J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

COMPUTER ARCHITECTURE

The term *computer architecture* was coined in the 1960s by designers at IBM to mean the structure of a computer that a programmer must understand to write a program (1). It represents the programming model of the computer, including the instruction set and the definition of register file, memory, etc. Over time, the concept of computer architecture has evolved to include both the functional specification and the hardware implementation. At the system level, it defines the processor-level building blocks, such as processors and memories, and the interconnection among the building blocks. At the microprocessor level, computer architecture determines the processor's programming model and its detailed implementation. The implementation of a microprocessor is also known as *microarchitecture*.

The task of a computer architect is to understand the state-of-the-art technologies at each design level and the changing design tradeoffs for their specific applications. The ultimate goal is to design the best computer system within the required cost and power budgets. The tradeoff of cost, performance, and power consumption is fundamental to a computer system design. Each application will require a different optimum design point. For high-performance server applications, chip and system costs are less important than performance. Computer speedup can be accomplished by constructing more capable processor units or by integrating many processors units on a die. For cost-sensitive embedded applications, the goal is to minimize processor die size and system power consumption.

Technology Considerations. There are numerous technical considerations in computer architecture. In general, designers have to provide whole system solutions rather than treating logic design, circuit design, and packaging as independent phases of the design process. Modern computer implementations are based on silicon technology. The two driving parameters of this technology are die size and feature size. Die size largely determines cost. Feature size determines circuit density and circuit delay. Current feature sizes range from 0.13 μm to 0.25 μm. With the continuing improvements in lithography, feature sizes below 0.1 μm (100 nm) can soon be realized. Feature sizes below 0.1 μm are also known as *deep submicron*.

Deep submicron technology allows microprocessors to be increasingly more complicated. According to Semiconductor Industry Association (*SIA*) (2), the number of transistors (Fig. 1) and the on-chip clock frequencies (Fig. 2) for high-performance microprocessors will continue to grow exponentially in the next ten years. However, there are physical and program behavioral constraints that limit the usefulness of this complexity. Physical constraints include interconnect and device limits as well as practical limits on power and cost. Program behavior constraints result from program control and data dependences and unpredictable events during execution (3).

Much of the improvement in computer performance has been due to technology scaling that allows increased circuit densities at higher clock frequencies. As feature sizes shrink, device area shrinks roughly as the square of the scaling factor, and device delay improves approximately linearly with feature size. On the other hand, there are a number of major technical challenges in the deep submicron era. The most important of these is that interconnect delay does not scale with the feature size. When all three dimensions of an interconnect wire are scaled down by the same scaling factor, the interconnect delay remains roughly unchanged. Consequently, interconnect delay becomes a limiting factor in the deep-submicron era.

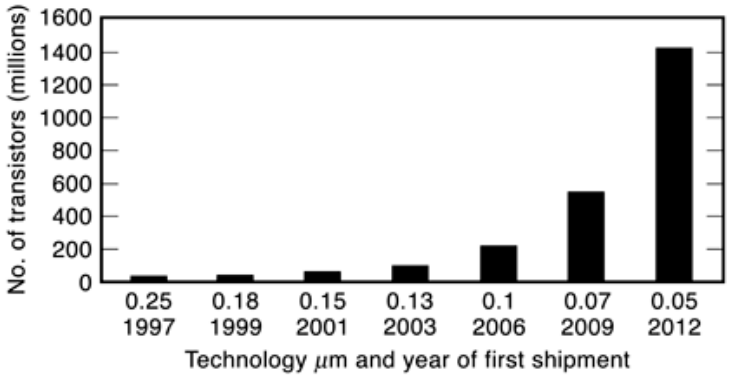

Fig. 1. Number of transistors per chip. (Source: National Technology Roadmap for Semiconductors.)

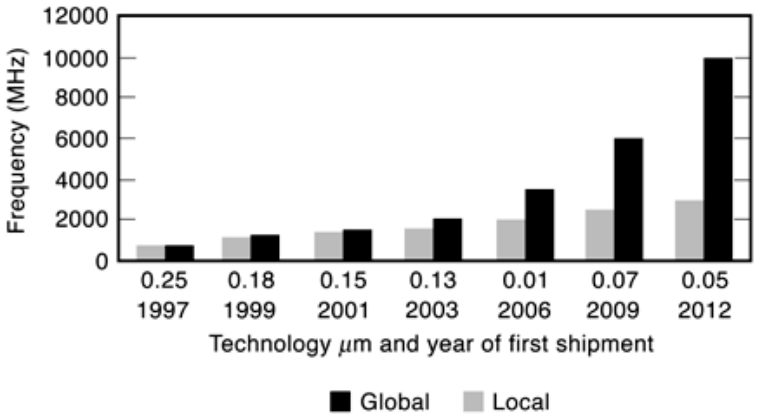

Fig. 2. On-chip local and global clock speeds. (Source: National Technology Roadmap for Semiconductors).

There are other technical challenges for high-performance microprocessors. For example, custom circuits are necessary to enable gigahertz signals to travel in and out of chips. Special cooling techniques are needed for processors that consume more than 100 W of power. All these challenges require designers to consider all aspects in the computer system and provide an optimal solution based on the design objectives.

Performance Considerations. Microprocessor performance has improved by approximately 50% per year in the last twenty years. This can be attributed to higher clock frequencies, deeper pipelines, and improved exploitation of *instruction-level parallelism*. However, the cycle time at a given technology must not be too small, or we will sacrifice overall performance by incurring too much clock overhead and suffering long pipeline breaks. Similarly, the instruction-level parallelism is usually limited by the application, which is further diminished by code-generation inefficiencies, processor resource limitations, and execution disturbances. The overall system performance may deteriorate if the hardware to exploit the parallelism becomes too complicated.

Another important consideration is that there is a growing disparity between microprocessor speed and memory speed. Current main memory systems, using synchronous DRAM or RAMBUS, have much longer read and write access times than microprocessor clock speeds. Consequently, the overall performance is often limited by the so-called "memory wall." Memory hierarchy, a crucial part of a computer system, is described in more detail in the subsection "Memory" below.

High-performance server applications, where chip and system costs are less important than performance considerations, encompass a wide range of requirements, from computation-intensive to memory-intensive. In manufacturing these high-performance servers, the need to customize implementation to specific applications often require expensive small-volume microproduction instead of mass production.

Power Considerations. Power consumption has received increasing attention because of growing demands for wireless and portable electronic applications. At the silicon chip level, the total power dissipation has three major components:

(1) Switching loss
(2) Leakage current loss
(3) Short-circuit current loss

Switching loss is by far the dominant factor among these components. It is proportional to the operating frequency and also proportional to the square of the supply voltage. In general, the operating frequency is also roughly proportional to the supply voltage. Thus, lowering the supply voltage can effectively reduce the switching loss. If the supply voltage is reduced by 50%, the operating frequency is also reduced by 50%, and the total power consumption becomes one-eighth of the original power. For many embedded applications, acceptable performance can be achieved at a low operating frequency by exploiting the available program parallelism using suitable parallel forms of processor configurations.

Improving the battery technology can also allow processors to run for an extended time. Currently, conventional nickel–cadmium battery technology has already been replaced by high-energy-density technology such as the *NiMH* (nickel metal hydride) battery. However, the energy density is unlikely to improve indefinitely, mainly for safety reasons. When the energy density becomes too high, a battery becomes virtually as dangerous as an explosive.

Die Size. Another important design tradeoff is to optimize the die size. In the high-performance server market, the silicon chip cost may be small compared with the overall system cost. Increasing the chip cost by ten times may not significantly affect the overall system cost. On the other hand, many system-on-chip implementations are cost-sensitive. For these applications, the optimum die size is extremely important.

The die area available to a designer depends largely on the manufacturing technology. This includes the overall control of the diffusion and process technology, the purity of the silicon crystals, and so on. Improving manufacturing technology allows larger dies with higher yields, and thus lowering manufacturing costs. At a given technology, die size can affect manufacturing costs in two ways. First, as the die size increases, fewer dies can be realized from a wafer. Second, as die size increases, production yield decreases, following generally a Poisson distribution of defects. In some cases, doubling the die size can increase the die cost by more than ten times.

Other Considerations. As VLSI technology continues to improve, there are new design considerations for computer architects. The simple traditional measures of processor performance—cycle time and cache size—are becoming less relevant in evaluating application performance. Some of the new considerations include:

(1) Creating high-performance processors with enabling compiler technology
(2) Designing power-sensitive system-on-chip processors in very short turnaround time
(3) Improving features that ensure the integrity and reliability of the computer
(4) Increasing the adaptability of processor structures, such as cache and signal processors

Performance–Cost–Power Tradeoffs. In the era of deep-submicron technology, two classes of microprocessors are evolving: (1) high-performance server processors and (2) embedded client processors. The majority of implementations are commodity system-on-chip processors devoted to end-user applications. These

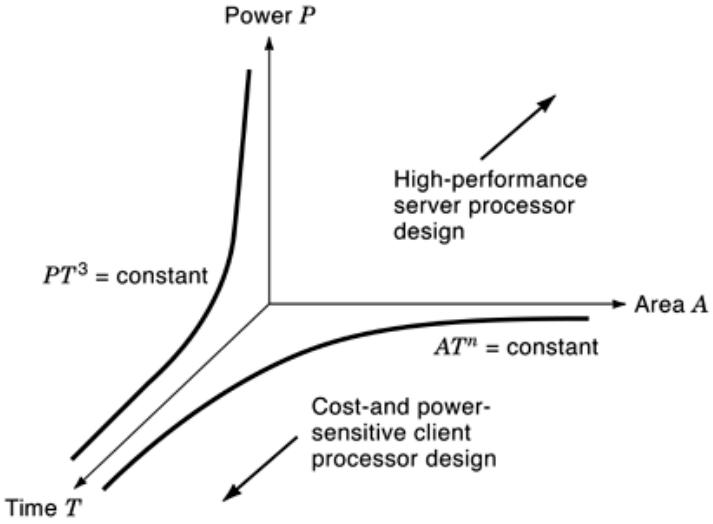

Fig. 3. Design tradeoffs for high-end and low-end processors.

highly cost-sensitive client processors are used extensively in consumer electronics. Individual application may have specific requirements. For example, portable and wireless applications require very low power consumption, and the high-end server processors are usually performance-driven.

At a fixed feature size, area can be traded off for performance (expressed in terms of the execution time T). VLSI complexity theorists have shown that there is an AT^n bound for microprocessor designs (4), where n usually falls between 1 and 2. By varying the supply voltage, it is also possible to trade off time T for power P with a PT^3 bound. Figure 3 shows the possible tradeoffs involving area, time, and power in a processor design (3). Embedded and high-end processors operate in different design regions of this three-dimensional space. The power and area axes are typically optimized for embedded processors, whereas the time axis is typically for high-end processors.

Alternatives in Computer Architecture. In computer architecture, a designer must understand the technology and the user requirements as well as the available alternatives in configuring a processor. The designer must apply what is known of user program behavior and other requirements to the task of realizing an area–time–power optimized processor. User programs offer differing types and forms of parallelism that can be matched by one or more processor configurations. A primary design goal is to identify the most suitable processor configuration that matches cost constraints.

This article describes different types of computer architecture and focuses on the design tradeoffs among them. The following section describes the principal functional elements of a processor. The section after presents the various types of parallel and concurrent processor configuration. The final section compares a few recent architectures and presents some concluding remarks.

Processor Architecture

The processor architecture consists of the *instruction set*, the *memory* that it operates on, and the *execution units* that implement and interpret the instructions. While the instruction set implies many implementation details, the resulting implementation is a great deal more than the instruction set. It is the synthesis of the physical device limitations with area–time–power tradeoffs to optimize cost performance for specified user

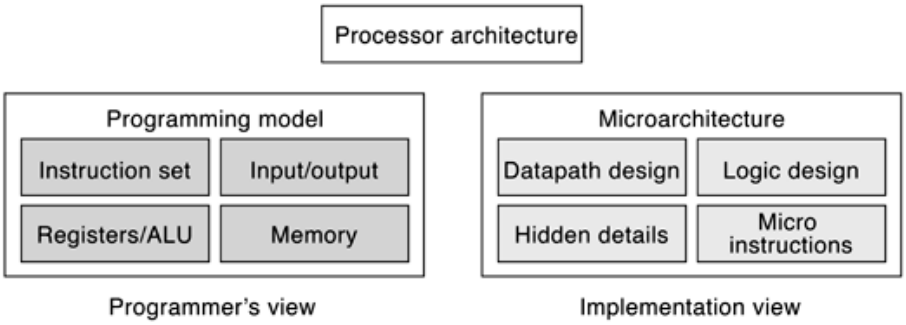

Fig. 4. Processor architecture: block diagram.

requirements. As shown in Fig. 4, the processor architecture may be divided into the high-level programming model and the low-level microarchitecture. In general, the microarchitecture model allows engineers to build or evaluate the hardware, whereas the high-level programming model allows programmers to develop programs for this particular architecture.

Instruction Set. Computers deal with many different kinds of data and data representations. The operations available to perform the requisite data manipulations are determined by the data types and the uses of such data. Processor design issues are closely bound to the instruction set. Instruction-set behavior data affects many of these design issues.

The *instruction set* for most modern machines is based upon a register-set to hold operands and addresses. The register-set size varies from 8 to 64 words, each word consisting of 32 to 64 bits. An additional set of floating-point registers is usually available to most modern computers. A typical instruction set specifies a program status word, which consists of various types of control status information, including condition codes set by the instruction. Common instruction sets can be classified by format differences into three types:

(1) L/S, or *load–store*, architecture
(2) R/M, or *register–memory*, architecture
(3) $R+M$, or *register-plus-memory*, architecture

The L/S, or *load/store*, instruction set characterizes many of the *RISC* (reduced instruction set computer) microprocessors (5). All values must be loaded into registers before an execution can take place. An ALU ADD instruction, for example, must have both operands and result specified as registers. The purpose of the RISC architecture is to establish regularity of execution and ease of decoding in an effort to improve overall performance. RISC architects have tried to reduce the amount of complexity in the instruction set and regularize the instruction format so as to simplify decoding of the instruction. A simpler instruction set with straightforward timing can more readily be implemented. For these reasons, it is assumed that implementations based on the L/S instruction set will result in higher clock rates than other classes, other parameters being generally the same.

The R/M, or *register/memory*, architectures include instructions that can operate both on registers and with one of the operands residing in memory. Thus, for the R/M architecture, an ADD instruction might be defined as the sum of a register value and a value contained in memory, with the result going to a register. The R/M instruction sets generally trace their evolution to the IBM System 360. Many mainframe computers (e.g. IBM, Hitachi), as well as the popular Intel x86 microprocessors, follow the R/M architecture.

The R+M, or *register-plus-memory*, architectures allow formats to include operands that are either in memory or in registers. For example, an ADD may have all of its operands in registers or all of its operands in

memory. The R+M architecture generalizes the formats of R/M. An example of the R+M architecture is Digital Equipment's VAX series of machines. The use of an extended set of formats and register modes allows a powerful and varied specification of operands and operation type within a single instruction. Unfortunately, format and mode variability complicates the decoding process, so that the process of interpretation of instructions can be slow. On the other hand, R+M architectures make excellent use of memory and bus bandwidth.

From the architect's point of view, the tradeoff in instruction sets is an area–time compromise. Both R/M and R+M architectures offer a more concise program representation, using fewer instructions of variable size, than the L/S architecture. Programs occupy less space in memory, and smaller instruction caches can be used effectively. (Variable instruction size makes decoding more difficult: the decoding of multiple instructions requires predicting the starting point of each instruction.) The register-memory processors require more circuitry and area to be devoted to instruction fetch and decode. Nonetheless, the success of Intel-type x86 implementations in achieving high clock rates and performance has shown that the limitations of register-memory instruction set can be overcome.

Memory. The *memory system* comprises the physical storage elements in the memory hierarchy. These elements include those specified by the instruction set (registers, main memory) as well as those elements that are largely transparent to the user's program (cache and virtual memory).

The register file, the fastest memory available to a processor, is also the most often referenced type of memory in program execution. Usually, the processor cycle time is determined by the register access time. Although the size of the register file is very small compared with the main memory, the main memory is much slower than the register file, typically 20 to 30 times slower. The main memory is almost always based on *DRAM* (dynamic static random access memory) technology, although *SRAM* (static random access memory) and flash technologies can also be used. Currently, most computer systems consist of between 64 Mbyte and 1 Gbyte of main memory. The capacity of a hard disk is many times larger than the main memory. The hard disk contains all the programs and data available to the processor. Its addressable unit (sector) is accessible in 1 ms to 10 ms, with typical single-unit disk capacity of 5 to 30 Gbyte. Large server systems may have hundreds or more of such disk units. As different memory elements have very different access times, additional levels of storage (buffer and cache) are added to hide the access-time differences, and this leads to a *memory hierarchy*.

Memory Hierarchy. There are basically three parameters that define a memory system: latency, bandwidth, and capacity. Latency is the time for a particular access request to be completed. Bandwidth is the number of requests supplied per unit time. Capacity is the total amount of information that can be stored in the memory element. In order to provide large memory spaces with desirable access-time latency and bandwidths, modern memory systems use a multiple-level memory hierarchy. Smaller, faster levels are more expensive than larger, slower levels. The multiple levels in the storage hierarchy can be ordered by their size and access time from the smallest, fastest level to the largest, slowest level. The goal of a good memory-system design is to provide the processor with an effective memory capacity of the largest level and with an access time close to the fastest.

Suppose we have a memory-system hierarchy consisting of a cache, a main memory, and a disk. The disk contains the contents of the entire virtual memory space. Typical size and access time ratios are:

Size:	memory/cache	$\approx$	1000
Access time:	memory/cache	$\approx$	30
Size:	disk/memory	$\approx$	100 to 1000 or more
Access time:	disk/memory	$\approx$	100000

Caching and *paging* are mechanisms to manage memory space efficiently. *Caches* are memory buffers located between processor and main memory. The purpose of caches is to hide the latency of the main memory.

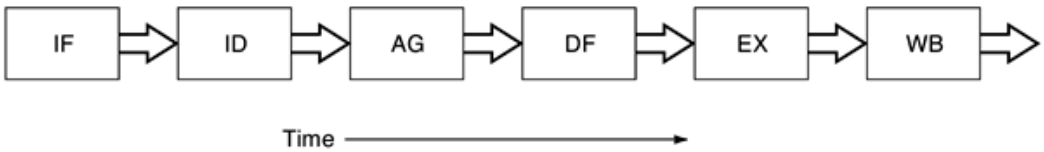

Fig. 5. Instruction execution sequence.

Instruction caches are used for buffering instruction code, and data caches are used for buffering dynamic program data. The purpose of *paging* is to hide the latency of the hard disk. Paging is a mechanism by which the operating system brings pages (or fixed-size blocks) of data on demand from the hard disk into the main memory. To allow efficient memory usage, the pages can be loaded into different segments in the main memory, and an address translation table is used to translate from the program logical address to the main-memory physical address.

The address translation table usually resides in the main memory, and the access time of the table can be quite slow. Special hardware, known as the translation lookaside buffer (*TLB*), can be used to speed up this translation. The TLB is a fast register system, typically consisting of between 64 and 256 entries, which save recent address translations for reuse.

Program Execution Unit. Each instruction code determines a sequence of actions in order to produce the specified results (Fig. 5). These actions can be overlapped (as discussed in the sub-subsection "Instruction-Level Parallelism" below), but the results always appear in a specified sequential order. These actions include:

(1) Fetching the instruction into the instruction register (IF)
(2) Decoding the instruction code (ID)
(3) Generating the address for any memory reference (AG)
(4) Fetching data operands into executable registers (DF)
(5) Executing the specified operation (EX)
(6) Returning the result to the specified register (WB)

To perform these actions efficiently, the hardware of a CPU consists of two distinct types of components: control logic and data paths. The control logic basically represents the state machine of the processor, and the data paths are functional units used for various sorts of calculations. First, the program instruction code is fetched from the instruction cache or main memory to the instruction decoder. Based on the instruction code, the control logic generates a sequence of actions. The control logic produces the necessary control signals directing the functional units to execute the instruction. Every cycle the control logic produces a new set of control values connecting various registers and functional units.

In some implementations, the control logic is based on the Boolean equations that directly implement the specified actions. When these equations are implemented with logic gates, the resultant decoder is called a *hardwired decoder*. In other implementations, a different style of design may be used that can support a more complicated control sequence and instruction set. The control sequence is determined by the *microcode* located inside the control logic. The microcode contains the control information in every cycle. The microcode implementation is more expensive but more flexible than the hardwired implementation.

The data paths of the processor include all the functional units needed to implement the vocabulary of the instruction set. Typical functional units are the arithmetic logic unit (*ALU*) and the floating-point unit (*FPU*). They also contain the buses and other structured interconnections between the registers and the functional units that complete the data paths.

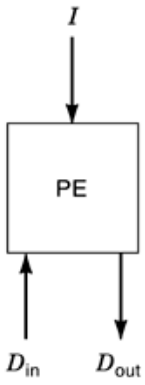

Fig. 6. SISD—single instruction, single data stream.

Program Parallelism and Parallel Architecture

Exploiting program parallelism is one of the most important elements in computer architecture design. In general, programs can encompass the following four levels of parallelism:

(1) Parallelism at the instruction level (fine-grained)
(2) Parallelism at the loop level (middle-grained)
(3) Parallelism at the procedure level (middle-grained)
(4) Parallelism at the program level (coarse-grained)

Instruction-level parallelism (*ILP*) means that multiple operations can be executed in parallel within a program. ILP may be achieved with hardware, compiler, or operating-system techniques. At the instruction level, multiple instructions can be executed in parallel provided that their results do not affect one another. At the loop level, consecutive loop iterations are ideal candidates for parallel execution. At the procedure level, the availability of parallel procedures depends largely on the algorithms used in the program. At the program level, different programs can execute in parallel provided that there is no resource conflict.

Different computer architectures have been built to exploit these inherent parallelisms. These architectures can be conveniently described using the *stream concept*. An instruction stream represents a sequence of processor instructions, and a data stream represents a sequence of data manipulated by the processor. There are four simple combinations that describe the most common architectures (6):

(1) *SISD*—Single Instruction, Single Data Stream This is the traditional uniprocessor (Fig. 6).
(2) *SIMD*—Single Instruction, Multiple Data Stream This includes array processors and vector processors (Fig. 7).
(3) *MISD*—Multiple Instruction, Single Data Stream These are typically systolic arrays (Fig. 8).
(4) *MIMD*—Multiple Instruction, Multiple Data Stream This includes traditional multiprocessors as well as the newer work on networks of workstations (Fig. 9).

The stream description serves as a programmer's view of the computer architecture. There are other factors, such as the interconnection network, that can affect the overall effectiveness of a processor organization.

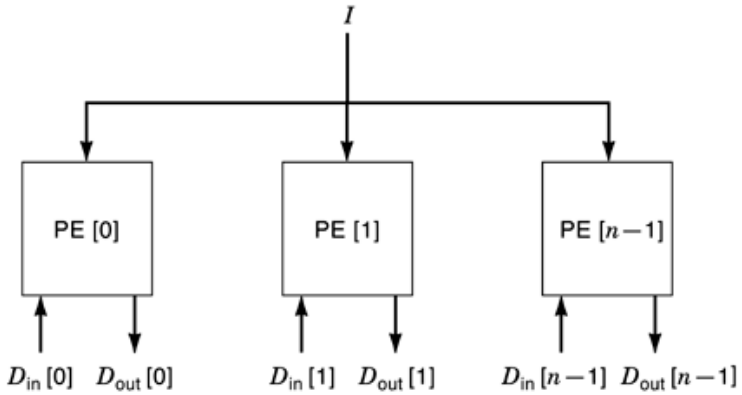

Fig. 7. SIMD—single instruction, multiple data stream.

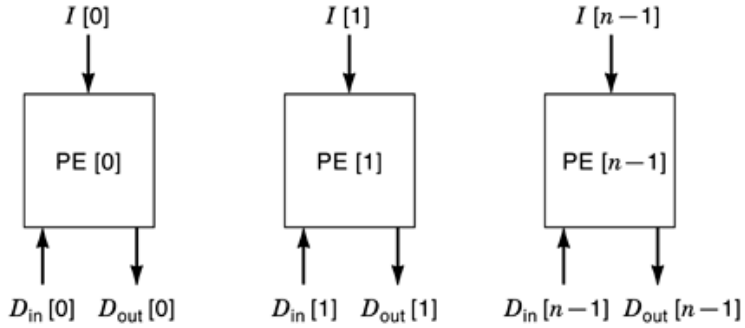

Fig. 8. MISD—multiple instruction, single data stream.

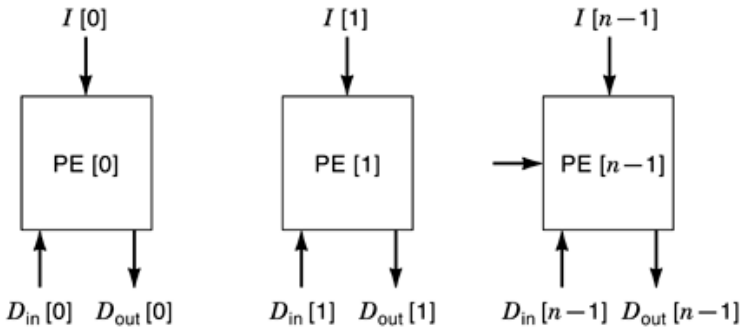

Fig. 9. MIMD—multiple instruction, multiple data stream.

In addition to the stream model, other types of characterizations may be needed to accurately evaluate the processor performance.

SISD—Single Instruction, Single Data Stream. The SISD class of processor architectures includes most commonly available computers. These processors are known as uniprocessors and can be found in millions of embedded processors in home appliances as well as standalone processors in home computers. While a

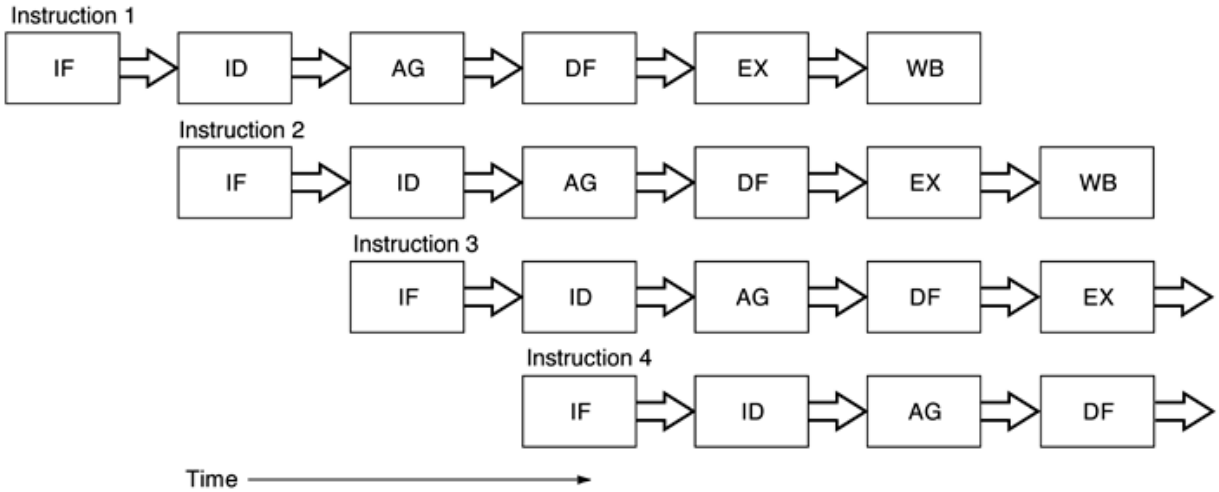

Fig. 10. Instruction timing in a pipelined processor.

programmer may not realize the inherent parallelism within these processors, a good deal of instruction concurrency may be available. For instance, pipelining is a powerful hardware technique that is used in almost all current processor implementations. There are many other software and hardware techniques exploiting instruction-level parallelism.

During execution, a SISD processor executes one or more operations per clock cycle from the instruction stream. An instruction represents the smallest execution packet managed explicitly by the processor. One or more operations can be contained within an instruction. Scalar and superscalar processors consume one or more instructions per cycle, where each instruction contains a single operation. On the other hand, VLIW processors consume a single instruction per cycle where this instruction contains multiple operations. Tables 3 4, and 5 below describe some representative *sequential scalar processors*, *superscalar processors*, and *very long instruction word* (*VLIW*) *processors*.

Sequential Scalar Processor. Scalar processors process at most one instruction per cycle and execute at most one operation per cycle. The simplest scalar processors, sequential scalar processors, process instructions atomically one after another. They process the instructions sequentially from the instruction stream. The next instruction is not processed until the execution for the current instruction is completed. Although conceptually simple, executing each instruction sequentially has significant performance drawbacks—a considerable amount of time is spent in overhead and not in actual execution.

Pipelined Processor. Pipelining is a straightforward approach to exploiting parallelism that is based on the fact that different phases of an instruction (e.g., instruction fetch, decode, execution) can be running concurrently. Pipelining assumes that these phases are independent and can therefore be overlapped. Multiple operations can be processed simultaneously with each operation at a different phase of its processing. Figure 10 illustrates the instruction timing in a pipelined processor, assuming that the instructions are independent. The meaning of each pipeline stage is described in the subsection "Program Execution Unit" above.

At any given time, only one operation is in each phase—thus one operation is being fetched, one operation is being decoded, one operation is accessing operands, one operation is in execution, and one operation is storing results. The most rigid form of a pipeline, sometimes called the *static* pipeline, requires the processor to go through all stages (phases) of the pipeline. A *dynamic* pipeline allows the bypassing of one or more of the stages, depending on the requirements of the instruction.

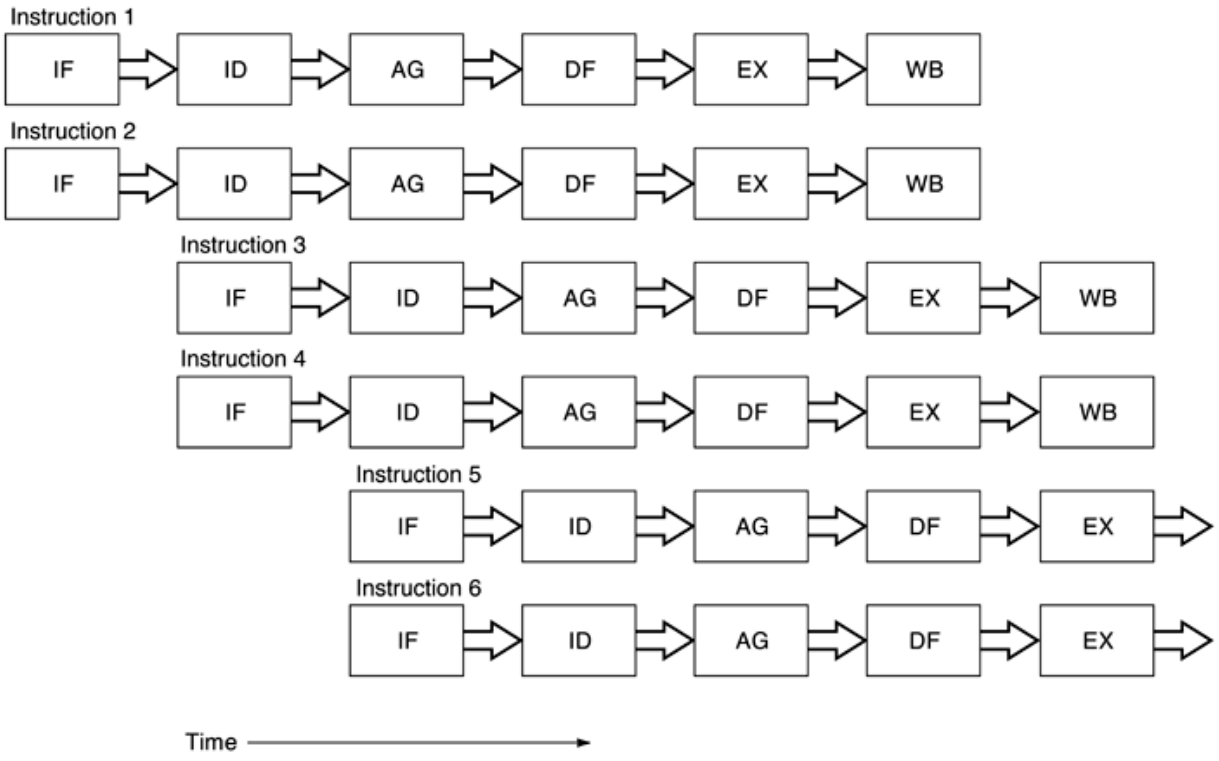

Fig. 11. Instruction timing of a pipelined ILP processor.

Instruction-Level Parallelism. While pipelining does not necessarily lead to executing multiple instructions at exactly the same time, there are other techniques that do. These techniques may use some combination of static scheduling and dynamic analysis to perform concurrently the actual evaluation phases of several different operations—potentially yielding an execution rate greater than one operation per cycle. This kind of parallelism exploits concurrency at the computation level. Since historically most instructions consist of only a single operation, this kind of parallelism is called *instruction-level parallelism* (ILP).

Two common computer architectures that exploit ILP are *superscalar* and *VLIW* processors. Interestingly, these two processors use radically different approaches to execute more than one operation per cycle. A superscalar processor examines the instruction stream dynamically in hardware to determine which operations are independent and can be executed. A VLIW processor relies on the compiler to analyze the available operations and to schedule independent operations into wide instruction words; the processor then executes these operations in parallel with no further analysis. Figure 11 shows the instruction timing of a pipelined superscalar or VLIW processor executing two instructions per cycle. In this case, all the instructions are independent, so that they can be executed in parallel.

Superscalar Processor. Pipelined processors are limited by their scalar nature to executing a single operation per cycle. This limitation can be avoided with the addition of multiple functional units and a dynamic scheduler to process more than one instruction per cycle. These resulting superscalar processors can achieve execution rates of more than one instruction per cycle. The most significant advantage of a superscalar processor is that processing multiple instructions per cycle is done transparently to the user, and that it can provide binary compatibility while achieving better performance.

Compared to a pipelined processor, a superscalar processor adds a scheduling instruction window that dynamically analyzes multiple instructions from the instruction stream. Although processed in parallel, these instructions are treated in the same manner as in a pipelined processor. Before an instruction is issued for execution, dependencies between the instruction and its prior instructions must be checked by hardware. There are two types of instruction dependencies: control dependency and data dependency. Two instructions are *control-dependent* if the result of one instruction can determine the outcome of a conditional branch, which in turn affects the execution of another instruction. Two instructions are *data-dependent* if the result of one instruction directly affects the data input of another instruction. When the issue width gets larger, the checking of control and data dependencies can become very complicated.

Because of the complexity of the dynamic scheduling logic, high-performance superscalar processors are limited to processing four to six instructions per cycle (refer to the subsection "Examples of Recent Architectures" below). Although superscalar processors can take advantage of dynamic execution behavior and exploit instruction-level parallelism from the dynamic instruction stream, exploiting high degrees of instruction requires a different approach. One such approach (the VLIW processor) relies on the compiler to perform the dependency analyses and to eliminate the need for complex analyses performed in hardware.

VLIW Processor. Instead of performing dynamic analyses, VLIW processors rely on static analyses in the compiler to determine which operations can be executed in parallel. Hence, VLIW processors are less complicated and have the potential for higher performance. A VLIW processor executes operations from statically scheduled instructions that contain multiple independent operations. Although it is not required that statically scheduled processors exploit instruction-level parallelism, most statically scheduled processors use wide instruction words. Because the complexity of a VLIW processor is not significantly greater than that of a scalar processor, the improved performance comes without complexity penalties.

On the other hand, VLIW processors cannot take advantage of any dynamic execution characteristics. In real systems, execution rarely proceeds exactly along the path defined by the code scheduler in the compiler. These are two classes of execution variations that can arise and affect the scheduled execution behavior:

(1) Delayed results from operations whose latency differs from the assumed latency scheduled by the compiler
(2) Interruptions from exceptions or interrupts, which change the execution path to a completely different and unanticipated code schedule

The delays arise from many causes, including special-case conditions that require additional cycles to complete an operation. One example is data cache miss; another example is a floating-point operation that requires an additional normalization cycle. Although stalling or freezing the processor can control delayed results, this solution can result in significant performance penalties. For processors without hardware resource management, delayed results can cause resource conflicts and incorrect execution behavior. VLIW processors typically avoid these situations by not using data caches and by assuming worst-case latencies for operations.

Interruptions are usually harder to control than delayed results. Managing interruptions is a significant problem because of their disruptive behavior and because the origins of interruptions are often completely beyond a program's control. Interruptions can arise from external sources (hardware interrupts) or internal sources (software interrupts). Whatever the source, these interruptions cannot be predicted at compile time.

There are many research papers comparing the merits of superscalar and VLIW processors. In general, the optimum solution depends very much on the target application. Currently, superscalar processors are more popular in general-purpose applications, whereas VLIW processors are more popular in graphical and DSP applications.

SIMD—Single Instruction, Multiple Data Stream. The SIMD processor is a natural response to the use of vectors and matrices in programs. From a programmer's standpoint, programming SIMD architecture is similar to programming a SISD processor except that some operations perform computations on aggregate

data. Since these regular structures are widely used in scientific programming, the SIMD processor has been very successful in these environments.

Recently many microprocessors, such as the HP Precision Architectureand the popular Intel x86 family, incorporated a number of *vector instructions* to speed up graphical and other multimedia applications. These instructions enable a SISD processor to perform SIMD operations, as if they were SIMD processors.

The two popular types of SIMD processor are the array processor and the vector processor. They differ both in their implementations and in their data organizations. An array processor consists of many interconnected processor elements that each have their own local memory space. A vector processor consists of a single processor that references a single global memory space and has special function units that operate specifically on vectors. Tables 4 and 5 below describe some representative vector processors and array processors.

Array Processors. The array processor is a set of parallel processor elements connected via one or more networks, possibly including local and global interelement communications and control communications. Processor elements operate in lockstep in response to a single broadcast instruction from a control processor. Each processor element has its own private memory, and data are distributed across the elements in a regular fashion that is dependent on both the actual structure of the data and also the computations to be performed on the data. Direct access to global memory or another processor element's local memory is expensive, so intermediate values are propagated through the array through local interprocessor connections.

Since instructions are broadcast, there is no means local to a processor element of altering the flow of the instruction stream; however, individual processor elements can conditionally disable instructions on the basis of local status information. Typically an array processor is coupled to a general-purpose control processor. The control processor performs the scalar sections of the application, interfaces with the outside world, and controls the flow of execution; the array processor performs the array sections of the application as directed by the control processor.

A suitable application for use on an array processor has several key characteristics: a significant number of data that have a regular structure; computations on the data that are uniformly applied to many or all elements of the data set; simple and regular patterns relating the computations and the data. Basically, any application that has significant matrix manipulation is likely to benefit from the concurrent capabilities of an array processor. One good example is to find the solution of the Navier–Stokes equations.

Vector Processors. A vector processor resembles a traditional SISD processor except that some of the function units and registers operate on *vectors*—sequences of data values that are seemingly operated on as a single entity. These function units are deeply pipelined and have a high clock rate. While the vector pipelines have as long or longer latency than a normal scalar function unit, their high clock rate and the rapid delivery of the input vector data elements results in a large throughput that cannot be matched by scalar function units.

Early vector processors processed vectors directly from memory. The primary advantage of this approach was that the vectors could be of arbitrary lengths and were not limited by processor resources; however, the high startup cost, limited memory-system bandwidth, and memory-system contention proved to be significant limitations. Modern vector processors require that vectors be explicitly loaded into special vector registers and stored back into memory. However, since vector registers can rapidly produce values for or collect results from the vector function units and have low startup costs, modern register-based vector processors achieve significantly higher performance than the earlier memory-based vector processors for the same implementation technology.

Modern processors have several features that enable them to achieve high performance. One feature is the ability to concurrently load and store values between the main memory and the vector register file while performing computations on values in the vector register file. This is an important feature because the limited length of vector registers requires that longer vectors be processed in segments—a technique called *strip mining*. Not being able to overlap memory accesses and computations would pose a significant performance bottleneck.

Most vector processors support result bypassing, known as *chaining*, which allows a follow-on computation to commence as soon as the first value is available from the preceding computation. Instead of waiting for the entire vector to be processed, the follow-on computation can be significantly overlapped with the preceding computation that it is dependent on. Sequential computations can be efficiently compounded and behave as if they were a single operation with a total latency equal to the latency of the first operation plus the pipeline and chaining latencies of the remaining operations, but none of the startup overhead that would be incurred without chaining. For example, division can be synthesized by chaining a reciprocal with a multiply operation. Chaining typically works for the results of load operations as well as normal computations.

A typical vector processor configuration consists of a vector register file, one vector addition unit, one vector multiplication unit, and one vector reciprocal unit. The vector register file contains multiple vector registers. In addition to the vector registers there are also a number of auxiliary and control registers, such the vector length register. The vector length register contains the length of the vector and is used to control the number of elements processed by vector operations.

The vector processor has one primary characteristic, which is the location of the vectors—vectors can be memory- or register-based. There are many other features that vector processors have that are not discussed here, due to their large number and many variations. These include variations on chaining, masked vector operations based on a Boolean mask vector, indirectly addressed vector operations (scatter/gather), compressed/expanded vector operations, reconfigurable register files, and multiprocessor support. Vector processors have developed dramatically from simple memory-based processors to modern multiple processors that exploit both SIMD vector and MIMD processing.

MISD—Multiple Instruction, Single Data Stream. While it is easy to envision and design MISD processors, there has been little interest in this type of parallel architecture. The main reason is that there are no ready programming constructs that easily map programs into the MISD organization. Conceptually, MISD architecture can be represented as multiple independently executing function units operating on a single stream of data, forwarding results from one function unit to the next. At the microarchitecture level, this is exactly what the vector processor does. However, in the vector pipeline the operations are simply fragments of an assembly-level operation, as distinct from being a complete operation. Interestingly, some of the earliest attempts at computers in the 1940s could be seen as the MISD concept. They used plug boards for programs, where data on a punched card were introduced into the first stage of a multistage processor. A sequence of actions was taken where the intermediate results were forwarded from stage to stage until at the final stage a result would be punched into a new card.

There are, however, more useful applications of the MISD organization. Nakamura (see Reading list) has described the use of an MISD machine called the SHIFT machine. In the SHIFT machine, all data memory is decomposed into shift registers. Various function units are associated with each shift column. Data are initially introduced into the first column and are shifted across the shift-register memory. In the SHIFT machine concept, data are regularly shifted from memory region to memory region (column to column) for processing by various function units. The purpose behind the SHIFT design is to reduce memory latency. In a traditional organization, any function unit can access any region of memory, and the worst-case delay path for accessing memory must be taken into account. In the SHIFT machine, we need only allow for access time to the worst element in a data column. The memory latency in modern machines is becoming a major problem; the SHIFT machine has a natural appeal for its ability to tolerate this latency.

MIMD—Multiple Instruction, Multiple Data Stream. The MIMD class of parallel architecture brings together multiple processors with some form of interconnection. In this configuration, each processor executes completely independently, although most applications require some form of synchronization during execution to pass information and data between processors. While there is no requirement that all processor elements be identical, most MIMD configurations are homogeneous with all processor elements identical.

MIMD Implementation Considerations. The MIMD processor, with its multiple processor elements interconnected by a network, may appear very similar to the SIMD array processor. However, in the SIMD

processor the instruction stream delivered to each processor element is the same, while in the MIMD processor the instruction stream delivered to each processor element is independent. In the MIMD processor, the instruction stream for each processor element is generated independently by that processor element as it executes its program.

When communication between processor elements in the MIMD processor is performed through a shared memory address space, a number of significant problems arise. The biggest problem is bus contention, which limits the number of processors in a shared bus. One effective way to reduce bus contention is to use private cache and memory for each processor. However, this creates cache and memory coherency problems, as different caches and memories may share the same data.

The memory consistency problem is usually solved through a combination of hardware and software techniques. At the MIMD processor level, memory consistency is often only guaranteed through explicit synchronization between processors. In this case, nonlocal references are ordered only relative to these synchronization points. The cache coherency problem is usually solved exclusively through hardware techniques. This problem is significant because of the possibility that multiple processor elements will have copies of data in their local caches with different values.

The primary characteristic of a MIMD processor is the nature of the memory address space—it is either separate or shared for all processor elements. The interconnection network is also important in characterizing a MIMD processor and is described in the next section. With a separate address space, the only means of communications between processor elements is through messages, and thus these processors force the programmer to use a message-passing paradigm. With a shared address space, communications between processor elements is through the memory system—depending on the application needs or a programmer preference, either a shared-memory or a message-passing paradigm can be used.

MIMD processors usually are chosen for at least one of two reasons: fault tolerance and program speedup. Ideally, if we have n identical processors, the failure of one processor should not affect the ability of the multiprocessor to continue program execution. Many multiprocessor ensembles have been built with the sole purpose of high-integrity, fault-tolerant computation. Generally, these systems may not provide any program speedup over a single processor. Systems that duplicate computations or that triplicate and vote on results are examples of designing for fault tolerance. Since multiprocessors simply consist of multiple computing elements, each computing element is subject to the same basic design issues. These elements are slowed down by branch delays, cache misses, and so on. The multiprocessor configuration, however, introduces speedup potential as well as additional sources of delay and performance degradation. The sources of performance bottlenecks in multiprocessors generally relate to the way the program was decomposed to allow concurrent execution on multiple processors. The achievable MIMD speedup depends on the amount of parallelism available in the program and how well the partitioned tasks are scheduled.

Partitioning is the process of dividing a program into tasks, each of which can be assigned to an individual processor for execution at run time. The program partitioning is usually performed with some *a priori* notion of program overhead. The *program overhead o* is the added time a task takes to be loaded into a processor prior to beginning execution. The larger the size of the minimum task set by the partitioning program, the smaller the effect of program overhead. Table 1 gives an instruction count for some various program grain sizes.

Scheduling can be performed statically at compile time or dynamically at run time. Static scheduling information can be derived on the basis of the probable critical paths. This alone is insufficient to ensure optimum speedup or even fault tolerance. In general, the processor availability is difficult to predict and may vary from run to run. While run-time scheduling has obvious advantages, handling changing systems environments, as well as highly variable program structures, it also has some disadvantages—primarily its run-time overhead. Run-time scheduling can be performed in a number of different ways. The scheduler may run on a particular processor, or it may run on any processor. It is usually desirable that the scheduling not be assigned to a particular processor, but rather that any processor be able to initiate it, and then the scheduling process itself be distributed across all available processors.

Table 1. Grain Size

Grain Description	Program Construct	Typical Number of Instructions
Fine grain	Basic block: instruction-level parallelism	5 to 10
Medium grain	Loop/procedure: loop-level parallelism; procedure-level parallelism	100 to 100,000
Coarse grain	Large task: program-level parallelism	100,000 or more

Types of MIMD Processors. While all MIMD architectures share the same general programming model, there are many differences in programming detail, hardware configuration, and speedup potential. Most differences arise from the variety of shared hardware, especially the way the processors share memory.

Multithreaded or Shared-Resource Multiprocessing. The simplest and most primitive type of multiprocessor system is what is sometimes called multithreaded or what we call here *shared-resource multiprocessing* (*SRMP*). In the SRMP, each of the processors consists of basically only a register set—program counter, general registers, instruction counter, and so on. The driving principle behind SRMP is to make the best use of processor silicon area. The functional units and buses are time-shared. The objective is to eliminate context-switching overhead and to reduce the realized effect of branch and cache miss penalties. Each "processor" executes without significant instruction level concurrency, so it executes more slowly than a more typical SISD.

Symmetric Multiprocessing. In the simplest symmetric multiprocessing (*SMP*) configurations several processors share a common memory via a common bus. They may even share a common data cache or level-2 cache. Since bus bandwidth is limited, the number of processors that can be usefully configured in this way is also limited. Since this configuration provides a uniform memory access time, it is also called uniform memory architecture. Computers made by Sequent and Encore employ this type of architecture.

Nonuniform Memory Architecture. Realizing multiprocessor configurations beyond a shared bus requires a distributed shared-memory system. This is known as nonuniform memory architecture (*NUMA*). The interconnection network provides multiple switched paths, thereby increasing the intercluster bandwidth at the expense of the switch latency in the network and the overall cost of the network. Programming such systems may be done in either a shared-memory or a message-passing paradigm. The shared-memory approach requires significant additional hardware support to ensure the consistency of data in the memory. Message passing has simpler hardware but is a more complex programming model.

Clusters: Networked Multiprocessors. Simple processor-memory systems with LAN or even Internet connection can, for particular problems, be quite effective multiprocessors. Such a configuration is sometimes called a *cluster* or a *network of workstations* (*NOW*) (7).

Table 2 illustrates some of the tradeoffs possible in configuring multiprocessor systems. Note that the application determines the effectiveness of the system. As architects consider various ways of facilitating interprocessor communication in a shared-memory multiprocessor, they must be constantly aware of the cost.

Table 2. Multiprocessor Configurations

Type	Physical Sharing	Programmer's Model	Remote Data Access Latency	Comments
Multithreaded	ALU, data cache, memory	Shared memory	No delay	Eliminates context switch overhead but limited possible speedup
SMP	Bus and memory	Shared memory	Small delay due to bus congestion	Limited speedup due to bus bandwidth limits.
NUMA (1)	Interconnection network and memory	Shared memory	Order of 100 cycles.	Typically 16 to 64 processors; requires memory consistency support.
NUMA (2)	Interconnection network	Message passing	Order of 100 cycles plus message decode overhead	Scalable by application; needs programmer's support.
Clusters	Only LAN or similar network	Message passing	More than 0.1 ms.	Limited to applications that require minimum communications.

In a typical shared-memory multiprocessor, the cost does not scale linearly—each additional processor requires additional network services and facilities. Depending on the type of interconnection, the cost for an additional processor may increase at a greater than linear rate.

For those applications that require rapid communications and have a great deal of interprocessor communications traffic, this added cost is quite acceptable. It is readily justified on a cost-performance basis. However, many other applications, including many naturally parallel applications, may have limited interprocessor communications. In many simulation applications, the various cases to be simulated can be broken down and treated as independent tasks to be run on separate processors with minimum interprocessor communication. For these applications, clusters of workstations provide perfectly adequate communications services. Table 8 below shows some representative MIMD computer systems from 1990 to 2000.

Comparisons and Conclusions

Examples of Recent Architectures. This section describes some recent microprocessors and computer systems, and illustrates how computer architecture has evolved over time. In the subsection on SISD above, scalar processors are described as the simplest kind of SISD processor, capable of executing only one instruction at a time. Table 3 describes some commercial scalar processors released from 1978 to 1997 (8,9).

Table 3. Typical Scalar Processors (SISD)

Processor	Year of Introduction	Number of Function Units	Issue Width	Scheduling	Number of Transistors
Intel 8086	1978	1	1	Dynamic	29 K
Intel 80286	1982	1	1	Dynamic	134 K
Intel 80486	1989	2	1	Dynamic	1.2 M
HP PA-RISC 7000	1991	1	1	Dynamic	580 K
Sun SPARC	1992	1	1	Dynamic	1.8 M
MIPS R4000	1992	2	1	Dynamic	1.1 M
ARM 610	1993	1	1	Dynamic	360 K
ARM SA-1100	1997	1	1	Dynamic	2.5 M

The Intel 8086, which was released in 1978, consists of only 29 thousand transistors. In contrast, Pentium III (from the same x86 family) contains more than 28 million transistors. The huge increase in the transistor count is made possible by the phenomenal advancement in VLSI technology. These transistors allow simple scalar processors to emerge to a more complicated architecture and achieve better performance. Many processor families, such as Intel x86, HP PA-RISC, Sun SPARC, and MIPS) have evolved from scalar processors to superscalar processors, exploiting a higher level of instruction-level parallelism. In most cases, the migration is transparent to the programmers, as the binary codes running on the scalar processors can continue to run on the superscalar processors. At the same time, simple scalar processors (such as MIPS R4000 and ARM processors) still remain very popular in embedded systems, because performance is less important than cost, power consumption, and reliability for most embedded applications.

Table 4 shows some representative *superscalar processors* from 1992 to 2000 (8,9). In this period, the number of transistors in a superscalar processor escalated from a million to more than 100 million. Interestingly, most transistors are not used to improve the instruction-level parallelism in the superscalar architectures. Actually, the instruction issue width remains roughly the same (between 2 and 6), because the overhead to build a wider machine in turn can adversely affect the overall processor performance. In most cases, many of these transistors are used in the on-chip cache to reduce the memory access time. For instance, most of the 140 million transistors in HP PA-8500 are used in the 1.5 Mbyte on chip cache.

Table 5 presents some representative VLIW processors (8,10). There have been very few commercial VLIW processors in the past, mainly on account of the poor compiler technology. Recently, however, there has been major advancement in VLIW compiler technology. In 1997 TI TMS320/C62x became the first DSP chip using VLIW architecture. The simple architecture allows TMS320/C62x to run at a clock frequency (200 MHz) much higher than traditional DSPs. After the demise of Multiflow and Cydrome, HP acquired their VLIW technology and codeveloped the IA-64 architecture (the first commercial general-purpose VLIW processor) with Intel.

Some SISD processors described earlier, such as ARM and TMS320/C62, are *embedded* processors, meaning that they are embedded in a system running some specialized target application. Embedded processors represent the fastest-growing processor market segment. In deep-submicron era, embedded-processor cores are

Table 4. Typical Superscalar Processors (SISD)

Processor	Year of Introduction	Number of Function Units	Issue Width	Scheduling	Number of Transistors
HP PA-RISC 7100	1992	2	2	Dynamic	850 K
Motorola PowerPC 601	1993	4	3	Dynamic	2.8 M
MIPS R8000	1994	6	4	Dynamic	3.4 M
DEC Alpha 21164	1994	4	4	Dynamic	9.3 M
Motorola PowerPC 620	1995	4	2	Dynamic	7 M
MIPS R10000	1995	5	4	Dynamic	6.8 M
HP PA-RISC 7200	1995	3	2	Dynamic	1.3 M
Intel Pentium Pro	1995	5	3, 6[a]	Dynamic	5.5 M
DEC Alpha 21064	1992	4	2	Dynamic	1.7 M
Sun Ultra I	1995	9	4	Dynamic	5.2 M
Sun Ultra II	1996	9	4	Dynamic	5.4 M
AMD K5	1996	6	4, 4[a]	Dynamic	4.3 M
Intel Pentium II	1997	5	3, 6[a]	Dynamic	7.5 M
AMD K6	1997	7	2, 6[a]	Dynamic	8.8 M
Motorola PowerPC 740	1997	6	3	Dynamic	6.4 M
DEC Alpha 21264	1998	6	4	Dynamic	15.2 M
HP PA-RISC 8500	1998	10	4	Dynamic	140 M
Motorola PowerPC 7400	1999	10	3	Dynamic	6.5 M
AMD K7	1999	9	3, 6[a]	Dynamic	22 M
Intel Pentium III	1999	5	3, 6[a]	Dynamic	28 M
Sun Ultra III	2000	6	4	Dynamic	29 M
DEC Alpha 21364	2000	6	4	Dynamic	100 M

[a] For some Intel ×86 family processors, each instruction is broken into a number of micro-operation codes in the decoding stage. In this article, two different issue widths are given for these processors—the first one is the maximum number of instructions issued per cycle, and the second one is the maximum number of micro-operation codes issued per cycle.

the crucial components in system-on-chip designs. Interestingly, some processors, such as PowerPC and Pentium families, are used in both server and embedded applications, although the implementations are usually different in view of the two different design objectives.

Table 5. Typical VLIW Processors (SISD)

Processor	Year of Introduction	Number of Function Units	Issue Width	Scheduling	Issue/Complete Order
Multiflow Trace 7/200	1987	7	7	Static	In-order/in-order
Multiflow Trace 14/200	1987	14	14	Static	In-order/in-order
Multiflow Trace 28/200	1987	28	28	Static	In-order/in-order
Cydrome Cydra 5	1987	7	7	Static	In-order/in-order
Philips TM-1	1996	27	5	Static	In-order/in-order
TI TMS320/C62x	1997	8	8	Static	In-order/in-order
Intel IA-64	2001	9	6	Static	In-order/in-order

While SISD processors and computer systems are commonly used for most consumer and business applications, SIMD and MIMD computers are used extensively for scientific and high-end business applications. As described in the preceding section, *vector processors* and *array processors* are the two types of SIMD architecture. In the last twenty-five years, vector processors have developed from a single-processor unit (Cray 1) to 512-processor units (NEC SX-5), taking advantage of both SIMD and MIMD processing. Table 6 shows some representative vector processors. On the other hand, there have not been many array processors, due to a limited application base and market requirement. Table 7 shows several representative array processors.

For MIMD computer systems, the primary considerations are the characterization of the memory address space and the interconnection network among the processing elements. The comparison of shared-memory and message-passing programming paradigms is discussed in the subsection on MIMD above. At this time, shared-memory programming is more popular, mainly because of its flexibility and ease of use. As shown in Table 8, the latest Cray supercomputer (Cray T3E-1350), which consists of up to 2,176 DEC Alpha 21164 processors with distributed memory modules, adopts shared-memory programming. As for computer clusters, IBM Sysplex and Parallel Sysplex are popular examples that provide coarse-grained instruction parallelism.

Concluding Remarks. Computer architecture has evolved greatly over the past decades. It is now much more than the programmer's view of the processor. The process of computer design starts with the implementation technology. As the semiconductor technology changes, so to does the way it is used in a system. At some point in time cost may be largely determined by transistor count; later, as feature sizes shrink, wire density and interconnection may dominate cost. Similarly, the performance of a processor is dependent on delay, but the delay that determines performance changes as the technology changes. Memory access time is only slightly reduced by improvements in feature size, because memory implementations stress size, and the access delay is largely determined by the wire length across the memory array. As feature sizes shrink, the array simply gets larger.

The computer architect must understand technology—not only today's technology, but the projection of that technology into the future. A design begun today may not be broadly marketable for several years. It is the technology that is actually used in manufacturing, not today's technology, that determines the effectiveness of a design.

The foresight of the designer in anticipating changes in user applications is another determinant in design effectiveness. The designer should not be blinded by simple test programs or benchmarks that fail to project the dynamic nature of the future marketplace.

Table 6. Typical Vector Computers (SIMD)

Processor	Year of Introduction	Memory- or Register-based	Number of Processor Units	Maximum Vector Length
Cray 1	1976	Register	1	64
CDC Cyber 205	1981	Memory	1	65,535
Cray X-MP	1982	Register	1–4	64
Cray 2	1985	Register	5	64
Fujitsu VP-100/200	1985	Register	3	32–1024
ETA ETA	1987	Memory	2–8	65,535
Cray Y-MP/832	1989	Register	1–8	64
Cray Y-MP/C90	1991	Register	16	64
Convex C3	1991	Register	1–8	128
Cray T90	1995	Register	1–32	128
NEC SX-5	1998	Register	1–512	256

Table 7. Typical Array Processors (SIMD)

Processor	Year of Introduction	Memory Model	Processor Element	Number of Processors
Burroughs BSP	1979	Shared	General purpose	16
Thinking Machine CM-1	1985	Distributed	Bit-serial	Up to 65,536
Thinking Machine CM-2	1987	Distributed	Bit-serial	4,096–65,536
MasPar MP-1	1990	Distributed	Bit-serial	1,024–16,384

The computer architect must bring together the technology and the application behavior into a system configuration that optimizes the available process concurrency. This must be done is a context of constraints on cost, power, reliability, and usability. While formidable in objective, a successful design is a design that provides lasting value to the user community.

Table 8. Typical MIMD Systems

System	Year of Introduction	Processor Element	Number of Processors	Memory Distribution	Programming Paradigm
Alliant FX/2800	1990	Intel i860	4–28	Central	Shared memory
Stanford DASH	1992	MIPS R3000	4–64	Distributed	Shared memory
Cray T3D	1993	DEC 21064	128–2048	Distributed	Shared memory
MIT Alewife	1994	Sparcle	1–512	Distributed	Message passing
Convex C4/XA	1994	Custom	1–4	Global	Shared memory
Thinking Machines CM-500	1995	SuperSPARC	16–2048	Distributed	Message passing
Tera Computers MTA	1995	Custom	16–256	Distributed	Shared memory
SGI Power Challenge XL	1995	MIPS R8000	2–18	Global	Shared memory
Convex SPP1200/XA	1995	PA-RISC 7200	8–128	Global	Shared memory
Cray T3E-1350	2000	DEC 21164	40–2176	Distributed	Shared memory

BIBLIOGRAPHY

In preparing this article, our general presentation follows that found in Refs. (3,10,11). Some material presented here is abstracted from these sources. There are a number of texts available for further reading on this subject. In addition to Ref. 11 mentioned above, Refs. 12 to 16 are widely available:

1. G. M. Amdahl G. H. Blaauw F. P. Brooks Architecture of the IBM System/360, *IBM J. Res. Devel.*, **8**(2): 87–101, 1964.
2. Semiconductor Industry Association, *The National Technology Roadmap for Semiconductors*, San Jose, CA, 1997.
3. M. J. Flynn P. Hung K. W. Rudd Deep-submicron microprocessor design issues, *IEEE Micro Mag.* **19**(4): 11–22, July–August, 1999.
4. J. D. Ullman *Computational Aspects of VLSI*, Rockville, MD: Computer Science Press, 1984.
5. W. Stallings *Reduced Instruction Set Computers*, Tutorial, 2nd ed., New York: IEEE Computer Society Press, 1989.
6. M. J. Flynn Very high speed computing systems, *Proc. IEEE*, **54**: 1901–1909, 1966.
7. G. Pfister *In Search of Clusters*, Upper Saddle River, NJ: Prentice Hall, 1998.
8. MicroDesign Resources, Microprocessor Report, various issues, Sebastopol, CA, 1992–2001.
9. Tom Burd *General Processor Information* [online], CPU Info Center, University of California, Berkeley, 2001. Available http://bwrc.eecs.berkeley.edu/CIC/summary/
10. M. J. Flynn K. W. Rudd Parallel architectures, *ACM Comput. Surv.*, **28**(1): 67–70, 1996.
11. M. J. Flynn *Computer Architecture: Pipelined and Parallel Processor Design*, Sudbury, MA: Jones and Bartlett, 1995.
12. D. Culler J. P. Singh A. Gupta *Parallel Computer Architecture: A Hardware / Software Approach*, San Francisco: Morgan Kaufman, 1988.
13. D. Sima T. Fountain P. Kacsuk *Advanced Computer Architectures: A Design Space Approach*, Essex, England: Addison-Wesley, 1997.
14. W. Stallings *Computer Organization and Architecture*, 5th ed., Upper Saddle River, NJ: Prentice Hall, 2000.
15. K. Hwang *Advanced Computer Architecture*, New York: McGraw Hill, 1993.
16. J. Hennessy D. Patterson *Computer Architecture: A Quantitative Approach*, San Francisco: Morgan Kaufman, 1996.

READING LIST

W. M. Johnson *Superscalar Microprocessor Design*, Englewood Cliffs, NJ: Prentice-Hall, 1991.
P. M. Kogge *The Architecture of Pipelined Computers*, New York: McGraw-Hill, 1981.

S. Kunkel J. Smith Optimal pipelining in supercomputers, *Proc. 13th Annual Symposium on Computer Architecture*, 1986, pp. 404–411.
C. D. Lima *et al.* A technology-scalable multithreated architecture, *Proc. 13th Symposium on Computer Architecture and High Performance Computing*, 2001, pp. 82–89.
K. W. Rudd VLIW Processors: Efficiently exploiting instruction level parallelism, Ph.D. Thesis, Stanford University, 1999.
A. J. Smith Cache memories, *Comput. Surv.*, **14**(3): 473–530, 1982.

M. FLYNN
P. HUNG
Stanford University