P. W. Agnew
A. S. Kellerman

Microprocessor Implementation of Mainframe Processors by Means of Architecture Partitioning

The benefits of Large-Scale Integration (LSI) implementations have applied quite naturally to processors with relatively low performances and simple architectures; e.g., the one-chip microprocessors used in personal computers contain several thousand logic gates. Mainframe processors, however, have so far been limited to using logic chips that contain several hundred logic gates. The best use of LSI logic employs microprocessors to keep critical paths on chip, thus keeping pin counts and power dissipations within reasonable limits. Microprocessors have been extensively used to implement peripheral functions, such as I/O device control. However, as of this writing, a single state-of-the-art microprocessor cannot contain a mainframe processor function. Therefore, new machine organizations are needed to use today's state-of-the-art microprocessors to implement a mainframe processor. This paper examines several methods for applying LSI and microprocessors to the design of processors of increasing performance and complexity, and describes a number of specific approaches to microprocessor-based LSI implementation of System/370 processors. The most successful approaches partition the System/370 instruction set into subsets, each of which can be implemented by microcode on a special microprocessor or by programs written for an off-the-shelf microprocessor.

Introduction

One characteristic of the era of integrated circuits has been that higher-performance computers use lower levels of integration. This is the result of individual optimizations across the performance spectrum. In 1980, first customer shipments of a microcomputer [1], a mainframe [2], and a supercomputer [3] used, respectively, a 68 000-transistor microprocessor chip, 2000-transistor LSI chips, and 500-transistor chips. The price of a silicon chip is roughly independent of the level of integration, so the price per gate is lower for microcomputers than for supercomputers.

One result of this situation has been the repeal of Grosch's Law [4], which says that if one pays twice as much for a computer, one obtains the square of that or four times as much processing power. This implied that one obtained the best cost/performance from the largest computer that could be justified by sharing it among many unrelated users and applications. In recent years, the exponent in that relationship has dropped from two to less than one [5]. As a result, one now obtains the best cost/performance from the smallest computer that can perform one's application in an acceptable time. Note that the exponent in the relation between function and cost is still greater than one for disk files [6], printers, and some other peripheral devices. Therefore, it is still economical to share these peripherals among several users and applications.

LSI has been very effective in reducing the costs of memory of all sizes, but has been much more effective in reducing the costs of low-performance processors than in reducing the costs of high-performance processors with complex architectures. This favors the implementation of high-performance computers using large numbers of low-performance processors and memories. However, this implementation is difficult to apply to existing complex architectures intended mainly for uniprocessors which process a single stream of instructions. Application to multiple-instruction multiple-data architectures and single-instruction multiple-data architectures [7] is more straightforward and will not be treated here.

This paper summarizes the ways in which LSI has been applied to low-performance digital systems and is being applied to systems of increasing performance and complexity. It then describes seven approaches to the partitioning strategy that was developed for implementing the processor portions of a complex architecture, specifically System/370, using state-of-the-art microprocessors such as the Motorola 68000. The advantages and disadvantages of these approaches are also discussed. Each of these approaches might be the appropriate choice for a specific set of objectives. Some of these trade-offs are also discussed.

Use of LSI in low-performance systems

Recent improvements in the cost/performance of digital computer systems have been driven by the availability of increasingly dense LSI chips. Denser LSI memory chips, with reduced costs per bit stored, have direct and obvious applicability to digital systems over the entire range from hand-held calculators to supercomputers. However, denser LSI logic chips apply most naturally to digital systems near the low end of the performance and complexity spectrum.

LSI applies very naturally to very small digital systems. The logic portion of a hand calculator, microwave oven, or wristwatch, including the necessary memory and I/O device interfaces, can be implemented on a single LSI microcomputer chip.

The processor of a small personal computer can be implemented on a single microprocessor chip which implements the instruction set of the computer, together with other LSI chips which implement the interfaces between the microprocessor and the memory, keyboard, display, disks, printers, and communication lines. This is an example of partitioning the function of a digital system for implementation by several LSI chips.

The partitioning method is simple, well known, and straightforward because the instruction-processing function can be implemented entirely by a single chip.

Use of LSI in larger systems

Strategies of applying LSI technology to the implementation of still more powerful digital systems, in which the state of the LSI art does not permit implementing the entire instruction-processing function on a single LSI chip, are far less obvious. Some strategies to be considered are as follows.

1. Procrastination

Wait until technology advances far enough to allow the desired architecture to be contained within a single chip. For example, the architecture of each generation's state-of-the-art microprocessor was determined by the then-current capability of the technology, which explains why today's leading microprocessors lack floating-point instructions. The significant disadvantage of this method is that it precludes implementing a predefined architecture that does not happen to fit within one chip in the current technology. This has led to the major software problems inherent in having each generation of microprocessors implement an essentially new architecture.

2. Off-chip microcode

The second method is to partition the instruction execution function so that the data flow is on one chip and the microcode that controls the data flow is on one or more separate chips. This method is the obvious application of LSI technology separately to the data flow and to the control store.

Unfortunately, this relinquishes the main advantage of LSI implementations, namely, the advantage of having both the control store and the data flow that it controls on the same chip so that the critical path remains on one chip. In most processors, the critical path runs from the control store, to the data flow, to the arithmetic result, and to the address of the next control-store word. Its length, in nanoseconds, determines the microcycle time and hence the instruction-processing rate of the processor. For a given power dissipation, a critical path that remains within one LSI chip means a shorter cycle time than one that must traverse several inches and several chip-to-card pins.

This method also requires what LSI technology is least adept at, namely, large numbers of pins. The data-flow chip needs at least a dozen pins to tell the control store what microword to give it next. Worse, the data-flow chip needs from 16 to 100 pins to receive that control word. A processor using this method is often limited to roughly 16-bit control words (and hence a vertical microprogram that can control only one operation at a time), whereas a processor with far higher performance could be designed if a 100-bit control word were available (to allow a horizontal microprogram that can control several operations in each microcycle and thus perform a given function in fewer cycles).

The off-chip microcode partitioning method has been particularly successful when applied to bit-slice processors, in which the data flow is not yet reduced to a single chip but rather is a collection of chips, each of which implements a particular group of bits throughout the data flow. Bit-slice processors usually employ bipolar technologies whose densities are limited by the number of gates available (or the ability to cool them) rather than by the numbers of pins on the chips. The off-chip partitioning method applies to FET implementations only in more unusual cases where many pins are available and the chip density is a good match for the number of gates needed to implement just the data flow of a

desired processor. The Toshiba T88000 16-bit microprocessor [8] happens to meet these conditions. Such an implementation can be best viewed as a bit-slice design in which the implementable slice width has widened to encompass the entire desired data flow.

3. Instruction-set partitioning

A third method of implementing an architecture that is too complex to implement on one chip is to partition the instruction set of the architecture itself, that is, select subsets of the instructions and provide a microprocessor to implement each subset. This method preserves the main advantage of a one-chip implementation, namely, keeping each critical path on a single chip. For each subset of the instructions, the corresponding microprocessor chip contains the data flow (including the registers) necessary for the execution of that subset and also contains the microcode that controls execution. The application of this strategy requires the following:

- A partitioning that makes each subset fit on one microprocessor in the current state of the technology.
- A way to pass control back and forth between the microprocessors quickly.
- A suitable way to pass data back and forth between the microprocessors.
- A technology in which it is economically feasible to have several copies of a complex data flow and control-store mechanism.

The last requirement implies that instruction-set partitioning never applied to low-performance or medium-performance processors before the current generation of low-cost LSI technology. The other requirements imply that instruction-set partitioning was not considered until the disadvantages of the other two strategies had become very clear.

State of the industry

Each major microprocessor manufacturer has faced the need to implement an architecture more complex than can be put onto a single LSI chip [9–11]. Some needed to implement pre-existing architectures to achieve software compatibility with installed machines. Others needed to enhance the functions of existing successful one-chip microprocessors by adding more instructions. Some examples employing 16-bit microprocessors, and the methods used to extend their architectures, follow.

Digital Equipment Corporation, having identified the need for a low-end implementation of their ®PDP-11 [12] minicomputer architecture, chose the off-chip microcode method. The result was the ®LSI 11 four-chip set manufactured first by Western Digital and then by ®DEC itself [13].

®Intel [14] determined a need for additional hardware computational power, particularly floating-point instructions, for its 8086 microprocessor systems. They developed a "coprocessor," the 8087 [15]. A processor containing both an 8086 chip and an 8087 chip operates as follows. The chips fetch each instruction simultaneously. If the instruction is one that the 8086 can execute, it executes the instruction and both chips fetch the next instruction. If the instruction is one that the 8087 executes, the 8087 starts to execute it. In the usual case where a main store address is required, the 8086 computes the address and puts it on the bus shared with the 8087. The 8087 uses that address to complete execution of the instruction and then signals the 8086 that it is ready for both of them to fetch the next instruction. Thus, each chip looks at each instruction and executes its assigned subset, but only the 8086 computes addresses.

®Zilog similarly identified a need to add floating-point instructions to its Z8000 microprocessor [16], and it developed an Extended Processing Unit or EPU [17]. A system containing a Z8000 and one or more EPUs works as follows. The Z8000 fetches an instruction, and if it can execute the instruction, it does so. Otherwise, the Z8000 issues a request for service by an EPU and supplies an identifier (ID) that it determines by examining the instruction. One EPU recognizes that ID as its own and begins executing. The EPU can use special wires to the Z8000 to instruct the Z8000 to move necessary data back and forth between the EPU and the main store. The Z8000 proceeds to fetch and execute more instructions while the EPU is working, and only stops to wait for the EPU if it needs to request service by the same EPU while that EPU is still busy. Thus, it is the responsibility of the Z8000 to start the EPU and respond to commands from the EPU. A great deal of execution overlap is possible in such a system.

National Semiconductor Corporation had a similar requirement to add floating-point instructions to its NS-16000 microprocessor systems. It called the NS-16000 a "master" and called the computational processor a "slave" [18, 19]. In a system containing a master and a slave, the master fetches instructions and executes them if it can. When the master fetches an instruction it cannot execute, it selects a slave to begin execution. The master sends the instruction and any needed data to the slave, waits for the slave to signal completion, receives the result, and proceeds to fetch the next instruction. Thus, the master never overlaps its execution with the slave's execution and is responsible for knowing what the slave is doing and what the slave needs.

Data General Corporation wanted an LSI implementation of its ®Eclipse minicomputer architecture [20]. The resulting ®MicroEclipse family [21] employs a one-chip processor that contains the data flow as well as the horizontal (35-bit) and vertical (18-bit) microcode for executing the most performance-critical instructions in the architecture. This proces-

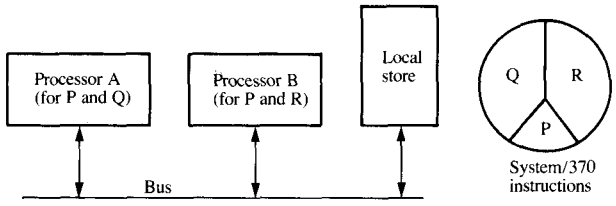

Figure 1 Two overlapping subsets.

sor can call for vertical microwords from an off-chip control store, as necessary, to execute the rest of the instructions of the architecture by making use of the on-chip horizontal microwords. This is a variant containing some of the advantages of both the off-chip control store method and the instruction-set partitioning method.

Seven approaches for implementing a complex architecture

Our group began studying ways to use LSI logic to implement complex processor architectures in November 1977. At that time, off-chip control store designs were common, and designs that partitioned off I/O functions for implementation on dedicated microprocessors were becoming common. None of the advanced microprocessor partitioning methods previously discussed had yet appeared. Partitioning of functions within a central processing unit for implementation on separate processors had been employed in supercomputers. Their goal was separate execution units for fixed-point, floating-point, and perhaps decimal instructions, that could overlap execution to achieve maximum throughput. Our goal was to optimize cost/performance (not performance) at the low end of the mainframe spectrum. Our investigations did not consider high-end processors. It required a year of analysis of many different approaches to using LSI for implementing a mainframe architecture to determine that one of the best approaches to the design of very low-cost mainframes was to use a supercomputer organization such as that exemplified by the IBM System/360 Model 91 [22] without execution overlap.

During the second year of our investigation, we selected one specific mainframe architecture, the System/370 [23] and one specific microprocessor, the ®Motorola [24] 68000 [25, 26]. This was shortly after the 68000 was first mentioned in the trade press. We selected this microprocessor because it was heavily microcoded, was intended to be general-purpose, and had 17 32-bit general registers.

Before launching an investigation of partitioning, we estimated how long it would take for advances in technology to allow implementation of all of System/370 architecture on one microprocessor; *i.e.*, we looked at the technology represented by the 68000, and at the silicon requirements of System/370, which required about four times as much on-chip microcode space as the 68000 provides. We projected where HMOS and comparable FET technologies would be over time. The result of the analysis was that we would have to wait at least a few years.

Our group subsequently developed several approaches to partitioning the System/370 architecture for implementation using 68000 microprocessors. We determined that the resultant cost/performance of many of the approaches was of more than academic interest. The following is a description of some of these approaches, selected for discussion either because of their good cost/performance with the System/370 architecture or because of their applicability to other complex architectures.

These descriptions are limited to the instruction processor portion of a computer. Each approach provides a local bus within the processor on which one or more microprocessor chips can communicate with each other and with a local store. Each approach assumes that the local bus can be connected to a global bus to allow the processor to communicate with I/O devices and main memory. At other times, the local bus is disconnected from the global bus so that separate communications can occur over the two buses.

- *Two overlapping subsets*

Our first approach to partitioning a mainframe architecture employs two specially microcoded microprocessors that implement overlapping subsets of the architecture. Each of these microprocessors has on-chip microcode that replaces the microprograms on a normal 68000. This approach is implemented as follows. Partition the mainframe architecture into three sets named P, Q, and R, where most of the high-usage instructions are in set P. Write microcode for P and Q to reside in Processor A and write microcode for P and R to reside in Processor B, as shown in Fig. 1. At any one time, one of the processors is "active" and the other processor is "passive." Only the active processor fetches and executes instructions and controls the bus. There is no contention between the processors.

Assume that the last several instructions have all been either in set P or in set Q. Thus, Processor A is active and Processor B is passive. Note that the internal values of Processor A (I-counter, general registers, condition code, etc.) are up-to-date, and the internal values of Processor B are not. If the next instruction is in set R, Processor A fetches this instruction and performs the following operations.

1. It places into a mailbox in a local store all of its internal values that Processor B might need in order to execute any instructions in sets P or R.

2. Processor A then signals Processor B telling it to become the active processor, that is, to read new internal values from the mailbox and then execute instructions as long as instructions remain in set R or set P.
3. Processor A then becomes the passive processor until, sometime later, it receives a signal from Processor B telling it to read internal values, execute an instruction in set Q, and then continue executing all instructions up to the occurrence of the next instruction in set R.

The sets P, Q, and R are selected on the basis of the following criteria. First, having all of the high-usage instructions in set P, which is common to both processors, greatly reduces the frequency of swapping the active and passive processors. This is desirable because, between swaps, instructions are executed as fast as if they were all implemented in the microcode of a single processor. Second, the frequency of processor swaps is reduced still further if sets Q and R are selected in such a way that instructions in these two sets seldom interleave with each other.

One particularly suitable partition selection for the sets is

- P contains fixed-point, branch, and load/store instructions;
- Q contains floating-point instructions; and
- R contains decimal and privileged instructions.

This selection satisfies both criteria. First, the fixed-point, branch, and load/store instructions represent about 75% of the execution time in a typical instruction mix. Second, although there is frequent interleaving of floating-point, branch, and load/store instructions with either fixed-point instructions or decimal instructions, there is much less frequent interleaving of floating-point instructions with decimal instructions. Therefore, there is relatively little performance lost to swapping active and passive processors if this selection of P, Q, and R is made. In fact, the need for both floating-point and decimal instructions in the same application is sufficiently rare that special-purpose systems containing only one of Processor A or Processor B could be attractive.

If a selection is made in which instructions in sets Q and R frequently interleave but have rather independent internal-value-modification characteristics, then an additional technique could be used to shorten the processor-swap overhead time. This would be to have the passive processor actually executing instructions in set P along with the active processor, listening to the bus, and updating its internal values but not controlling the bus or affecting any external values. Also, the passive processor would decode those instructions not implemented in its own microcode just enough to see whether each such instruction would affect its internal values other than the I-counter and Condition Code (CC). If so, it would set a bit indicating that it must read internal values from the mailbox when it again becomes the active processor. If it becomes the active processor when this bit is still reset, then it reads in only the I-counter and CC values from the mailbox. This often reduces the time required to swap the active and passive processors, although it does not reduce the frequency of swapping.

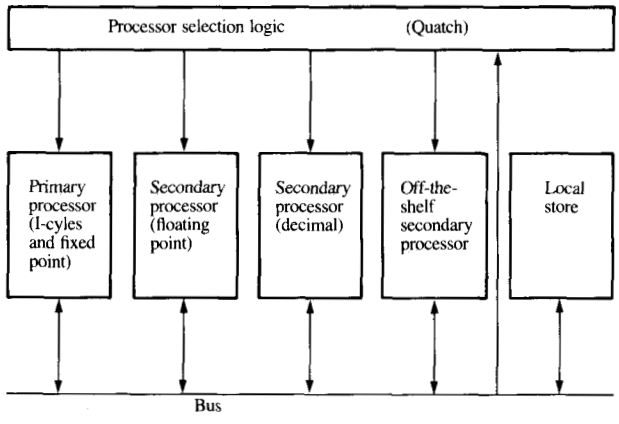

Figure 2 Four subsets, three microcoded.

- *Four subsets, three microcoded*

The second approach to partitioning employs four microprocessors as shown in Fig. 2. Three of these (the Primary processor and the first two Secondary processors) have special on-chip microprograms that replace the microprograms on a normal 68000. These three processors implement, respectively, the following functions:

- I-cycles (instruction fetch and decode and effective-address calculation) for all instructions, and E-cycles (instruction execution) for the fixed-point, load/store, and branch instructions. The register space of this processor is used for the general registers (GRs). Note that its on-chip microcode implements all functions that make heavy use of the GRs, so the critical path is contained within one chip.
- E-cycles for floating-point instructions. Half of the register space in this microprocessor is used for the Floating-Point Registers (FPRs) and the other half is used for work space. Again, the microcode is on the same chip as the registers (and, of course, the data flow) that it controls. An alternative design employs a different microprocessor chip that can execute floating-point instructions faster because its data flow is wide enough to process most common floating-point variables in parallel.
- E-cycles for decimal instructions. All of the register space in this microprocessor is available for work space, since decimal instructions have the storage-to-storage format.

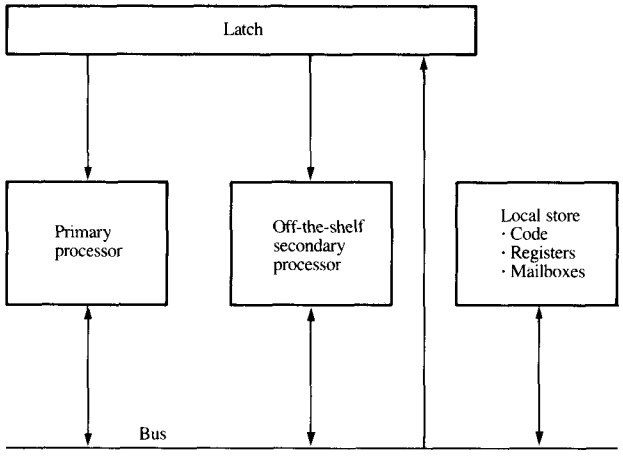

Figure 3 Two subsets, one microcoded.

The fourth microprocessor is "off-the-shelf." That is, it contains the ordinary Motorola microcode that implements the instruction set of the 68000. The part of the System/370 architecture that is not implemented by microcode, namely, the privileged instructions and such functions as interrupt handling, are simulated by sequences of 68000 instructions that are stored in a separate local store rather than on a microprocessor chip. This is appropriate because these instructions and functions are used infrequently (so maximum speed is not required), are error-prone (so early models should have them in easily changed PROMs), and are voluminous (so they can be written more economically in the relatively high-level 68000 machine language rather than in the very low-level 68000 horizontal microcode language).

A system containing these four microprocessors operates as follows. The first ("Primary") microprocessor fetches an instruction. If it can execute the instruction, it does so. If not, the Primary hands off control to one of the other ("Secondary") microprocessors. This involves, first, passing necessary data such as the operation code and effective address and, second, setting a new value into a four-state circuit ("Quatch") whose state determines which microprocessor has control of the local bus that connects all four microprocessors and their local store, in parallel, to the rest of the system. The selected Secondary runs, with full control of the local bus and full access to the main store and I/O system, until it has completed execution of the instruction it was given. Then it sets the original value back into the Quatch, handing control back to the Primary. At this point the Primary fetches the next instruction, and execution proceeds.

Note that this mechanism for passing control allows the Secondary responsible for floating-point instructions to call on the off-the-shelf Secondary to complete an instruction that detected an error. Thus the error-handling function, which is voluminous and not critical to performance, need not occupy valuable control store space on the floating-point Secondary chip.

The desirability of this approach to partitioning of the System/370 architecture can be appreciated by noting that the Primary runs more than 75% of the time when executing typical job mixes and has to hand only one instruction in twenty over to a Secondary.

- *Two subsets, one microcoded*

Our third approach to partitioning is similar to the second, but employs only a single specially microcoded microprocessor and a coded microprocessor. This approach combines the excellent cost/performance of on-chip microcode for the most critical functions with the flexibility, extendibility, and low development cost of off-chip microprocessor code for less critical functions. It uses the structure shown in Fig. 3 and operates as follows. One processor, called the "Primary" processor, contains the general registers (GRs) and contains the microcode for all functions that make heavy use of GRs. It performs I-cycles for all instructions. It also performs E-cycles for the most-used instructions, *i.e.*, for almost all instructions except floating-point, decimal, and privileged instructions. In a typical instruction mix, the instructions that the Primary processor executes constitute about 95% of the instructions by frequency of occurrence and about 50% of the instructions by execution time. Because the Primary processor also performs I-cycles for all instructions, it actually runs more than 50% of the time.

The Primary processor is also responsible for detecting instructions for which it does not contain the execution microcode. It hands over control to the other or "Secondary" processor to complete such instructions. Most of the decimal, floating-point, and privileged instructions do a relatively large amount of data processing or are used very infrequently in typical instruction mixes. Therefore, the time to pass control from the Primary processor to the Secondary processor and back is relatively small. The Secondary processor carries out the necessary processing under control of code contained in the local store. The same local store contains other registers, such as the floating-point registers, and the mailboxes in which the processors leave instruction codes, operand addresses, condition codes, and other necessary data as they pass control back and forth. Control of the two processors is simple because only one of them is ever running at any one time. There is no overlap and no bus contention. Either processor can pass control to the other by inverting the state of the two-state latch that determines which of them is granted use of the bus.

Note that a state-of-the-art microprocessor implements a reasonably high-level machine language. This is the language in which most of the mainframe architecture is coded, when using this approach to partitioning. Development of this code is rapid and inexpensive in comparison to writing in a low-level microcode language. Moreover, the code resides in local store, where it is easy to change in comparison to microcode residing on a microprocessor chip. The corresponding disadvantage is that code implementing instructions tends to run longer than microcode implementing the same instructions. Therefore, there is a performance imbalance between the high-usage instructions, which are implemented in microcode, and the low-usage instructions, which are implemented in code.

- *Subset with emulation*

Our fourth approach relies heavily on software to implement parts of the architecture that cannot be placed on a single microprocessor chip, as illustrated in Fig. 4. In using this approach, the steps are to define a suitable subset of the mainframe architecture, to implement this subset as the "machine" architecture of the microprocessor chip, and to write a first layer of software to raise the level of the subset to the level of full mainframe architecture. The subset must include sufficient instructions and functions to enable the first layer of software to simulate the rest of the mainframe architecture, including preservation of system integrity.

In some applications, no such first software layer is necessary. It might be possible to run some System/360 software (which does not use new functions introduced in System/370) directly on the machine interface of the microprocessor chip. The selected subset might suffice for many OEM-type applications, such as intelligent terminals, intelligent printers, and test-equipment control. Applications in turnkey "applications machines" could be written for the subset with customers never knowing that the subset was there. In other applications, missing instructions can be replaced by subroutine calls at compile time. In the remaining applications, the operating system, viewed as a many-layered entity, as depicted in Fig. 4, can have a first layer that handles "invalid operation" program interruptions by simulating the missing instructions instead of passing these interruptions up to the next-higher layer.

This solution to the problem of insufficient control-store space has the advantages of minimal hardware development cost, risk, and time, as well as excellent product cost/performance for applications that employ only the selected subset. However, it has the disadvantages of a large mix imbalance, in any sort of software simulation of missing instructions, and an increased maximum interrupt latency time.

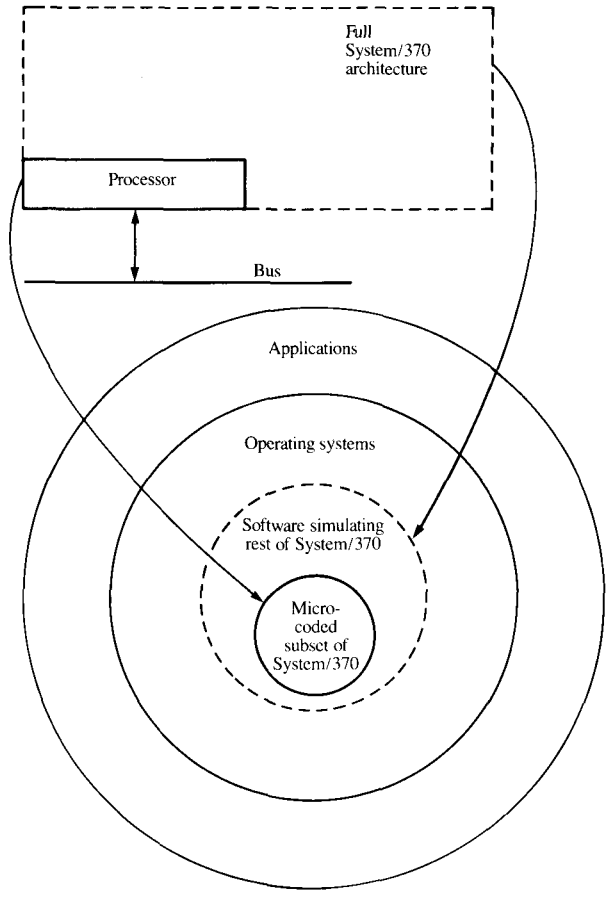

Figure 4 Subset with emulation.

- *Off-chip vertical microcode*

Our three remaining approaches employ two levels of microcode. The fifth approach, shown in Fig. 5, has the advantages of two levels of microcode with different widths.

Current microprocessors achieve excellent cost/performance by allowing a single chip to contain both the control store and the data flow that it controls. Their cost/performance is further improved if the control store is wide, or "horizontal," rather than narrow or "vertical." A wide control store eliminates most decoding, so it reduces both complexity and propagation delay. A wide control store can control several simultaneous operations, so it improves performance. However, a wide control store usually needs to contain more bits than a narrow one in order to implement a given function.

One common solution to the problem of a large, wide control store has been described [27] with reference to the

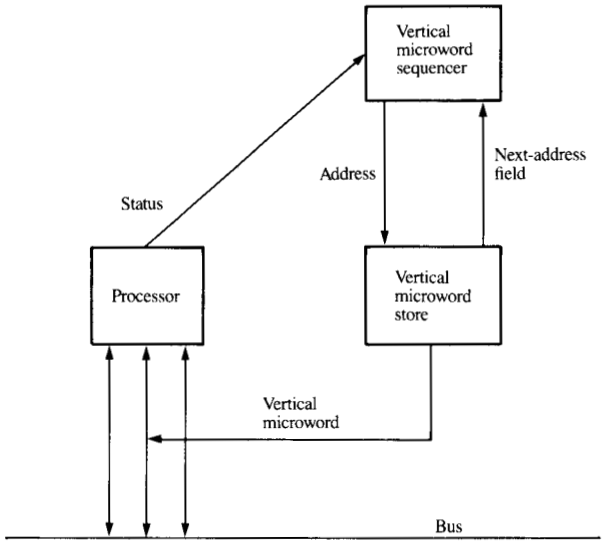

Figure 5 Off-chip vertical microcode.

Motorola 68000 microprocessor. The solution is based on noting that the information in a wide control store is highly redundant; many control words have bits that are identical. The solution is to have both a wide horizontal store and a narrow vertical store. The horizontal store contains the few (nonredundant) control bit patterns required by the data flow. The vertical store contains the many bit patterns that are necessary for sequencing through many machine instructions. Such an approach is said to reduce the total control store size by about a factor of two in the Motorola 68000 microprocessor.

Even with this approach, current microprocessors have insufficient on-chip control store to implement all of the microcode that is necessary to implement System/370 architecture. Yet there is a major cost/performance advantage in having all of the horizontal microcode on the same chip as the data flow (to avoid the many pins or bus cycles required to bring a wide control word onto the chip), and there is a cost/performance advantage in having the most-frequently-used vertical microwords on the same chip as the data flow (to avoid any accesses to the off-chip bus in most microcycles). This leaves only the infrequently used vertical microwords to be stored off the microprocessor chip in a microprocessor-based implementation of a large system or mainframe architecture.

Implementing this fifth approach requires solutions to two major problems. These problems and their solutions are as follows. First, branch from on-chip to off-chip vertical microcode by

- setting a latch attached to a microprocessor output pin, or
- restricting on-chip vertical micro read-only memory (ROM), for example to 512 words, and branching to a word whose address exceeds 511, or
- branching to the highest valid on-chip vertical microword address after setting the off-chip vertical microword branch address onto the data bus.

Second, allow conditional branches to depend on status bits by

- bringing up to 16 raw status bits off the chip, by way of the data bus or dedicated pins, just before the data bus or other dedicated pins are used to bring the next vertical microword on chip, or
- using the branch control fields of the horizontal microwords to select just the desired status information and bring off chip just the low two bits of the address of the next off-chip microword.

Note that most horizontal microwords will probably be used by both on-chip and off-chip vertical microwords. However, some specially written horizontal microwords will have to be put onto the chip just for the use of the off-chip vertical microcode. That is, the microprocessor, as seen by the off-chip vertical control store, should interpret a thoroughly general and flexible vertical microcode language. This provides the ability to implement a complex mainframe architecture. The on-chip vertical microcode provides very high performance for the most-frequently-used portions of that architecture.

Other advantages of this method of partitioning microcode are that

- it allows microcoding for high speed, since coding for smallest size is not necessary;
- it allows off-chip vertical microcode, written for a first product, to be put in the on-chip vertical microstore in subsequent products whose microprocessors have larger ROM; and
- it encourages a microprogramming methodology of first selecting a set of useful horizontal microwords and then stringing them together with vertical microwords, which increases microprogrammer productivity.

- *Off-chip horizontal microcode*

Our sixth approach, shown in Fig. 6, employs two sets of microwords that have the same width. One set is on the microprocessor chip and executes very rapidly. The other set is in an external store and can be very large. In a typical instruction mix, fixed-point, branch, and load/store instructions account for 95% of the instructions by frequency of occurrence, and for 60% to 75% of the instructions by execution time. Thus, these instructions are suitable candi-

dates for this partitioning scheme to have on-chip. The remaining microwords, kept in an off-chip control store, are brought onto the chip one by one for execution. This could be done in several cycles using existing address and/or data pins for microword bits, or it could be done using dedicated pins. The off-chip control store must be wide enough for both the microword bits required by the data flow and the microword-selection bits required by the sequencer. The off-chip microword sequencer must have access to on-chip status information in order to perform conditional microprogram branches and in order to pass control back and forth between on-chip and off-chip functions and instructions.

This method of partitioning the microcode has the following advantages:

- an architecture of unlimited complexity can be implemented by a sufficiently large off-chip control store;
- difficult parts of the architecture can be placed off chip, where they can be corrected without altering the microprocessor chip itself;
- off-chip microcode may be placed on chip, with minimal modifications, if a subsequent product uses a microprocessor chip with larger on-chip control store;
- with care, patches to the on-chip microcode can be implemented in the off-chip microcode if errors are found;
- since off-chip instructions are executed in the same engine as on-chip instructions, they have full access to registers, condition code, and other facilities of the machine; and
- all accesses to main storage and channels are made by the same microprocessor.

The arrangement for partitioning microcode between on-chip and off-chip control stores allows the most-frequently-used instructions to run with the cost/performance of microprocessors (due to the short critical path produced by on-chip microcode). Unfortunately, this arrangement runs the rest of the instructions and functions with the cost/performance characteristic of bit slices (with the longer critical path produced by off-chip microcode).

- *Subset with primitives*

Our last approach, shown in Fig. 7, could produce a very economical processor at the expense of a difficult and prolonged development process. The difficulty is defining suitable "primitive" operations.

In principle, a microprocessor that contains on-chip microcode for a mainframe system's fixed-point, branch, and load/store instructions can be programmed to emulate the remainder of that system's architecture, as described under "Subset with Emulation." In practice, that design produces relatively poor performance for the instructions and functions that are emulated by off-chip code rather than microcoded on the microprocessor chip.

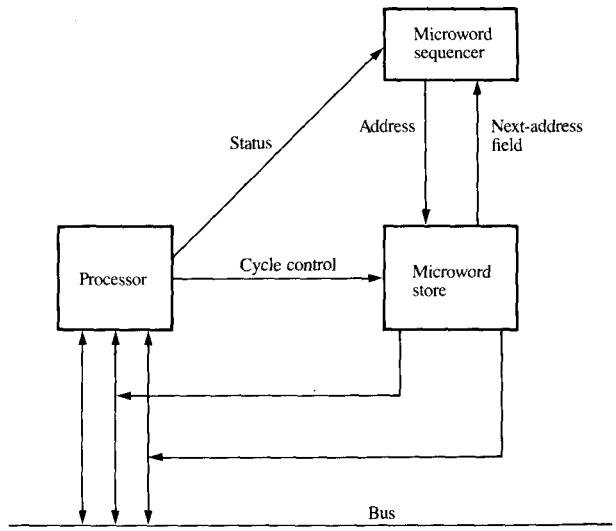

Figure 6 Off-chip horizontal microcode.

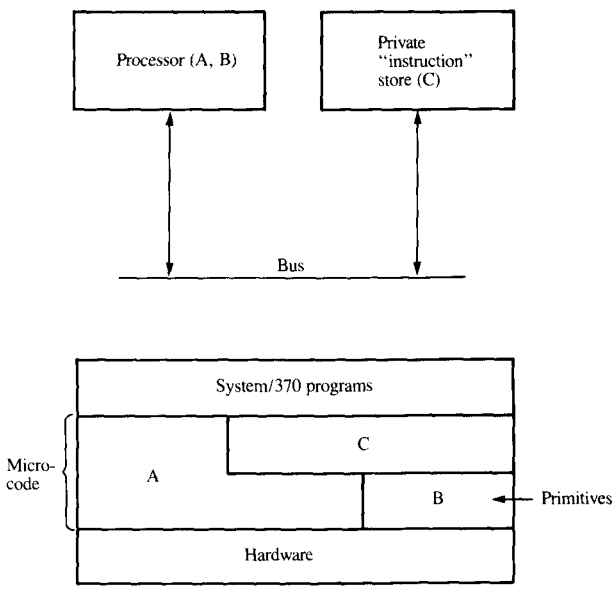

Figure 7 Subset with primitives.

Microcoding some "primitives" (instead of some instructions that could occupy the same on-chip control store space) can produce significantly higher performance on a complete instruction mix. A primitive is not itself a system instruction, but rather it executes a simple function that is useful in the emulation of more complicated instructions or functions. An emulation program can achieve higher performance if it has primitives available as well as the basic instructions. Exam-

ples of primitives are "load registers with contents of instruction fields," "set condition code according to arithmetic result," and "compute effective address."

This method of implementing a large system architecture on a microprocessor is implemented by subdividing the microprocessor's operation code space into the following three sets:

A. codes of high-usage instructions, each of which is implemented by a sequence of on-chip microcode;
B. codes assigned to primitives which are useful for emulating instructions, each of which is implemented by a sequence of on-chip microcode; and
C. codes of the remaining low-usage instructions, each of which is implemented by a sequence of high-usage instructions (A) and primitives (B).

In operation, an instruction stream is being fetched from store. As long as instruction codes are found to be in set A, execution is controlled by on-chip microcode. Any codes in set B are illegal in this mode. When an instruction code is found to be in set C, direct execution of on-chip microcode is terminated after completion of that instruction's I-cycles (which can include effective address generation). The instruction code selects a starting address in a private program store, and the microprocessor fetches its next "instruction" from this address. That "instruction" code will be in set A or B, so it initiates a sequence of on-chip microcode. This sequence ends by fetching another "instruction" which initiates another sequence of on-chip microcode, and so on, until the instruction whose code was in set C has been completely emulated. Then the next instruction is fetched from store, not from the private program store. That instruction, too, is either executed directly by a sequence of on-chip microcode, or simulated by "instructions" in the private program store, which are in turn executed by sequences of on-chip microcode.

Note that the emulation mode used to program a low-usage instruction, whose code is in set C, has the following special characteristics:

- "Instructions" are fetched from the private program store, not from main store.
- The instruction counter is not incremented.
- Codes in both sets A and B are legal while emulating an instruction in set C.
- Interrupts must be held pending until all of the "instructions" that emulated one instruction in set C are completed.
- Any instructions in set A that are used along with primitives in set B to simulate an instruction in set C must be prevented from changing the condition code or taking their ordinary exceptions.

Some advantages of this method of partitioning the architecture between on-chip microcode and off-chip emulation code are as follows.

- An instruction in set C can be simulated with relatively few bus cycles. An "instruction" brought in from the private instruction store (by one or two bus cycles) initiates a sequence of many microwords which do not require bus cycles.
- Constant data needed by difficult instructions or by interrupts (such as the implied register of the Translate and Test instruction or the many implied storage addresses for interrupts) can be brought in easily as immediate fields of "instructions" fetched from the private program store. Such constants may be difficult to introduce by way of on-chip microcode.
- An architecture of unlimited complexity can be emulated by a sufficiently large private program store if the codes in sets A and B supply functions of sufficient generality.
- The private program store can be relatively small, because it stores relatively powerful "instructions," each of which is interpreted by many microwords. This is especially true if powerful branch and subroutine call "instructions" are used to save space.

Note that the transfer of control from on-chip microcode to an off-chip emulation program need not be limited to the time when an I-cycle completes. On-chip microcode should be allowed to call for simulation of the rest of an instruction whenever it detects an unusual condition (so it does not require high performance) that is difficult to handle (so it would otherwise consume many valuable on-chip microwords). For example, the on-chip microcode for the Move Characters instruction should be able to call an off-chip program if it detects operand overlap.

Conclusion

Each successive generation of computers has achieved improved cost/performance by using denser technologies and machine organizations that are appropriate to those technologies. The introduction of LSI allowed immediate spectacular improvements in the cost/performance of computers whose architectures can be implemented entirely within a single LSI chip. The application of LSI to high-performance processors with complex uniprocessor architectures has proceeded more slowly and with less spectacular results. This is because application of LSI to such processors requires use of new machine organizations for which the existing architectures were not originally intended. We have described several approaches to using LSI to implement processors with complex architectures at the low end of the traditional mainframe processor spectrum. Appropriate approaches can be selected to achieve particular performance goals, as listed in Table 1.

Table 1 Comparison of approaches.

Number	*Approach name*	*Rank**				*Main advantage*	*Main disadvantage*
		P	*B*	*D*	*R*		
1	Two overlapping subsets	7	7	2	1	Low build cost, good balance	Cannot implement rich architecture
2	Four subsets, three microcoded	6	2	4	7	High performance	High build cost
3	Two subsets, one microcoded	4	6	6	7	Good cost performance	Unbalanced performance
4	Subset with emulation	1	6	7	6	Low cost	Low and unbalanced performance
5	Off-chip vertical microcode	3	5	5	5	—	Need complete set of horizontal microwords
6	Off-chip horizontal microcode	2	1	3	7	Can implement rich architecture	Low performance
7	Subset with primitives	5	6	1	5	Good cost/performance System/370	Need complete set of primitives

*Key (7 is best):
P = Performance
B = Build cost
D = Development cost
R = Richness of implementable architecture

There are two aspects of these partitioning approaches that should be especially noted. First, they do not apply to the high end of the mainframe processor spectrum. It still remains true that higher-performance processors must use lower levels of integration. Second, development of increasingly sophisticated ways to apply LSI technology to implementation of existing uniprocessor architectures should not dampen enthusiasm for developing new architectures that fit LSI more naturally. All multiple-instruction multiple-data architectures fit LSI better than uniprocessors do; and the ten-year trend of implementing successively higher-level architectures within a single microprocessor chip should be continued to and then beyond the architecture levels implemented by today's mainframes.

Acknowledgments

A significant fraction of both the ideas and analyses reported in this paper are the work of Joseph P. Buonomo, Steven R. Houghtalen, Raymond E. Losinger, James W. Valashinas, A. James Albert, and R. C. Huang.

References and notes

1. "Color Graphics Computer Uses 68000 Central Processor," *Electronics* **53,** 203 (New Products Section) (November 6, 1980).
2. A. Durniak, "VLSI Shakes the Foundations of Computer Architecture," *Electronics* **52,** 111 (May 24, 1979).
3. Rita Shoor, "CDC 205 Runs 800 Million Operations/Sec," *Computer World,* pp. 1–2 (June 9, 1980).
4. C. Gordon Bell and Alan Newell, "Grosch's Law," *Computer Structures—Readings and Examples,* McGraw-Hill Book Co., Inc., New York, 1971, p. 561.
5. "More Tumult for the Computer Industry," *Business Week,* pp. 58–68 (May 30, 1977).
6. Len Yencharis, "Micro/Mini Storage Peripherals Driven by Disk, Tape Advances," *Electron. Design* **27,** 42–64 (October 25, 1979).
7. David J. Kuck, "A Survey of Parallel Machine Organization and Programming," *Computing Surv.* **9,** 29–57 (March 1977).
8. Tsuneo Kinoshita, Tai Sato, Hiroyuki Tango, and Jun Iwamura, "Sapphire Substrate Boosts Microprocessor Density," *Electronics* **54,** 112–115 (October 6, 1981).
9. "Special Report on Microprocessors," *Electron. Engineering Times,* pp. 64–70 (May 12, 1980).
10. Carol Ogdin, "Sixteen-bit Micros," *Mini-Micro Syst.* **12,** 64–72 (January 1979).
11. Robert Sugarman, "Computers: Our 'Microuniverse' Expands," *IEEE Spectrum* **16,** 32–37 (January 1979).
12. ®PDP-11, ®LSI 11, and ®DEC are registered trademarks of Digital Equipment Corporation, Maynard, MA.
13. D. Dickhut, B. Hashizume, and W. Johnson, "LSI Trio Calls the Tunes in Microcomputer's CPU," *Electronics* **53,** 130–135 (July 17, 1980).
14. ®Intel is a registered trademark of Intel Corporation, Santa Clara, CA.
15. "Co-processor Cooperates with 8 or 16-bit Microprocessors," *Electron. Design* **28,** 19 (News & Technology) (March 1, 1980).

16. ®Zilog is a registered trademark of Zilog Inc., Cupertino, CA.
17. F. Faggin, "How VLSI Impacts Computer Architecture," *IEEE Spectrum* **15,** 28–31 (May 1978).
18. S. Bal, G. Chao, and Z. Soha, "Bilingual, 16-bit, Microprocessor Summons Large-Scale Computer Power," *Electron. Design* **28,** 66–70 (January 18, 1980).
19. S. Bal, E. Burdick, R. Barth, and D. Bodine, "System Capabilities Get a Boost From a High-Powered Dedicated Slave," *Electron. Design* **28,** 77–82 (March 1, 1980).
20. ®Eclipse and ®MicroEclipse are registered trademarks of Data General Corporation, Southborough, MA.
21. M. Druke, D. Carberry, R. Gusowski, and E. Buckley, "LSI Processor Mirrors High-Performance Minicomputer," *Electronics* **53,** 119 (February 14, 1980).
22. *The IBM System/360 Model 91,* special issue, *IBM J. Res. Develop.* **11,** No. 1 (January 1967).
23. R. P. Case and A. Padegs, "Architecture of the IBM System/370," *Commun. ACM* **21,** 73–96 (January 1978).
24. ®Motorola is a registered trademark of Motorola, Inc., Chicago, IL.
25. I. LeMair, "Complex Systems Are Simple to Design with the 68000," *Electron. Design* **26,** 100–107 (September 1, 1978).
26. Edward Stritter and Tom Gunter, "A Microprocessor Architecture for a Changing World: the Motorola 68000," *Computer* **12,** 43–51 (February 1979).
27. E. Stritter and N. Tredennick, "Microprogrammed Implementation of a Single Chip Microprocessor," *ACM SIGMICRO Newsletter* (SIGMICRO is ACM's Special Interest Group on Microcode) **9,** 43–51 (December 1978).

Received September 8, 1981; revised February 3, 1982

The authors are with the IBM System Products Division's Advanced Systems Department, located at the Endicott laboratory, P.O. Box 6, Endicott, New York 13760.