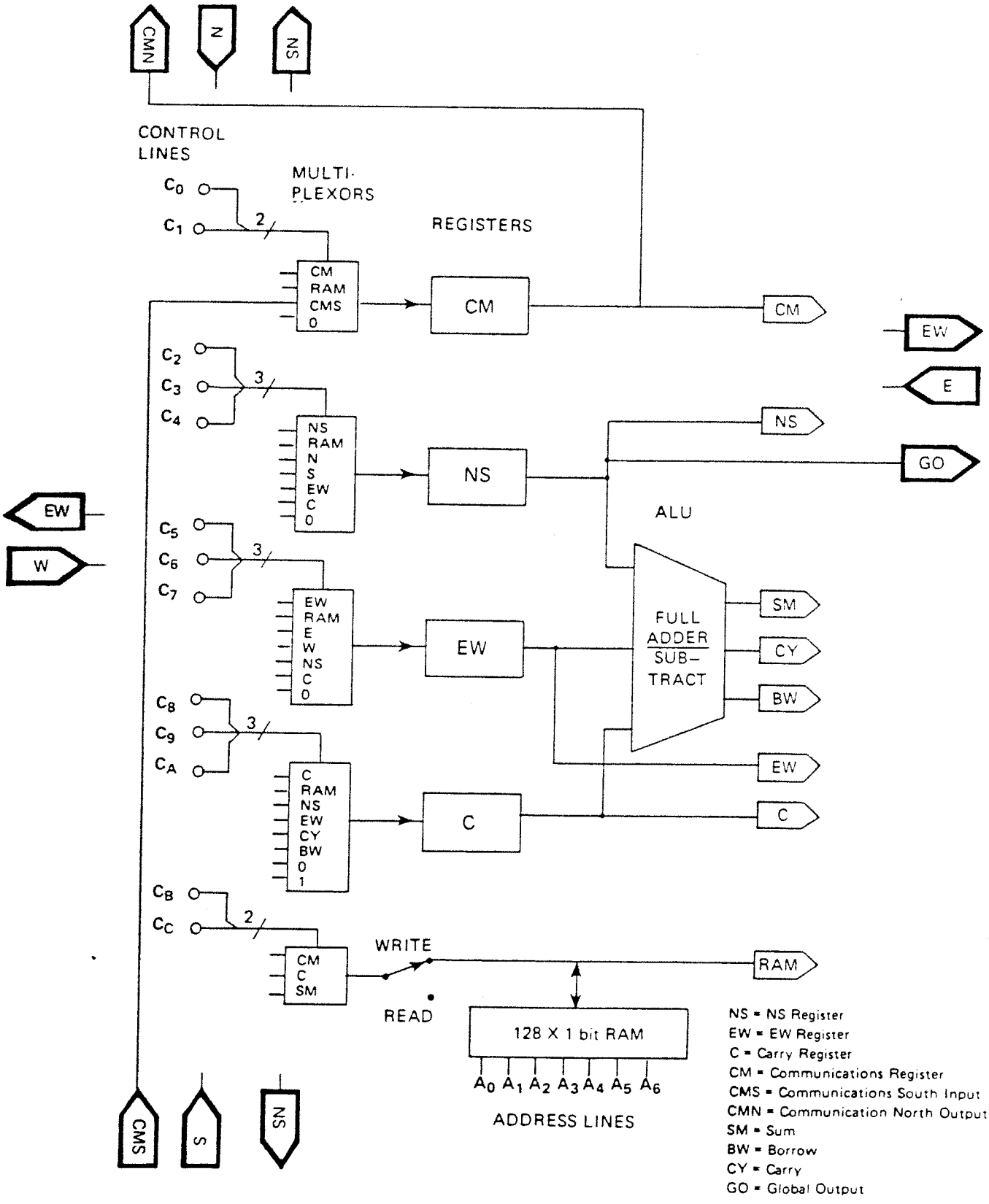


Figure 22. NCR GAPP brochure (cont'd)

■ INSTRUCTION SET

Register Operation	Mneumonic	Control Lines C_C	C_B	C_A	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	Description
CM	CM: = CM	X	X	X	X	X	X	X	X	X	X	X	0	0	MICRO-NOP
	CM: = RAM	X	X	X	X	X	X	X	X	X	X	X	0	1	LOAD CM FROM RAM
	CM: = CMS	X	X	X	X	X	X	X	X	X	X	X	1	0	MOVE FROM CMS INTO CM
	CM: = 0	X	X	X	X	X	X	X	X	X	X	X	1	1	LOAD Ø INTO CM
NS	NS: = NS	X	X	X	X	X	X	X	X	0	0	0	X	X	MICRO-NOP
	NS: = RAM	X	X	X	X	X	X	X	X	0	0	1	X	X	LOAD NS FROM RAM
	NS: = N	X	X	X	X	X	X	X	X	0	1	0	X	X	MOVE FROM N INTO NS
	NS: = S	X	X	X	X	X	X	X	X	0	1	1	X	X	MOVE FROM S INTO NS
	NS: = EW	X	X	X	X	X	X	X	X	1	0	0	X	X	MOVE FROM EW INTO NS
	NS: = C	X	X	X	X	X	X	X	X	1	0	1	X	X	MOVE FROM C INTO NS
	NS: = 0	X	X	X	X	X	X	X	X	1	1	0	X	X	LOAD Ø INTO NS
EW	EW: = EW	X	X	X	X	X	0	0	0	X	X	X	X	X	MICRO-NOP
	EW: = RAM	X	X	X	X	X	0	0	1	X	X	X	X	X	LOAD EW FROM RAM
	EW: = E	X	X	X	X	X	0	1	0	X	X	X	X	X	MOVE FROM E INTO EW
	EW: = W	X	X	X	X	X	0	1	1	X	X	X	X	X	MOVE FROM W INTO EW
	EW: = NS	X	X	X	X	X	1	0	0	X	X	X	X	X	MOVE FROM NS INTO EW
	EW: = C	X	X	X	X	X	1	0	1	X	X	X	X	X	MOVE FROM C INTO EW
	EW: = 0	X	X	X	X	X	1	1	0	X	X	X	X	X	LOAD Ø INTO EW
C	C: = C	X	X	0	0	0	X	X	X	X	X	X	X	X	MICRO-NOP
	C: = RAM	X	X	0	0	1	X	X	X	X	X	X	X	X	LOAD C FROM RAM
	C: = NS	X	X	0	1	0	X	X	X	X	X	X	X	X	MOVE FROM NS INTO C
	C: = EW	X	X	0	1	1	X	X	X	X	X	X	X	X	MOVE FROM EW INTO C
	C: = CY	X	X	1	0	0	X	X	X	X	X	X	X	X	LOAD C FROM CARRY
	C: = BW	X	X	1	0	1	X	X	X	X	X	X	X	X	LOAD C FROM BORROW
	C: = 0	X	X	1	1	0	X	X	X	X	X	X	X	X	LOAD Ø INTO C
	C: = 1	X	X	1	1	1	X	X	X	X	X	X	X	X	LOAD 1 INTO C
RAM	READ	0	0	X	X	X	X	X	X	X	X	X	X	X	READ FROM RAM
	RAM: = CM	0	1	X	X	X	X	X	X	X	X	X	X	X	LOAD RAM FROM CM
	RAM: = C	1	0	X	X	X	X	X	X	X	X	X	X	X	LOAD RAM FROM C
	RAM: = SM	1	1	X	X	X	X	X	X	X	X	X	X	X	LOAD RAM FROM SUM

■ ARITHMETIC OPERATIONS

Adder/Subtracter Operations

INPUT			OUTPUT		
NS	EW	C	SM	CY	BW
0	0	0	0	0	0
0	1	0	1	0	1
1	0	0	1	0	0
1	1	0	0	1	0
0	0	1	1	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	1	1

■ LOGIC OPERATIONS

LOGICAL OPERATION	DESCRIPTION	CONDITIONS
INV	$SM = \overline{NS}$	EW = 0, C = 1
	$SM = \overline{EW}$	NS = 0, C = 1
	$SM = \overline{C}$	NS = 0, EW = 1
AND	$CY = NS \bullet EW$	C = 0
	$CY = EW \bullet C$	NS = 0
	$CY = NS \bullet C$	EW = 0
	$BW = \overline{NS} \bullet EW$	C = 0
OR	$CY = NS + EW$	C = 1
	$BW = \overline{NS} + EW$	C = 1
	$BW = EW + C$	NS = 0
XOR	$SM = NS \oplus C$	EW = 0
	$SM = NS \oplus EW$	C = 0
	$SM = EW \oplus C$	NS = 0
XNOR	$SM = \overline{NS \oplus EW}$	C = 1

Figure 22. NCR GAPP brochure (cont'd)

■ PLASTIC CHIP CARRIER PACKAGE

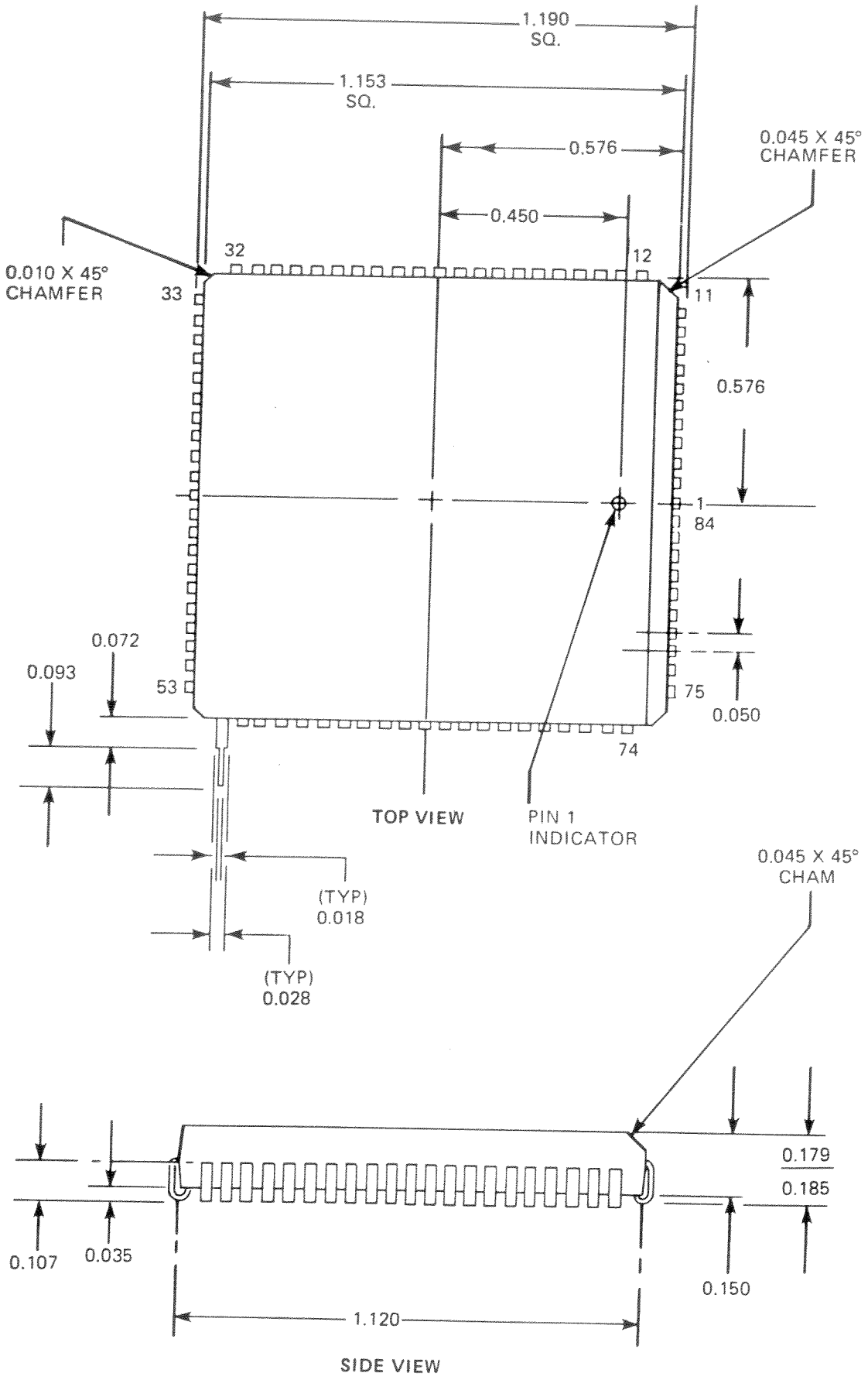


Figure 22. NCR GAPP brochure (cont'd)

▪ TABLE OF SIGNAL NAMES VS. PIN NUMBERS (CERAMIC AND PLASTIC)

SIGNAL NAMES	CER PIN	PLA PIN	SIGNAL NAMES	CER PIN	PLA PIN	SIGNAL NAMES	CER PIN	PLA PIN
CMS_{B0}	B3	14	N_{04}	K7	60	C_7	J9	52
CMS_{B1}	C4	16	N_{05}	H7	58	C_8	A10	30
CMS_{B2}	A4	18	S_{B0}	A2	13	C_9	C7	31
CMS_{B3}	A9	25	S_{B1}	A3	15	C_A	B9	32
CMS_{B4}	A6	27	S_{B2}	B4	17	C_B	C8	33
CNS_{B5}	B8	29	S_{B3}	A8	24	C_C	B10	34
CMN_{00}	K1	72	S_{B4}	B6	26	RA_0	C5	19
CMN_{01}	J4	70	S_{B5}	B7	28	RA_1	K2	67
CMN_{02}	J5	68	W_{00}	J1	76	RA_2	B5	20
CMN_{03}	H6	61	W_{10}	H2	77	RA_3	K3	66
CMN_{04}	J7	59	W_{20}	H1	78	RA_4	A5	21
CMN_{05}	K8	57	W_{30}	G3	79	RA_5	K4	65
E_{05}	G8	51	W_{40}	G1	80	RA_6	A7	23
E_{15}	K10	50	W_{50}	F3	81	GO	J8	54
E_{25}	H9	49	W_{60}	D1	4	VSS	C6	22
E_{35}	G9	48	W_{70}	C1	5	VSS	H5	64
E_{45}	F9	47	W_{80}	E2	6	VDD	E3	82
E_{55}	J10	46	W_{90}	E1	7	VDD	F8	41
E_{65}	E10	40	W_{A0}	C2	8	VDD	G10	45
E_{75}	E9	39	W_{B0}	A1	9	$VDD_{(sub)}$	–	53
E_{85}	D10	38	CLK	K6	63	N.C.	B1	1
E_{95}	D9	37	C_0	H3	75	N.C.	D2	2
E_{A5}	C10	36	C_1	J2	74	N.C.	D8	3
E_{B5}	C9	35	C_2	D3	10	N.C.	E8	42
N_{00}	H4	73	C_3	B2	11	N.C.	F1	43
N_{01}	J3	71	C_4	C3	12	N.C.	F2	44
N_{02}	K5	69	C_5	K9	56	N.C.	F10	83
N_{03}	J6	62	C_6	H8	55	N.C.	G2	84
						N.C.	H10	–

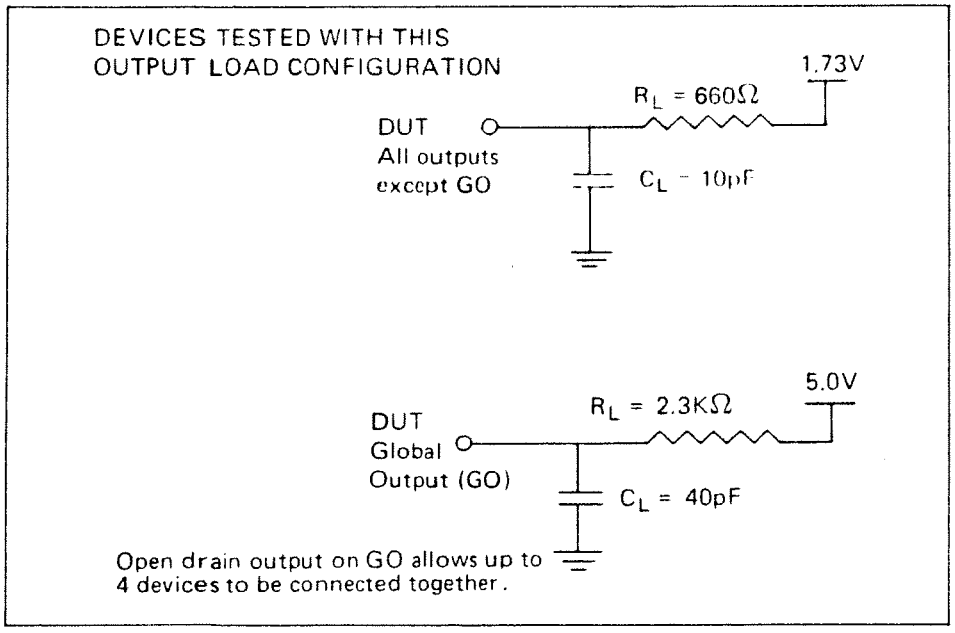


Figure 22. NCR GAPP brochure (cont'd)

XIII. AN INTERACTIVE OPERATING SYSTEM FOR AP120B ARRAY PROCESSOR

1. INTRODUCTION

Developing software for array processors(AP) is always a laborious and tedious task because one has to deal with a host of software tools to write and to test a program. Typically one has to write AP programs using a microcode assembler to assemble the microcode, which has to be bound with a main line program to be executed by the host computer. During execution, AP programs have to be loaded into the AP program memory. Arrays of data have to be formatted and moved into the data memory in AP. After AP routines are executed, resulting data have to be moved back to the host for examination or further processing.

Most AP manufacturers provide FORTRAN callable library packages with the hardware, assuming that the user will use FORTRAN to do all their work with the array processor. Since FORTRAN is a compiler language, it is very difficult to build an interactive system which allows the user to control the array processor intimately. One has to go through the editing-compiling-linking-loading process to get a shot at the program and repeat the process if anything has to be changed.

Floating Point System supplies a big software toolbox for the AP120B Array Processor, which has been a work horse for many scientific and engineering applications. Among the tools, APDBUG and APSIM do allow the user to try out various things interactively and observe the reaction of the array processor. However, the command set is severely limited and does not give the user full control over the entire range of AP120B's capability. Ideally, one should be able to access all the facilities in the array processor, direct the array processor to do elementary operations as defined by programs loaded in the program memory, and also construct and execute high level commands built from these elementary commands in an interactive fashion.

FORTH is a very powerful software tool, allowing the user to access and utilize all the facilities in a computer system. It was originally developed for instrumental control and programming. User can program at the lowest machine code level to take advantage of the resources provided in the computer system. Yet, it is also a high level language which enables the user to express his algorithm by strings of English- like commands, which can be either executed by an interpreter to cause immediate action in the computer system or by a compiler to construct more powerful high level commands to be executed later when called.

In our laboratory, an AP120B Array Processor from Floating Point System was installed on an Harris 80 Computer for experiments in real time signal processing. A FORTH interpreter-compiler was written for the Harris 80 Computer to facilitate the testing and maintenance of the interface between this computer and a number of peripheral equipments to collect real time data and to control some experiments actively. Since the AP120B is a very important link to close the control loop in this type of experiment, an effort was initiated to put the AP120B under the control of this FORTH system so that we might be able to test the entire system interactively.

2. THE FORTH OPERATING SYSTEM

This implementation of FORTH on the Harris 80 Computer was essentially a transcription of the fig-FORTH for the NOVA computer(1), released by the FORTH Interest Group in 1981. The NOVA FORTH model was chosen because the architecture of the Harris computer is very similar to that of the NOVA computer produced by Data General Corp. These two CPU's have the same types of registers and similar addressing modes. A requirement was to follow the fig-FORTH model as closely as possible so that the programs developed on the Harris computer can be transported to other computers with minimal modifications.

The most serious problem in adapting the fig-FORTH model to the Harris computer is that the Harris computer is a 24-bit machine while the fig-FORTH model presumes a 16-bit machine with byte addressing capability. Standard fig-FORTH commands use the same address to access bytes and 16 bit words. This is not feasible in Harris 80 because memory is accessed in 24-bit words. The solution adopted here was to make a clear differentiation between byte- addressing commands and word addressing commands. For the byte addressing commands, addresses are manipulated in a byte memory space. Two special commands were used to convert addresses from the byte space to the word space and vice versa. If proper care is exercised in the addressing modes, programs can be transported from the fig-FORTH system to this Harris FORTH system.

The dictionary in this FORTH system occupies about 3 Kwords in the core. With stacks and buffers, the entire FORTH load module is about 8 Kwords in size. Two disk buffers are allocated to interface with disk files, each one is 1 Kword in length or 3 Kbytes, equivalent to three standard FORTH blocks. This is a convenient size for the Harris operating system to handle the data transfer between a random access disk file and the FORTH module.

The FORTH module contains the FORTH nucleus, the text interpreter, and the colon compiler, with some extra utilities. A text editor and a Harris machine code assembler were written in FORTH and saved in a disk file, which can be loaded if needed. It was anticipated that the assembler will be necessary

to write the device driver for the FPS Array Processor. It turned out that we only had to implement a small code routine APDR in the FORTH nucleus to call the array processor service routines already installed in the Harris computer. Everything else needed to interface to the AP120B were written in high level FORTH language.

This FORTH implementation is described more fully elsewhere in this book(2).

3. PROGRAMMING MODEL OF AP120B ARRAY PROCESSOR

The array processor AP120B is a very complicated machine, with many processing elements, a multitude of interconnecting buses, and a large number of memories and registers. It takes a fair amount of time to learn this machine well enough to use this machine successfully. In our design of the AP operating system using FORTH as the underlying control executive, one major design goal was to simplify as much as possible the AP software as viewed from the programmer's direction and hide the internal complexity of the AP to shorten the learning curve in using the array processor. The programming model, or the structures in the AP relevant for its utilization, is shown schematically in Fig. 23.

In this model of AP120B, there are only four elements in the AP to be dealt with by the user: the main data memory MD, the program storage memory PS, the scratchpad registers SPAD, and the interface registers including the switch register SWR, the lights register LITES, and the function register FN. The user only has to manage these few elements in order to command the array processor to perform desired functions.

Among these elements, the most crucial element is the program storage memory PS, where AP program or subroutine modules are stored. To simplify the implementation and the use of this AP control system, it is assumed that the program storage memory is preloaded with a collection of AP subroutines selected from the AP math library. To ease the task in calling these subroutines from the FORTH operating system, a table of subroutine entry points is constructed and stored at the beginning of the PS memory. Each entry in this table contains two AP instructions: a JSR instruction to call the appropriate subroutine, and a HALT instruction to stop the AP activity after the subroutine is successfully executed. This way, the AP math library routines can be used without any modification. It is possible to optimize the library subroutines to eliminate the calling overhead required by APEX and to halt the AP at the end of the subroutine. These modifications will speed up the execution of AP functions and reduced the program memory size. However, this optimization will be left as future projects.

In the main data memory MD, the first 100 memory elements are reserved to store scalar constants and variables, as needed by

some AP functions. Starting at memory element 100 is the vector stack, which will be used by most AP functions implicitly. The vector stack is managed by two variables in FORTH, the variable VP, vector stack pointer, pointing to the top of the vector stack, and the variable FRAME, defining the size of the vector elements. In this FORTH system, all vector elements are of the same size. This may be a limitation if the applications requires vector elements of different sizes. However, for dedicated real time applications in which data are generally of the same size and format, this uniform vector format may be quite adequate. The advantage is that the language syntax is greatly simplified because the user does not have to specify explicitly the addresses, lengths, and increments of the vectors involved in an AP function, as in most AP FORTRAN subroutine calls.

The scratchpad registers SPAD are used by the AP to retrieve parameters needed in executing an AP function. Up to 16 parameters can be passed to an AP subroutine via SPAD. Before the FORTH system commands the AP to start executing an AP subroutine, it has to fill the SPAD with the necessary list of parameters into a buffer in the Harris computer. The address of this buffer is then passed to the FORTH command SPLDGO (SPAD LoaD & Go) so that this parameter list is read into the SPAD registers in the AP before the AP function is executed.

The AP120B is attached to the Harris 80 computer as an I/O peripheral device. From the side of the Harris computer, the AP120B appears as three registers in an I/O device: two read/write registers SWR and FN, and a read-only register LITES. Commands and parameters are transferred to the AP and AP status can be examined by the Harris computer through these registers. The programming task is essentially developing specific commands which control these three registers.

4. IMPLEMENTATION OF THE AP OPERATING SYSTEM

The entire program to control AP120B from the FORTH operating system is only three pages of FORTH source codes, which are shown in its entirety in Listing 14. It is equivalent to about a hundred pages of FORTRAN codes for the equivalent functions. For the readers who are not familiar with the FORTH operating system and its peculiar language syntax, a few guidelines are offered here to help reading the source code.

1. Source codes are arranged in screens of 1024 bytes. In each screen, codes are grouped into 16 lines, each of 64 bytes in length. A line number precedes the line of code, but is not part of the code.

2. Words are separated by one or more spaces. A word can be a command, a number, or a string which must follow a string command.

3. (is a comment command, causing the FORTH interpreter to ignore all the text up to and including the delimiter ')'.

4. : is a command causing the FORTH interpreter to construct a new command and add it to the dictionary in the FORTH operating system. The syntax of a new definition is:

: <name> <list of valid FORTH words> ;

; is the command to terminate a new command and make it available for execution or compilation.

5. The stack effect of a command is documented as a comment after the name of the new command. Items before the --- marks are those on the top of the stack before executing the command, and the items after the --- marks are those numbers left on the stack after executing the command.

6. DECIMAL and OCTAL alternate the number base between the regular decimal system and the octal system often using in addressing AP registers and the Harris memory.

7. EXIT terminates the interpretation of a screen of text. The texts after EXIT are ignored by the interpreter.

5. AP DEVICE DRIVER

Two physical devices are assigned to the AP120B in the Harris I/O structure. Device 65 is used to handle the SWR, LITES and FN interface registers, and device 66 is used to handle interrupts from the AP120B. The AP device driver routines are installed in the Harris I/O service package, and the elementary AP functions can be called directly through the standard Harris I/O calling protocol:

```
        TLO     IOPAR           Transfer address of parameter
                                list to the K register.
        BLU     $IOW            Call the I/O service.
        ...
        ...
        ...

IOPAR   DATA    'xxxyy          Device number xxx and function
                                code yy in octal.
        DATA    word count
        DAC     buffer-address
```

Two FORTH commands I/O and APDR were implemented in the FORTH module. I/O is to handle general input/output service to all the Harris peripheral devices, and APDR is a adaption of I/O to the AP120B. I/O requires three parameters as input on the FORTH data stack: the device-function code, the word count and the parameter list address. APDR also requires three parameters on

the data stack: the AP driver function code, the pattern to be copied into SWR register, and the command pattern to be copied into the FN register. APDR assumes that the device to be addressed is AP120B, Device 65 in Harris. I/O was used only once to define the function APOPEN, because it has to address Device 66 to initialize the interrupt handler for AP120B. All other elementary AP functions are derived from APDR.

I/O (buffer-addr word-count function-code ---)

The most elementary I/O command passing control to the I/O service routines in the Harris operating system. The top item specifies the I/O channel and the function to be performed. If the function requires the transferring of additional parameters, the buffer address and the the size of the parameter buffer must be specified as the next two items under the function code. If address and count are not needed, dummy numbers must be supplied.

APDR (parameter1 parameter2 function ---)

Fill the I/O parameter buffer with the two parameter values and the function code on the stack and executed the AP function. It calls the AP driver routines installed in the Harris computer.

6. ELEMENTARY AP FUNCTIONS

Elementary AP functions are simple derivatives of APDR. For some functions, two parameters and the function code used by APDR are sufficient to specify the operations and APDR is executed immediately. For more complicated functions, more parameters have to be moved into a buffer called IOPAR and one of the parameter is used to point to IOPAR. APDR then picks up these additional parameters in IOPAR for its execution. A few supporting functions are also included in this category. They are used to move data among buffers, memory and registers.

APOPEN (---)

Open the logic Devices 65 and 66, which must be assigned to the physical devices associated with AP120B hardware interfaces to the Harris computer by Harris system commands. This command must be executed before any other AP commands.

APIN (register --- value)

Read the contents of one of the AP interface register whose number is placed on the stack. Returned value on the stack is its contents.

APOUT (value register ---)

Store the value into the AP interface register whose number is given on the top of the stack.

RREG (function --- lites)

Examine an AP register or memory by writing the FUNCTION register and reading the LITES register.

WREG (function switch ---)

Deposit into an AP register or memory by writing the SWITCH register and the FUNCTION register.

WTRUN (--- error)

Wait for the current AP program to finish and return the completion code. Error occurred if the returned code is not zero.

WTDMA (--- error)

Wait for the completion of a DMA transfer. A completion code is returned on the stack.

@REG (register --- value)

Use RREG to fetch the contents of an AP register.

!REG (register value ---)

Store a value into an AP register.

APBUF (--- buffer-addr)

Return the address of an array where the I/O parameters are stored to be retrieved by the command I/O.

SPAD (--- address)

Return the address of an array where parameters are stored and moved into the SPAD in AP before an array processing function is executed.

!PAR (heap addr count ---)

A set of numbers piled as a heap on the stack are dumped to the memory starting at addr. The number of items moved is given on the top of the stack as a count.

!DMA (control word-count ap-addr host-addr ---)

The four parameters on the stack are stored in the I/O parameter buffer to specify the DMA actions to be followed.

RUNDMA (---)

Executed a DMA transfer. The detailed actions must be specified by an appropriate !DMA command.

!SPAD (nspads slist start function errloc noload psa ---)

Store seven parameters required of an AP process into the I/O parameter buffers. Nspads is the numbers of SPAD items to be used, slist is the address of the array SPAD from which SPAD items are to be passed into AP, start is the starting address of the executable code in AP, function is the functional command, errloc is the address where an error code will be returned, noload indicates whether codes are to be loaded from the host, and psa is the address of the PS memory.

RUNAP (---)

Execute an AP process. The process must be specified by the !SPAD command.

APERR (--- error)

Return the error code produced by the last AP operation.

APRSET (---)

Reset AP by stopping any DMA activities, halting the AP, resets the interface, and initializes various flags and data values.

7. VECTOR STACK MANAGEMENT

Most AP math library subroutines require long lists of parameters to specify the addresses, lengths, and increments of the vectors involved in their operations. These parameters are passed to the AP subroutines via SPAD registers. Using the elementary AP function defined above, we can pass all the necessary parameters explicitly with !SPAD command. However, to pass long lists of parameters on the data stack in FORTH is very messy and often ensures the unreadability of the program. Assuming that the vectors to be are of the same size and format, we can construct a vector stack in the MD memory to manipulate these vectors. AP stack operations will remove their required vector operands from the top of the vector stack, and leave only the explicit results on the vector stack for the subsequent operations to used as operands. Thus all references to the vector operands are implicit and the user does not have to supply the parameter lists. This vector stack greatly simplifies the syntax and the programming of this AP operating system, similar to the use of a data stack in FORTH.

VP and FRAME are two basic tools for managing the vector stack in the MD memory. They are defined as variables so that the structure and the location of the vector stack can be changed

dynamically. Other commands are defined to support AP functions to handle the vector stack more efficiently.

VP (--- pointer)

Return the current pointer to the top of the vector stack. It is initialized to point at MD location 100.

FRAME (--- size)

Return the frame size of the vectors on the vector stack. it is initialized as 10 for demonstration purposes.

?VP (---)

Check the vector stack pointer. If it points below 100, abort the current AP process and re-initialize VP to 100.

+VP (---)

Increase the vector stack pointer VP by one frame.

-VP (---)

Decrease the vector stack pointer VP by one frame. The net effect of this command is popping the top vector off the vector stack.

V@ (host-address ---)

Read one frame of data from the buffer starting at host-address in the Harris computer into AP through DMA transfer and push this frame of data on the vector stack.

V! (host-address ---)

Pop the top frame on the vector stack and write this frame of data to the host buffer starting at host-address.

V? (---)

Remove the top frame on the vector stack and print its contents on the CRT terminal.

V. (---)

Display the contents of the entire vector stack on the CRT terminal This is a very useful command to inspect the vector stack without disturbing the size and contents of the vector stack. It calls a low level command (V.) to do the printing.

SAME (---)

Fill the SPAD with three parameters: the address, size, and increment of the top frame on the vector stack. This command is used to initialize the SPAD for AP function involving only one vector.

BINARY (---)

Fill the SPAD with 7 parameters, specifying that the topmost two frames are to be the source vectors and the second frame on the vector stack to be the destination vector for the following AP function. It is used to set up SPAD for binary vector functions like add, subtract, multiply, and divide.

INPLACE (---)

Fill the SPAD with 5 parameters, specifying an inplace AP function which uses the topmost vector frame as the source and the destination.

8. VECTOR STACK OPERATIONS

Most of the vector stack operators call their corresponding library subroutines installed in the PS memory. Taking their operands off the vector stack and leaving their results on the vector stack, the parameter lists required by the library subroutines can be generated automatically by the supporting commands like SAME, BINARY, and INPLACE. All the vector operators eventually call SFLDGO command, which passes the SPAD parameter list and executes the AP operation at the PS addresses given to it on the data stack.

The mnemonic names chosen in this FORTH system for AP operations mimic the names of their corresponding subroutine defined in the FPS math library, except those for which FORTH type generic names are more appropriate, like V+, V-, V# and V/. The AP operations included in this sample system are very limited in its scope. The AP functions are limited only by the availability of the subroutine library and the size of the PS memory.

SPLDGO (address ---)

Reset AP120B, copy parameters into SPAD. and start AP120B at the PS address given on the data stack.

VO (---)

Clear the top frame on the stack to zero's.

VDUP (---)

Duplicate the top frame on the vector stack.

VOVER (---)

Duplicate the second topmost frame and push it on the top.

VDROP (---)

Discard the topmost frame on the vector stack.

V+ (---)

Add the top two frames on the vector stack, pop the topmost frame, and replace the original second frame with the sum.

V- (---)

Remove the topmost two frames on the vector stack and push the difference vector on the stack (second-first).

V* (- -)

Remove the topmost two frames on the vector stack and push the product vector on the stack.

V/ (---)

Remove the topmost two frames on the vector stack and push the quotient vector on the stack (second/first).

VSIN (---)

Convert the top frame on the stack to its sines.

VCOS (---)

Convert the top frame on the stack to its cosines.

VFILL (source ---)

Push a new frame on the vector stack and fill this frame with the value stored in the MD address by the number on the data stack.

VRAMP (source increment ---)

Push a new frame on the vector stack and fill this frame with a ramp function whose starting value and increment value are in the MD memory addresses by the top two numbers on the data stack.

9. DIAGNOSTIC TOOLS

A few commands for diagnosing the AP system and its operations are also defined for the user to inspect the registers and some parts of the memories in the AP120B.

STATUS (---)

Display the contents of all the registers in AP120B addressible by the Harris computer. It is a power tool to inspect the

current status of the AP for diagnostics.

APSTEP (---)

Causes the AP to execute the next instruction in the PS memory.

APCONT (---)

Continue the AP execution from the point of last interruption.

10. A DEMONSTRATION SESSION

To demonstrate the interactive features of this AP operating system a short terminal session was recorded and shown in Figure 24. Prior to entering this AP operating system, an AP load module must be loaded into the PS memory. The load module listing is shown in Figure 25, which contains only a set of entry points with subroutine calls to the AP math library. It was assembled using APAL and the load module was generated by APLINK. The module with the library routines is then loaded into PS memory by APDBUG. A list of constants is also loaded via APDEBUG into the MD memory for testing. This list of constants is shown in Figure 26. After the AP is thus configured, the FORTH module is loaded into Harris 80 computer. The AP control program shown in Figure 2 is then compiled into the FORTH dictionary. At this point all the AP commands defined in the control program are available for execution and compilation, as commanded by the user through the Harris console terminal.

In the demonstration session shown in Figure 24, we started by pushing some constant vectors on the vector stack and performed a vector add. Contents of the stack were displayed using the V. command. Then a ramp vector was pushed on the stack and converted to a sine vector. The last part of the demonstration showed that a new function VTAN were defined and tested. The ability to compiled new commands from existing command set interactively is one of the very powerful features of the FORTH language.

11. CONCLUSION

It is demonstrated that the FPS AP120B array processor can be used interactively under the control of a FORTH operating system. Here the AP120B is used as a fixed instruction set slave processor to process vector arrays of fixed length and identical format. These restrictions reduces the APEX overhead to a minimum because the library subroutines are loaded into PS memory only once and remain static in the PS memory. FORTH operating system only has to start the AP at known entry points in order to invoke needed AP functions. Fixed vector format

allows the manipulating of vector arrays using a LIFO vector stack structure. The vector stack greatly reduces the overhead in passing SPAD parameters because vector addresses can be computed automatically, and the AP commands can be issued, using very simple syntax, making AP a very friendly computing device.

The limitations on the fixed instruction set and on the fixed data format are artificial for this demonstration implementation. It is possible to reload or overlay PS memory with new subroutines, so that the AP can be reconfigured dynamically for multi-task applications. Any vector data structure can be accommodated if the user are willing to specify all the SPAD parameter explicitly.

A microassembler(3) to develop microcodes for bit-slice microprocessor was completed in this laboratory, under a FORTH operating system. It can be adapted to the Harris-AP120B system so that AP microcodes can be written and tested interactively. This facility, once completed, will allow us to bypass all the vendor software tools and used the array processor under a single operating system using a single language.

REFERENCES

1. C. H. Ting, 'fig-FORTH for NOVA-Computer', FORTH Interest Group, San Carlos, CA. . 1981.

2. See 'FORTH for the Harris 80 Computer' in this volume.

3. C. H. Ting, 'Microassembler' in 'Forth Notebook', Offete Enterprises, 1983, pp. 136-169.

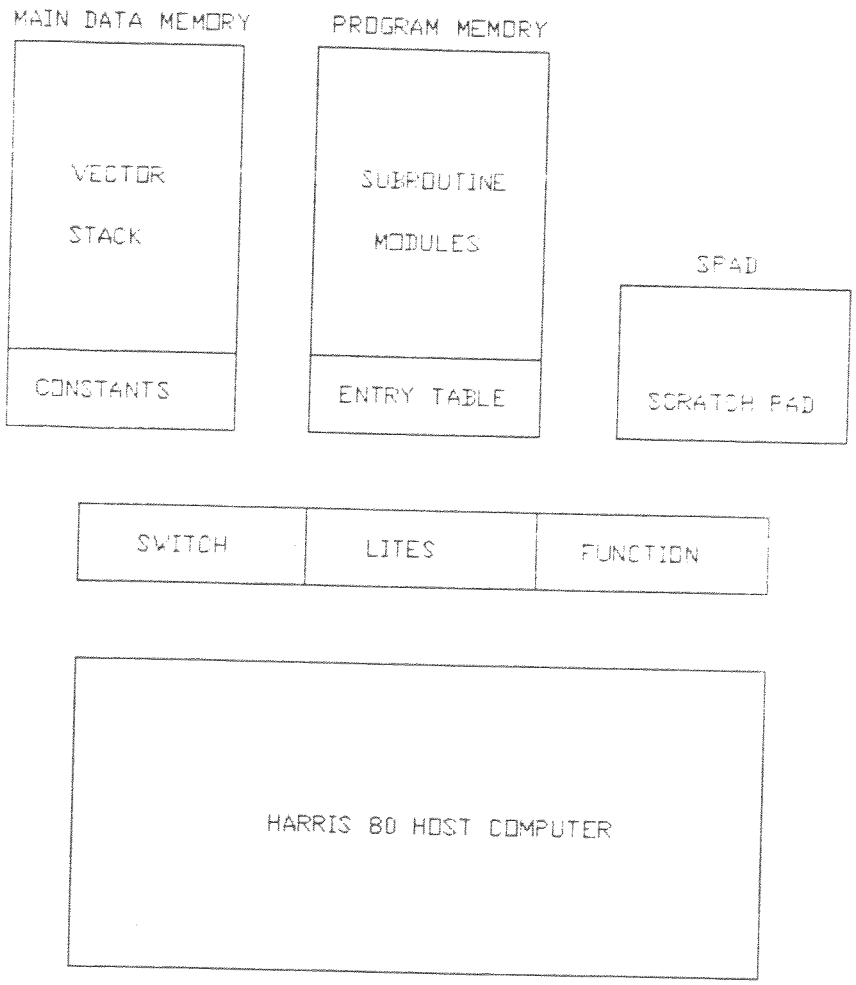


Figure 23. Programming model of AP120B array processor

```
1 VFILL    ( Push a vector of 1's on the vector stack.)
2 VFILL    ( Push a vector of 2's on vector stack.)
V.    ( Display the vector stack.)

     11F4     4ØØØØØ        2   4ØØØØØ        2   4ØØØØØ        2   4ØØØØØ        2
     11FC     4ØØØØØ        2   4ØØØØØ        2   4ØØØØØ        2   4ØØØØØ        2
     12Ø4     4ØØØØØ        2   4ØØØØØ        2         Ø        Ø   4Ø4352   2Ø562E
     11F4     4ØØØØØ        1   4ØØØØØ        1   4ØØØØØ        1   4ØØØØØ        1
     11FC     4ØØØØØ        1   4ØØØØØ        1   4ØØØØØ        1   4ØØØØØ        1
     12Ø4     4ØØØØØ        1   4ØØØØØ        1         Ø        Ø   4Ø4352   2Ø562E

V+    ( Add the two vectors.)
V.    ( Show the result.)

     11F4     6ØØØØØ        2   6ØØØØØ        2   6ØØØØØ        2   6ØØØØØ        2
     11FC     6ØØØØØ        2   6ØØØØØ        2   6ØØØØØ        2   6ØØØØØ        2
     12Ø4     6ØØØØØ        2   6ØØØØØ        2         Ø        Ø   4Ø4352   2Ø562E

VDROP    ( Discard the result.)
V.    ( Inspect the vector stack again.)
V-stack empty.


Ø 14 VRAMP    ( Push a ramp vector on the vector stack.)
V.    ( Inspect it.)

     11F4          Ø       81   51EB85   1ØØØFA   51EB85   1ØØØFB   7AE147   58ØØFB
     11FC     51EB85   1ØØØFC   666666   3ØØØFC   7AE147   58ØØFC   47AE14   4ØØØFD
     12Ø4     51EB85   1ØØØFD   5C28F5   6ØØØFD         Ø        Ø   4Ø4352   2Ø562E

VSIN    ( Convert it to a sine vector.)
V.    ( Inspect the result.)

     11F4          Ø       81   51EB2B   5ØØØFA   51EA1F   2ØØØFB   7ADC8F   6ØØØFB
     11FC     51E5ED   48ØØFC   665B7A   48ØØFC   7ACE68   6ØØØFC   479F19       FD
     12Ø4     51D528   28ØØFD   5CØ91F   2ØØØFD         Ø        Ø   4Ø4352   2Ø562E

VDROP    ( Clean the vector stack.)


Ø 14 VRAMP    ( Push a ramp vector on stack for testing.)
V.    ( Take a look.)

     11F4          Ø       81   51EB85   1ØØØFA   51EB85   1ØØØFB   7AE147   58ØØFB
     11FC     51EB85   1ØØØFC   666666   3ØØØFC   7AE147   58ØØFC   47AE14   4ØØØFD
     12Ø4     51EB85   1ØØØFD   5C28F5   6ØØØFD         Ø        Ø   4Ø4352   2Ø562E
VTAN    ( Now test the new tangent operator.)
V. VDROP    ( See if the result looks reasonably right.)

     11F4          Ø       81   51EC38       FA   51EE51       FB   7AEAB8   3ØØØFB
     11FC     51F6B6   2ØØØFC   667C44   3ØØØFC   7BØ715   28ØØFC   47CC1C   3ØØØFD
     12Ø4     52185F   68ØØFD   5C68DE   1ØØØFD         Ø        Ø   4Ø4352   2Ø562E
```

Figure 24. A demonstration session of AP120B Forth

```
;*****************************************************************
;
;       AP TEST  USING APDBUG
;
;       C. H. TING, DEC-14-83
;
;*****************************************************************
;
;
        $TITLE                  TESTG
;
;
        $EXT    VCLR,VMOV,VSWAP,VNEG,VADD,VSUB,VMUL,VABS
        $EXT    VFILL,VRAMP,VDIV,VSIN,VCOS,VRAND
;
;
ORG:    HALT
        HALT
;
;
        JSR     VCLR
        HALT
        JSR     VMOV
        HALT
        JSR     VSWAP
        HALT
        JSR     VNEG
        HALT
        JSR     VADD
        HALT
        JSR     VSUB
        HALT
        JSR     VMUL
        HALT
        JSR     VFILL
        HALT
        JSR     VRAMP
        HALT
        JSR     VSIN
        HALT
        JSR     VCOS
        HALT
        JSR     VDIV
        HALT
        $END
```

Figure 25. Entry points of the AP load module

```
  17 10 1
 0.0000000000      "MD    000000
 1.0000000000      "MD    000001
 2.0000000000      "MD    000002
 3.0000000003      "MD    000003
 4.0000000000      "MD    000004
 5.0000000000      "MD    000005
 6.0000000000      "MD    000006
 7.0000000000      "MD    000007
 8.0000000000      "MD    000010
 9.0000000000      "MD    000011
 10.0000000000     "MD    000012
 100.0000000       "MD    000013
 1000.000000       "MD    000014
0.9999999963E-01   "MD    000015
0.1000000001E-01   "MD    000016
0.1000000004E-02   "MD    000017
 3.141592562       "MD    000020
EOF..
```

Figure 26. Constants in the MD memory

```
SCR#  21
  Ø ( AP CONTROL, CHT, 2Ø-DEC-83)                    OCTAL
  1 : APOPEN    Ø 1 1Ø113 I/O   Ø 1 1Ø213 I/O    ;
  2 : APIN    ( REGISTER --- VALUE )
  3         PAD SWAP 12 APDR    PAD @    ;
  4 : APOUT    ( VALUE REGISTER --- )
  5         32 APDR    ;
  6 : RREG    ( FUNCTION --- LITES )
  7         PAD SWAP 13 APDR    PAD @    ;
  8 : WREG    ( FUNCTION SWITCH --- )
  9         34 APDR    ;
 1Ø  APOPEN  DECIMAL EXIT
 11 : WTRUN    ( --- APXERR )    Ø Ø 5 APDR    APPAR 3 + @    ;
 12 : WTDMA    ( --- APXERR )    Ø Ø 6 APDR    APPAR 3 + @    ;
 13 : REGTEST    15 1 DO    CR I . I APIN . LOOP    ;
 14 : REGTEST1 ( VALUE --- ) 15 1 DO DUP I APOUT LOOP DROP REGTEST ;
 15


SCR#  22
  Ø ( ACCESSING AP REGISTERS, CHT, 21-DEC-83)
  1 OCTAL
  2 : @REG    ( REG --- DATA )
  3         2Ø6Ø + RREG    ;
  4 : !REG    ( REG DATA --- )
  5         SWAP 1Ø6Ø + SWAP    WREG    ;
  6 : SP?    CR 2Ø Ø DO 1 I !REG 5 @REG 1Ø U.R
  7         I 7 = IF CR THEN    LOOP    ;
  8 DECIMAL                    EXIT
  9 : CTEST    16 Ø DO I @REG . LOOP ;
 1Ø : !TEST ( DATA --- )    16 Ø DO I OVER !REG LOOP DROP @TEST ;
 11 EXIT
 12
 13
 14
 15


SCR#  23
  Ø ( RUNDMA & RUNAP, CHT, 22-DEC-83)
  1 VARIABLE APBUF 6 ALLOT    VARIABLE SPAD 15 ALLOT
  2 : !PAR    ( HEAP ADDR N --- )
  3           OVER + SWAP DO    I !    LOOP    ;
  4 : RUNDMA    APBUF Ø 18 APDR    ;
  5 : RUNAP    APBUF Ø 3Ø APDR    ;
  6 : !DMA    ( CTRL NUM APMA HOST --- )
  7           APBUF 4 !PAR    ;
  8 : !RUN    ( NSPADS SLIST STRT FN BRKLOC NOLOAD PSA --- )
  9           APBUF 7 !PAR    ;
 1Ø : WTRUN    Ø Ø 5 APDR    ;
 11 : WTDMA    Ø Ø 6 APDR    ;
 12 : APXERR    ( --- ERR )    APPAR 3 + @    ;
 13 : APRSET    Ø Ø 23 APDR    ;
 14 EXIT
 15

C. H. Ting                                          Harris 8Ø
 ??
```

Listing 14. Controlling AP120B array processor

```
SCR#  24
  Ø ( VECTOR STACK I/O, CHT, 22-DEC-83)
  1 VARIABLE VP ( VECTOR STACT POINTER)   1ØØ VP !
  2 VARIABLE FRAME   ( VECTOR SIZE)   1Ø FRAME !
  3 : ?VP   VP @ 1ØØ <   IF 1ØØ VP ! 29 ERROR THEN   ;
  4 : +VP   FRAME @ VP +!   ?VP   ;
  5 : -VP   FRAME @ NEGATE VP +!   ?VP   ;
  6 OCTAL
  7 : V@     ( HOST --- )
  8         >R 3Ø4 FRAME @ 2* +VP VP @ R>   WTDMA !DMA RUNDMA ;
  9 : V!     ( HOST-ADDR --- )
 1Ø         >R 344 FRAME @ 2* VP @ R> WTDMA !DMA RUNDMA -VP   ;
 11 : V?    ( --- )
 12         PAD V!   PAD FRAME @ 2* DUMP   ;
 13 : (V.)   VP @   144 OVER DO V? FRAME @ NEGATE +LOOP   VP ! ;
 14 : V.   VP @ 144 > IF (V.) ELSE ." V-stack empty." THEN ;
 15 DECIMAL EXIT


SCR#  25
  Ø ( SPLDGO, CHT, 23-DEC-83)
  1 OCTAL
  2 : SPLDGO   ( START --- )
  3         WTRUN   APRSET   ( Reset AP to clear error flags)
  4         >R 2Ø SPAD -1 2Ø4ØØ R 1+ Ø R>   !RUN RUNAP   ;
  5 : !SPAD   ( HEAP N --- )   SPAD SWAP !PAR   ;
  6 DECIMAL
  7 : FRAME@   FRAME @   ;
  8 : VP@   ( --- INC ADDR )   1 VP @   ;
  9 : SAME   FRAME@   VP@   3 !SPAD   ;
 1Ø : BINARY   FRAME@   -VP VP@ 2DUP +VP VP@ -VP   7 !SPAD   ;
 11 : INPLACE   FRAME@   VP@ 2DUP   5 !SPAD   ;
 12 EXIT
 13
 14
 15


SCR#  26
  Ø ( VECTOR STACK OPERATORS, CHT, 23-DEC-83)
  1 : VØ   SAME   2 SPLDGO   ;
  2 : VDUP   FRAME@ VP@ +VP VP@ 2SWAP   5 !SPAD   4 SPLDGO   ;
  3 : VOVER   FRAME@   -VP VP@ +VP +VP VP@   2SWAP
  4         5 !SPAD   4 SPLDGO   ;
  5 : VDROP   -VP   ;
  6 : V+   BINARY   1Ø SPLDGO   ;
  7 : V-   BINARY   12 SPLDGO   ;
  8 : V*   BINARY   14 SPLDGO   ;
  9 : VSIN   INPLACE   2Ø SPLDGO   ;
 1Ø : VCOS   INPLACE   22 SPLDGO   ;
 11 EXIT
 12
 13
 14
 15
```

C. H. Ting Harris 8Ø

Listing 14. Controlling AP120B array processor (cont'd)

```
SCR#  27
  Ø ( VFILL AND VRAMP, CHT, 3-JAN-84)
  1 : VFILL    ( source --- )
  2         >R    FRAME@ +VP VP@ R>    4 !SPAD    16 SPLDGO    ;
  3 : VRAMP    ( start inc --- )
  4         >R >R    FRAME@ +VP VP@ R> R> SWAP    5 !SPAD
  5         18 SPLDGO    ;
  6 EXIT
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15


SCR#  28
  Ø ( AP STATUS, CHT, 5-JAN-84)
  1 : NAMES    ( line --- )
  2         CR    64 * 29 BLOCK BYTE + CELL 64 TYPE ;
  3 : .REGISTERS
  4         1 NAMES CR    8 Ø DO I @REG 8 U.R LOOP
  5         2 NAMES CR    16 8 DO I @REG 8 U.R LOOP ;
  6 : .APIN
  7         3 NAMES CR    8 Ø DO I APIN 8 U.R LOOP
  8         4 NAMES CR    15 8 DO I APIN 8 U.R LOOP ;
  9 : STATUS    BASE @ OCTAL .REGISTERS .APIN BASE !    ;
 1Ø EXIT
 11
 12
 13
 14
 15


SCR#  29
  Ø ( AP REGISTER NAMES, CHT, 5-JAN-84)        EXIT
  1        PSA       SPD        MA       TMA      DPA SP(SPD)  APSTAT       DA
  2  PS(TMA) IOB(DA)                     DPX      DPY  MD(MA)           TM(TMA)
  3                 SWR        FN    LITES     APMA      HMA       WC     CTRL
  4      FMTH      FMTL     RESET   IFSTAT     MASK    APMAE      MAE
  5 CTL
  6
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15

C. H. Ting                                                  Harris 8Ø
 ??

SCR#  3Ø
  Ø ( APRSET, CHT, 5-JAN-84)
  1 OCTAL
  2 : APRSET    Ø Ø 27 APDR    ;
  3 : APSTEP    1ØØØØ 2 32 APDR    ;
  4 : APCONT    2ØØØØ 2 32 APDR    ;
  5 DECIMAL
  6 EXIT
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15
```

Listing 14. Controlling AP120B array processor (cont'd)