PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

A VLSI Building Block For Massively Parallel Computation

Abhaya Asthana
Boyd Mathews
Cheryl J. Briggs
Mark R. Cravatts

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

SMOKE is an experimental 32-bit pipelined processor that integrates arithmetic, logic and communication functions in a single VLSI chip. Packaged in a 256 pin grid array it provides separate paths to local program and data memories, four parallel ports for fast inter-processor communication and a fifth parallel port for communication with global memory and global control unit. This is in addition to providing full 32 bit integer and floating point operations. The combination of computation power and communication facilities present in SMOKE make it ideally suited as a building block for constructing processor arrays. In this paper the architecture for the SMOKE processor is described, and its application in constructing massively parallel machines is illustrated with a system architecture that we are currently developing.

1. INTRODUCTION

SMOKE is an experimental 32 bit floating point processor designed and fabricated in 1.25 micron CMOS technology for upto 25 MFLOPs operation [1]. It is intended to be used as a building block for parallel architectures in which computation is performed cooperatively by many processors of modest size and capability. Smoke features a fully integrated 32-bit floating point/integer unit, four parallel ports for inter-processor communication, a parallel port for global communication, and a small but powerful instruction set that includes floating point, integer, logic, control and communication instructions. Figure 1 shows a block diagram of the Smoke processing element module consisting of the Smoke processor with its local data and program memories. The processor chip consists of 60,000 transistors and has an area of about 8mm x 6mm. Figure 2 shows a photograph of the chip layout.

Our main motivation for building Smoke is to study the issues in architecture, hardware and software for parallel processing systems by actually constructing a test bed system. Our research goal is to discover organizational approaches that are well suited for VLSI implementation and to provide feedback to our design effort. One particular area of interest to us is the serial *overhead* involved in execution of parallel programs. Even a very small overhead can significantly limit the speedup of a parallel processor. For example, a 0.1% overhead in a system with 1000 processors will limit the peak performance to 100x in accordance with Amdahl's law [2]. A significant part of our effort is focussed at

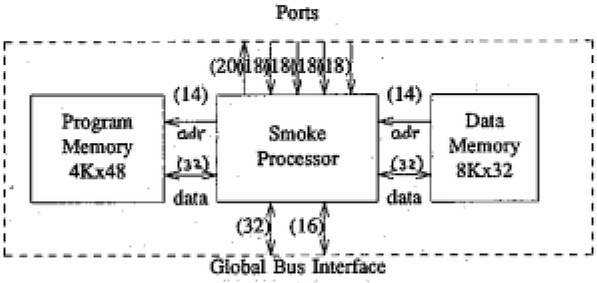

Figure 1. Smoke Processing Element Module

understanding the factors that give rise to this overhead and in developing techniques and principles to reduce it.

The following needs of structured parallel computation have influenced the architecture and design of the Smoke processor:

- The first property of structured numerical computation that we exploit is *locality of reference*. What this means is that once a task is activated within a processor, it will continue to execute without making many global references for a significant period of time. It is this period of time that determines the communication to computing ratio of the system. The lower the ratio, the more efficient is the system. In order for the task to execute independently it requires local resources namely, program and data memory. The Smoke architecture provides these resources.
- Secondly, much of the parallelism inside a task is of a structured nature that can be extracted efficiently by a pipelined architecture at the processor level. While we do not provide a vector instruction set, we have designed a software reorganizer to recognize and provide high throughput on vector type constructs [3].
- The third need of parallel computation is *communication of partial results* between processing elements as opposed to communication with a global shared memory. As the computation proceeds from one iteration to the next, results may have to be written to a shared memory structure, passed to a neighboring processing element, or transmitted to a distant processing element. The frequency with which the communication happens and the amount of data transmitted is application dependent. What is important in achieving high execution rates is low

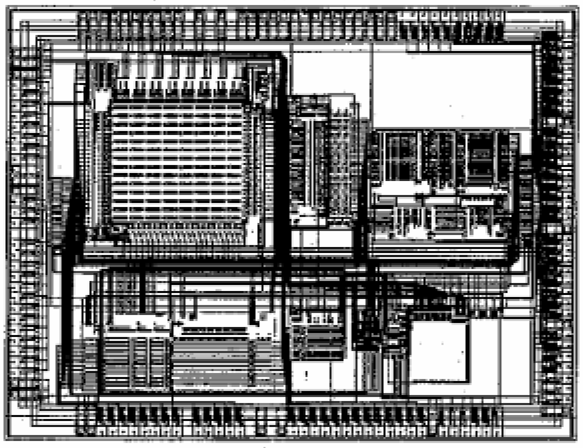
Figure 2. Smoke Chip Layout

latency and high bandwidth for communication between processing elements. Smoke addresses this for numerical algorithms that map well into processor arrays. For this architecture, it provides the ability to communicate results to nearest neighbors via ports in one cycle. Such an interconnect could be a nearest neighbor mesh or a binary hypercube, for instance.

- The fourth requirement of many parallel computations is an *efficient means for accessing large data structures stored in shared memory*. This implies a low latency, high bandwidth path to memory from every processing element. Also implied is hardware for resolution of contention for this path, if any, and for locking and unlocking of test_and_set flags for coordinating access to shared resources.
- Finally, *synchronizing* the activities going on in various processors is another requirement. This depends on the control scheme used for a particular multiprocessor application. In an SIMD scheme the host processor requires the ability to control the execution of the processing elements individually and in groups. In an MIMD scheme the processing elements need the ability to be able to synchronize with each other in a reasonable way without undue loss of code density or performance. Smoke provides facilities to handle these synchronization issues.

We envision the Smoke processing module to be embedded in a global architecture such as that shown in Figure 3. Smoke processing elements are combined into *n* groups where each group has upto eight processing elements in it. As mentioned earlier, there are two distinct types of communication that are supported in this architecture. At the computational level, the processing elements communicate with each other through the inter-pe-network. This is a point to point network making use of the parallel ports in the processor, that could range from a 2D Mesh to a Hypercube depending on the application. For control and access to shared data, the processing elements use the global network. The Global Control Unit (GCU) provides the host machine functions in this architecture. It also provides the control and coordination functions in applications that use SIMD strategies. For applications that use MIMD operation, the GCU acts merely as a more capable processing element. Each group has a Global Interface Unit (GIU) associated with it. The function of the GIU is to provide the means for passing data and control information between the processing elements and the GCU and shared memory.

The architecture is hierarchical with respect to data storage elements and also with respect to execution control. It represents a merger of a number of techniques for obtaining speedup. At the lowest level we have the Smoke processor and its associated local program and data memories. We get speedup at this level through pipelining and locality of reference. At the next level the Smoke processors are combined into groups (arrays, clusters, etc) and the speed up is obtained through array parallelism. At the highest level, the architecture is a shared memory multiprocessor.

2. SMOKE ARCHITECTURE

Smoke is a seven stage pipelined architecture (see Figure 4). By providing a wide instruction word format, and separating the instruction and data paths for the most

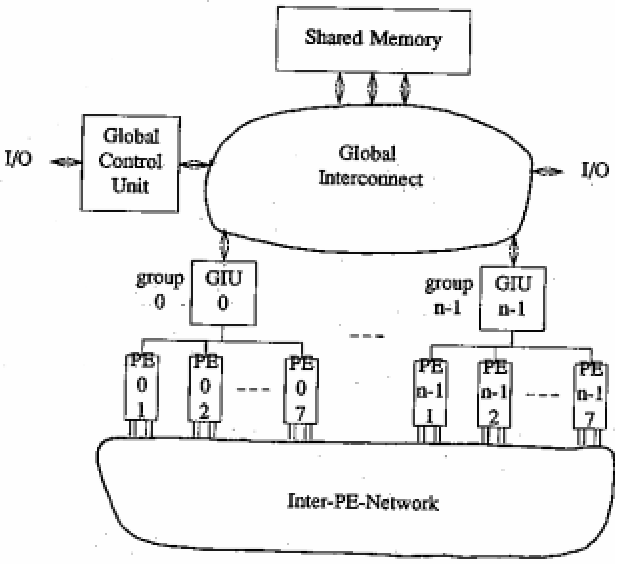

Figure 3. System Context for Smoke

part, the effective address computation and the instruction decoding proceed in parallel. The number of execution stages depends on the instruction type and the context. Two considerations dominated the design of Smoke execution unit. First, elimination of the co-processor model for performing floating-point operations [5,6], and second, achieving a closer match between the peak and the sustained performance of the processor [7,8]. We felt that a co-processor approach to constructing such a compute engine is inefficient in its operation at the hardware level and is difficult to use at the program level. Low latency in Smoke is achieved by using redundant hardware to realize separate pipelines for different operations.

2.1 Programmer's Model

The programmer's view of Smoke consists of local program memory, local data memory, index registers, status word and global memory as shown in Figure 5. There are eight index registers numbered ixr0-ixr7. These

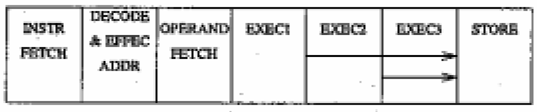

Figure 4. Pipeline Stages in The Smoke Processor

registers are 13 bits wide and serve as pointers to data stored in the local data memory. All of these registers can be used as stack pointers since push and pop instructions apply to all of them uniformly. However, register ixr7 is special and is called the "frame pointer". It is used by the processor as a push down stack pointer during call, return, interrupts and exceptions.

The local program memory is 48 bits wide and has storage for 4K instructions. The lower 9 locations are reserved for vectors. The data memory is an 8K word 32-bit wide memory. It is both the primary storage and the scratch-pad area in Smoke because there are no data registers. Typically, the upper part of the data memory is used as the stack area. Also mapped in the topmost address locations are the ports as shown in the figure.

The global memory is visible to the programmer as a large 32 bit wide memory that is indirectly accessible through move instructions. Smoke provides the facility to move data between its local memory and global memory. It also provides the facility to move programs from global memory into the local program memory.

The status word contains flags which indicate the result of the previous operations. These status bits are capable of causing exceptions unless they are disabled by setting the corresponding mask bit to zero. By default the machine is initialized with the mask set to zero. Additionally, upon entry into an interrupt or exception procedure, the mask is set to zero by default while the status word including the mask prior to the interrupt (or exception) is saved on the stack.

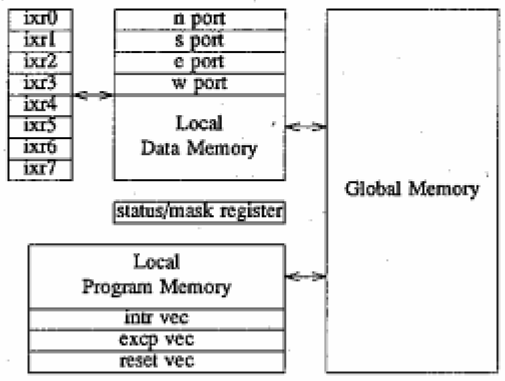

Figure 5. Programmer's View of Smoke

2.2 Instruction Format and addressing Modes

The instructions are 48 bits long in Smoke and are all of fixed length. We realize that the fixed length restriction is wasteful of program memory space but it does simplify the parsing of the instructions during the decode cycle. The form for a typical three operand Smoke instruction is shown in Figure 6(a). It consists of a six bit opcode and three operand specifiers: src1, src2, and dest. The destination operand could be a memory location, an index register or a jump address as in the case of branch or call instructions. The operand specifiers are interpreted based on the instruction and the addressing modes in keeping with a streamlined architecture. To keep things simple, Smoke provides only three addressing modes: absolute, indexed and immediate. The formats for these modes is shown in Figure 6(b). The first bit in an operand specifier is used to distinguish between absolute mode and the other two modes. The second bit is used to distinguish between the immediate and indexed modes.

2.3 Arithmetic/Logic Instructions

A major fraction of the Smoke processor hardware is devoted to arithmetic processing. Full 32 bit floating point add and multiply operations are supported directly in hardware. In addition, 32 bit integer arithmetic and logic instructions are provided. As mentioned earlier, special effort was made to keep the latency of these integer operations to a minimum. Conversion from 24 bit integers to floating point representation and from floating point to 24 bit integer representation is supported. Integer multiplication is done by the same array that is used in the floating point multiplier. Integer multiplication operates on two 24 bit numbers and yields a 32 bit result.

2.4 Program Control Instructions

All branch instructions in Smoke have delayed semantics. Unconditional and conditional branches have a two cycle delay. Call and return instructions are provided to handle procedure calls and also have a two cycle delay. These instructions use the frame pointer to push the return program counter on the stack. Since Smoke allows any index register to be treated as a stack pointer using push and pop instructions, arguments can be pushed on the frame or can be passed via a separate argument stack.

Because of differences in the lengths of the integer and floating point arithmetic pipelines, it is difficult to determine the separation between a compare and the following conditional branch instruction. To simplify the task of the programmer for integer compare and branch operations, Smoke provides a special test_and_branch instruction. This "tbra" instruction compares two integer values and branches based upon a specified condition always in a deterministic number of cycles (four).

2.5 Loop and Index Register Instructions

The index registers can be loaded, stored, and incremented by an arbitrary value. Push and pop instructions also apply to all index registers and automatically increment or decrement the specified register by one.

Smoke implements the Fortran DO-loop (or a form of the C FOR) statement in a single instruction. The increment_test_and_branch, "ixtb", instruction takes an index register as a source, increments it by a specified value, compares it to a specified final value and, if the condition is true, the program branches to a specified destination address. Availability of this instruction facilitates the implementation of inner loops commonly found in numerical applications.

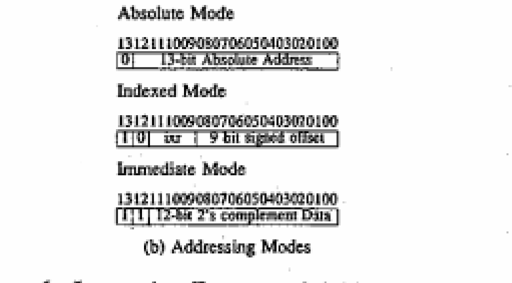

Figure 6. Instruction Format and Addressing Modes

2.6 Global Communications

In addition to providing floating point capability, Smoke dedicates a significant fraction of its on-chip resources to both local and global communications. This is most evident when the pinout for Smoke is considered. Two styles of global communication are supported:

- Processor initiated, in which a request for data movement or a signal is generated by a program executing within the Smoke processor.
- External control unit initiated, in which a program action in the global control unit causes data to be transferred or a control signal to sent to a processing element.

The processor can request data of arbitrary sizes to be moved between global memory and its local program or data memory. The *global_data_read* operation has the following syntax.

global_data_read(global_address,local_address,count)

This instruction will result in *count* words being copied from global memory location *global_address* into the processor's local data memory starting at location *local_address*. There is a separate instruction, *pmld*, to read code into program memory. The *global_data_write* operation, which has the same syntax, does the reverse move from local data memory to global memory. These operations are non-blocking in that the processor continues to execute subsequent instructions. The processor can check for completion of the transfer by monitoring the global access done bit (GAD) in the processor status register or can synchronize using a wait instruction as described later.

Communication initiated by the Global Control Unit is either for reading or writing the processor's memory, for starting a processor's execution or causing an interrupt. The reads and writes are block transfers and are transparent to any program that may be executing in the processor. This facility is useful for initializing a processing element with a code and or data for the next phase of computation while the current phase is still in progress. Similarly, the results of a previous phase of computation can be read transparently while the next phase is in progress. Another use is in debugging. The program or data memory can be read even though the program executing within the processor may have stopped. This is possible because the external read and write transactions are interpreted and executed completely in hardware.

2.7 Port Communications

There are four parallel ports in Smoke each of which allows 32 bits to be written to or read from every major cycle. They are all memory mapped, so accessing a port is like accessing memory. Naturally, all addressing modes that apply to memory apply to ports as well. A program can output a result to only *one* of the four ports in any given instruction. However, a result can be input into all of the four ports every major cycle. The ports can be used by a program in either blocking or non-blocking mode. Physically the ports are implemented using five sets of 16 lines, one set for output and four for input. Two clock phases are used to complete a 32 bit transfer in a cycle. There are additional lines for indicating port full condition and for data latching.

Smoke has been designed with efficient transfer between directly connected neighbors in mind. However, its use with arbitrary interconnection networks is not precluded. For use with an arbitrary network, an appropriate packet structure and protocol will have to be designed by the user.

2.8 Synchronization

Programs executing in a processing element have to synchronize with activities in other processing elements and with activities in the global control unit. In an MIMD environment, the synchronization with nearest neighbors is achieved through port reads and writes. Synchronization with distant processors can be achieved through shared data structures (e.g. semaphores) present in the global memory. The Smoke usage model supports this by reserving certain global locations for read modify write references. The same method may be used for synchronizing with GCU. Alternatively, if the programmer so chooses, programs running within processing elements could use message passing to transmit and receive events or data.

In an SIMD environment, the GCU needs the facility to simultaneously control the execution of programs in all processing elements. It also requires the ability to sense the completion of a task on a given processing element or a group of processing elements. This is achieved with two signals *go* and *done*. The protocol that uses *go* and *done* is very simple. The signal *go* is an input which when asserted by the GCU signifies to a processing element that computation can proceed. The processing element performs that task and thereafter signals the GCU of its completion by asserting done. The GCU on its end can assert the go lines to all processing elements in the systems at once, for a group of processing elements at a time or individual processing elements depending on the control algorithms suitable for a given application. Similarly, the *done* bits from all processing elements are monitored by the GCU to determine when the next stage of the computation should begin.

The synchronization function is manifested in the wait instruction in Smoke.

wait(event(s)_specifier)

The wait instruction can specify one or more events on which the processor can wait. When a wait is issued, the execution suspends until all the specified events have occurred and then the execution is resumed. Thus the wait implements an AND semantic. By waiting on the port buffers to empty before writing, and to fill up before reading the processor can make the ports into blocking ports. If such synchronization is not done by the program the ports are essentially non-blocking. The same applies for global operations.

2.9 Interrupt

An interrupt mechanism is provided in Smoke to allow an external agent to preempt the execution of the current task in a processor and redirect it to execute a pre-specified interrupt procedure. The actions taken by the interrupt procedure are under the programmer's control. Any further interrupts are masked upon entry into the interrupt procedure. Physically, the interrupt communication is through an interrupt pin and an interrupt acknowledge pin.

3. IMPLEMENTATION OVERVIEW

The main modules that make up SMOKE are shown in Figure 7. The program control unit is responsible for fetching the instructions, decoding them and managing the program counter and the processor status word. The

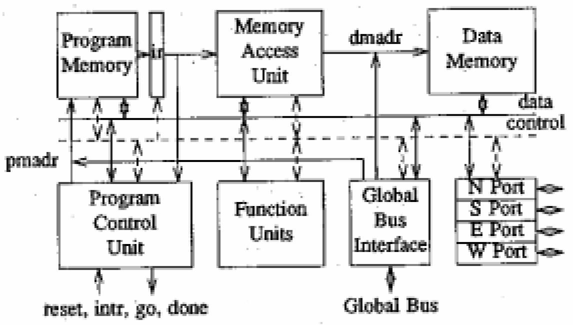

Figure 7. Smoke Organization

memory access controller performs the effective address computation and fetches the operands from the memory. It also manages the address queue for storing results back into memory. The function units perform the arithmetic and logic operations. The global bus interface unit provides the interface to the global bus and the GCU. The ports provide interface to the neighboring processors.

Smoke has separate instruction and data spaces. This allows the accesses to the program and data memories to be completely overlapped every cycle. Most operations in smoke are memory to memory operations due to the absence of data registers. Treatment of memory as the local register set places a heavy demand on the data bus. The processor uses a three phase clocking scheme as shown in Figure 8. This allows two source operands to be read in C1 and C2, and one result to be stored in C3 respectively, every major cycle. The memory must, therefore, run three times as fast as the processor pipeline stages to make this possible.

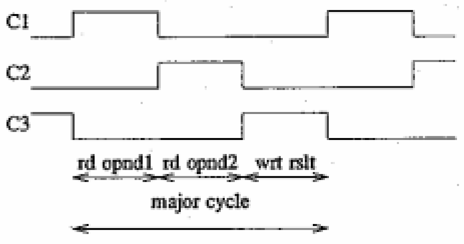

Figure 8. Three Phase Clocking

3.1 Pipelining

Smoke is a seven stage pipelined architecture. The seven stages of the pipeline are shown in Figure 4. The effective address computation and the instruction decoding proceed in parallel. This is possible because of the wide instruction format and the orthogonality of the addressing modes from the instruction type. The number of execute stages depends on the instruction type and the context, as mentioned in the last section.

Smoke has a 32-bit multi-function pipelined arithmetic and logic unit designed to operate at peak rate of 20+ MFLOPS [11]. The unit is physically divided into three separate pipelines: an integer ALU which is a single stage pipeline, a three stage floating-point adder, and a three stage floating-point and integer multiplier. The format of floating-point numbers entering and exiting the function unit is a restriction of the IEEE standard 754 for single precision floating-point numbers [9]. The integer ALU performs the standard arithmetic and logic operations. The floating-point adder performs the operations of add, subtract, fixed-point to floating-point conversion, and floating-point to fixed-point conversion [10]. The floating-point multiplier performs both floating-point multiplication and integer multiplication. The organization of the function unit pipelines is shown in Figure 9. The use of separate pipelines simplifies the hardware implementation because the need for resource scheduling is eliminated. A new set of operands can enter the function unit every major cycle.

The length of the function unit pipes was kept as short as possible. Another objective was to keep low latency for logic and integer operations in the presence of floating point operations. In short, we concluded that best improvements in the overall system performance would come only by improving the vector and the scalar performances in a balanced manner [7]. Low latency in integer operations is achieved by having a short integer pipe. However, this requires a mechanism for resolving conflicts that we describe in the next section.

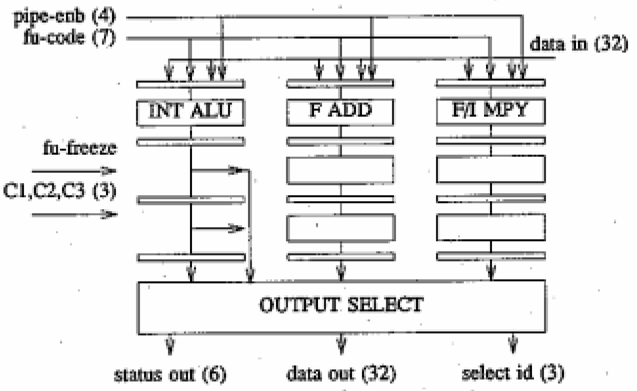

Figure 9. Function Unit Pipelines

3.2 Conflict Resolution and Reorganizing

A contention for the output bus occurs when results from two pipelines of unequal lengths arrive at the output in the same cycle. The function unit module handles this by adding buffers to the integer pipeline to effectively make all pipelines the same length (We still allow results from the short integer pipe to be written as early as they can be). When all pipelines are the same length we are guaranteed that only one can have a result at the output in any cycle because only one operation can enter the pipeline in any cycle. In cycles where no result has come to the end of a long pipe, and a buffered result from a short pipe is ready to be stored, it will get written into data memory. Specifically, the result at the integer pipeline is normally enabled unless there is something at the end of either of the longer pipelines. In that case, the result at the short pipeline is buffered and must wait until the next cycle. This ensures that in case of a conflict, the result that has stayed in a pipeline the longest gets written out first.

Consider the example shown in Figure 10. The major clock ticks are shown along the horizontal axis and the pipeline stages along the vertical. The *addf* instruction is the first to enter the pipeline followed by *addi, subi, subf* and *addi*. The *addi* is the first to complete in tick T6 and its result is stored. In the next cycle both *addf* and *subi* finish simultaneously and compete for the store operation. Since *addf* has been in the pipe longer, it gets precedence. The result of *subi* gets passed to the next pipeline stage, and gets stored in cycle T8. In tick T9 the result of *addi* is written from the shortest store path, and finally, result of *subf* is stored in T10.

The ill effects that could arise from reordering of the result stream are avoided by proper reorganization and pacing of the input instruction stream [3]. We believe that a multiple latency pipeline creates a scheduling problem which is manageable in software and yields improved performance over a multifunction pipeline of uniform length. Note that reordering of independent operations can give faster and denser code than automatic insertion of *nops* where stalling is required. Hardware interlocks to handle data dependencies can yield denser code but not faster execution since the *nops* are merely being implemented as waits in hardware.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
IFETCH	addf	addi	subi	subf	addi					
DEC		addf	addi	subi	subf	addi				
OPFET			addf	addi	subi	subf	addi			
EXEC1				addf	addi	subi	subf	addi		
EXEC2/S					addf	(addi)	subi	subf	(addi)	
EXEC3/S						addf		(subi)	subf	
STORE							(addf)			(subf)

Figure 10. An Example of Conflict Resolution

3.3 Delayed Branching

Because of pipelined execution in Smoke, the branches are delayed. This implies that a number of instructions following the branch instruction will get executed whether the branch is taken or not. Once an instruction has been decoded, its processing continues on its own as it moves down the pipeline. All the necessary state information is passed down from stage to stage. Smoke does not do branch prediction, instead, it relies on instruction reorganization to get maximum possible performance. The number of instructions that get executed after a branch depends on the particular type of branch instruction. The number is two in case of *call, ret, bra* and *cbra* and four in case of *test_and_branch* and *increment_test_and_branch*. If a program counter modifying instruction is placed in the two or four cycle window following a branch, it will lead to the undesirable program behavior.

3.4 Safe Period and Handling of Abnormal Conditions

The streamlined execution of a pipelined machine can get severely disrupted when an abnormal event occurs. Instructions that do not modify the program counter directly, such as arithmetic, logic, load and store, do not present much of a problem because once the instruction has been decoded its operation affects the state of only the succeeding stages of the pipe. An instruction that modifies the program counter, however, presents a difficulty.

Consider the case in which the decoding of the instruction immediately following a branch is disrupted due to an interrupt. In the normal course of events, the two instructions following the branch instruction would get executed. Since in the previous cycle the branch instruction was decoded, in this cycle (in which the following instruction is being decoded) the program counter will get updated to its new value. At the same time the interrupt is recognized and the machine enters the interrupt sequence. In the interrupt sequence, the return value of the program counter will be the new branch address that got loaded. Thus, when the program is resumed after return from the interrupt procedure, execution will begin from this branch address. The error is that the original two instructions following the branch that were supposed to have been executed did not get executed.

To ensure proper program behavior, the logic in Smoke allows the instruction following the branch to finish execution before an interrupt or hold is accepted. In a sense there is time period during which it is not "safe" to allow abnormal conditions to come in. In designing this circuit another problem we came across was deciding on the length of this window, because it is dependent on the instruction type. A close examination of the instruction set revealed three window sizes. For non-branch instructions the size was 0, meaning that interrupts do not get delayed at all. For branch instructions the window is 2 cycles. For test_and_branch and loop branch instructions, the window is 4 cycles. Note that these represent the worst case delay experienced by the interrupt or wait conditions. The size of the window is encoded by the control PLA with every instruction it decodes. This is then used to generate an inhibit signal of that length. This inhibit signal prevents the abnormal condition from being recognized until it is "safe" to do so.

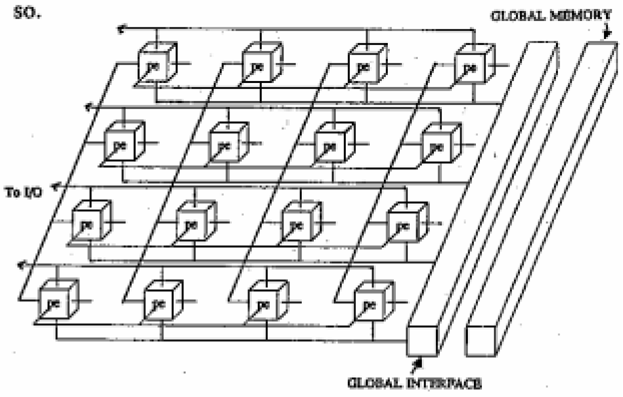

Figure 11. A Two Dimensional Processor Array

4. APPLICATION

Smoke processors can be configured in various ways to produce large processor arrays. One such array is shown in Figure 11. A two dimensional bus structure is used to interconnect the processors along rows and columns. The communication protocol supported in this structure allows a processor to send or receive messages to any other processor in the system. Furthermore, it allows broadcast communication along any row or column. The broadcast communication is blocking in nature. Thus, a processor doing a broadcast will block until all the recipients can receive the message. The scheme can be extended to three dimensions for larger arrays while still maintaining the regularity of interconnect and uniform wire lengths. As shown in Figure 12 the fourth port is used for high bandwidth connection to I/O, while the global port for all processors along a column in a given computational plane provides the connection to global control and memory. A complete system architecture is shown in Figure 13. We envision each computational plane to be eventually constructed on a wafer substrate using the advanced VLSI packing technology [4,12,13].

The architecture described above has several characteristics that match well with many important applications both in the area of structured numerical computation and in symbolic computation. One application that is of particular interest to us is circuit simulation which has a mix of both symbolic and structured numerical computation. The input phase in circuit simulation where the node equations are being formulated and the matrix being set up is mostly symbolic. The *load* phase in which the matrix elements are being computed by evaluating the device models for each iteration of the Newton's method is computationally expensive but has little structure to it. The *solve* phase in which the system of linearized equations have to be solved using Gaussian Elimination method is computationally intensive and is structured as well. The broadcast communication is ideal for transmitting the pivot element to all members in a row and for transmitting the factors along the rows and columns that are required in update operations. By properly mapping the large matrix to be solved on a finite-sized array, we manage to preserve the regularity of communication. The fact that a sparse matrix representation is used for the matrix to be solved requires an organization of the data within each of the local data memories such that the access time is minimized. We are currently designing hardware solutions to further reduce this access delay.

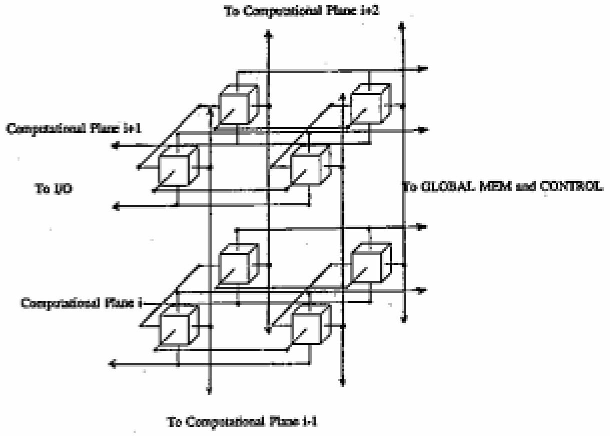
To Computational Plane i+2
Computational Plane i+1
To I/O
To GLOBAL MEM and CONTROL
Computational Plane i
To Computational Plane i-1

Figure 12. A Three Dimensional Processor Array

5. PACKAGING

The parallel architecture shown in Figure 13 places new requirements on the packaging technology used to interconnect devices and assemble systems. These requirements include the assembly of high pinout (up to 500 IOs) devices, the ability to sustain synchronous system operation at frequencies up to 100 MHz, the propagation of pulses with rise times less than 2 nsec, and cooling at thermal loads greater than 1 watt/sq.cm. A system packaging which is based upon the use of individually packaged devices and PWBs tend to limit the performance of VLSI devices due to parasitic effects. The AVP technology [4,12,13] overcomes many of the limitations of conventional packaging. The substrate is a silicon wafer. An integral bypass capacitor is fabricated on the surface of the wafer. Devices are attached to the interconnection substrate by means of solder, which allows pads to be placed over the area of the chip, is repairable and provides low inductance connection. Multichip packaging allows ICs to be tested individually before attachment so that only functional devices are assembled.

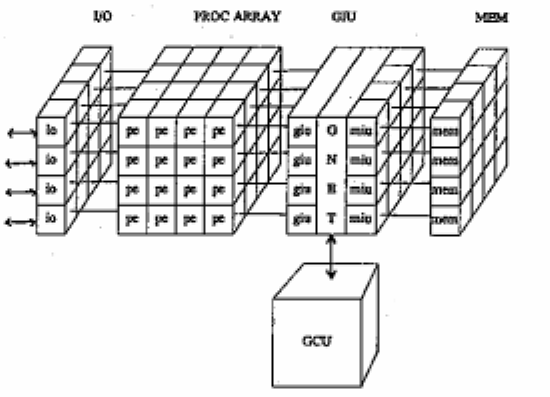

Figure 13. System Architecture

We hope to package a 2x2 array in our prototye system on a single wafer. As the design rules shrink to 0.5 mcron feature sizes and the wafer sizes for AVP grow to 12"x12", it will be possible to construct larger arrays such as 8x8 or 16x16 directly on a single wafer substrate. Three dimensional structures can be constructed by stacking wafers. With such a packaging technology, a modest sized 4x4x4 array with a 25 MFLOP Smoke processors, will be capable of delivering 1.6 GFLOPs in a cube of size 6"x6"x6". And, a 8x8x8 array would result in a peak performance of 12.8 GFLOPs in a cube of size 10"x10"x10".

6. CONCLUSION

The combination of computation power and communication facilities present in SMOKE make it ideally suited as a building block for constructing processor arrays. In this paper the architecture for the SMOKE processor was described, the rationale for the design is discussed and the system architecture of our machine currently under development was described. Given the architecture and the packaging technology described here, we believe it is possible to build a system to deliver 10+ GFLOPs in a size of 1 cu.ft.!

Our research focus has been at exploring techniques that will help reduce the gap between peak and sustained performance of parallel systems. This, of course, is possible only with a careful combination of VLSI design, machine organization, compiler technology, and, ultimately, understanding of the application programs and reducing them to efficient parallel algorithms.

7. REFERENCES

[1] A. Asthana, B. Mathews, K. Padmanabhan, "Architectural description of the MAP Processing Element (SMOKE)," AT&T Bell Laboratories Internal Report, Nov. 1985.

[2] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," AFIPS Conf. Proc., Vol. 30, Thompson, Washington, DC, 1967, pp. 483-485.

[3] S. Abraham and K. Padmanabhan, "SRS: An Instruction Reorganizer and Simulator for the SMOKE Processor," AT&T Bell Laboratories Internal Report, Nov. 1985.

[4] C. J. Bartlett, "Advanced Packaging for VLSI," Solid State Technology, June, 1986, pp 119-123.

[5] Moshe Gavrielov and Lev Epstien, "The NS32081 Floating-Point Unit," IEEE Micro, April 1986, pp. 6-12.

[6] Howard Sachs and Walt Hollingsworth, "A High Performance 846000 Transistor UNIX Engine- The Fairchild Clipper," Proc. ICCD 85, Oct. 1985, pp. 342-346.

[7] James J. Hack, "Peak Vs. Sustained Performance in Highly Concurrent Vector Machines," Computer, September 1986, pp. 11-19.

[8] G. S. Patterson, Jr. "Large Scale Scientific Computing - Future Directions," Computer Physics Communications, Vol. 26, Nos. 3 and 4, June 1982, pp. 217-225.

[9] "IEEE Standard for Binary Floating Point Arithmetic," (ANSI/IEEE Std. 754). IEEE. 1985.

[10] J. B. Gosling, "Design of Arithmetic Units for Digital Computers," Springer Verlag, New York, 1980.

[11] A. Asthana, C.J. Briggs, M. Cravats, B. Mathews, and K. Padmanabhan, "A High Speed Multiple Pipeline Function Unit as a Building Block for Parallel Architectures," Proc. ICCD 87, Oct. 1987, Port Chester, N.Y.

[12] H. J. Levinstein, C. J. Bartlett, W. J. Bertram Jr., "Multi-Chip Packaging Technology for VLSI Based Systems," Proc. ISSCC, (1987).

[13] W. J. Bertram, Jr., "High Density, Large Scale Interconnection for Improved VLSI System Performance," Proc. IEDM, 113 (1987).