IRA H. GILBERT

The Synchronous Processor

The Synchronous Processor is a single-instruction, multiple-data-stream, special-purpose computer, which can perform calculations at a rate of almost 400 million instructions per second (MIPS). To develop this computer, a new hardware design was implemented and a special programming language was constructed.

Built entirely of readily available, off-the-shelf components, the Synchronous Processor (SP) computer achieves a throughput of nearly 400 million instructions per second (MIPS). The SP is a parallel-architecture computer — its design makes extensive use of concurrency in control, computation, internal communication, and input/output. The throughput of the SP is almost an order of magnitude greater than that of a Cray 1 (the first conventional supercomputer), yet the electrical power requirement for the SP is approximately two orders of magnitude smaller than that of the Cray 1. The two machines are not directly comparable: the Cray is a general purpose, 64-bit floating point machine; the SP is a special-purpose, fixed-point 16/32-bit machine. But for the applications that it addresses, the SP is much more economical and powerful than its more conventional cousin.

SPL, the programming language developed for the SP, supports the concurrent operation of the processor. The language also enhances processing efficiency by giving the programmer control of individual hardware registers. Yet, although the intimate register control is similar to the level of control provided by assembler languages the SPL offers many of the capabilities of a high-level language. The FORTRAN language is available to the programmer at translate time, and supplementary machine-addressing modes are immediately available to the programmer. A feature that is particularly important for signal-processing applications is the language's capability of treating data arrays as objects, which eliminates the need to keep track of the physical addresses of the data in the arrays.

MULTIPROCESSOR COMPUTERS

Many computer applications require far greater throughput than a single processor can provide. This computational inadequacy has led to the development of multiprocessor computer architectures [1].

One type of multiprocessor computer, the single-instruction, multiple-data stream (SIMD) processor, is especially well suited for regular and predictable computations that require only loose coupling between data and control. (Loose coupling means that the flow of operations to be performed is relatively independent of the results of intermediate computations). Many computation-intense applications are of this nature. For example, the solution of systems of partial differential equations arising in fluid dynamic simulations and those used in seismic analysis for geophysical exploration are applications that exhibit loose coupling.

The SIMD computer is also suitable for solving the large systems of linear equations that can arise in optimization problems. Modern surveillance techniques often require complex, multidimensional image enhancement and signal-processing operations, which are also suitable for SIMD computers. Additionally, SIMD machines can simulate most neural network models. Large neural-network simulations (thousands or millions of synapses) require the sort of processor power only available from multiprocessor computers.

A SIMD computer consists of an array of identical processing elements that operate in lockstep under the control of one master processor. Only the master processor has access

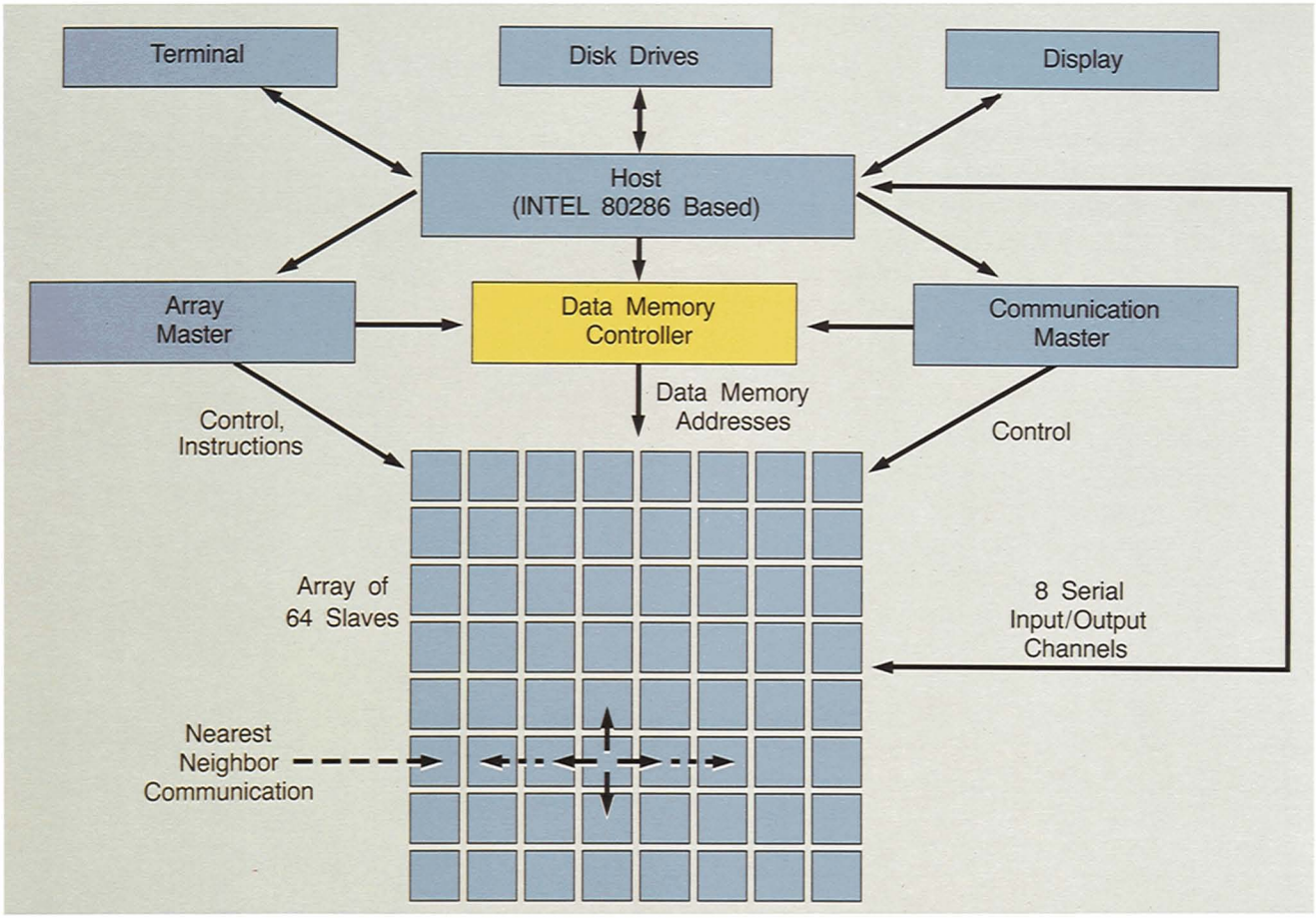


Fig. 1 — Organization of the SP.

to the single copy of the application program. The slave processing elements synchronously execute the instruction stream broadcast to them by the master. Thus, for only the incremental cost of additional processing elements, the computer's throughput can be arbitrarily increased. No additional master processors are required. Another benefit of the SIMD architecture is the decoupling of application software and processor size. That is, code may be initially developed on a scaled-down version of a processor but run, without alteration, on a much larger array.

Because the SIMD architecture requires only one copy of an application program, a programmer can easily optimize the speed of a program. Optimizing a program for speed may increase memory requirements, but, since a SIMD computer holds only one copy of a program (not one copy per processor), the increase in memory requirement is insignificant.

HISTORY

The concept of SIMD processing dates from 1962, when Slotnick *et al* [2] proposed the SOLOMON (Simultaneous Operation Linked Ordinal Modular Network) computer. This system comprised a 32×32 array of processing elements, with nearest-neighbor communication, under the control of a single master. Computation and communication were both bit-serial. Each processing element contained 4,096 bits of data storage. The first important SIMD processor, the ILLIAC IV, was designed and built at the University of Illinois by a team under the direction of Slotnick [3]. Built in the late 1960s and early 1970s, the computer was decommissioned in 1981. It consisted of 64 64-bit floating-point processors. In the 32-bit mode, ILLIAC IV also functioned as a 128-node array. Each of the processing elements had 2,048 64-bit words of memory.

Most subsequent SIMD processors returned to the original concept of a large number of very primitive bit-serial processing elements. Examples include the DAP (ICL), the Massively Parallel Processor (Goodyear Aerospace), and most recently, the Connection Machine of Hillis [4]. The Connection Machine is SIMD only in so far as computation is concerned. Communication within the Connection Machine is carried out asynchronously; messages are routed through a network of 4,096 nodes that are organized as the corners of a 12-dimensional hypercube.

THE SP

The SP is a SIMD machine that was designed primarily for signal-processing applications. The system comprises a host, two master controllers, and a processor array. The SP is organized hierarchically, as shown in Fig. 1. The host is responsible for interfacing the SP to the external world, for exercising overall control over the other masters, and for managing input to and output from the processor array.

Immediately beneath the host in the hierarchy are the two master controllers, the array master (AM) and the communication master (CM). The AM is responsible for routing instructions to the computational elements (CEs). The CM controls communication among the slave CEs.

At the bottom of the hierarchy is an array of 64 CEs arranged on an 8×8 grid. Each CE is built around a Texas Instruments TMS 320-20 digital signal processing (DSP) chip. The CEs contain 64K 16-bit words of data storage (Fig. 2). The DSP chips run at 6 MHz, can multiply two 16-bit numbers to a 32-bit result in one 167-ns cycle, and deliver a performance of 6 MIPS. The aggregate throughput of the array is, therefore, 380 MIPS. Some applications can even take advantage of the TMS 320-20's ability to multiply and accumulate simultaneously, yielding a peak performance of 760 MIPS.

The TMS 320-20 also provides 544 words of on-chip memory. Since the access time for the on-chip memory is one half to one third of the access time associated with the fastest off-chip memory, the arguments of arithmetic operations must be stored on chip — to achieve the peak throughput of these devices.

Although the TMS 320-20's data word width is 16 bits, its accumulator and product registers are 32 bits wide. The wider accumulator and product register enable the processor to accumulate sums of products with greater accuracy than a fixed-length 16-bit processor.

A serial 12-MHz data link connects each CE in the array to its four nearest neighbors. Inter-CE communication is purely lockstep. For example, when one CE passes data to its neighbor on the right, all CEs do so. The overall array topology is toroidal — the top and bottom edges are connected, as are the right and left edges. This topology is especially suited to such applications as performing two-dimensional fast Fourier transforms (FFTs). (For a detailed description of this application, see the Appendix "FFT Application.")

The toroidal connectivity may be reconfigured, under software control, to produce a finite, open topology. This reconfiguration allows the SP to filter four separate arrays simultaneously, for example, with no "spillover" or "wraparound" from one array to another. In addition, failed columns of CEs can be bypassed in software via the reconfigurable connectivity, thus providing a measure of fault tolerance.

As in all SIMD computers, the computational elements of the SP do not act as independent computers, but rather as slave processors that are executing a stream of instructions. The instructions are common to all slaves. As a result, only data-specific instructions (multiply, add, store, load) are executed by the CEs. Only the AM uses the remaining control instructions — to access the single copy of the program. In the interest of processing efficiency, the two types of instructions must be separated. This separation lets the SP execute the instructions in parallel. This separation can be achieved by building high-speed hardware that scans the composite instruction stream and separates the control instructions from the arithmetic instructions. However, this approach complicates the hardware and compromises its reliability.

In the SP, the control and arithmetic instructions are separated at translate time (Fig. 3). This separation method permits the use of high-speed computation elements without the need for even higher-speed instruction separator hardware.

SP ORGANIZATION

The host, the AM, and the CM are three loosely coupled processors, each with its own program. The host directly controls input and output between the array and the mass-storage devices. The host is also responsible for downloading the programs of the AM, the CM, and the CEs to their respective memories. Once these programs are downloaded, the host initiates execution by vectoring the AM and CM to starting addresses in their respective code spaces. These two masters then run independently of the host until they reach programmed HALTs. At this point, their execution is terminated and the host is notified. If the host has already queued up a successor vector and if the auto-execution mode is enabled, the HALT instruction then triggers execution of the next vector without further intervention by the host. This auto-execution operation eliminates the host's interrupt-response time penalty.

Communication between the host and mass storage is carried out by means of serial-communication links, with one link for each column of the array. Each link operates at 12 MHz, which yields a total data rate of 96×10^6 bps. Each link consists of nine 16-bit registers: one register for each of the eight CE processors in a column plus a ninth register located on a custom board in the host. To input data to the array the host loads each of the eight registers on the custom board, then shifts one bit at a time (all eight columns simultaneously) until the data are transferred from the host to the top CE in each column. The host loads and shifts the registers seven more times, until all eight CEs in each column have received their data. Then, in a single step, the data are transferred to the CEs' local memories. The output of data from array to host is the reverse of this process.

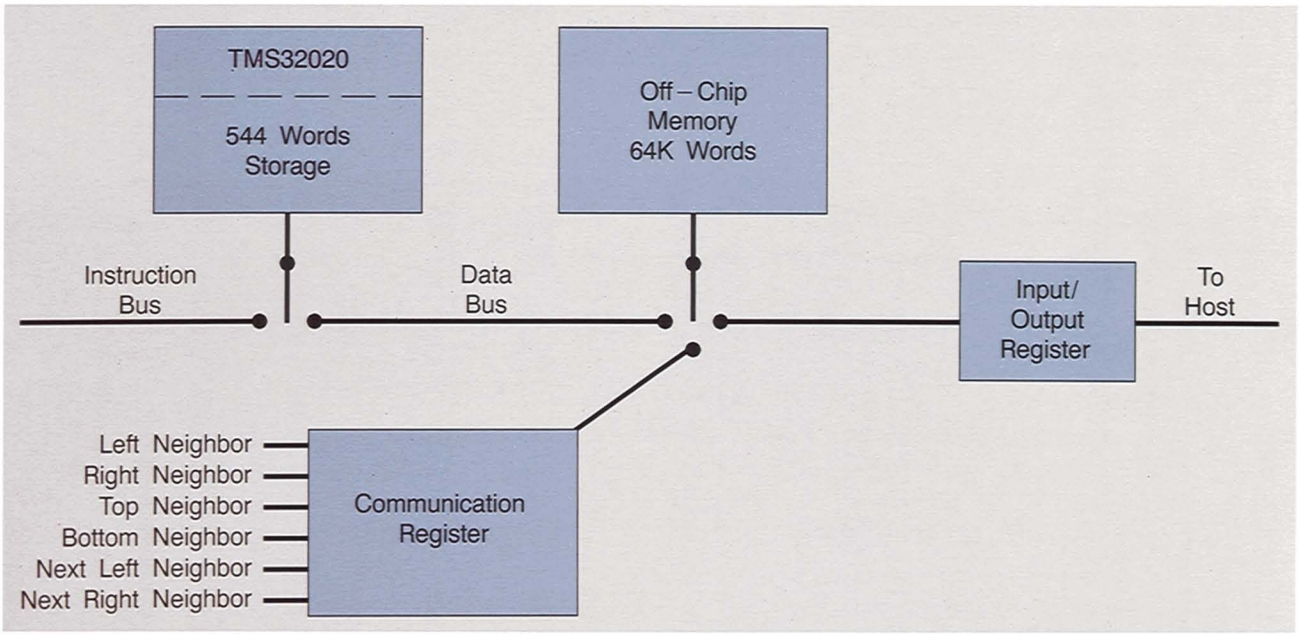


Fig. 2 — Each computation element in the SP includes a TMS 320-20, 64K of data memory, an input/output register, and a communication register. The communication register's multiplexed input accepts data from any of six sources, providing nearest-neighbor communication as well as a measure of fault tolerance. Requests for access to the off-chip memory can be made at any time by the AM, the CM, or the input/output system. These requests are arbitrated by the data-memory controller. The AM receives highest priority, the CM second priority, and input/output lowest priority.

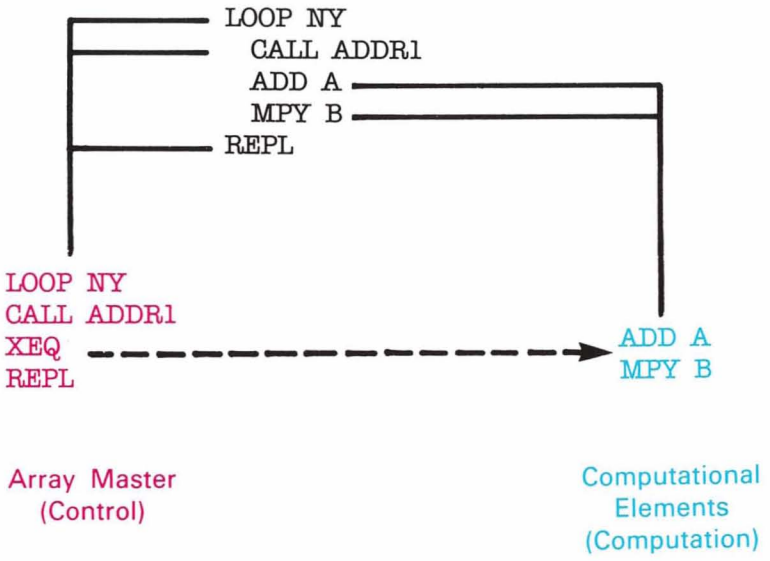


Fig. 3 — The SPL separates code into control and computation blocks. The control blocks are executed by the AM in parallel with the slave processor's execution of the computation instructions.

The AM is a simple programmable controller; it accesses the program memory of the CEs and broadcasts instructions to all the CEs in the array, so the CEs can execute the instructions in parallel. The AM contains two separate program memories. One contains the AM-sequencer control instructions, the other the data-related CE instructions. This division is transparent to the programmer. The CE and AM instructions are actually executed concurrently.

The CM controls all aspects of communication among the CEs. The communication is bit-serial and strictly lockstep. Each CE contains a 16-bit shift register, which is used for interprocessor communication (Fig. 2). A six-input multiplexer feeds data to the shift register. Four of the inputs to the multiplexer are the shift-register outputs of the four nearest-neighbor CEs. Two inputs to the multiplexer are from CEs that are two columns away. (This configuration lets the CM bypass failed columns.) The CM loads and unloads the CEs' shift registers, selects the data source, and performs the necessary data shifts.

Suppose, for example, that a CE is instructed to pass data that is located in its memory at address A1 to another CE's memory at address A2. The CM loads the communication register of every CE from each CE's memory location A1. The CM then sets every CE's multiplexer so that data are received from the appropriate neighboring processor and then the CM shifts the data. This process of multiplexer selection and shifting is repeated until the data arrive at the intended destination, at which point the data from all communication registers are simultaneously placed in the destination CE's A2 address.

SP SOFTWARE

As might be expected, the development of the SP required the development of a new programming language. The need for programmer efficiency biased the choice of language towards a high-level compiler language, but anything less than a highly optimizing compiler was considered inadequate for the intended applications of the SP. Resource requirements for the development of a highly optimizing compiler prevented the development of an optimizing compiler. Instead, we chose to achieve the necessary run-time efficiency by programming at the assembler level. However, SPL incorporates a number of features that eliminate many of the

```
      DO (I = 1,20)
        DO (J = 1,20)
          K = ALPHA × SIN (I × BETA)
          L = J × J
%         LTA ARRAY1 (K,J)
%         MPY ARRAY2 ((I,L)
        FIN
      FIN
```

Fig. 4 — This FORTRAN code produces multiple copies of executable assembler code (shown with a %).

burdens normally associated with assembler programming. Moreover, the full power of a high-level language, FORTRAN, is available at translate time. This feature makes it possible, for instance, to refer to multidimensional arrays as data objects and to generate "unrolled" or straight-line code (Fig. 4).

Of the eight lines appearing in Fig. 4, only two assembler instructions — LTA (load the T register) and MPY (multiply) — generate code that survives the translation process. These instructions are flagged by the symbol %. The translator program recognizes this symbol as a command to "hide" instructions from the FORTRAN compiler. The FORTRAN instructions, when compiled, create a group of these assembler instructions.

The operands of the assembler instructions in Fig. 4 are previously defined elements of two-dimensional arrays. Since the indices of these elements are complicated functions of the loop variables I and J, but are common to all CEs, it is undesirable to compute them in real time. Instead, the addresses are computed at translate time and associated with the LTA and MPY instructions. As a result, this pair of instructions is replicated 400 times, once for each instance of the combination (I,J). The illustrated code fragment will, therefore, generate 800 machine instructions and consume 800 words of memory. In this case the performance advantage gained by unrolling the code justifies the concomitant increase in storage-space requirements.

The code-unrolling technique is only one of the features that facilitates efficient array handling. The SP's hardware supplies a way to step through arrays at run-time and SPL supports this capability. The AM contains hardware that injects a 16-bit address onto the local memory bus at a time when the CE's TMS 320-20 is receiving data from a previous instruction. This feature provides two immediate benefits: the elimination of "dead time" when the TMS 320-20 is fetching data, and, because the injected address is 16 bits wide, the elimination of paging schemes that the programmer would normally be forced to use. In addition, SPL lets programmers define subarrays as dimensioned pieces of larger arrays, further easing the task of array manipulation.

A key element of SPL is support for concurrency of computation and control. Referring back to Fig. 3, a programmer writes code that integrates computation and control. In the illustrated code fragment, the loop length parameter, NY, is defined at translate time. It is not a run-time parameter. The LOOP (loop), REPL (repeat loop), and CALL (call) instructions are control instructions, used only by the AM. LOOP and REPL delineate the boundaries of a program loop; CALL transfers control to another block of code within the AM. The arithmetic instructions ADD (add) and MPY (multiply), however, are used only by the array of CEs. The SPL translator scans the illustrated segment and separates the arithmetic instructions from the control instructions. A single AM instruction, XEQ (execute), replaces the two arithmetic instructions in the AM code space.

The XEQ instruction references the base address of the CE instruction block in CE code space. An AM program is thus translated to a sequence of control instructions interspersed with XEQs that refer to CE code space. These XEQs are similar to CALLs because different XEQs can refer to the same CE code address. The key difference between CALL and XEQ instructions is that the CALL instruction transfers control from one location in AM code space to another in AM code space, but the XEQ instruction merely indicates the starting address of the next block of CE instructions to be broadcast to the CE array.

The next-to-last instruction in the block of

CE code tells the AM that the last instruction of the sequence follows. The AM responds to this information by queuing up the next block of CE code. This approach to broadcasting the CE commands provides the speed and efficiency of direct memory access (DMA) type transfer without needing additional hardware or setup time.

When the host vectors the AM to a starting address, the SP starts to execute. The AM then executes control instructions until the first XEQ instruction is encountered. At that point, the AM initiates the transfer of CE instructions (located at the referenced address) to the array of CEs. Once initiated, this transfer proceeds without further intervention of the AM, allowing the AM to perform control instructions while the CEs perform computation. When the AM encounters a second XEQ, it queues up the base address of the associated block of CE instructions, flags the XEQ as pending, and then continues to execute control instructions until it encounters another XEQ instruction. At this point, with one block of CE code pending and another XEQ awaiting execution, the AM halts operation on the control instructions. When the first block of CE code is completely broadcast, the AM begins to transmit the pending block of CE instructions immediately. The last XEQ to be encountered is queued up as pending and the AM proceeds with control instructions. With this instruction pipelining, the computational array is kept busy executing arithmetic instructions and the array spends a very small amount of time waiting for instructions.

DATA-DEPENDENT BRANCHING

Although the lockstep execution characteristic of SIMD processing poses no hardship for most signal-processing applications, in some circumstances it is useful to retain a version of data-dependent branching. For example, if the power of the SP were brought to a real-time medical imaging application, an intensity threshold that indicates the possible presence of cancerous tissue could be set. Then, if the threshold is exceeded, the transform operation could be modified to increase the image resolution in the area of suspect tissue.

To address such applications, SPL has a conditional NOP (null operation) capability by which any computational element can be forced to execute NOPs — effectively idling — depending on the results of an arithmetic or logic test. This lets one section of the array focus on the transform operation while other sections await the completion of the operation.

The implementation of this data-dependent branching feature makes unconventional use of the TMS 320-20's instruction counter. The counter was designed to address program memory, but since the CEs have no such memory, they simply execute the instruction stream broadcast by the AM. The program counter plays no role in fetching instructions. Instead, the most significant bit (MSB) of the instruction counter determines whether the CE executes the incoming instruction stream or NOPs. When the conditional-branch mode is enabled (by setting an appropriate flip-flop), the presence of a "1" in the MSB of the program counter substitutes NOPs in the CE's instruction stream. Conversely, a "0" in the MSB of the program counter permits the passage of the broadcast instruction stream to the TMS 320-20 for execution (see Table).

Table — Form of Data-Dependent Branching Maintained by SPL

	Accumulator ≠ 0		*Accumulator = 0*	
Instruction Cycle	*Program Counter*	*Instruction*	*Program Counter*	*Instruction*
1	0032	BZ	0032	BZ
2	0033	FFFC	0033	FFFC
3	0034	INST 1	FFFC	NOP
4	0035	INST 2	FFFD	NOP
5	0036	INST 3	FFFE	NOP
6	0037	INST 4	FFFF	NOP
7	0038	B	0000	B
8	0039	0000	0001	0000
9	0000	INST 5	0000	INST 5

Note: SPL maintains a form of data-dependent branching by the use of the branch-on-zero (BZ) instruction. The data-dependent branching is possible because the TMS 320-20's program counter is not used to access program memory; computation instructions are broadcast to the SP array by the AM. To execute either string of N instructions or a string of N NOPs (depending upon whether the accumulator flag is set), program a BZ instruction to an address that is N less than 64K. If the branch is taken, NOPs will be perfomed until the program counter rolls over.

IMPROVED TECHNOLOGY

Construction of the SP began five years ago. The technology used in its construction is, therefore, outdated. A larger version of the SP, consisting of 1,024 CEs built around the TMS 320-C30 (the successor to the TMS 320-20), could boost SP performance by almost two orders of magnitude — to 33 GFLOPs (billion floating-point operations per second). The TMS 320-C30 is a 32-bit floating-point processor with a clock speed of 16 MHz, which yields a computation speed of 33 MFLOPs. The device also offers the increased precision and dynamic range inherent to floating-point representation. On-chip storage for the TMS 320-C30 consists of 2K 32-bit words; eight 40-bit registers are also included on chip, to increase intermediate precision and dynamic range.

Despite the two orders-of-magnitude increase in performance, the enlarged SP array will occupy only about 4 ft^3. It will consume less than 4 kW. By taking advantage of the increased density of memory chips and by consolidating some of the circuit functions into gate-array integrated circuits, the number of integrated circuits required for each CE could be cut to about one fourth the number required for the present configuration.

To balance communication and computation capabilities in the enhanced SP, internal data paths must be byte-wide, rather than bit-wide. Byte-wide communication is the most practical method of increasing internal communication speed; it prevents internal communication from becoming a bottleneck to system throughput.

The word size of the TMS 320-C30 is twice that of the TMS 320-20. The TMS 320-C30 executes instructions at four times the speed of the older DSP chip — 33 MIPS vs 6 MIPS. This combination produces an eightfold increase in data throughput for each CE.

CONCLUSION

The SP project has convinced us that SIMD is indeed an efficient computer architecture and is practical for an important class of computation-intense problems. The SP is an effective realization of the SIMD concept. Much of the success of the project is due to the SPL language, which significantly simplifies SP programming and thereby enhances programmer productivity.

Appendix: FFT Application

The two-dimensional FFT is an example of an application that is well suited to the architecture of the SP. The implementation of a two-dimensional FFT also reveals how the SP combines communication and computation.

The FFT algorithm in this Appendix is based on the standard row-column technique. Each of the rows is Fourier transformed and then each of the resulting columns is Fourier transformed. For this example, the input data comprise an N × N-dimensional array; the array of CEs is dimensioned n × n. For the existing SP, n = 8, but there is no need to limit the description of the algorithm to this particular dimension. Both N and n are powers of two. Moreover, N is a multiple of n^2.

	C=0			C=1			C=2			
	00	01	02	03	04	05	06	07	08	
R=0	10	11	12	13	14	15	16	17	18	t=0
	20	21	22	23	24	25	26	27	28	
	30	31	32	33	34	35	36	37	38	
R=1	40	41	42	43	44	45	46	47	48	t=1
	50	51	52	53	54	55	56	57	58	
	60	61	62	63	64	65	66	67	68	
R=2	70	71	72	73	74	75	76	77	78	t=2
	80	81	82	83	84	85	86	87	88	

Fig. A — Natural order.

	C=0			C=1			C=2			
	00	01	02	13	14	15	26	27	28	
R=0	10	11	12	23	24	25	06	07	08	t=0
	20	21	22	03	04	05	16	17	18	
	30	31	32	43	44	45	56	57	58	
R=1	40	41	42	53	54	55	36	37	38	t=1
	50	51	52	33	34	35	46	47	48	
	60	61	62	73	74	75	86	87	88	
R=2	70	71	72	83	84	85	66	67	68	t=2
	80	81	82	63	64	65	76	77	78	

Fig. B — Column-dependent vertical roll.

Initially, the data are arranged in "natural order." That is, each row of data is assigned to a single row of CEs and is divided evenly among the n columns (Fig. A). The first N/n elements of the first row of data reside in the first CE, the second N/n elements reside in the second CE, and so on. In addition, the first N/n elements of the first column reside in the first CE, the next N/n elements of the first column reside in the next lower CE in the column of CEs, and so forth.

The data are first rearranged so that each row of data is contained in a single CE. Then, a one-dimensional FFT, performed simultaneously in all n^2 CEs, transforms n^2 rows of data at once. The entire data array of N rows is thus row-transformed in N/n^2 operations. Next, the data are rearranged so that each column of resultants is contained within a single CE. Simultaneous one-dimensional transforms then transform all columns of data in N/n^2 operations. Finally, the results of the column transformation are restored to natural row-column order.

Although simple to state, this prescription is not trivial to implement. The SP is restricted to lockstep communication, which requires all the CEs to fetch data from the same local source address, move it along the same relative path, and store it in the same local destination address. This lockstep constraint provides hardware and software simplicity, but it isn't obvious that the data movements can be performed without a great deal of wasted motion.

In fact, the entire task can be efficiently performed by successive applications of one simple transformation. This is the first publication of this algorithm. A key element in the success of this communication algorithm is the toroidal connectivity of the array.

Let the row and column address of a given CE be denoted (R,C) and the coordinates of the data elements within a CE be denoted (q,p). Each CE originally holds N/n elements of each row of data. The overall row and column address of a data element (i,j) are then given by

$$i = q + R(N/n) \qquad q, p = 0,\ldots, (N/n) - 1$$
$$j = p + C(N/n) \qquad i, j = 0,\ldots, N - 1$$
$$R,C = 0,\ldots, n - 1.$$

N is a multiple of n^2, so (N/n) - 1 (the total range of q and p) is a multiple of n. Any particular value of q or p may therefore be expressed as a multiple of n plus a remainder that is less than n:

$$p = r + sn \qquad r,\ t = 0,\ldots,\ n - 1$$
$$q = t + un \qquad s,\ u = 0,\ldots,\ (N/n^2) - 1.$$

Then,

$$i = t + un + R\ (N/n)$$
$$j = r + sn + C\ (N/n).$$

These equations express the coordinates of a numerical element in terms of the row and column address of its assigned CE and in terms of r,s,t, and u, which are internal coordinates repeated in each CE.

If it is possible to interchange t and C by a realizable lockstep communication, then for fixed R, C, and u (*ie*, within a portion of the memory of a single CE), the row index, i, will be held constant while the column index, j, will vary over the entire range (0 to N-1) and r, s, and t go through their respective ranges. With this variation, each row will be reassigned to a single CE, as desired. This transformation is realized by a sequence of three operations as displayed in the Table. All arithmetic is evaluated in *modulo* n.

App. 1 — Transformation Comprising Three Operations

	t	*C*	
t → t - C	t - C	C	Move each row of data in a CE up the number of rows equal to the column number of the CE.
C → C + t	t - C	t	Shift the rows of data to the right by a number of CEs equal to t, the row location within a CE.
t → C - t	C	t	Invert the data.

The first operation consists of rolling data vertically within each CE by an amount that depends on the column address of that CE. For example, move all rows of data within the CEs of column 2 up two rows within each CE (Fig. B). The second operation is a transfer of data among CEs. As shown in Fig. B, shift the rows containing 00, 30, and 60 zero CEs to the right, shift the rows containing 10, 40, and 70 one CE to the right, and shift the rows containing 20, 50, and 80 two CEs to the right. The results of carrying out such shifts are illustrated in Fig. C. The parameter t (the increment in the column address C) is independent of the CE coordinates, so this transfer can be performed in lockstep. The third operation is an inversion (t → -t) followed by a column-dependent roll (Fig. D).

	C=0			C=1			C=2			
R=0	00	01	02	13	14	15	26	27	28	
	06	07	08	10	11	12	23	24	25	t=0
	03	04	05	16	17	18	20	21	22	
R=1	30	31	32	43	44	45	56	57	58	
	36	37	38	40	41	42	53	54	55	t=1
	33	34	35	46	47	48	50	51	52	
R=2	60	61	62	73	74	75	86	87	88	
	66	67	68	70	71	72	83	84	85	t=2
	63	64	65	76	77	78	80	81	82	

Fig. C — Lockstep shift.

	C=0			C=1			C=2			
R=0	00	01	02	10	11	12	20	21	22	
	03	04	05	13	14	15	23	24	25	t=0
	06	07	08	16	17	18	26	27	28	
R=1	30	31	32	40	41	42	50	51	52	
	33	34	35	43	44	45	53	54	55	t=1
	36	37	38	46	47	48	56	57	58	
R=2	60	61	62	70	71	72	80	81	82	
	63	64	65	73	74	75	83	84	85	t=2
	66	67	68	76	77	78	86	87	88	

Fig. D — Inverted.

The result of applying this three-step transformation to naturally ordered data is

$$i = C + nu + (N/n)\ A$$
$$j = r + ns + (N/n)\ t.$$

Each row is placed within the local memory of a single CE. Then that row is Fourier transformed with a standard one-dimensional FFT, which is executed simultaneously by each CE. After all rows are similarly transformed, it is necessary to redistribute the results so that each column is placed within a single

CE. This rearrangement is accomplished by a double interchange:

$C \leftrightarrow t$ Return to natural order
$R \leftrightarrow r$ Transform to column order

or

$t \rightarrow t - C$
$r \rightarrow r - R$
$C \rightarrow C + t$
$R \rightarrow R + r$
$t \rightarrow C - t$
$r \rightarrow R - r.$

The result is

$i = t + nu + (N/n)\, r$ Column order
$j = R + ns + (N/n)\, C.$

As required, for fixed R, C, and s, the column index j is fixed while the row index i varies from 0 to N-1, and t, u, and r vary through their ranges. Each column has been packed into a single CE.

Finally, the data are restored to natural order:

$R \leftrightarrow r$

or

$r \rightarrow r - R$
$R \rightarrow R + r$
$r \rightarrow R - r.$

The result is

$i = t + nu + (N/n)\, R$ Natural order
$j = r + ns + (N/n)\, C.$

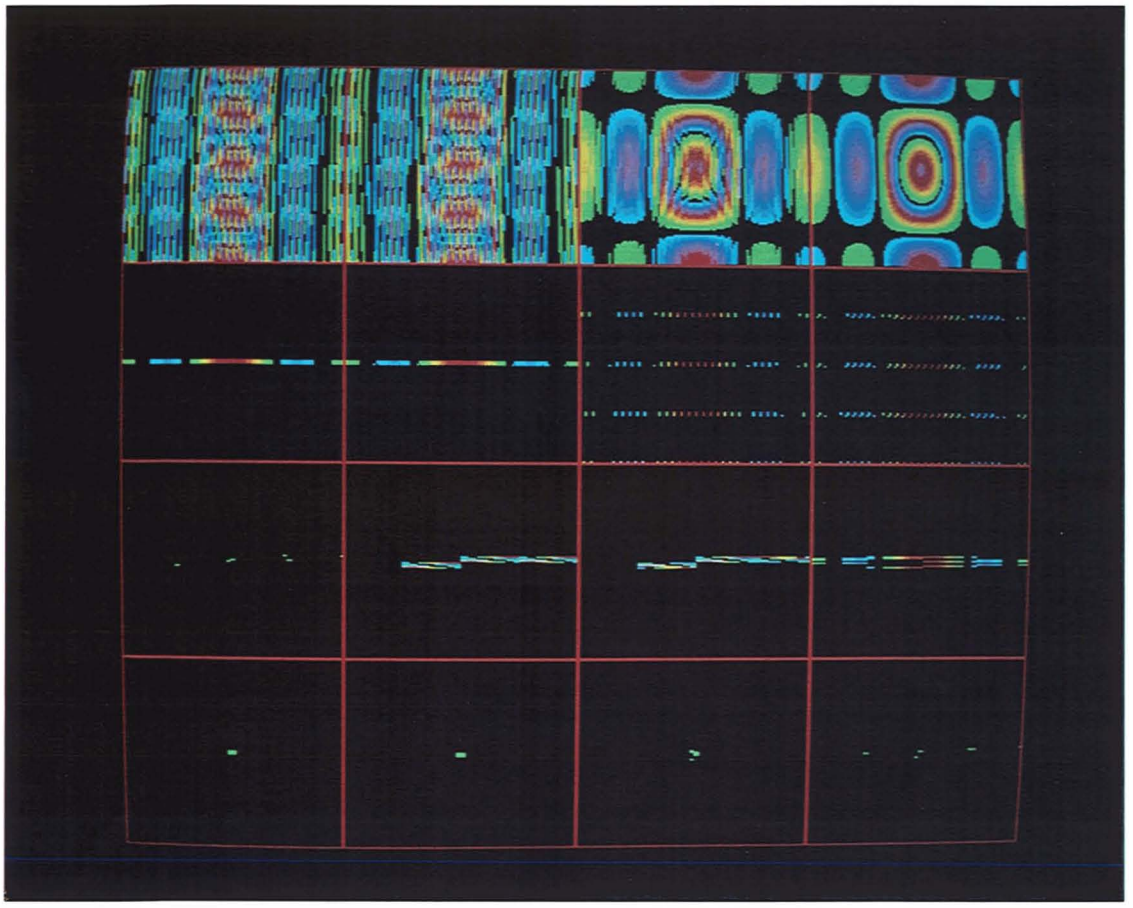


Fig. E — The SP generated this two-dimensional FFT image. The bottom left image is the original gating-function input. The images progress left to right, bottom to top, and include the transfer from natural order to row order. An FFT is performed on the shifted data. The results of the FFT are transferred to natural order and then from natural order to column order, at which point an FFT is performed. Finally the results are transferred back to natural order. The image at the top right shows the resultant synchronous function.

All inter-CE data communication for a two-dimensional FFT can be performed in lockstep without wasted motion. This algorithm for two-dimensional FFTs has been implemented on the SP — the results are shown in Fig. E. This technique is also applicable to matrix multiplication.

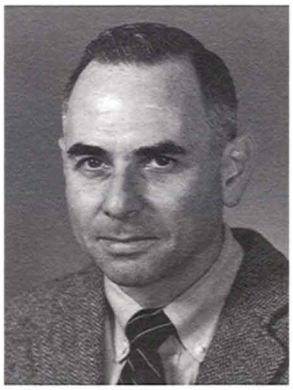
For a single CE, a complex FFT of length N takes $9\ N \log_2 N$ cycles. An $N \times N$ row-column transform takes $18\ (N/n)^2 \log_2 N$ cycles ($n = 8$ for the SP). The communications overhead required to access off-chip data for the location-dependent roll used to reconfigure the data adds $40\ (N/n)^2$ cycles. The communication time, which includes loading and unloading the communication register as well as performing the actual shifts, is $(N/n)^2\ (4n^2 + 4)$ cycles. Since the SP's clock rate is 6 MHz, the computation and communication times to perform a $1{,}024 \times 1{,}024$ full complex transform are

Computation time = 0.60 s
Communication time = 0.71 s.

If a single transform is performed, the communication and computation times are additive. But if several transforms are performed in succession, the communication and computation functions can be overlapped, which reduces the effective time to the greater of the two components. For transforms of larger size, the computation time eventually exceeds the computation time. Real to Hermitian transforms can be performed in half the time required for full complex transforms.

REFERENCES

1. M. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Computers* **C-21,** 948 (1972).
2. D.L. Slotnick, Borck, C.W., and McReynolds, R.C., "The Solomon Computer," *Proc. of the Fall Joint Computer Conf.* **22,** 97 (1962).
3. D.L. Slotnick, "The Fastest Computer," *Scientific American* **224,** 76 (February 1971).
4. D.W. Hillis, *The Connection Machine,* MIT Press, Cambridge, MA (1985).



IRA GILBERT is the group leader of the Processor Systems Group. He received a BS in physics from the Polytechnic Institute of Brooklyn and a PhD in physics from Harvard University. Ira's research is focused on digital signal processing. As a hobby, he studies neuroscience.