F.E. Hall and A.G. Rocco, Jr.

A Compact Programmable Array Processor

An array processor is a processor optimized to operate on arrays or vectors of data. Typically, a similar operation is performed repetitively on many inputs. Taking advantage of the repetitive nature of this processing, array processors can achieve higher throughput by performing operations simultaneously. This technique is known as parallel computation. Array processors also frequently use a form of parallel computation called pipelining, where an operation is divided into smaller steps, and the steps are performed simultaneously. For example, if we are building a large number of houses, instead of having one crew we can have three; while the first crew digs the cellar and pours the foundation, the next puts up the walls and roof, and the last does the finish work. When one crew finishes, it moves on to the next house site, and the next crew begins work. We could also increase the parallelism by having another set of three crews working on another group of houses at the same time.

While parallelism and pipelining increase the performance of array processors, these techniques also complicate software development. We have developed an array processor that simplifies code generation by decoupling the processes of computation and memory address generation. The processor is completely programmable, as opposed to processors that implement an algorithm in dedicated hardware. This is a general-purpose array processor, and can be used for a variety of real-time signal-processing applications.

We have developed a novel array processor, the Data-Stream Array Processor (DSAP), which provides complete programmability and ease of code generation, while reaping the benefits of parallelism and pipelining inherent in array operations. The DSAP consists of multiple independently programmable array processing elements. Each processing element (PE) divides the array operation into separate and conceptually asynchronous processes of computation and memory addressing, greatly simplifying the process of code generation.

The DSAP was developed to satisfy the volume, weight, and power constraints of radar carried by an Unmanned Air Vehicle (UAV). In such an experimental system, a completely programmable processor is required, yet the processor must have a real-time processing capability of approximately 100 million operations per second (MOPS). Although the DSAP architecture was developed for radar signal processing, it is a general-purpose fixed-point array processor that should be applicable to a wide range of real-time signal-processing tasks where power and weight are at a premium, yet programmability is necessary.

Processor System Architecture

The DSAP architecture consists of one or more array processing element (PE) boards and an application-specific high-speed data-transfer interface, all accessible by a general-purpose host computer residing on a VME bus (Fig. 1). The VME bus is an industry-standard 32-bit microprocessor bus that makes up the backplane of the so-called host computer that in an operational system is responsible for signal-processor program loading and control, as well as subsequent processing of the results of the signal processing. The VME-bus backplane and processor cards are commercially available. The VME bus is connected to the signal-processor bus via a Lincoln Laboratory–built VME-bus

interface card. In the DSAP developed for the UAV radar, the high-speed interface consisted of A/D converters and a radar-timing generator. The backplane, which contains a clock-generation circuit, interconnects the PEs with several data buses. The VME-bus interface provides signal redriving and address-space mapping between the VME-bus and the signal-processor host bus. Through this interface, a VME-bus master can access the entire DSAP state.

By providing concurrent computation and I/O operations, the DSAP architecture provides the efficiency necessary for real-time processing. By using multiple identical array processors, a real-time program can be algorithm-partitioned, with each PE performing a particular set of operations on the data set, or a program can be data-partitioned, with each PE running the same algorithm on a part of the input data. The inter-PE communication paths support either of these techniques.

Processing-Element Architecture

The computational components of a PE (Fig. 2) are an arithmetic processor (AP) and two address generators (AG) — the input AG and the output AG. A large two-port data memory stores input data and results. A third AG, which functions independently of the computational processors, transfers data between the data memory and the world external to the PE. This data transfer uses the second port of the data memory and thus may proceed without interfering with processing.

The AP performs the adds, multiplies, and other operations for computations. It contains three arithmetic functional units and a large multiport register bank, which stores intermediate results. The AP does not, however, directly access the data memory for either its inputs or its outputs; rather, it processes streams (a stream is an ordered series of data tuples) of operands and emits streams of results.

Each AG consists of an arithmetic unit, a two-port register bank, and iteration-control hardware that implements nested program loops efficiently. The three AGs are identical, but they perform three distinct functions within the stream processing. One AG transfers data from memory to the AP. The second AG stores the AP output stream in the data memory. The third AG handles transfers between the data memory and

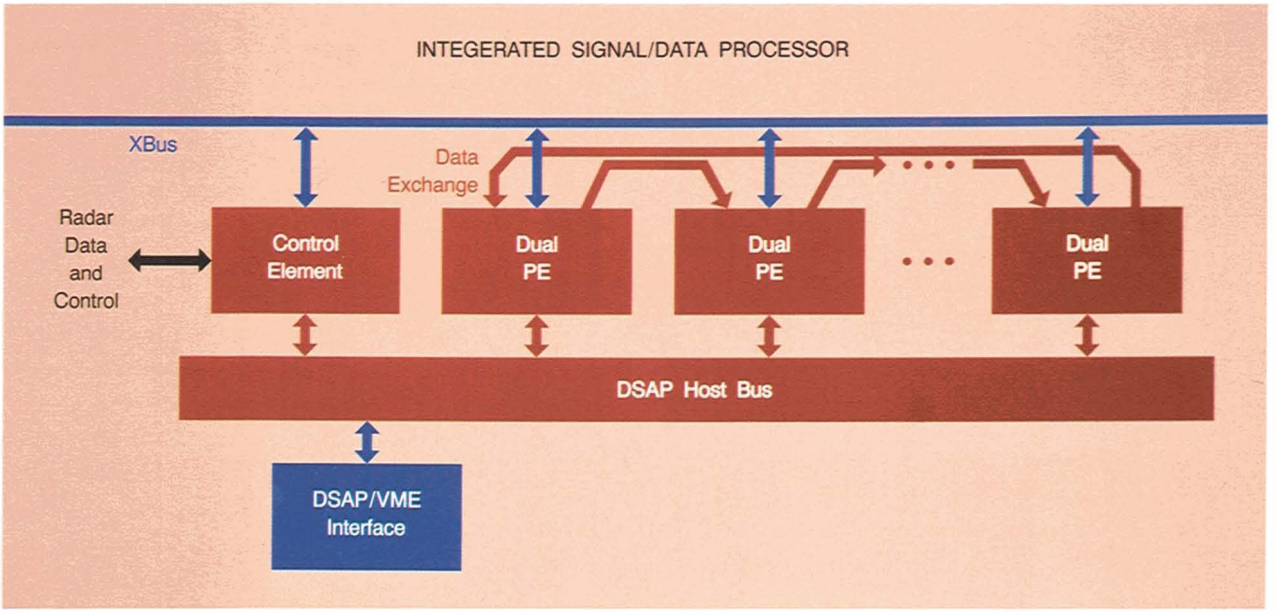


Fig. 1 — The processor elements of the Data-Stream Array Processor (DSAP) communicate with one another and with the outside world through the three buses shown in this block diagram: the external bus (XBUS), the data-exchange bus, and the DSAP host bus.

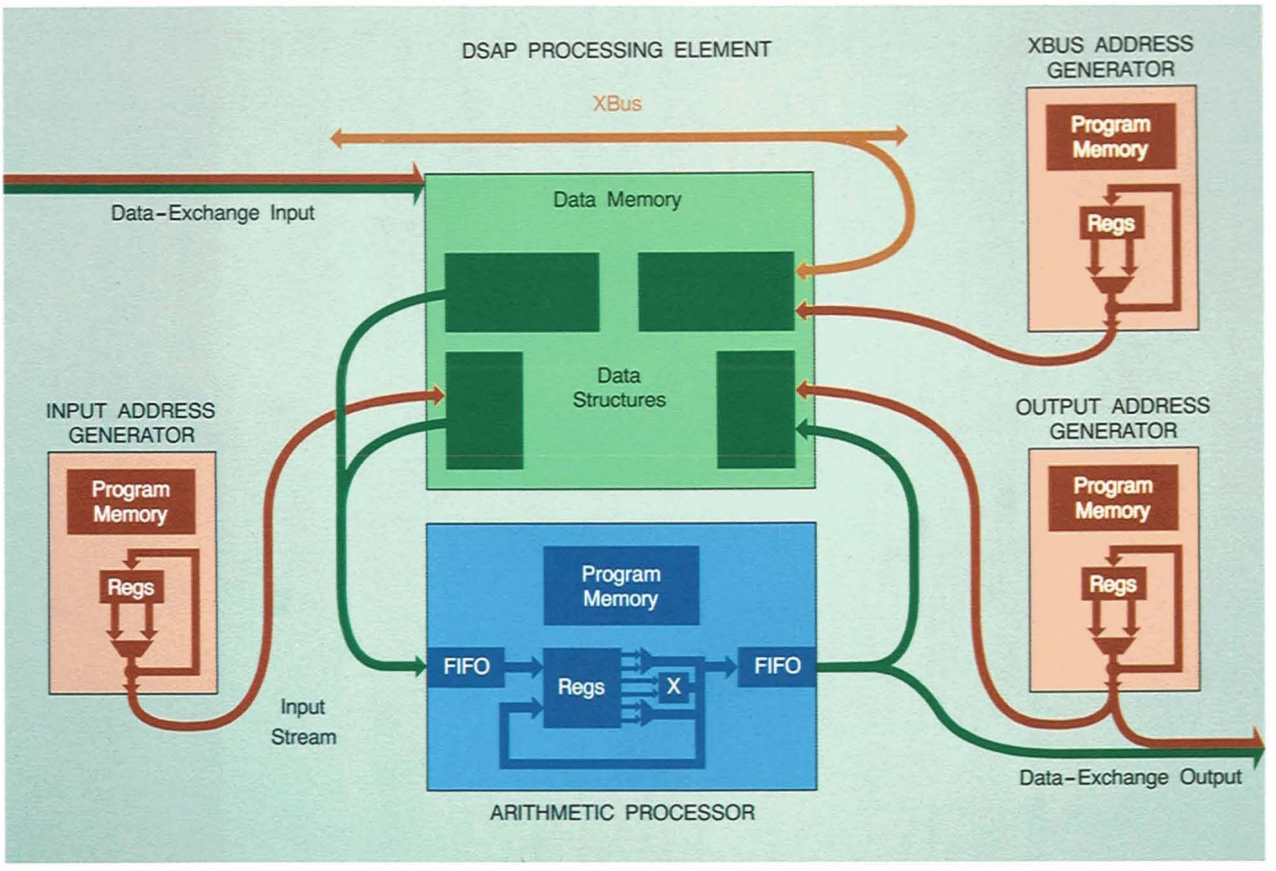


Fig. 2 — Data flow within PEs is controlled by the input, output, and XBUS AGs.

a backplane bus called the external bus (XBUS).

The partition of data processing and address calculation into asynchronous processes is one of the fundamental innovations of the DSAP architecture. Address calculation is separated from data processing; the AP processes data and the AGs calculate addresses. The interface between the three processes of input, calculation, and output are data streams.

Many signal processors contain address-calculation hardware, but the process of address generation, and therefore of data-memory access, typically runs in lockstep with the data processing. Memory addressing, as well as computation, is often coded into a very wide, horizontally microcoded instruction word. Since addressing and computation must operate in instruction lockstep, and since the code fragments that perform these different functions are often of different lengths or shapes (in the sense of nested loops), hardware resources are frequently left idle — except for the few applications for which the hardware was optimized. To complicate code generation further, there is typically a pipeline delay between such instructions and the resultant memory accesses.

In the DSAP architecture, the three processes of input-stream formation, computation, and output-stream storage are conceptually asynchronous. In fact, the processes are executed on three independent processors, each with its own hardware state and thread of execution. All interprocessor synchronization and the memory-access pipelining are hidden by the hardware. To smooth out any relative timing differences between the three processes, first-in/first-out (FIFO) buffers store a few stream items, which improves the overall hardware utilization.

Processing-Element Hardware

The principal hardware components of a PE are the three identical AGs, the AP, a dual-port data memory, and two gate arrays that contain

Very Large-Scale Integration Device Development

The first AG and AP chips (Figs. A and B, respectively) were designed in an NMOS process using the MAGIC layout editor, the MEXTRA circuit extractor, and the RNL transistor-level simulator. The AG and AP contain approximately 60,000 and 75,000 transistors, respectively. The chips were laid out almost entirely by hand, and development of the two chips required nearly three person-years of effort. A major problem in completing the layout was that the development continuously pushed the limits of the design tools. The AG and the AP were both fabricated in single-metal NMOS processes; the AG had a 3-μ geometry and the AP had a 2.25-μ geometry. The execution speed of the NMOS devices is approximately 3.5 million instructions per second (MIPS).

The very large-scale integration (VLSI) devices have now been implemented with a commercial silicon compiler that takes a schematic input and optimizes a standard cell layout, with minimal designer interaction. The design time for the second implementation was reduced to about nine months for each chip, and the execution-speed goals were achieved. In addition to improving the speed, the capabilities of the integrated circuits were increased. Program-memory sizes for the AG and AP were doubled, program memory was changed from dynamic to static, and the multiplier, which had been external to the AP, was brought on-chip. Moreover, the greatest advantage of the silicon-compiler approach is that the design is in a technology-independent form, so future implementation in faster integrated-circuit processes should be fairly straightforward.

The new version of the AG is implemented in a 1.5-μ double-metal CMOS process. It contains approximately 238,000 transistors, of which about 200,000 are used for static RAMs.

As of January 1989 we do not yet have the CMOS versions of the AG and AP. The CMOS AG has been simulated at the 10-MIPS rate, is in fabrication, and is expected in April. The AP is in the final stages of design and is expected sometime in the summer. With the current NMOS AGs and APs we are able to run at between 3 and 3.75 MIPS (we have speed-sorted the integrated circuits).

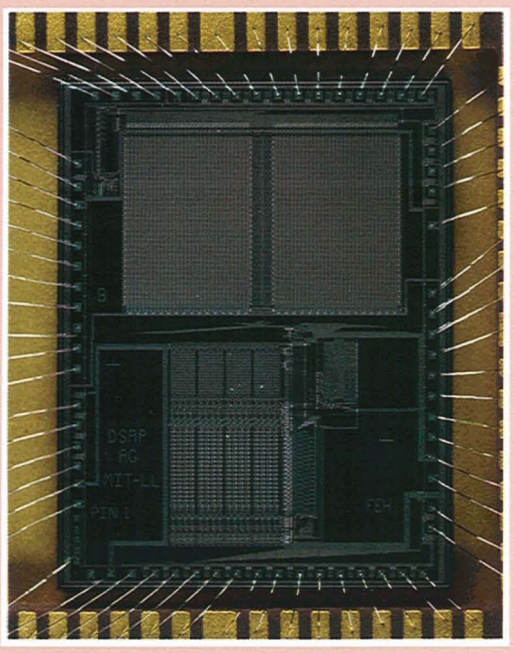


Fig. A — The address generator is a 60,000-transistor, 3-μ-NMOS integrated circuit. The die measures 7 mm × 9 mm and is packed in a 68-lead package.

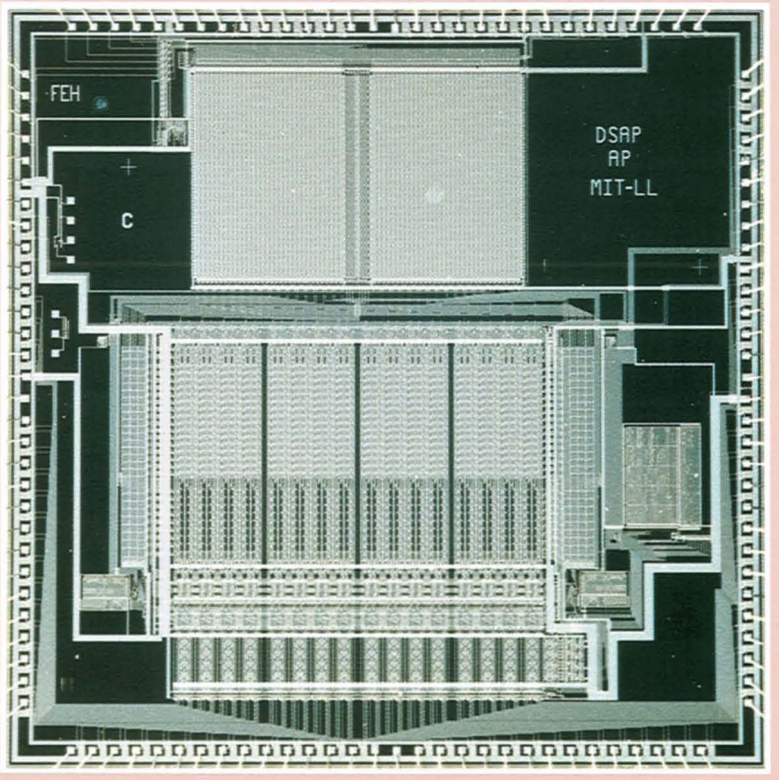


Fig. B — The arithmetic processor is a 2.25-μ-NMOS device that has 75,000 transistors packed on an 11-mm² die. The integrated circuit is mounted in a 120-lead pin-grid-array package.

the logic necessary to synchronize the processors and their accesses to the data memory. The AGs that generate the AP input and output streams are designated "AGI" and "AGO," respectively. The AG designated "AGX" moderates transfers between the data memory and the external world via the 32-bit-wide data-only XBUS. Both the AGs and the AP are implemented on custom VLSI devices, described in detail in the box, "Very Large-Scale Integration Device Development." All processors on the PE board operate at 10 million instructions per second (MIPS). The processors are asynchronous in the sense that each has its own thread of program execution, but in the hardware domain they are driven by a common clock and interconnected by completely synchronous interfaces.

Each PE is connected to the external world by three data paths: the XBUS, the data-exchange network, and the host bus.

The XBUS is a 32-bit bus that interconnects all PEs on the backplane and effects transfers at a rate of 10 million words per second. The XBUS can be used to get data in and out of the processor, as well as between PEs. Transfers on the XBUS are accomplished by the execution of a send operation by the AGX on the sending PE, while the receiving PE or PEs are waiting for data via receive instructions. No address travels with the data; the AGXs on both ends of the transfer produce the local data-memory address. For

fixed-length XBUS transfers, the AGXs can maintain synchronicity by all counting XBUS transfers and receiving the ones they need. For variable-length transfers the length can be sent as part of the data or the AGXs can be synchronized after each transfer. Synchronization is accomplished by using one technique or a combination of several techniques. There are four event lines on the backplane, which may be set, cleared, and monitored to mark various time epochs in a program's execution. For data streams coming in from outside the processor, the interface can send an end-of-stream (EOS), which will force all the AGXs on the PEs to vector to a predetermined location. AGXs can also be synchronized with a stop (rendezvous) request, which causes them to wait until the stop request is granted. Once all the AGXs have requested a stop and an enable bit has been set, all the AGXs simultaneously receive a grant allowing them to continue at the same time. Under software control, the event lines, the EOS signal, and/or a stop request can be used to synchronize all the AGXs to a particular starting point.

The second data path interconnecting PEs is the data-exchange network, a unidirectional link from one PE to another. Although the present DSAP implementation connects the PEs in a circular array that has no latency, the electrical protocol is compatible with more complex interconnection schemes — such as a butterfly network that might have one or more processor cycles of pipelining for the data transfers. The data-exchange path operates at a 5-million-words-per-second transfer rate; the address within the destination PE's memory location where the datum is to be stored accompanies each transferred datum. Data-exchange transfers are initiated by the AGO on the sending PE. The operation can be thought of as a diversion of the AP's output stream (or selected items thereof) into the data memory of another PE.

The third data path is the host bus, which is a logical extension of the VME bus. Through the host bus, a VME-bus master can directly access the data memory or the program memory of any processor (AGX, AGI, AGO, or AP) on any PE board. In addition to memory accesses for program loading and data transfer, several host-bus signals control DSAP execution. These sig-

Backplane and VME-Bus Interface

The DSAP PE boards are mounted on a custom backplane that provides the clock signals and the inter-PE buses: XBUS, host bus, event lines, and data-exchange network. Since the clock signals contain fundamental frequency components at 80 MHz, special care was taken to provide impedance matching to minimize ringing.

Each dual-PE board occupies 1 MB of host-bus address space. The address space consists of 512 kB for each PE, one half for the data memory, and the other half for the program memories of the AGs and AP (not exhaustively decoded).

The VME-interface card plugs into a VME-bus socket and connects to the DSAP backplane through flat cables. The VME interface maps the host-bus address space into VME address space. Not all VME systems offer the luxury of devoting many megabytes to a DSAP; PE pointers and mapping registers can optionally reduce the address space to the size of one PE, or any power of two PEs. The DSAP base address in VME space is jumper-selectable.

The VME interface has VME-bus–accessible circuits that reset and control the overall DSAP operation, including a single stepping capability for software testing.

VME-bus interrupts can be generated by transitions of the DSAP event lines. An interrupt can also be generated when all PEs conclude their processing and indicate the end of processing by executing a stop instruction. The interrupt level is jumper-selectable and the interrupt-vector location is software-selectable.

nals are discussed in more detail in the box, "Backplane and VME-Bus Interface."

Each PE's data memory is organized as 64k 32-bit words. A data-memory word can be considered as two 16-bit words that represent a complex fixed-point number. All data-stream transfers between the AP and the data memory are 32-bit values, regardless of their representation. The data memory is dual-ported by time multiplexing; that is, the memory operates at 20 MHz, twice the instruction rate of the computational units.

A pair of Lincoln Laboratory–developed gate arrays, which provide a 32-bit crossbar switch and a finite-state machine (FSM) to generate the timing signals, moderate the contenders for the data memory. The contenders are divided into two groups: one for each half-cycle, or logical-memory port. One half-cycle is dedicated to processing. Highest priority is given to direct accesses from the AP; next-highest priority goes to data-exchange write operations from another PE. A lower priority is assigned to AP stream output and AGO accesses. Finally, AP stream inputs and AGI accesses receive the lowest priority. The other half-cycle is dedicated to I/O, during which AGX accesses are given priority over host-bus accesses.

Data-memory scheduling is effected through a protocol of requests and grants. During each machine cycle, each processor (AG or AP) encodes the memory accesses for which it will be ready at the conclusion of the cycle. The gate array's FSM examines all the memory-access requests and grants those of highest priority. Note that some requests require the concurrent availability of more than one processor for execution. An AP input-stream read cycle, for instance, requires that there be space in the AP input-stream buffer and that the AGI generate the memory address. When both of these conditions are satisfied, the FSM grants both processors (the AP and the AGI) an access and schedules the memory cycle.

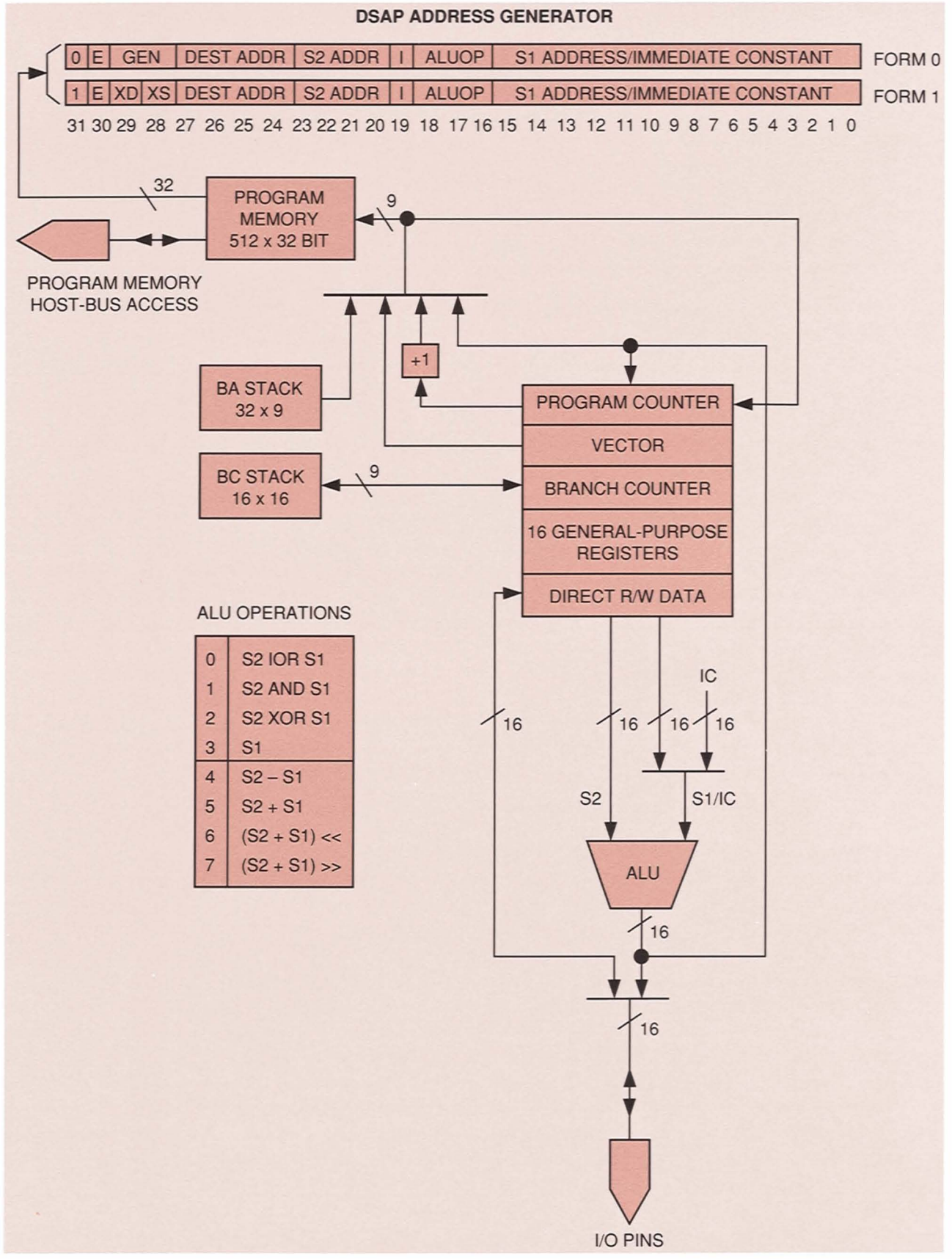
If the AP input FIFO is full but the AGI is ready, the AGI will not be granted an access, so its program execution will pause until the grant is received. The AP can simultaneously request input- and output-stream transfers; the FSM determines which transfer to schedule based on the AG requests and on priority. Memory accesses are pipelined over two machine cycles; the order of events is completely deterministic and synchronous once a transaction is granted. If a request is denied, the processor may pause, depending on what the pending request and the next instruction are. For example, if an AGI generates an address for AP input, it won't actually pause until the program comes to another instruction that will generate a request. The mechanism of requests and grants hides the details of interprocessor synchronization and the logistics of data-memory accesses from the programmer.

Address Generator

The purpose of each of the three identical AGs is to produce the 16-bit values used as PE data-memory addresses. The AG is a self-contained three-address processor with on-chip program memory; its architecture is straightforward (Fig. 3). The AG's sixteen 16-bit data registers can be read simultaneously from two ports. One of these ports can be replaced by an immediate constant from the instruction word. The other port can read various special registers such as the program counter. The two operands feed an eight-function ALU, which writes its results back into the data registers at a third address. The ALU result can also be used by the AG as an address for the PE board.

Because the AGs don't include hardware for implementing special array-addressing schemes (e.g., bit-reversing hardware for FFTs), the general-purpose nature and broad applicability of the AGs is maintained.

There is special hardware in the AGs that facilitates program loops. A dedicated branch counter can be loaded with a value indicating the number of iterations in a loop. A begin-loop instruction loads the branch counter and copies the program address of the first instruction of the loop into a branch-address (BA) register (actually the top of the BA stack). A bit in the instruction word is assigned to indicate the final


DSAP ADDRESS GENERATOR
0 E GEN DEST ADDR S2 ADDR I ALUOP S1 ADDRESS/IMMEDIATE CONSTANT
FORM 0
1 E XD XS DEST ADDR S2 ADDR I ALUOP S1 ADDRESS/IMMEDIATE CONSTANT
FORM 1
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
32
PROGRAM MEMORY 512 x 32 BIT
9
PROGRAM MEMORY HOST-BUS ACCESS
+1
BA STACK 32 x 9
BC STACK 16 x 16
9
PROGRAM COUNTER
VECTOR
BRANCH COUNTER
16 GENERAL-PURPOSE REGISTERS
DIRECT R/W DATA
ALU OPERATIONS
0 S2 IOR S1
1 S2 AND S1
2 S2 XOR S1
3 S1
4 S2 – S1
5 S2 + S1
6 (S2 + S1) <<
7 (S2 + S1) >>
IC
16
16
16
16
S2
S1/IC
ALU
16
16
I/O PINS

DSAP ADDRESS GENERATOR

REGISTER ADDRESS SPACE			
	SOURCE 2	DESTINATION	DST WITH E BIT
0 - F	GEN REGISTERS	GEN REGISTERS	END LOOP
10	BRANCH COUNTER	BEGIN LOOP	POP BC/BA STK
11	BRANCH COUNTER	BRANCH COUNTER	STOP
12	VECTOR	VECTOR	END LOOP
13	I/O DATA	I/O DATA	END LOOP
14		WRITE LOW	END LOOP
15		READ LOW	END LOOP
16		WRITE HIGH	END LOOP
17		READ HIGH	END LOOP
18		CALL SUB	RETURN
19		BR IF ALU = 0	POP BA STACK
1A	BC STACK PNT	BR IF ALU ≠ 0	BC STACK PNT
1B	BA STACK PNT	BR IF ALU ≤ 0	BA STACK PNT
1C	CHECKSUM MSBs	BR IF ALU < 0	
1D	CHECKSUM LSBs	BR IF ALU ≥ 0	
1E	STATUS/CONTROL	BR IF ALU > 0	STATUS/CONTROL
1F	CUR PC + 1	BRANCH ALWAYS	

Fig. 3 — Block diagram and instruction summary of the address generator.

instruction in the loop. When an instruction with this bit set is executed, the branch counter is automatically tested for zero in parallel with all other instruction operations. If it is zero, execution continues past the end of the loop; if it is nonzero, the branch counter is decremented and execution branches back to the top of the loop. Once a loop is initiated, iteration proceeds with no overhead associated with the loop, allowing the AG to generate an address every machine cycle. Last-in/first-out stacks associated with both the branch counter and the branch-address register handle nested loops.

Along with each address generated, there is a 2-bit tag. The tag tells the PE board what to do with the address. As shown in Table 1, the disposition of the different types of addresses depends on which AG is producing it.

In addition to performing data-stream operations, any AG can directly access a 16-bit word in data memory. The current ALU result can be used as a memory address, and the data can be transferred to or from the I/O data register. Direct accesses are typically used to load pointers or parameters from data memory, or for diagnostic purposes.

The AG was developed for the PE board, but it is a general-purpose microcoded sequencer that is suitable for other possible applications. In the prototype DSAP constructed for the UAV radar application, for example, two AG chips were used as a programmable radar-timing generator.

Arithmetic Processor

The arithmetic processor is a special-purpose VLSI device that performs array operations on streams of data. It efficiently supports complex arithmetic on 16-bit integers or fractions. Sup-

Table 1. Disposition of Address Tags

Tag Type	*ADDRESS SOURCE*		
	Input AG	*Output AG*	*XBUS AG*
Type 1	Read a Value for the AP Input Stream	Write an Output-AP Stream Item to Data Memory	Read a Value from the XBUS
Type 2	Tag This AP Input Datum as End-of-Stream	Send the Stream Item to Another PE over the Data-Exchange Network	Wait for an XBUS Transfer to Occur
Type 3		Write the Item into Local Data Memory and Retain It for a Subsequent Transfer to Another PE	Send a Value to the XBUS

port for multiple-precision operations is available. In each instruction cycle, the AP can use two identical ALUs and a parallel multiplier to perform as many as three arithmetic functions. Analysis of several common signal-processing tasks indicates that this combination of functional units provides a well-balanced processing environment and that neither computation nor data accesses present a bottleneck to system throughput.

A typical stream-processing module on the AP is implemented as a program loop (see "Appendix: Programming Example"). In a typical program loop, a set of input-stream values are read, arithmetic operations are performed, and finally either output-stream values are produced or values are accumulated. The loop is repeated until it is interrupted by an input datum tagged by the AGI program as the EOS.

Efficient AP code often involves pipelining a computation on one set of inputs through more than one loop iteration. For instance, a complex multiplication operation that involves four multiplications and two additions can be implemented in a four-instruction loop, by deferring the additions and the outputs until the next time around the loop. At that time, the multiplier can be concurrently working on the next input-stream values.

A unique design innovation of the AP architecture is a shifting-register bank that stores intermediate results. This design greatly simplifies code generation for pipelined operations. The registers can be thought of as a FIFO, in which new values (from the ALUs, multiplier, input stream, or any combination) are written into the top (i.e., lowest-order register) and push existing data down (to higher-register addresses). The oldest values disappear off the bottom of the register bank. Writing is always sequential. Nonetheless, by specifying register addresses in an instruction word, the 16 registers can be read randomly.

The AP architecture (Fig. 4) is built around the register banks. Each register is 32 bits wide, but, for most purposes, the two 16-bit halves (called the X and Y sides) can be accessed independently. In every instruction cycle, the 64-bit instruction word specifies six independent register-read addresses, three from the X side and three from the Y side. The register address space is described by five bits for each

read port, which is selected from the 16 shifting registers, eight general registers, and special registers that are used for program flow control and direct data-memory accesses. The three operands from each side pass through swapping multiplexers; in these multiplexers, the X and Y parts of the corresponding words can be exchanged. Two X ports feed the X ALU and two Y ports feed the Y ALU. One port from each side is directed to the multiplier.

ALU operations are independently selectable for the two sides and include, in addition to the usual suite of operations, such selection functions as maximum, minimum, and absolute value. The ALU results pass through a 1-bit left- or right-shifter; all results are available for storage in the register bank within a single instruction cycle. As a result of the arithmetic functions, four 16-bit values are available for storage, two from the ALUs and a 32-bit product. As discussed above, up to three items can be written into the shifting-register bank on each instruction cycle, and the two sides can be independently written. For instance, the least significant part of a product can be ignored and the most significant part written into one of the sides of the shifting registers. The data in the shifting-register bank move down corresponding to the number of items stored, and the items are always written in reverse alphabetical order: product, input stream, and ALU result; or *P, I, A.*

The general and special registers are written as a single 32-bit entity from both ALU results. The write address is specified by a 4-bit address field in the instruction.

Two of the special-register addresses correspond to the output stream. One address is for ordinary stream items and one tags the item written as the EOS. When the EOS is stored in data memory with the assistance of the output AG, the AG's program flow is interrupted and forced to branch to a pre-specified location. The input and output streams pass through FIFO memories in the AP, effectively decoupling the AP and AGs. The AP program can both read an input and write an output item in the same instruction, although there is actually only one port to the data memory.

Two of the 32-bit special registers, which are actually made up of four 16-bit registers, are used for flow control. Two of the 16-bit registers are branch-address registers, and, by setting the appropriate value in the 2-bit flow-control instruction field program, flow can be directed to branch with no overhead. The third register is the program counter; a read returns the current program-memory address plus one, and a write causes a branch. The fourth register is the EOS vector register. Program execution branches to the address contained in this register when an input-stream item is read that the AGI has tagged as the EOS.

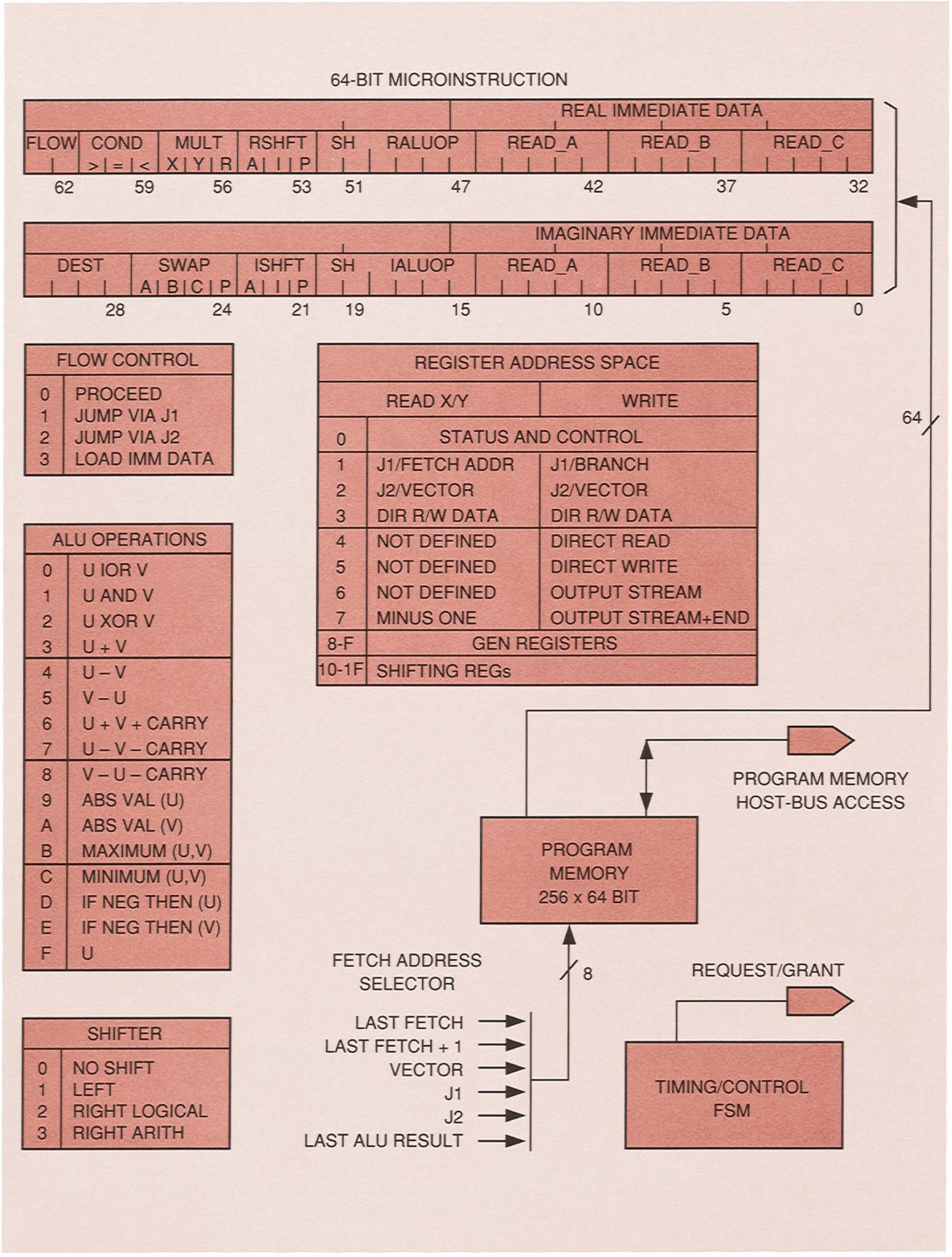
The data register is a special register that is used in direct data-memory accesses. These accesses circumvent the stream FIFOs and let the AP program access any data-memory location. This facility is useful for parameter loading, table-lookup operations, and diagnostics. The Y ALU result is used as the data-memory address, and data are transferred to or from the 32-bit data register. Two write-only special-register addresses initiate the direct accesses. One address initiates a read, and the other initiates a write.

The write operations of the ALU data to the general or special registers may be made conditional upon the arithmetic result of the X ALU on the previous cycle. The 3-bit condition field of the AP instruction can test the previous result for zero, negative, or various combinations, and if a condition is met, a write operation can be specified. This capability is useful for conditional branches, although conditional branches are in fact quite rare in application code, or for conditional writes to the output stream to output-only selected items.

Unmanned–Air-Vehicle Radar Application

A DSAP has been built for a prototype moving-target–detection radar carried by a UAV. Figure 5 shows the DSAP chassis, which includes a custom radar interface and six dual-PE boards. The radar interface generates radar-timing signals and sends digitized radar data on the XBUS in real time.

The dual-PE printed-circuit board, shown in


64-BIT MICROINSTRUCTION
REAL IMMEDIATE DATA
FLOW
COND
> | = | <
MULT
X | Y | R
RSHFT
A | I | P
SH
RALUOP
READ_A
READ_B
READ_C
62
59
56
53
51
47
42
37
32
IMAGINARY IMMEDIATE DATA
DEST
SWAP
A | B | C | P
ISHFT
A | I | P
SH
IALUOP
READ_A
READ_B
READ_C
28
24
21
19
15
10
5
0
64
FLOW CONTROL
0 PROCEED
1 JUMP VIA J1
2 JUMP VIA J2
3 LOAD IMM DATA
REGISTER ADDRESS SPACE
READ X/Y
WRITE
0 STATUS AND CONTROL
1 J1/FETCH ADDR J1/BRANCH
2 J2/VECTOR J2/VECTOR
3 DIR R/W DATA DIR R/W DATA
4 NOT DEFINED DIRECT READ
5 NOT DEFINED DIRECT WRITE
6 NOT DEFINED OUTPUT STREAM
7 MINUS ONE OUTPUT STREAM+END
8-F GEN REGISTERS
10-1F SHIFTING REGs
ALU OPERATIONS
0 U IOR V
1 U AND V
2 U XOR V
3 U + V
4 U – V
5 V – U
6 U + V + CARRY
7 U – V – CARRY
8 V – U – CARRY
9 ABS VAL (U)
A ABS VAL (V)
B MAXIMUM (U,V)
C MINIMUM (U,V)
D IF NEG THEN (U)
E IF NEG THEN (V)
F U
SHIFTER
0 NO SHIFT
1 LEFT
2 RIGHT LOGICAL
3 RIGHT ARITH
PROGRAM MEMORY HOST-BUS ACCESS
PROGRAM MEMORY 256 x 64 BIT
FETCH ADDRESS SELECTOR
8
REQUEST/GRANT
LAST FETCH
LAST FETCH + 1
VECTOR
J1
J2
LAST ALU RESULT
TIMING/CONTROL FSM

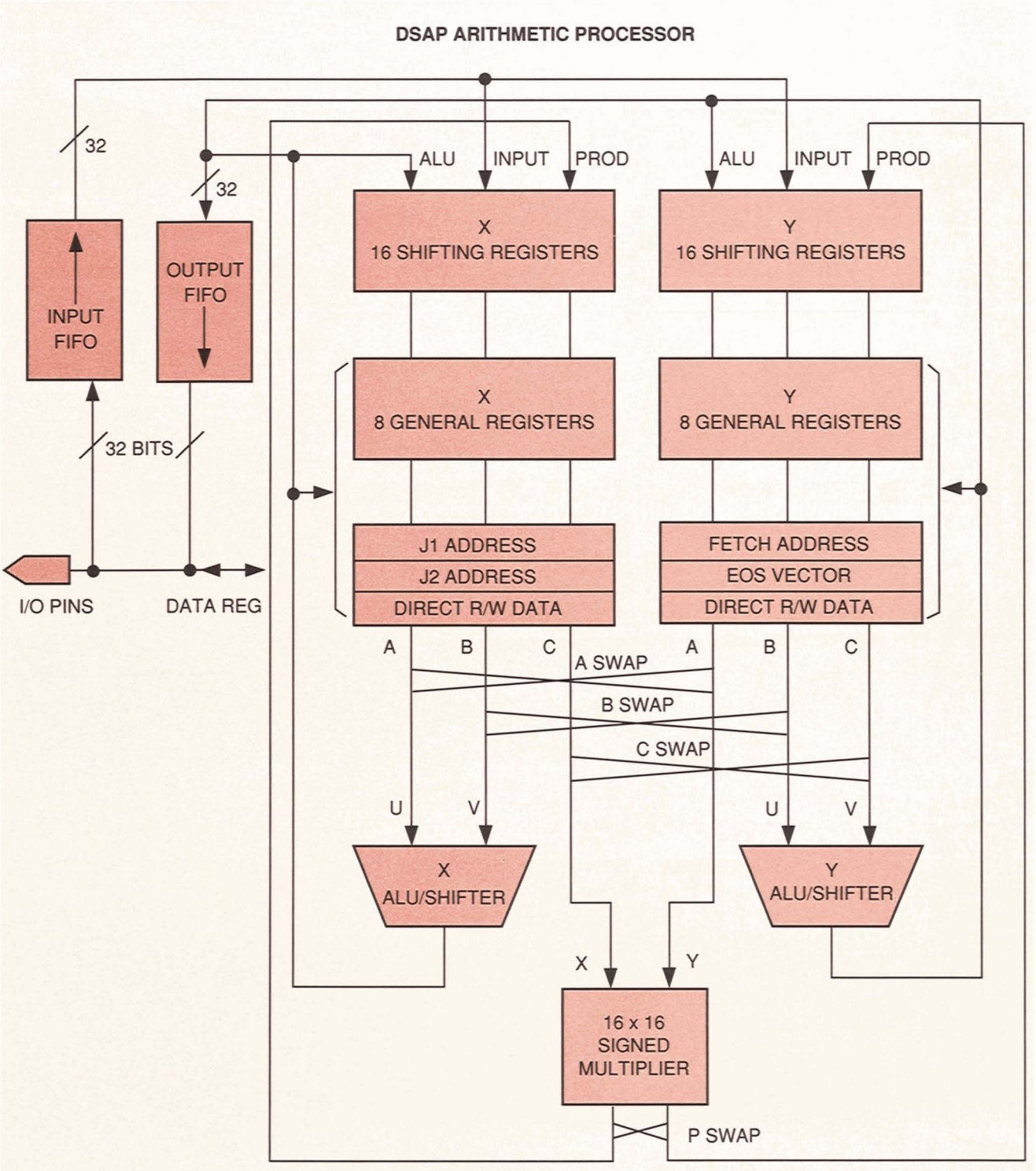


Fig. 4 — Block diagram and instruction summary of the Arithmetic Processor.

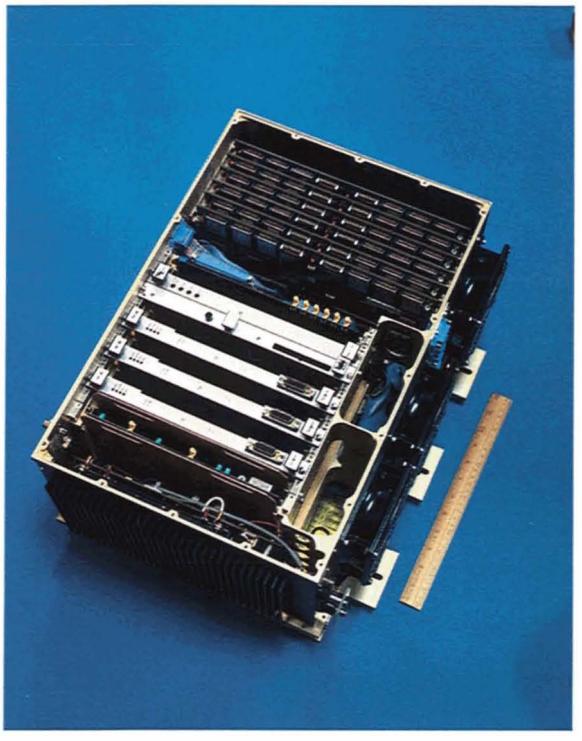
Fig. 6, measures 6.5×12 in^2. In addition to data memory, the AGs, and the AP, the board contains several discrete integrated circuits. The integrated circuits redrive clock lines and other backplane signals. Most integrated circuits are mounted in small-outline packages, which reduce the board size, yet are easier to handle than leadless chip carriers (LCC). The data memory, however, is assembled with LCC 16k $\times$ 4-bit static RAMs. The RAMs are mounted on two sides of a ceramic board and inserted into pin sockets on the PE board. The dual-PE board consumes 17 W of 5-V power and weighs 29 oz.

The DSAP shares the chassis with a 10-slot VME backplane (Fig. 5). The backplane contains commercial 68020-based processors and interface boards, which provide interfaces to platform location and communication equipment, as well as post-detection processing of radar data.

The DSAP package is a complete radar-processing system. It provides clean moving-target location reports, which can be transmitted over a low-bandwidth data link to a ground-based display.

Radar signal processing begins when the radar-interface card sends 32-bit complex numbers, which represent digitized radar-video data, down the XBUS. An EOS from the radar-interface card signals the boundary between radar pulses, and the AGXs on all PE boards count XBUS transfers from the EOS. The PEs are individually programmed to store a group of samples from each radar pulse. The processing is range-partitioned; each PE processes a subset of the total range swath of the radar.

When the data from 64 radar pulses have been collected, the AP, AGI, and AGO commence the moving-target–detection processing on that batch of data. At the same time, the AGXs on all boards switch to a second block of memory



UAV PROCESSOR

- Integrated Signal and Data Processor
- Programmable
- 360 Million Operations per Second
  - — 6 Dual-Processing-Element Boards
  - — 3 VME 68020 Single-Board Computers
  - — Integral Radar Timing and Control Unit
  - — Integral Radar A/D Converters
- Power – 400 W
- Weight – 55 lb (25 kg)
- Volume – 1.6 ft^3 (.05 m^3)
  (13-1/2" x 10-1/4" x 20")

Fig. 5 — This Data-Stream Array Processor operates at 360 MOPS and includes six dual–processor-element boards, three 68020-based single-board computers, integral radar timing and control unit, and integral radar A/D converters.

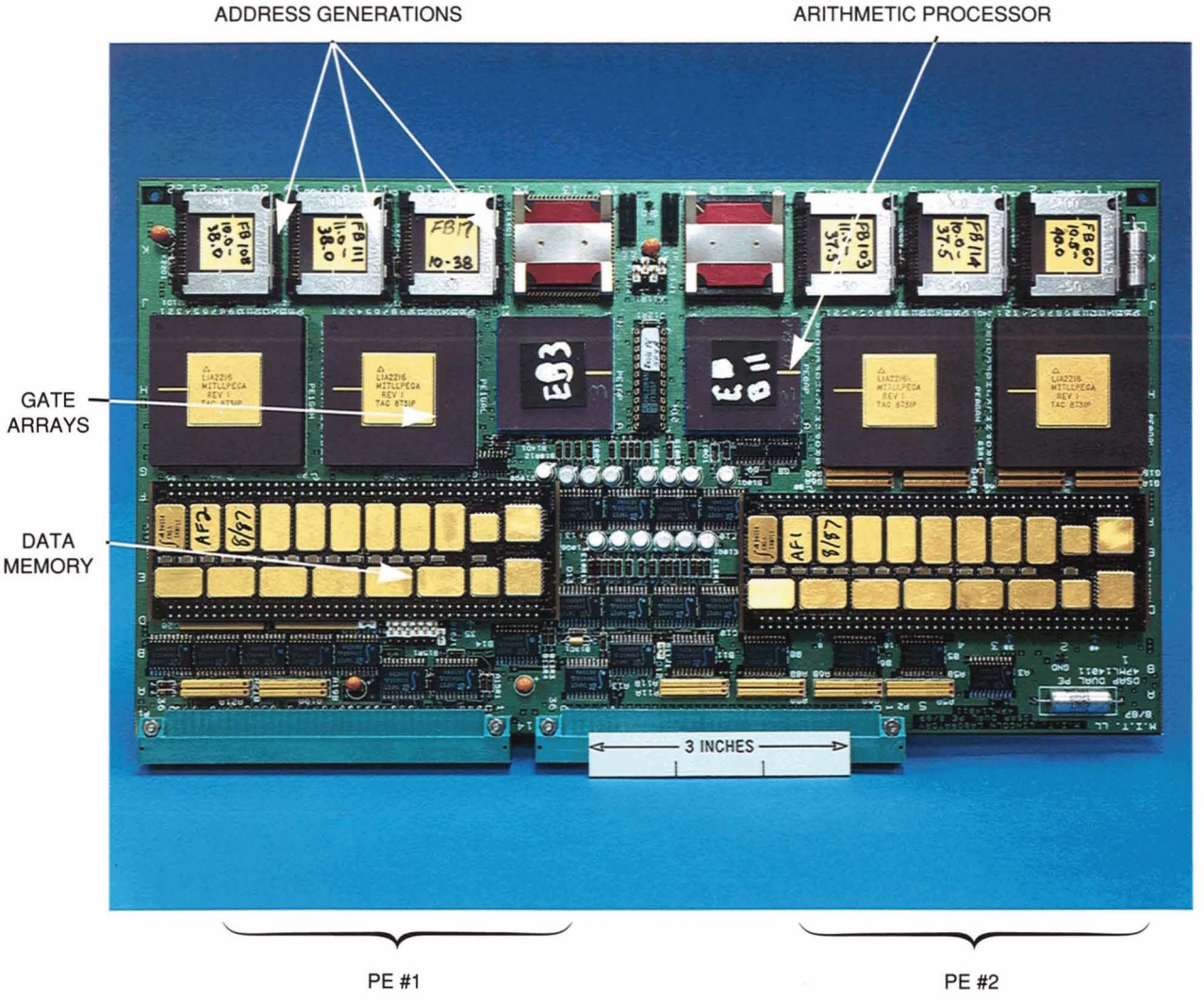


Fig. 6 — Each dual–processing-element board is a 12-layer printed-circuit board. The dual-PE boards include six address generators, two arithmetic processors, four custom gate arrays, and two data-memory modules.

and concurrently collect data from the next 64 pulses.

The moving-target–detection processing consists of a series of array-processing steps. First, an FFT is performed on the 64 values in each range cell. A radix-4 FFT kernel is used to maximize efficiency and the computation is permuted to use a non-bit-reversing addressing method. The complex FFT results are then passed through a magnitude approximation, resulting in a range-Doppler matrix; the Doppler values indicate radial velocity relative to the radar. Following this, the data are accessed along the range dimension of the matrix for each Doppler cell and an average signal strength is computed. A second pass through these data detects cells whose signal strength is significantly higher than average. The AP program uses the conditional-store feature to send only those range-Doppler cells which cross the threshold to its output stream. By selecting only the above-threshold cells, the volume of data is limited only to those cells which contain possible targets of interest.

When the processing is concluded, the DSAP generates a VME-bus interrupt. The interrupt indicates that the target data can be picked up for subsequent stages of processing performed in the general-purpose VME-based processors.

One figure of merit for analyzing processors

such as the DSAP is the amount of its peak capability that is used in real applications. The AP on each PE can perform three operations per instruction, indicating a peak capability of 30 MOPS. An examination of the radar signal-processing application gives a utilization that is better than 75%, or a true processing power of over 22 MOPS. Note that the 22 MOPS given here do not include the processing done by the AGs in generating addresses (3 × 10 MOPS peak).

Another application-specific measure of performance is the number of radar range samples per second that can be processed by moving-target detection. The complete processing algorithm, including the steps — input validity checking, weighted 64-point FFT, magnitude approximation, threshold calculation, and primitive target report generation — requires 17.5 processor cycles per input sample. This translates to 1.75 μsec per sample, at a 10-MIPS execution rate, or 570,000 input samples per second per PE.

Conclusions

The DSAP architecture provides a completely programmable array processor. Each PE in the DSAP is capable of over 20 MOPS in actual operation, yet requires less than 10 W and 24 in^3. The performance capability of the DSAP was achieved by using commercial VLSI and gate-array processes and standard printed-circuit boards, thus minimizing both development and production costs. The data-stream architecture of the PE board offers complete flexibility in programming and greatly simplifies code generation.

Future development efforts will focus on expanding the existing AP architecture to a 32-bit floating-point processor. Reductions in VLSI geometry sizes should permit this doubling of the data-path size.

Acknowledgments

The authors would like to thank Quentin Klein for his contributions to this architecture. We would like to thank Dave Craska, John Duncan, Don Malpass, Tony Marsala, Jim Noonan, John Reilly, and Paula Rygiel for their aid in writing support/diagnostic software and debugging the hardware. We would also like to thank Gerald Morse and Ed Schwartz for their continued support and guidance.

Appendix: Programming Example

This section describes the process of implementing a signal-processing computation on a DSAP. To limit the length of the example, we will assume that the data are already in the data memory of a PE, and will concentrate on the programming of the AP, AGI, and AGO. We will use this example to illustrate the philosophy and the use of the DSAP's stream architecture.

Generally, programming of the processors on the PE board is done in a macroassembly language, which hides some of the details of the AP, including shifting-register allocation and use of swaps. For this example, a pseudo-English tabular form will relate the instruction fields of the AG and AP (shown in Figs. 3 and 4).

The first stage of software implementation for a DSAP is to consider the signal-processing problem as a series of array-processing steps. An array-processing step can be loosely defined as one or more array operations that can be performed within an AP program loop; a step therefore defines the semantics of the input and output streams. Once the stream ordering has been dictated and the organization of structures in the data memory is known, the input and output AG programs can be written.

The programming example describes a complex convolution of two equal-length vectors

$$(P * Q)_i = \sum_{j=0}^{i} P_j \, Q_{i-j}$$

where P and Q are the arrays to be convolved, i and j are the indices, running from 0 to $N-1$, and N is the array length.

A repeatedly invoked complex–inner-product kernel is at the core of the convolution operation. For this kernel, the AP is programmed to accept pairs of complex values, multiply them, and add them to a running sum until the AP encounters an EOS flag. At that point, the AP emits the single complex result.

The stream architecture's separation of computation and data-memory addressing enables a programmer to compose, without concern about data-memory addressing, the AP program as an independent module that forms the inner product. This separation simplifies programming. Furthermore, once the AP inner-product module is created, it can be used for subsequent applications such as matrix multiplication and discrete Fourier transforms (DFT) by simply changing the AG code. The internal details of an AP module need not be considered in order to use it; the stream protocol (ordering of operands), EOS semantics, and pipeline delay through the AP computation are the only necessary interface specifications.

The complex inner product requires four multiplications and two additions for the complex multiplication, followed by two additions for accumulation. Each iteration of the kernel requires two inputs. Since we have two ALUs and a multiplier at our disposal and since the additions and I/O can be performed in the background while the multiplications are taking place, the AP program loop will be four instructions long, limited by the multiplier. Fractional data are assumed. Therefore, only the 16 most significant bits of the product will be preserved.

We first write the procedure as a series of assignments that use minimal parallelism and assign names to the values involved in the computation. All intermediate results except the sum are assigned to the shifting registers; the sum will be accumulated in general register 8. The 32-bit values stored in either the shifting registers or the general registers are referred to by name, and the individual 16-bit parts of them can be referred to with the x or y subscripts.

Variables:

P, Q	: shifting	... complex inputs
J, K, L, M	: shifting	... partial products
C	: shifting	... complex product
S	: gen reg 8	... accumulator

Code:

Loop:
Input P
Input Q
$J_x = P_x * Q_x;$
$K_y = P_x * Q_y;$
$L_x = P_y * Q_y;$
$M_y = P_y * Q_x;$
$C_x = J_x - L_x;$
$C_y = K_y + M_y;$
$S = S + C;$
End loop.

The complex-multiply sequentially produces four 16-bit partial products that are pushed onto one side or the other of the shifting registers during each instruction. This piecemeal shifting will misalign the X and Y sides of the shifting registers by one location. For an automatic code generator, this misalignment is no problem, but, for illustration, we will keep the two halves of values together by always pushing a 32-

bit product and by ignoring the least significant part of the product. We then use a product swap to store the significant part on the conceptually correct side. Although this method squanders registers, the program is not register-limited.

The code may be condensed into a four-instruction loop by concurrent use of the arithmetic units.

In general, it may be useful to try various combinations of parallel operations to obtain maximal AP utilization. We write the program listed in Table 2 in tabular form to keep track of the parallel operations. Note that the named values do not correspond to the same register location at each instruction; rather, the values are pushed down the register bank as new values are written.

Processing of a set of input-stream values is pipelined over three iterations of the loop. During any individual iteration, the ith values are being input while the $(i-1)$st values are being multiplied and the $(i-2)$nd values are being added and accumulated. Any values older than $(i-2)$ are of no interest and eventually disappear off the bottom of the shifting-register bank. Because of the pipelining, the complete AP program must be longer than four instructions. The first two iterations must be coded inline, rather than looped, essentially to prime the data pipeline.

The EOS protocol must also be defined. We shall dictate that EOS will be sent with a dummy value after Q_{N-1}. The instruction during which the EOS is received will execute, but the following instruction will be executed at the address contained in the vector register. Following the EOS, the AP must finish the back end of the loop kernel inline, to empty the data pipeline and emit the result value. An EOS indicates that one inner product is done and the result emitted, and the AP can then begin another inner product (since it will be invoked repeatedly) until some signal indicates that the entire step is done. We can define a protocol that specifies that if two EOSs are received in a row, then the entire step (one complex convolution) is complete and an EOS signal is passed to the output AG.

With these details in mind, the complete AP program for the complex inner product can be coded. The program will have two loops, the inner of which is the four-instruction kernel described in Table 2 , and the outer of which encompasses the entire processing of one inner product. The $J1$ and $J2$ registers will be used to loop back to the tops of the two respective loops. The $J2$ and vector register are simultaneously written, and the $J1$ and branch register are simultaneously written. Immediate-constant data are used to load the flow-control registers and to clear the sum register. The complete program is listed in Table 3 and starts at program-memory location zero. Program origin affects the values assigned to the flow-control registers. All numeric values are hexadecimal.

Register assignments can now be performed. Shifting-register assignment is rather tedious and is easily performed with software tools, but we walk through it here to illustrate the process. Table 4 describes the contents of the shifting registers at the conclusion of each instruction that writes them. At the far left is the register address (hexadecimal — shifting-register addresses are 10–1F), followed by the symbolic values contained in the X and Y sides of the register. The column at the far right describes the values written during that instruction. Subscripts indicate to which input-stream values the intermediate results correspond. The values labeled "junk" are the insignificant part of products.

The assignments of the shifting registers can be determined from Table 4. Fields left blank in Table 5 are unused and may be set to zero in the AP instructions. The conditional and multiply control fields are set to ones in all instructions. When an instruction's flow-control field calls for immediate data, the ALU operation field and the register addresses are replaced by a 16-bit immediate constant, denoted IC, on both the X and Y sides. IC appears as the ALU result, and may be stored through the ALU destination into any of the general or special registers, and/or pushed

Table 2

ALU Push	*Input*	*Prod Push*	*General-Register Assign*	*Flow*
$C = J_x - L_x,\ K_y + M_y$	P	$J_x = P_x * Q_x$ $K_y = P_x * Q_y$ $L_x = P_y * Q_y$		
	Q	$M_y = P_y * Q_x$	$S = S + C$	End loop

Table 3. Symbolic Representation of the Complex–Inner-Product Program

			SHIFTING-REGISTER WRITES		
Loc	*ALU*	*Input*	*Prod*	*General-Register Write*	*Flow*
	Loop back to here to start a new vector; clear the sum.				
0:				$J2/V = [0, 18]$	immed
1:				$S = [0, 0]$	immed
	Input the first stream pair.				
2:		P			
3:		Q			
4:				$J2/V = [0, 14]$	immed
5:				$J1/PC = [10, 7]$	immed
	Begin first complex-multiply.				
	Push of [0, 0] from the ALU maintains shifting-register assignment.				
6:			$J = P_x * Q_x$		
7:	$C = [0, 0]$	P	$K = P_y * Q_x$		
8:			$L = P_y * Q_y$		
9:		Q	$M = P_x * Q_y$		
	Four-instruction kernel:				
10:			$J = P_x * Q_x$		
11:	$C = J - L, K + M$	P	$K = P_y * Q_x$		
12:			$L = P_y * Q_y$		
13:		Q	$M = P_x * Q_y$	$S = S + C$	$J1$
	End of kernel; get here when EOS interrupts kernel.				
	Perform last two actions of kernel.				
14:			$L = P_y * Q_y$		
15:			$M = P_x * Q_y$	$S = S + C$	
	Perform addition from last complex-multiply.				
16:	$C = J - L, K + M$				
	Perform final accumulation and write directly to output stream.				
	Loop back to start a new vector.				
17:				OUT = $S + C$	$J2$
18:				OUTEOS = [0, 0]	immed

into the shifting registers. Refer to Fig. 4.

Once the AP module is written, only its stream protocol must be considered in order to invoke it in a particular application. We have written an inner-product module that is the core of the convolution example but may also be used for matrix multiplies, DFTs, and other applications, simply by changing the AG code.

To summarize the protocol: AGI sends a stream of pairs of values, P_i and Q_i, to be processed; following the final valid data, AGI sends a dummy value tagged as EOS. At this point, it can commence sending the next stream of pairs. When the entire processing is complete, AGI sends another EOS-tagged dummy value. Meanwhile, AGO receives a stream of valid outputs followed by an EOS. On receiving two EOSs in a row the AP sends an EOS to the AGO.

To perform the discrete convolution of two vectors of length N, the AGI must send to the inner-product engine N streams of successively increasing length

$$
\begin{aligned}
&[P_0, Q_0]\\
&[P_0, Q_1], [P_1, Q_0]\\
&[P_0, Q_2], [P_1, Q_1], [P_2, Q_0]\\
&[P_0, Q_3], [P_1, Q_2], [P_2, Q_1], [P_3, Q_0]\\
&\ldots\\
&[P_0, Q_{N-1}], [P_1, Q_{N-2}], \ldots, [P_{N-2}, Q_1], [P_{N-1}, Q_0].
\end{aligned}
$$

The AGI program can be written rather simply as two nested loops, which we will describe first symbolically, using variables that will be assigned to general registers. All looping will use the dedicated iteration hardware in which the branch counter is loaded with

Table 4. Contents of Shifting Registers after Each Instruction

Instruction 2:			
10:	P_0	P_0	input
Instruction 3:			
10:	Q_0	Q_0	input
11:	P_0	P_0	
Instruction 6:			
10:	J_0	junk	prod
11:	Q_0	Q_0	
12:	P_0	P_0	
Instruction 7:			
10:	0	0	dummy ALU result
11:	P_1	P_1	input
12:	junk	K_0	prod
13:	J_0	junk	
14:	Q_0	Q_0	
15:	P_0	P_0	
Instruction 8:			
10:	L_0	junk	prod
11:	junk	K_0	
12:	P_1	P_1	
13:	0	0	
14:	J_0	junk	
15:	Q_0	Q_0	
16:	P_0	P_0	
Instruction 9:			
10:	Q_1	Q_1	input
11:	junk	M_0	prod
12:	L_0	junk	
13:	junk	K_0	
14:	P_1	P_1	
15:	0	0	
16:	J_0	junk	
17:	Q_0	Q_0	
18:	P_0	P_0	
Instruction 10:	Beginning of kernel; i indicates values being input, and runs from 2 to $N-1$. Kernel repeats until it receives an EOS on Nth P value.		
10:	J_{i-1}	junk	prod
11:	Q_{i-1}	Q_{i-1}	
12:	junk	M_{i-2}	
13:	L_{i-2}	junk	
14:	junk	K_{i-2}	
15:	P_{i-1}	P_{i-1}	
16:	C_{i-3}	C_{i-3}	
17:	J_{i-2}	junk	
Instruction 11:			
10:	C_{i-2}	C_{i-2}	ALU
11:	P_i	P_i	input
12:	junk	K_{i-1}	prod
13:	J_{i-1}	junk	
14:	Q_{i-1}	Q_{i-1}	
15:	junk	M_{i-2}	
16:	L_{i-2}	junk	
17:	junk	K_{i-2}	
18:	P_{i-1}	P_{i-1}	
Instruction 12:			
10:	L_{i-1}	junk	prod
11:	C_{i-2}	C_{i-2}	
12:	P_i	P_i	
13:	junk	K_{i-1}	
14:	J_{i-1}	junk	
15:	Q_{i-1}	Q_{i-1}	
16:	junk	M_{i-2}	
17:	L_{i-2}	junk	
18:	junk	K_{i-2}	
19:	P_{i-1}	P_{i-1}	
Instruction 13:			
10:	Q_i	Q_i	input
11:	junk	M_{i-1}	prod
12:	L_{i-1}	junk	
13:	C_{i-2}	C_{i-2}	
14:	P_i	P_i	
15:	junk	K_{i-1}	
16:	J_{i-1}	junk	
Instruction 14:	EOS processing; beginning of instruction similar to 11 above. Finish complex-multiply for $(N-1)$st data. Nth P value has been written but is not processed.		
10:	L_{N-1}	junk	prod
11:	C_{N-2}	C_{N-2}	
12:	P_N	P_N	
13:	junk	K_{N-1}	
14:	J_{N-1}	junk	
15:	Q_{N-1}	Q_{N-1}	
16:	junk	M_{N-2}	
17:	L_{N-2}	junk	
18:	junk	K_{N-2}	
19:	P_{N-1}	P_{N-1}	
Instruction 15:			
10:	junk	M_{N-1}	prod
11:	L_{N-1}	junk	
12:	C_{N-2}	C_{N-2}	
13:	P_N	P_N	
14:	junk	K_{N-1}	
15:	J_{N-1}	junk	
Instruction 16:	Final addition		
10:	C_{N-1}	C_{N-1}	ALU
11:	junk	M_{N-1}	
12:	L_{N-1}	junk	
13:	C_{N-1}	C_{N-1}	
14:	P_N	P_N	
15:	junk	K_{N-1}	
16:	J_{N-1}	junk	

Table 5. Tabular Representation of the Complex–Inner–Product Program with Register Assignments (register addresses in hexadecimal)

		X						Y				
Loc	*Flow*	*Shift*	*ALU*	*A*	*B*	*C*	*Dest*	*Swap*	*Shift*	*ALU*	*A*	*B*	*C*
0:	immed					IC = 0	*J2/V*						IC = 18
1:	immed					IC = 0	8						IC = 0
2:		*I*							*I*				
3:		*I*							*I*				
4:	immed					IC = 0	*J2/V*						IC = 14
5:	immed					IC = 10	*J1/PC*						IC = 7
6:		*P*		10	0	11		*A*	*P*		0	0	0
7:		*A, I, P*	XOR	0	0	11		*P*	*A, I, P*	XOR	12	0	0
8:		*P*		0	0	0		*C*	*P*		15	0	14
9:		*I, P*		0	0	16		*P*	*I, P*		15	0	0
10:		*P*		14	0	10		*A*	*P*		0	0	0
11:		*A, I, P*	*U* − *V*	17	13	15		*P*	*A, I, P*	*U* + *V*	11	14	12
12:		*P*		0	0	0		*C*	*P*		15	0	19
13:	*J1*	*I, P*	*U* + *V*	8	11	19	8	*P*	*I, P*	*U* + *V*	15	8	11
14:		*P*		0	0	0		*C*	*P*		14	0	18
15:		*P*	*U* + *V*	8	11	19	8	*P*	*P*	*U* + *V*	15	8	11
16:		*A*	*U* − *V*	15	11	0			*A*	*U* + *V*	0	14	10
17:	*J2*		*U* + *V*	8	10	0	6			*U* + *V*	0	8	10
18:	immed					IC = 0	7						IC = 0

the number of branches back to the top of the loop. A number of parameters are defined that are most conveniently coded as ICs in the AG instructions. As in the AP, each line of code below corresponds to one instruction.

PARAMETERS:

LENGTH	The length of the vectors
*P*_START	Start address of *P* vector minus one
*Q*_START	Start address of *Q* vector

VARIABLES:

STREAM_LENGTH	Register 0
*P*_POINTER	Register 1
*Q*_POINTER	Register 2

CODE:

```
STREAM_LENGTH = 1;
Loop = LENGTH;
  P_POINTER = P_START;
  Q_POINTER = Q_START + STREAM_LENGTH;
  Loop = STREAM_LENGTH – 1;
    Generate_address = P_POINTER = P_POINTER + 1;
    Generate_address = Q_POINTER = Q_POINTER – 1;
    End_loop;
  Generate_EOS = STREAM_LENGTH
     = STREAM_LENGTH + 1;
  End_loop;
Generate_EOS = 0;
```

This code can be transformed into AG instructions, shown in Table 6. When Source 1 is of the form "IC = ..." a 16-bit immediate constant is used; otherwise, a register address is given. Refer to Fig. 3.

The AGO program is even easier; it loads a pointer to the beginning of the result vector and loop-generates an incrementing address until either the loop counter is exhausted or an EOS is encountered. The EOS causes an interrupt that transfers program flow to the address contained in the vector register when the AGO generates an address that stores the EOS-tagged value. The write-to-data-memory will occur, as will any other effects of the AGO instruction, including internal register writes.

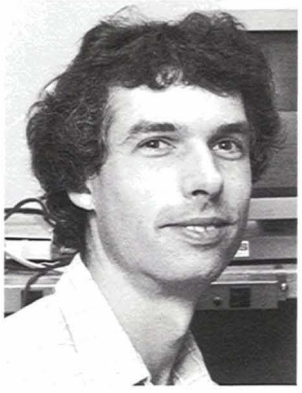
Note that if any or all of the vectors occur noncontiguously in data memory, the AG programs can perform the addressing by incrementing or decrementing the pointers by a value other than one.

One final comment: this example describes only one of many programming methods; another example deserves brief discussion. The AP program was written as a prologue of inline code, followed by a loop of high-density pipelined code, followed by an epilogue of inline code. Another method, which saves AP instructions but costs more execution time, is simply to write the pipelined kernel loop, along with the necessary prologue to set up the *J* and vector registers. In general, this method is useful for AP functions that do not have internal storage between one input-stream item and another, such as a complex-multiply

or FFT kernel. The inner product does not fall into this category. Because of the pipelining, an AP program thus coded will emit one or more "garbage" values that must be discarded by the AGO in advance of the legitimate-result stream. Similarly, the AGI must generate additional stream values at the end to expel the last of the useful results. This method, used extensively in the UAV radar application software, is entirely a matter of programmer preference.

Table 6

Loc	*End*	*Gen*	*Dest*	*Source 2*	*ALUOP*	*Source 1*
0:	0	0	0	0	3	IC = 1
1:	0	0	10	0	3	IC = LENGTH
2:	0	0	1	0	3	IC = *P*_START
3:	0	0	2	0	5	IC = *Q*_START
4:	0	0	10	0	4	IC = 1
5:	0	1	1	1	5	IC = 1
6:	1	1	2	2	5	IC = –1
7:	1	2	0	0	5	IC = 1
8:	0	2	0	0	0	0



F. EDWARD HALL is currently developing new architectures and software for signal processors. Ed received a B.S. in physics from Bucknell University. He has worked at Lincoln Laboratory for 11 years.

A. GREGORY ROCCO, Jr. works in the areas of computer architectures, digital signal processing, and computer-aided engineering. Greg received a B.S. in electrical engineering from the University of New Hampshire in 1977 and an M.S. in electrical engineering from Purdue University in 1982. He has been a Lincoln Laboratory staff member for 11 years.